

21 de enero de 2026

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. Análisis Léxico.....	2
5. La Tabla de Símbolos	7
6. Análisis Sintáctico	7
7. Análisis Semántico.....	14
8. Gestión de Errores	22
A. Casos de Prueba	27

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/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 Análisis Léxico

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 *Lexer* puede generar nueve excepciones distintas, siendo todas recuperables. De entre ellas, *Carácter inválido* sirve de *fallback*.

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

Sólo se emite un *warning*, *Secuencia de Escape inválida*. Como se muestra en Acciones Semánticas, al detectar una secuencia incorrecta 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 $G = (T, N, S, P)$, donde:

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

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

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:

$$\begin{aligned} A &\rightarrow = \mid \lambda \\ B &\rightarrow = \mid \lambda \\ C &\rightarrow = \mid \lambda \\ D &\rightarrow = \mid \lambda \\ E &\rightarrow = \mid \lambda \\ F &\rightarrow | \\ G &\rightarrow dG \mid .K \mid \lambda \\ K &\rightarrow dL \\ L &\rightarrow dL \mid \lambda \\ H &\rightarrow c_2H \mid \backslash M \mid " \\ M &\rightarrow nH \mid tH \\ I &\rightarrow c_3I \mid \lambda \\ J &\rightarrow *N \mid \lambda \\ N &\rightarrow c_4N \mid *O \\ O &\rightarrow c_5N \mid *O \mid /S \end{aligned}$$

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

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*.

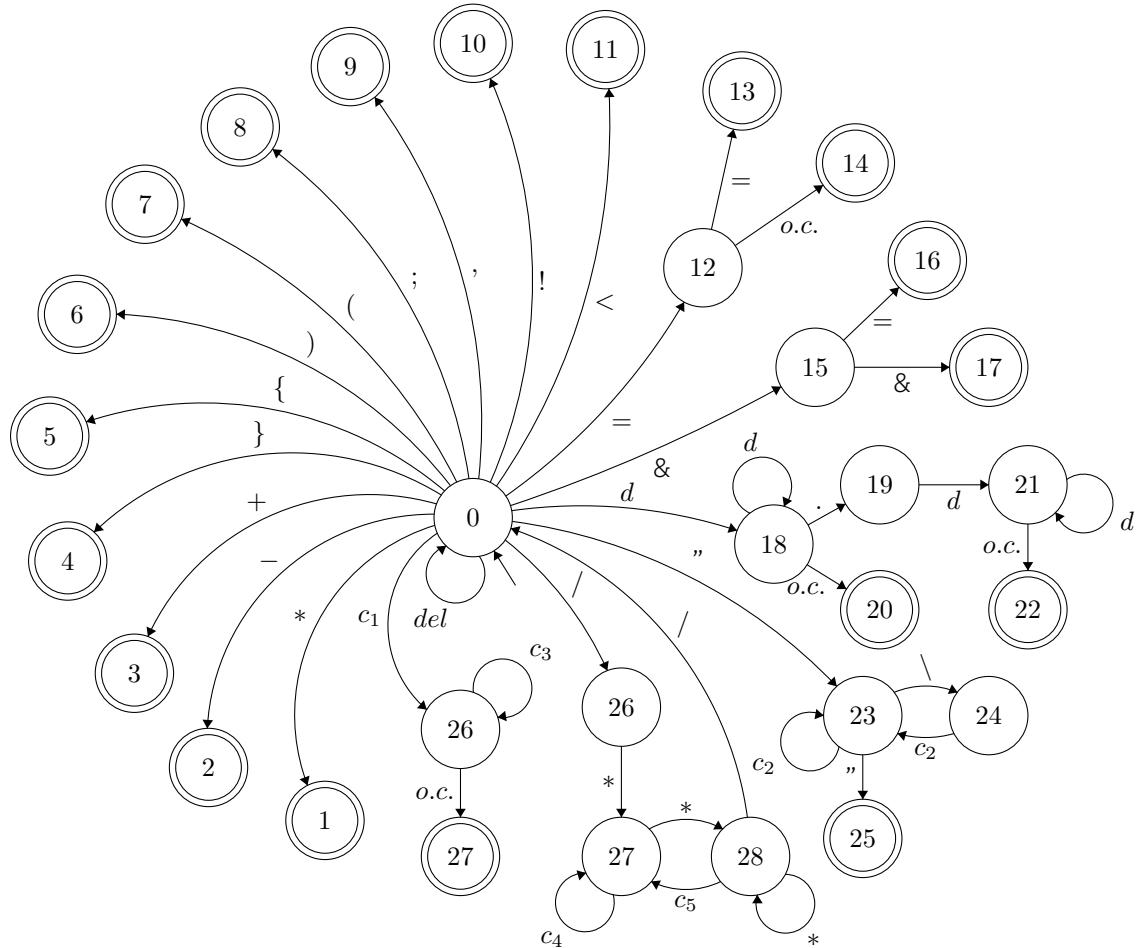
² La definición de delimitador se toma de la documentación oficial de *ECMAScript*.

4.4 Autómata

A continuación se muestra el autómata finito determinista o *FDA* 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

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

4.5.2 Errores

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

```
1 reporter.emit(MalformedStrLit)
```

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

```
1 reporter.emit(UntermStrLit)
```

INV_FLOAT_FMT Si en el estado 19 no se puede transitar

```
1 reporter.emit(InvFloatFmt)
```

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

```
1 reporter.emit(UntermComment)
```

INV_CHAR Ante cualquier error no manejado en el resