

25 de noviembre de 2025

Procesador de MyJS: *jsp*

Memoria del Grupo 59

Andrés Súnico

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.....	13
8. Diseño del Gestor de Errores.....	13
A. Casos de Prueba	14

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

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

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

P se compone de la regla del axioma:

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

Y del resto de reglas:

$$A \rightarrow = \mid \lambda$$

$$B \rightarrow = \mid \lambda$$

$$C \rightarrow = \mid \lambda$$

$$D \rightarrow = \mid \lambda$$

$$E \rightarrow = \mid \lambda$$

$$F \rightarrow \mid$$

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

$$K \rightarrow dL$$

$$L \rightarrow dL \mid \lambda$$

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

$$M \rightarrow nH \mid tH$$

$$I \rightarrow c_3I \mid \lambda$$

$$J \rightarrow *N \mid \lambda$$

$$N \rightarrow c_4N \mid *O$$

$$O \rightarrow c_5N \mid *O \mid /S$$

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

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*.

4.5.2 Errores

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

```
1 error("Malformed_string_literal")
```

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

```
1 error("Unterminated_string_literal")
```

INV_FLOAT_FMT Si en el estado 19 no se puede transitar

```
1 error("Invalid_float_literal_format")
```

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

```
1 error("Unterminated_comment")
```

INV_CHAR Ante cualquier error no manejado en el resto de acciones

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

4.5.3 Generación Directa

GEN_MUL En la transición 0:1

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

GEN_SUB En la transición 0:2

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

GEN_SUM En la transición 0:3

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

GEN_RBRACK En la transición 0:4

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

GEN_LBRACK En la transición 0:5

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

GEN_RPAREN En la transición 0:6

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

GEN_LPAREN En la transición 0:7

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

GEN_SEMI En la transición 0:8

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

GEN_COMMA En la transición 0:9

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

GEN_NOT En la transición 0:10

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

GEN_LT En la transición 0:11

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

GEN_EQ En la transición 12:13

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

GEN_ASSIGN En la transición 12:14

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

GEN_ANDASSIGN En la transición 15:16

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

GEN_AND En la transición 15:17

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

4.5.4 Generación de Números

INIT_NUM En la transición 0:18

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

INIT_DEC En la transición 20:21

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

GEN_DEC En la transición 21:22

```
1 if (num > 3.4028235e38) {
2     error("Float_literal_out_of_range")
3 } else {
4     gen_token(FloatLit, num)
5 }
```

ADD_INTDIG En la transición 18:18

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

ADD_DECDIG En las transiciones 21:21

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

GEN_INT En la transición 18:20

```
1 if (num > 32767) {
2     error("Integer_literal_out_of_range")
3 } else {
4     gen_token(IntLit, num)
5 }
```

4.5.5 Generación de Cadenas e Identificadores

INIT_STR En la transición 0:23

```
1 lex := ""
2 len := 0
```

ADD_CHAR_STR En la transición 23:23

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

ADD_CHAR_ID En la transición 0:26, 26:26

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

ADD_ESCSEQ En la transición 24:23

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

INIT_ID En la transición 0:26

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

GEN_STR En la transición 23:25

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

GEN_ID En la transición 26:27

```
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

El siguiente gran módulo del procesador es el Analizador Sintáctico o *Parser*. El *Parser* consume los *tokens* del *Lexer* y los utiliza para producir el árbol sintáctico abstracto o *AST*.

El *AST* es una *TAD* que representa la estructura sintáctica del fichero fuente, dónde cada nodo corresponde a una construcción sintáctica del lenguaje.

Para este proyecto, se ha escogido implementar un *Parser* ascendente de tipo $SLR(1) \in LR(1)$. A diferencia de un analizador descendente $LL(1)$, se construye el *AST* desde las hojas hasta la raíz, lo que suele permitir una generación más directa y eficiente del árbol.

6.1 Gramática

Se define la gramática del *Parser* como la tupla $G = (T, N, P, R)$, dónde:

$T = \{\text{Todo token definido por el } \textit{Lexer}\}$

$N = \{E_1, R_1, R_2, U_1, U_2, E_2, V, S, L, Q, X, B, T, M, F, F_1, F_2, F_3, H, A, K, C, P\}$

R se compone de:

$P \rightarrow BP \mid FP \mid \lambda$
 $C \rightarrow BC \mid \lambda$
 $F \rightarrow \text{Func } F_1 F_2 F_3 \text{ LBrack } C \text{ RBrack}$
 $F_1 \rightarrow H$
 $F_2 \rightarrow \text{Id}$
 $F_3 \rightarrow \text{LParen } A \text{ RParen}$
 $H \rightarrow T \mid \text{Void}$
 $A \rightarrow T \text{ Id } K \mid \text{Void}$
 $K \rightarrow \text{Comma } T \text{ Id } K \mid \lambda$
 $B \rightarrow \text{If LParen } E \text{ RParen } S \mid \text{Do LBrack } C \text{ RBrack While LParen } E \text{ RParen Semi}$
 $B \rightarrow S \mid \text{Let } MT \text{ Id Semi} \mid \text{Let } MT \text{ Id Assign } E \text{ Semi} \mid \text{Let } MT \text{ Id AndAssign } E \text{ Semi}$
 $M \rightarrow \lambda$
 $T \rightarrow \text{Int} \mid \text{Float} \mid \text{Bool} \mid \text{Str}$
 $S \rightarrow \text{Write } E \text{ Semi} \mid \text{Read Id Semi} \mid \text{Ret } X \text{ Semi}$
 $S \rightarrow \text{Id Assign } E \text{ Semi} \mid \text{Id AndAssign } E \text{ Semi} \mid \text{Id LParen } L \text{ RParen Semi}$
 $L \rightarrow EQ \mid \lambda$
 $Q \rightarrow \text{Comma } EQ \mid \lambda$
 $X \rightarrow E \mid \lambda$
 $E_1 \rightarrow E_1 \text{ And } R_1 \mid R_1$
 $R_1 \rightarrow R_1 \text{ Eq } R_2 \mid R_2$
 $R_2 \rightarrow R_2 \text{ Lt } U_1 \mid U_1$
 $U_1 \rightarrow U_1 \text{ Sum } U_2 \mid U_2$
 $U_2 \rightarrow U_2 \text{ Mul } E_2 \mid E_2$
 $E_2 \rightarrow \text{Not } E_2 \mid \text{Sub } E_2 \mid \text{Sum } E_2 \mid V$
 $V \rightarrow \text{Id LParen } L \text{ RParen} \mid \text{LParen } E \text{ RParen} \mid \text{IntLit} \mid \text{FloatLit} \mid \text{StrLit} \mid \text{True} \mid \text{False} \mid \text{Id}$

G se trata de una gramática de contexto libre y el lenguaje que genera, $L(G)$, está compuesto por la estructura de todos los programas sintácticamente correctos.

De este modo, un fichero fuente sintácticamente correcto se puede interpretar como una palabra $w \in L(G)$ y visto así, el AST se trata justamente del árbol de derivación de w , por lo que es fácil de obtener a partir de la secuencia de reglas aplicadas por el *Parser* para reducir w al axioma S .

Esta secuencia de reglas se llama el *parse* de un programa, y los próximos apartados se centran en como hallarlo.

6.2 Autómata

G se trata de una gramática de contexto libre, por lo su lenguaje no puede reconocerse directamente con un *AFD*. Sin embargo, si se aumenta su gramática a $G' = (T, N, P', R')$, con $R' = R \cup \{P' \rightarrow P\}$, y se verifica que $G' \in SLR(1)$, entonces la colección completa de conjuntos de ítems $LR(0)$ de G' sí que representa un *AFD*: el que reconoce todos los prefijos viables de G . Nótese que $L(G) = L(G')$ por lo que son equivalentes.

Los estados de dicho autómata representan los estados intermedios de análisis tras consumir una secuencia de símbolos potencialmente correcta, y se construye mediante las operaciones de *closure* y de *goto*.

6.3 Tablas de Acción y Goto

El autómata se puede codificar mediante las tablas de acción y *goto*, que si se pueden construir sin conflictos, garantizan que $G' \in SLR(1)$, es decir, que el analizador puede utilizar estas tablas para reconocer cualquier cadena de $L(G)$ de forma determinista y generar su *parse*.

Cuadro 3: Tabla de Acción

	if	do	while	int	float	str	bool	void	let	func	ret	read	write	true	false	FloatLit	IntLit	StrLit	Id	=	&=	,	;	()	{	}	+	-	*	&&	!	<	==	\$
0	s7	s5							s6	s4	s9	s10	s11						s12																r1
1																																			acc
2	s7	s5							s6	s4	s9	s10	s11						s12																r1
3	s7	s5							s6	s4	s9	s10	s11						s12																r1
4				s22	s21	s19	s20	s15																											
5																										s23									
6				r18	r18	r18	r18																												
7																										s25									
8	r28	r28							r28	r28	r28	r28	r28						r28								r28								r28
9														s30	s29	s32	s33	s31	s28					r16	s34				s37	s38		s36			
10																			s44																
11														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
12																				s48	s47				s46										
13																																			r2
14																																			r3
15																			r10																
16																			r11																
17																			r14																
18																			s49																
19																			r19																
20																			r20																
21																			r21																
22																			r22																
23	s7	s5							s6		s9	s10	s11						s12								r4								
24				s22	s21	s19	s20																												
25														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
26																																s55			
27																																			
28																																			
29																																			
30																																			
31																																			
32																																			
33																																			
34														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
35																																			
36														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
37														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
38														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
39																																			
40																																			
41																																			
42																																			
43																																			
44																																			
45																																			
46														s30	s29	s32	s33	s31	s28						s34	r31			s37	s38		s36			
47														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
48														s30	s29	s32	s33	s31	s28						s34				s37	s38		s36			
49																																			

Cuadro 6: Tabla de Goto

	E ₁	R ₁	R ₂	U ₁	U ₂	E ₂	V	S	L	Q	X	B	T	M	F	F ₁	F ₂	F ₃	H	A	K	C	P
0								8				3			2								1
1																							
2								8				3			2								13
3								8				3			2								14
4													16			18			17				
5																							
6														24									
7																							
8																							
9	26	43	42	41	40	39	35				27												
10																							
11	45	43	42	41	40	39	35																
12																							
13																							
14																							
15																							
16																							
17																							
18																	50						
19																							
20																							
21																							
22																							
23								8				51										52	
24													53										
25	54	43	42	41	40	39	35																
26																							
27																							
28																							
29																							
30																							
31																							
32																							
33																							
34	58	43	42	41	40	39	35																
35																							
36						59	35																
37						60	35																
38						61	35																
39																							
40																							
41																							
42																							
43																							
44																							
45																							
46	68	43	42	41	40	39	35		69														
47	70	43	42	41	40	39	35																
48	71	43	42	41	40	39	35																
49																							
50																		73					
51								8				51										74	
52																							
53																							
54																							
55		78	42	41	40	39	35																
56																							
57	68	43	42	41	40	39	35		79														
58																							
59																							

6.4 Algoritmo

El recorrido de las tablas se realizará con un *stack* dónde se irán apilando los símbolos no terminales hasta ser reducidos al axioma. Además, cada vez que se efectúe una reducción, el índice de la regla utilizada se añadirá al *parse*, quedando este completamente formado al terminar el procedimiento.

En la tabla acción las celdas quieren decir:

- sN : desplazar y apilar el estado n -ésimo
- rN : reducir por la regla n -ésima
- acc : aceptar la cadena

En la tabla goto:

- N : apilar el estado n -ésimo

En ambas tablas una celda en blanco indica un error sintáctico.

6.5 Conflictos

Hay dos tipos de conflictos, conflictos reducción-reducción y conflictos desplazamiento-reducción. Ambos ocurren cuando se intenta escribir una acción en una celda no vacía de la tabla acción.

Como no han habido colisiones durante la construcción de la tabla, se confirma que $G' \in SLR(1)$. Por tanto, se va a poder implementar el *Parser LR(1)* exitosamente con la gramática escogida.

6.6 Errores

Por ahora sólo se ha lanzado un tipo de error en el *Parser*³, *Token Inesperado*. Sin embargo, su mensaje varía según la celda en la que se lance, tomando el formato:

"expected <opción 1>, <opción 2>, ..., <opción n-1> or <opción n> before <token inesperado>"

Cuadro 9: Listado de errores del *Parser*

Error	Severidad
Token Inesperado	<i>error</i>

La lista de *tokens* esperados se obtiene realizando una búsqueda en profundidad en las tablas por cada *token* con una celda no vacía en la fila del fallo. Los *tokens* que desplazan siempre serán válidos, pero hay casos en los que un *token* reduce una o varias veces pero acaba terminando en error, por lo que es necesario simular cada uno de ellos hasta llegar a un desplazamiento.

7 Diseño del Semanter

Se detallará en la entrega final.

8 Diseño del Gestor de Errores

Se detallará en la entrega final.

³ Tras implementar el Analizador Semántico, se podrán enriquecer los errores del *Parser* con mayor facilidad y se añadirán más tipos.

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      let int a = 0;
10     let int b = 1;
11
12     do {
13         let int c = a + b;
14         a = b; b = c;
15
16         n = n + -1;
17     } while (!(n < 2));
18
19     return b;
20 }
```

Fichero de *tokens*:

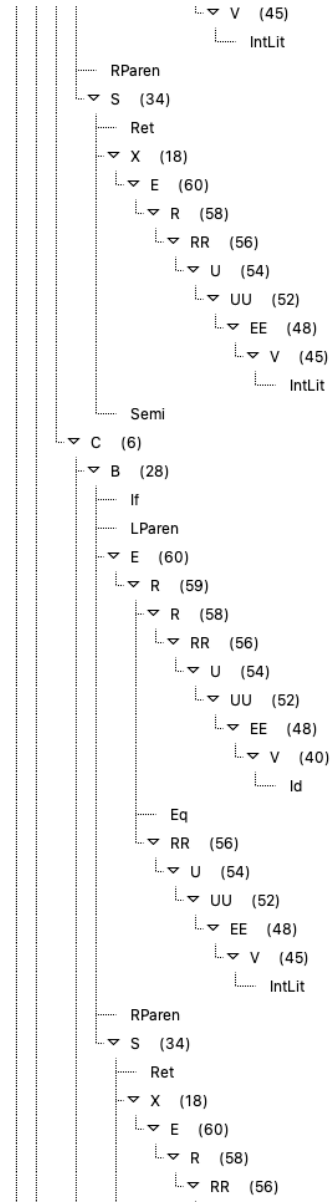
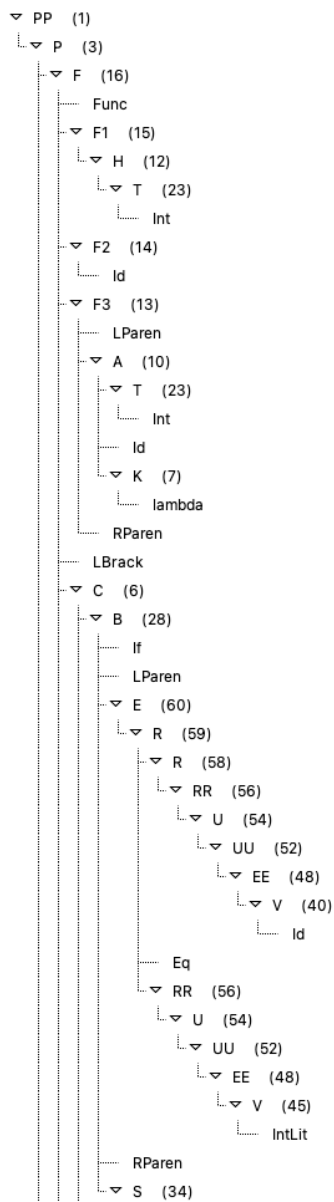
1	<Func, >	22	<IntLit, 1>	45	<Id, 2>	68	<LParen, >
2	<Int, >	23	<RParen, >	46	<Sum, >	69	<Id, 1>
3	<Id, 0>	24	<Ret, >	47	<Id, 3>	70	<Lt, >
4	<LParen, >	25	<IntLit, 1>	48	<Semi, >	71	<IntLit, 2>
5	<Int, >	26	<Semi, >	49	<Id, 2>	72	<RParen, >
6	<Id, 1>	27	<Let, >	50	<Assign, >	73	<RParen, >
7	<RParen, >	28	<Int, >	51	<Id, 3>	74	<Semi, >
8	<LBrack, >	29	<Id, 2>	52	<Semi, >	75	<Ret, >
9	<If, >	30	<Assign, >	53	<Id, 3>	76	<Id, 3>
10	<LParen, >	31	<IntLit, 0>	54	<Assign, >	77	<Semi, >
11	<Id, 1>	32	<Semi, >	55	<Id, 4>	78	<RBrack, >
12	<Eq, >	33	<Let, >	56	<Semi, >		
13	<IntLit, 0>	34	<Int, >	57	<Id, 1>		
14	<RParen, >	35	<Id, 3>	58	<Assign, >		
15	<Ret, >	36	<Assign, >	59	<Id, 1>		
16	<IntLit, 0>	37	<IntLit, 1>	60	<Sum, >		
17	<Semi, >	38	<Semi, >	61	<Sub, >		
18	<If, >	39	<Do, >	62	<IntLit, 1>		
19	<LParen, >	40	<LBrack, >	63	<Semi, >		
20	<Id, 1>	41	<Let, >	64	<RBrack, >		
21	<Eq, >	42	<Int, >	65	<While, >		
		43	<Id, 4>	66	<LParen, >		
		44	<Assign, >	67	<Not, >		

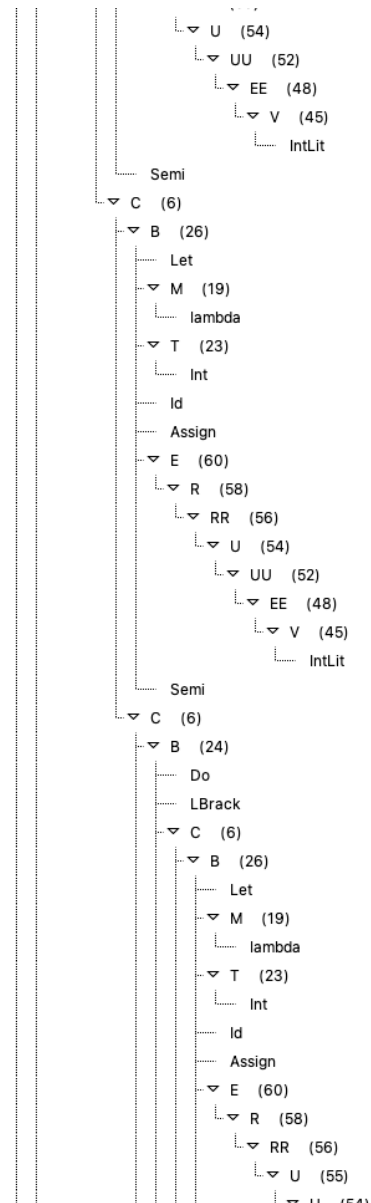
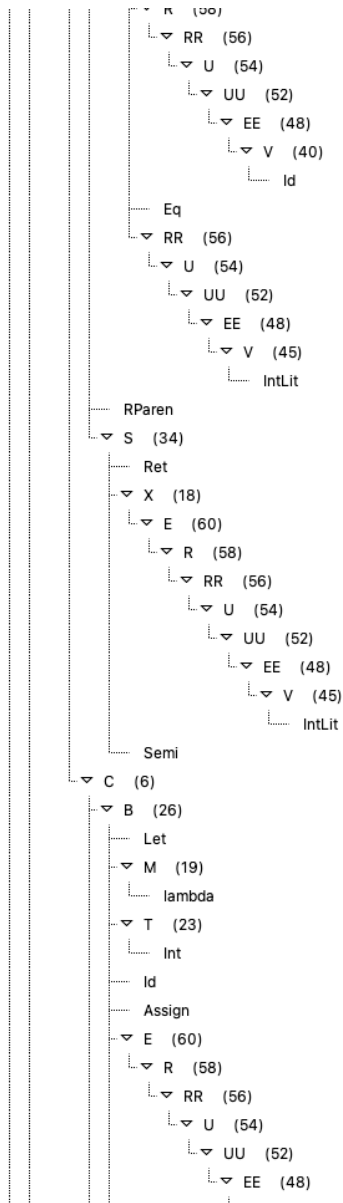
Tabla de símbolos:

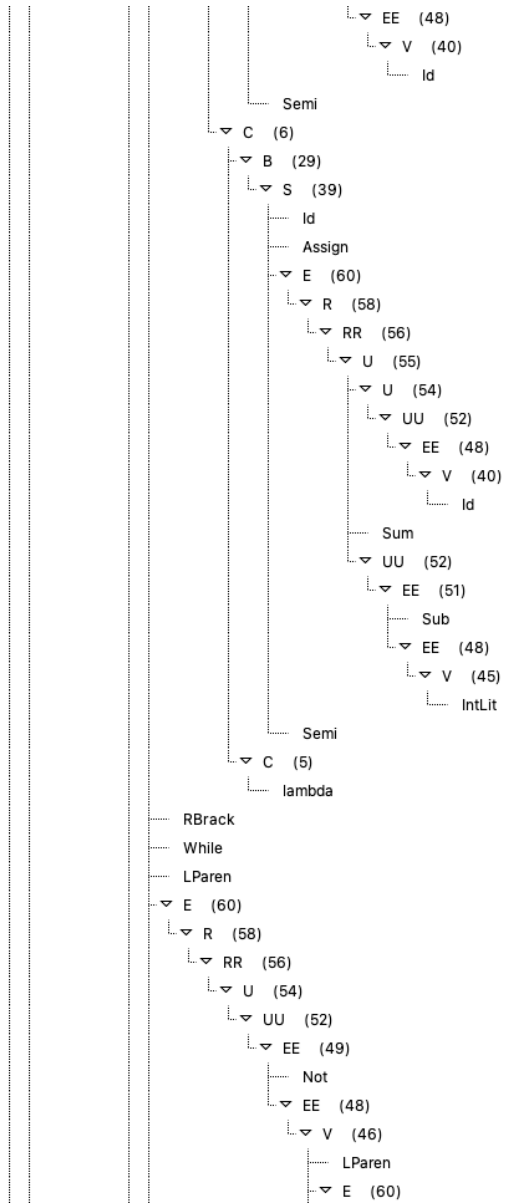
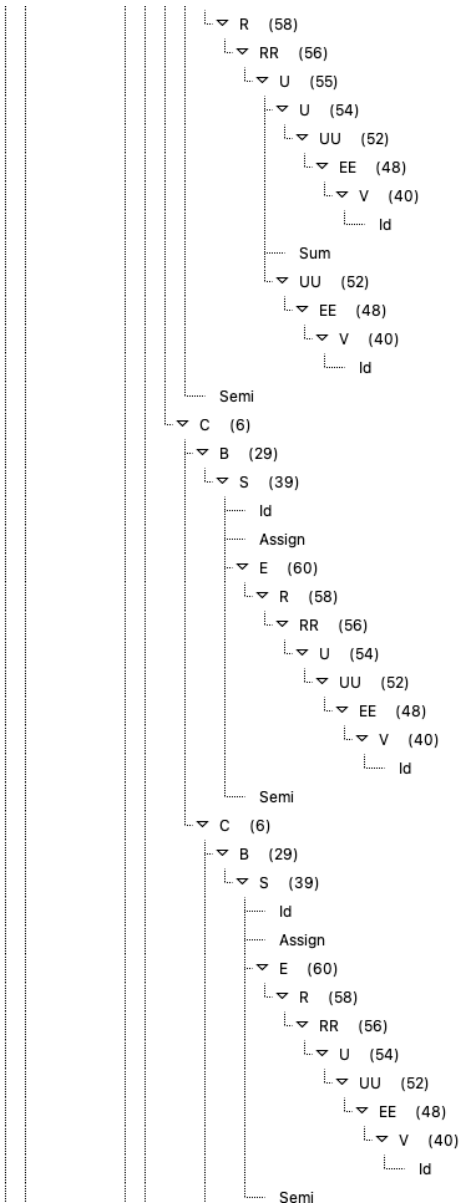
```
1  table #0:
2  * 'fib'
3  * 'n'
4  * 'a'
5  * 'b'
6  * 'c'
```

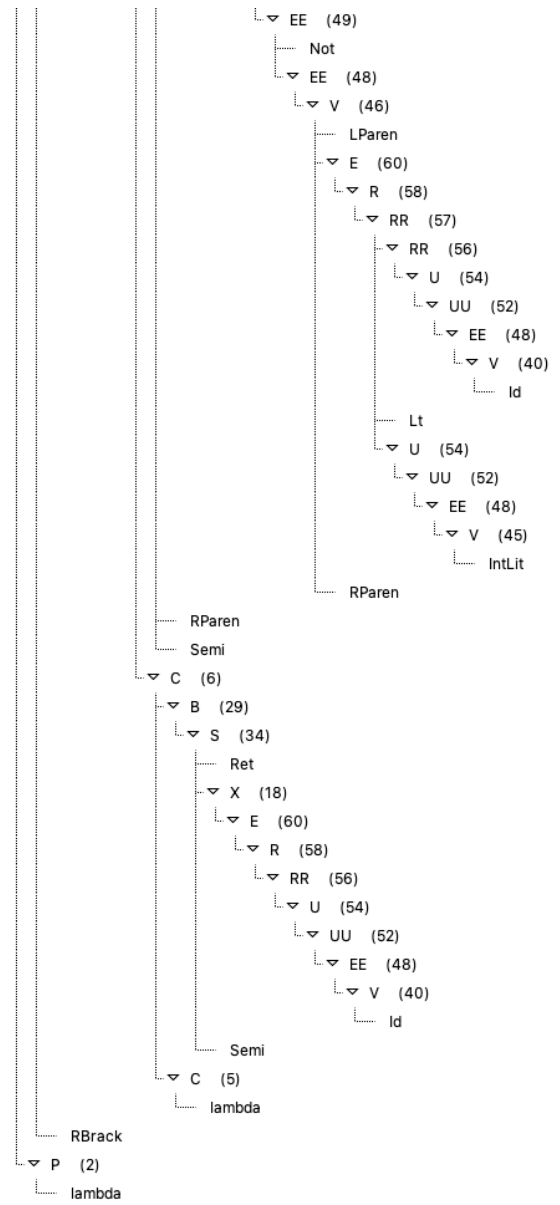
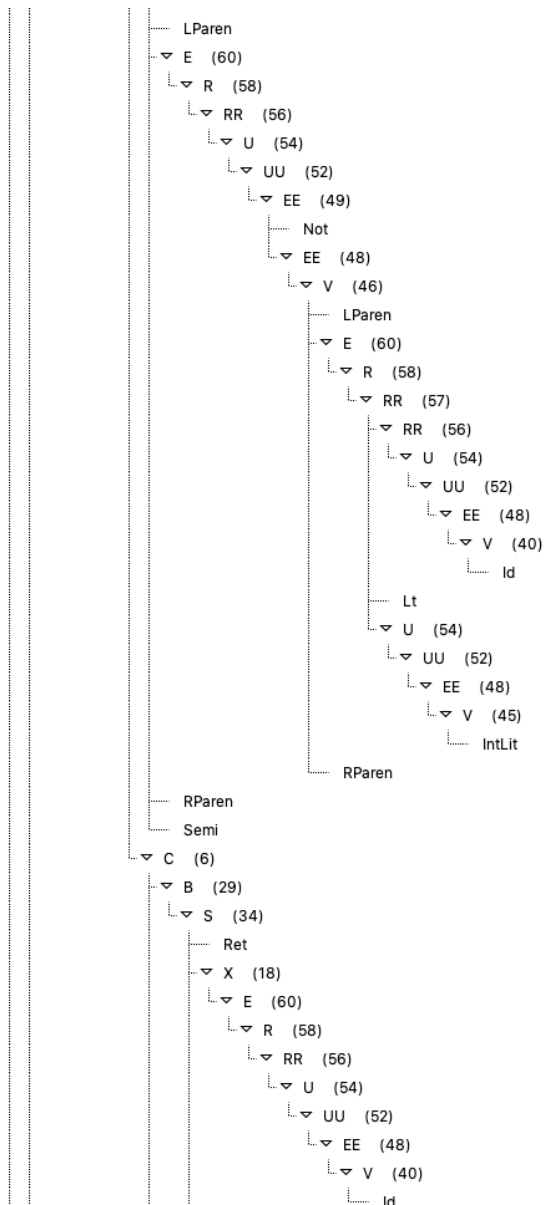
Parse:

```
1  ascending 23 12 15 14 23 7 10 13 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48 52 54 56 58 60 18 34
    28 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48 52 54 56 58 60 18 34 28 19 23 45 48 52 54 56
    58 60 26 19 23 45 48 52 54 56 58 60 26 19 23 40 48 52 54 40 48 52 55 56 58 60 26 40 48 52 54
    56 58 60 39 29 40 48 52 54 56 58 60 39 29 40 48 52 54 45 48 51 52 55 56 58 60 39 29 5 6 6 6 6
    40 48 52 54 56 45 48 52 54 57 58 60 46 48 49 52 54 56 58 60 24 40 48 52 54 56 58 60 18 34 29 5
    6 6 6 6 6 6 16 2 3 1
```









factorial.javascript 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(res);  
18  
19 /*** eof ***/
```

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 <Ret, >  
16 <IntLit, 1>  
17 <Semi, >  
18 <If, >  
19 <LParen, >  
20 <Id, 1>  
21 <Lt, >  
22 <IntLit, 2>  
23 <RParen, >  
24 <Ret, >  
25 <Id, 1>  
26 <Semi, >  
27 <Let, >  
28 <Int, >  
29 <Id, 2>  
30 <Assign, >
```

```
31 <Id, 1>  
32 <Semi, >  
33 <Do, >  
34 <LBrack, >  
35 <Id, 1>  
36 <Assign, >  
37 <Id, 1>  
38 <Sum, >  
39 <Sub, >  
40 <IntLit, 1>  
41 <Semi, >  
42 <Id, 2>  
43 <Assign, >  
44 <Id, 2>  
45 <Mul, >  
46 <Id, 1>  
47 <Semi, >  
48 <RBrack, >  
49 <While, >  
50 <LParen, >  
51 <Not, >  
52 <LParen, >  
53 <Id, 1>  
54 <Lt, >  
55 <IntLit, 2>  
56 <RParen, >  
57 <RParen, >  
58 <Semi, >  
59 <Ret, >  
60 <Id, 2>  
61 <Semi, >  
62 <RBrack, >
```

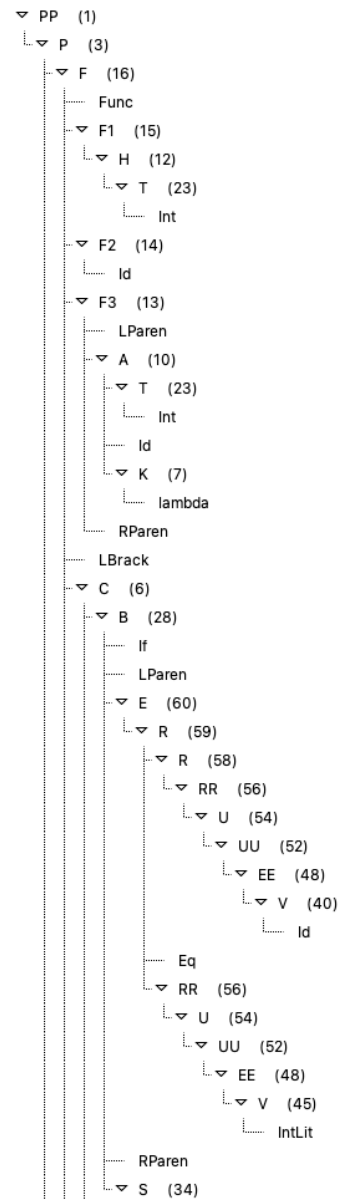
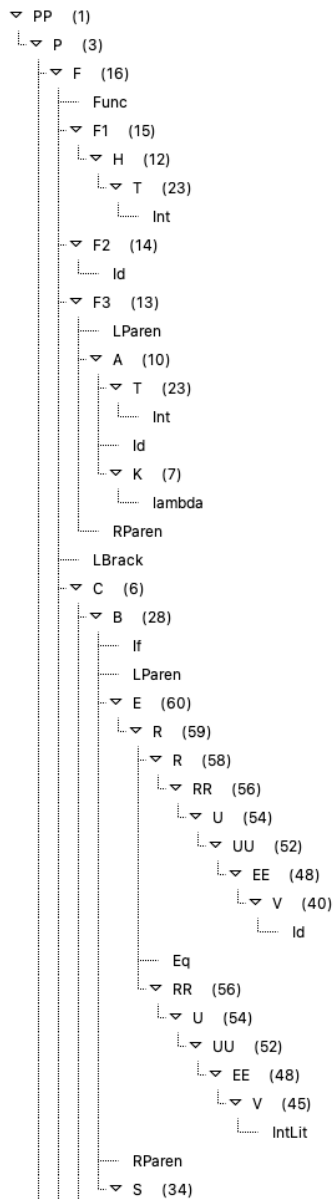
```
63 <Let, >  
64 <Int, >  
65 <Id, 1>  
66 <Assign, >  
67 <IntLit, 0>  
68 <Semi, >  
69 <Read, >  
70 <Id, 1>  
71 <Semi, >  
72 <Let, >  
73 <Int, >  
74 <Id, 2>  
75 <Assign, >  
76 <Id, 0>  
77 <LParen, >  
78 <Id, 1>  
79 <RParen, >  
80 <Semi, >  
81 <Write, >  
82 <LParen, >  
83 <Id, 2>  
84 <RParen, >  
85 <Semi, >
```

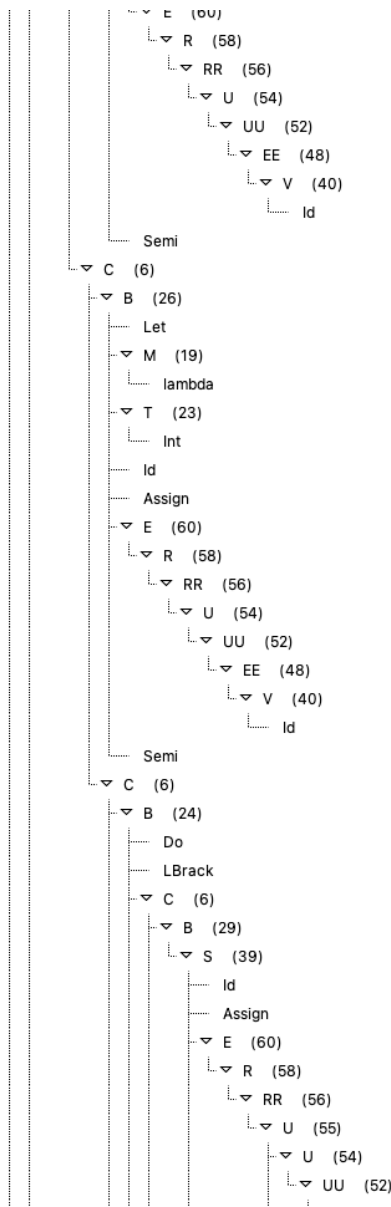
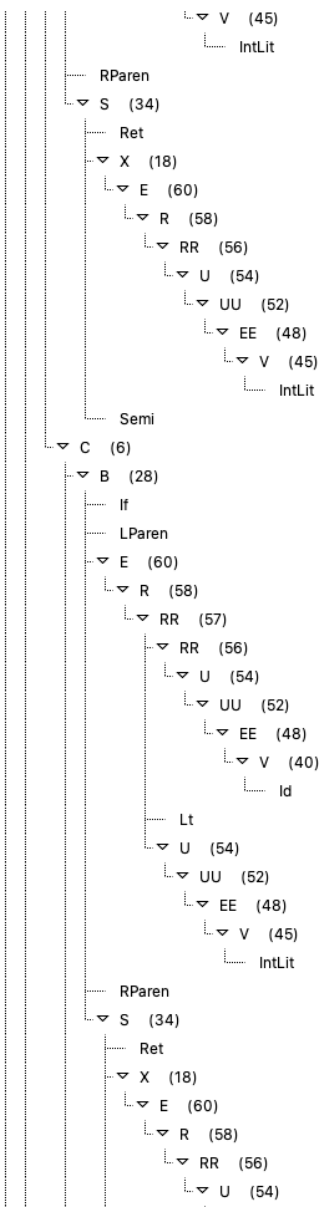
Tabla de símbolos:

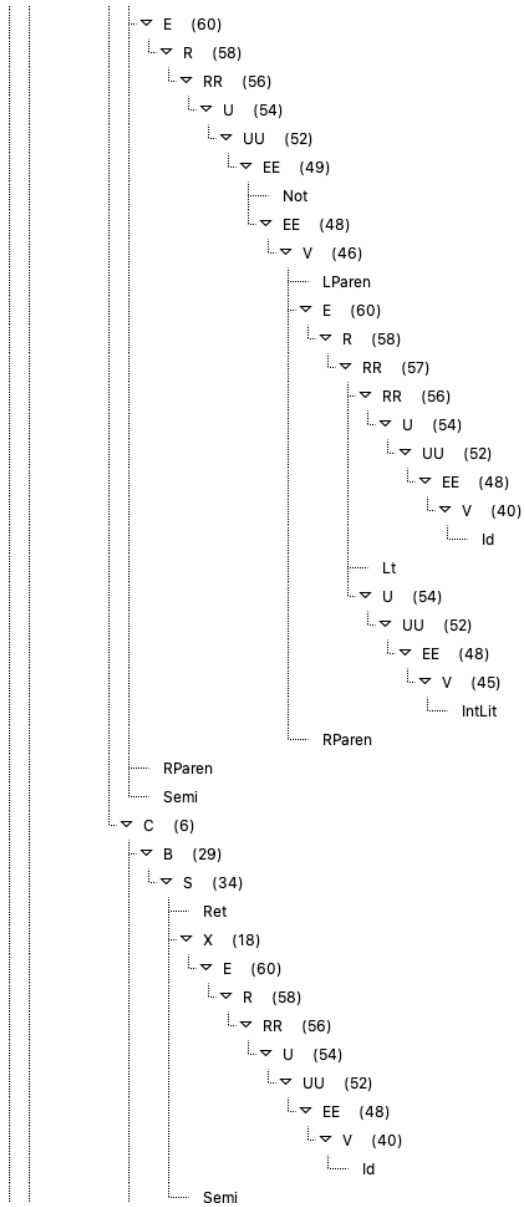
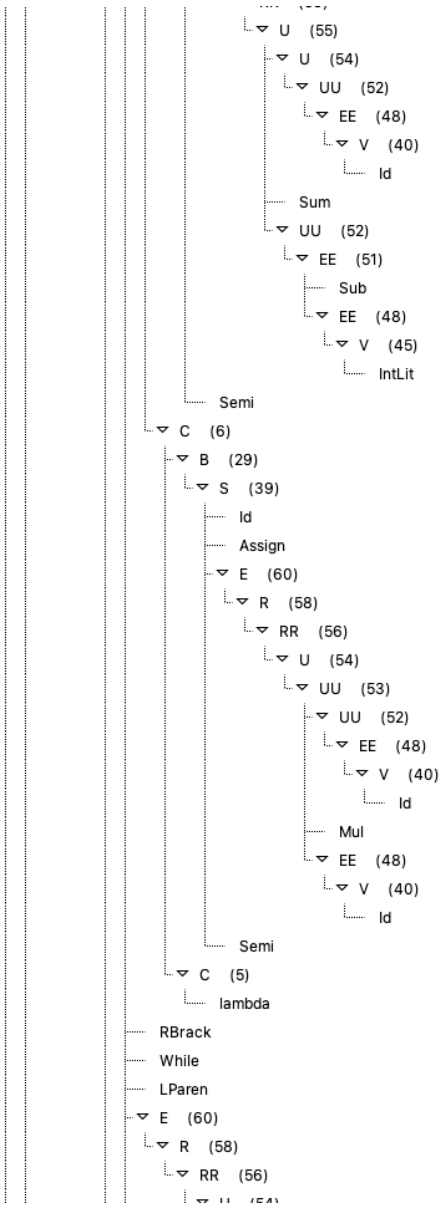
```
1 table #0:  
2 * 'factorial'  
3 * 'n'  
4 * 'res'
```

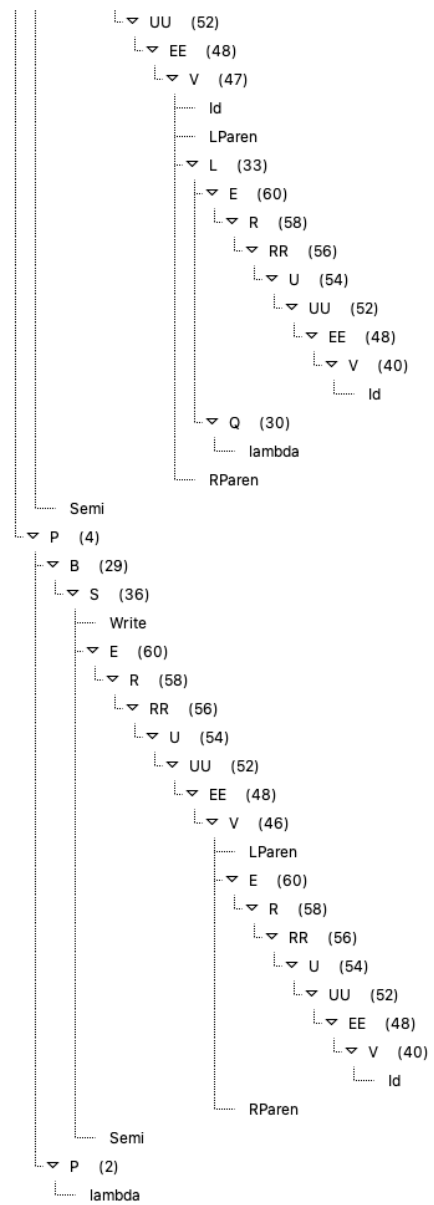
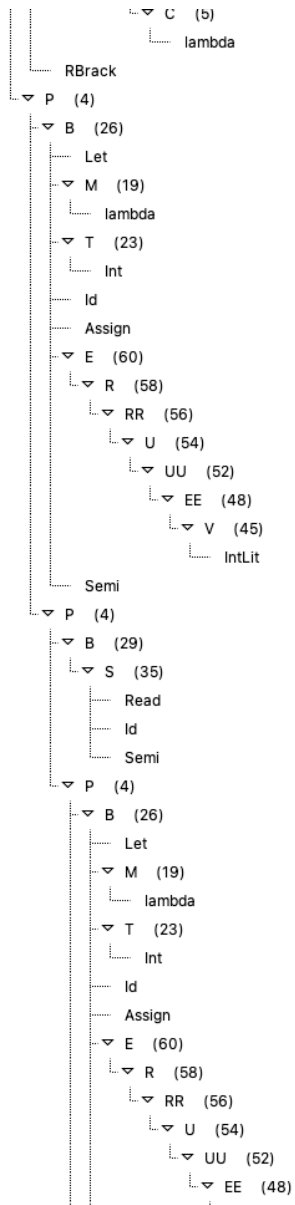
Parse:

```
1 ascending 23 12 15 14 23 7 10 13 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48 52 54 56 58 60 18 34  
23 40 48 52 54 56 45 48 52 54 57 58 60 40 48 52 54 56 58 60 18 34 28 19 23 40 48 52 54 56 58  
60 26 40 48 52 54 45 48 51 52 55 56 58 60 39 29 40 48 52 40 48 53 54 56 58 60 39 29 5 6 6 40  
48 52 54 56 45 48 52 54 57 58 60 46 48 49 52 54 56 58 60 24 40 48 52 54 56 58 60 18 34 29 5 6  
6 6 6 6 16 19 23 45 48 52 54 56 58 60 26 35 29 19 23 40 48 52 54 56 58 60 30 33 47 48 52 54 56  
58 60 26 40 48 52 54 56 58 60 46 48 52 54 56 58 60 36 29 2 4 4 4 4 3 1
```









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 int weird &= 3+-+--+--+--+--+5; let float oper=2.0000; 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 <Id, 0>
16 <Semi, >
17 <Do, >
18 <LBrack, >
19 <Id, 0>
20 <Assign, >
21 <Id, 0>
22 <Sum, >
23 <Sub, >
24 <IntLit, 1>
25 <Semi, >
26 <RBrack, >
27 <While, >
28 <LParen, >
29 <Not, >
30 <LParen, >
31 <Id, 0>
32 <Lt, >
33 <IntLit, 0>
34 <RParen, >
35 <RParen, >
36 <Semi, >
37 <RBrack, >
38 <Func, >
39 <Int, >
40 <Id, 2>
41 <LParen, >
42 <Float, >

```

```

43 <Id, 3>
44 <Comma, >
45 <Str, >
46 <Id, 4>
47 <RParen, >
48 <LBrack, >
49 <Let, >
50 <Str, >
51 <Id, 5>
52 <Assign, >
53 <StrLit, "this is a str">
54 <Semi, >
55 <Let, >
56 <Float, >
57 <Id, 6>
58 <Assign, >
59 <FloatLit, 1203.23>
60 <Semi, >
61 <Let, >
62 <Int, >
63 <Id, 7>
64 <Assign, >
65 <IntLit, 2398>
66 <Semi, >
67 <Let, >
68 <Int, >
69 <Id, 8>
70 <AndAssign, >
71 <IntLit, 3>
72 <Sum, >
73 <Sub, >
74 <Sum, >
75 <Sub, >
76 <Sum, >
77 <Sub, >
78 <Sum, >
79 <Sub, >
80 <Sum, >
81 <Sub, >
82 <Sum, >
83 <Sub, >
84 <Sum, >
85 <Sub, >
86 <Sum, >

```

```

87 <Sub, >
88 <Sum, >
89 <IntLit, 5>
90 <Semi, >
91 <Let, >
92 <Float, >
93 <Id, 9>
94 <Assign, >
95 <FloatLit, 2>
96 <Semi, >
97 <Let, >
98 <Bool, >
99 <Id, 10>
100 <Assign, >
101 <False, >
102 <Semi, >
103 <Let, >
104 <Bool, >
105 <Id, 11>
106 <Assign, >
107 <True, >
108 <Semi, >
109 <Id, 11>
110 <AndAssign, >
111 <Id, 10>
112 <Semi, >
113 <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'

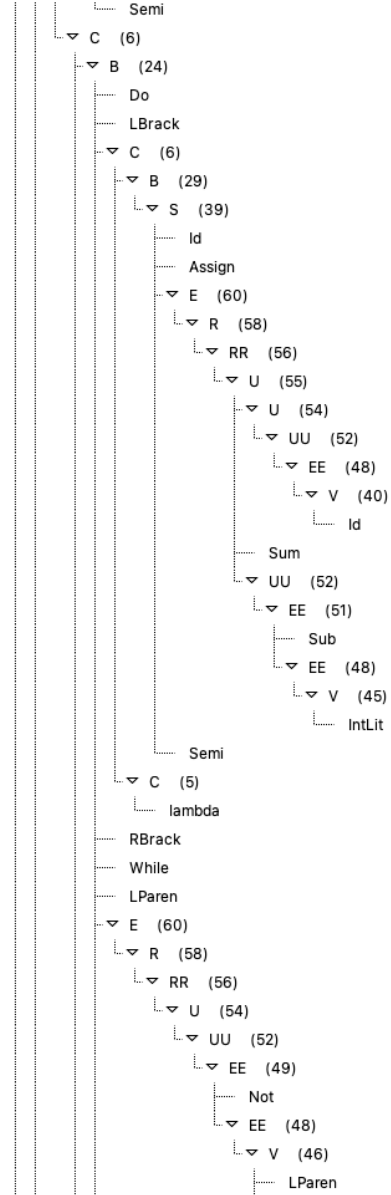
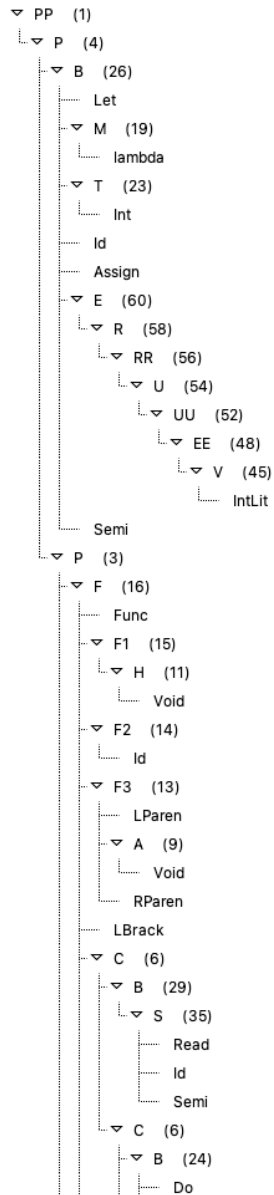
```

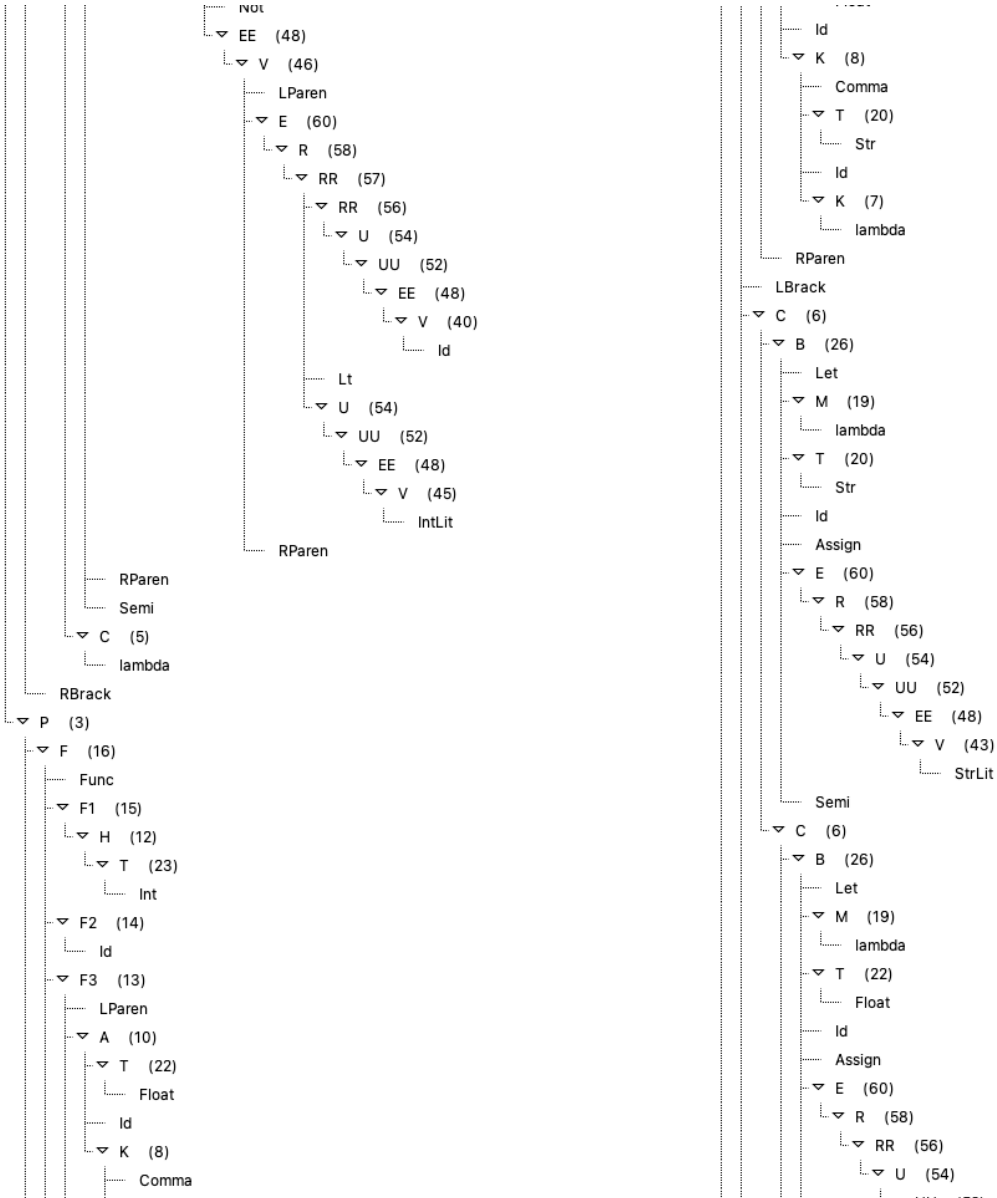
Parse:

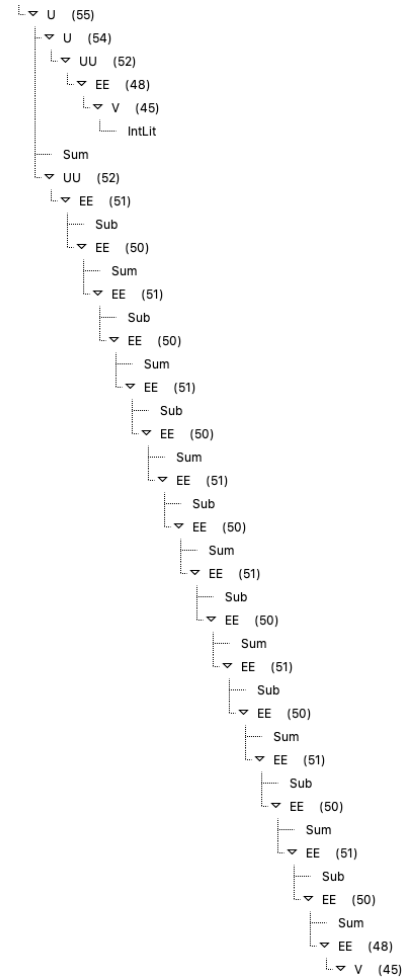
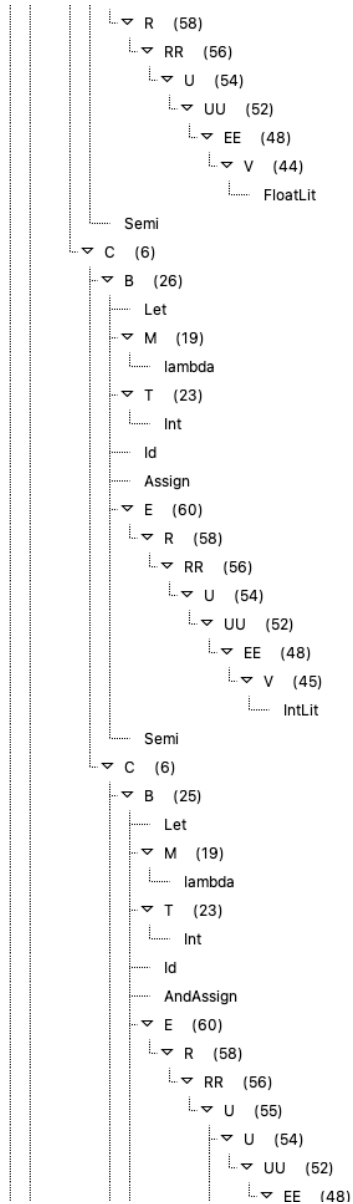
```

1 ascending 19 23 45 48 52 54 56 58 60 26 11 15 14 9 13 35 29 40 48 52 54 45 48 51 52 55 56 58 60 39 29 5
   6 40 48 52 54 56 45 48 52 54 57 58 60 46 48 49 52 54 56 58 60 24 5 6 6 16 23 12 15 14 22 20 7 8 10
   13 19 20 43 48 52 54 56 58 60 26 19 22 44 48 52 54 56 58 60 26 19 23 45 48 52 54 56 58 60 26 19 23
   45 48 52 54 45 48 50 51 50 51 50 51 50 51 50 51 50 51 50 51 52 55 56 58 60 25 19 22 44 48 52
   54 56 58 60 26 19 21 41 48 52 54 56 58 60 26 19 21 42 48 52 54 56 58 60 26 40 48 52 54 56 58 60 38
   29 5 6 6 6 6 6 6 6 16 2 3 3 4 1

```







A.2 Casos Incorrectos

unterm.javascript Fichero con errores de sentencias incompletas.

```

1  /* untermiated declaration */
2  let int = 2;
3
4  /* unfinished strings */
5  let string foo = "im not finishing this string
6  let string bar = "im not finishing this one either\
7  let string rep = "last one\q
8
9  /* unfinished float */
10 let float x = 478234.
11
12 /** * / * ///<** / this comment is finished /* /* ***/
13 /** this one is not ** ***** ***/ /*
14
15 function int main {
16     let string useless = "this variable won't be recognised";
17     return 0;
18 }
```

Diagnósticos:

```

1  unterm.javascript:5:18: error: missing terminating character ''' on string literal
2  |
3  5 | let string foo = "im not finishing this string
4  |                  ^ started here
5  |
6  unterm.javascript:6:18: error: missing terminating character ''' on string literal
7  |
8  6 | let string bar = "im not finishing this one either\
9  |                  ^ started here
10 |
11 unterm.javascript:7:27: warning: unknown escape sequence '\q'
12 |
13 7 | let string rep = "last one\q
14 |                  ^^ interpreted as \\q
15 |
16 unterm.javascript:7:18: error: missing terminating character ''' on string literal
17 |
18 7 | let string rep = "last one\q
19 |                  ^ started here
20 |
21 unterm.javascript:10:15: error: expected digit after '.' in float literal
22 |
23 10 | let float x = 478234.
24 |                ^^^^^^ perhaps you meant '478234.0'
25 |
26 unterm.javascript:13:1: error: unterminated block comment
27 |
28 13 | /** this one is not ** ***** ***/ /*
29 |    ^^ started here
30 |
31 unterm.javascript:2:9: error: expected 'identifier' before '='
32 |
33 2 | let int = 2;
34 |          ^ before this token
35 |
```

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

```

1  /* parse error, missing type */
2  function hello() {}
3
4  /* string with 64 characters */
5  let string foo = "ffffffffffffffffffffffffffffffffffffffffffffffffffffffff";
6
7  /* string with 65 characters */
8  let string bar = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
9
10 /* max int16 */
11 let int imax = 32767;
12 let int imax_plus_one = 32768;
13
14 /* max float */
15 let float fmax = 340282346638528859811704183484516925440.0;
16 let float fmax_plus_more = 3402823466385288598117041834845169254382923892341.0;
17 let float lots_of_decimals = 0.2347028349820934809218409238845290380928;

```

Diagnósticos:

```

1  overflow.javascript:8:18: error: string literal is too long
2  |
3  8 | let string bar = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
4  |                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ length is 65 but
5  |                   maximum is 64
6  overflow.javascript:12:25: error: integer literal out of range for 16-byte type
7  |
8  12 | let int imax_plus_one = 32768;
9  |                        ^^^^^ maximum is 32767
10 |
11 overflow.javascript:16:28: error: float literal out of range for 32-byte type
12 |
13  16 | let float fmax_plus_more = 3402823466385288598117041834845169254382923892341.0;
14 |                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ maximum is 3.4028235e38
15 |
16 overflow.javascript:2:10: error: expected 'int', 'float', 'string', 'boolean' or 'void' before 'hello'
17 |
18  2 | function hello() {}
19 |                   ^^^^^ before this token
20 |

```

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

```

1  if condition write "no parenthesis";
2  123.45.67 12abc 123456. 78
3  "bad\escape" "no closing quote
4  /* nested comment */ /* comment */ /* unclosed comment*/
5  != &== != <=> &&|!!
6  "unterminated string again /* unclosed comment
7  "bad string with \x illegal escape" "" ""empty""
8  "Hello^[World"
9  /* final comment */ /* another unclosed

```

Diagnósticos:

```

1  noise.javascript:2:7: error: illegal character '.' in program
2  |
3  2 | 123.45.67 12abc 123456. 78
4  |      ^ here
5  |
6  noise.javascript:2:17: error: expected digit after '.' in float literal
7  |
8  2 | 123.45.67 12abc 123456. 78
9  |                      ^^^^^^ perhaps you meant '123456.0'
10 |
11 noise.javascript:3:5: warning: unknown escape sequence '\e'
12 |
13 3 | "bad\escape" "no closing quote
14 |      ^^ interpreted as \\e
15 |
16 noise.javascript:3:14: error: missing terminating character '"' on string literal
17 |
18 3 | "bad\escape" "no closing quote
19 |      ^ started here
20 |
21 noise.javascript:5:14: error: illegal character '>' in program
22 |
23 5 | != &== != <=> &&|!!
24 |      ^ here
25 |
26 noise.javascript:5:18: error: illegal character '|' in program
27 |
28 5 | != &== != <=> &&|!!
29 |      ^ here
30 |
31 noise.javascript:6:1: error: missing terminating character '"' on string literal
32 |
33 6 | "unterminated string again /* unclosed comment
34 | ^ started here
35 |
36 noise.javascript:7:18: warning: unknown escape sequence '\x'
37 |
38 7 | "bad string with \x illegal escape" "" ""empty""
39 |      ^^ interpreted as \\x
40 |
41 noise.javascript:8:7: error: malformed string literal, contains control character '\u{1b}'
42 |
43 8 | "Hello\u{1b}World"
44 |      ^^^^^^ remove this character
45 |
46 noise.javascript:9:21: error: unterminated block comment
47 |
48 9 | /* final comment */ /* another unclosed
49 |      ^^ started here
50 |
51 noise.javascript:1:4: error: expected '(' before 'condition'
52 |
53 1 | if condition write "no parenthesis";
54 |      ^^^^^^^^^ before this token
55 |

```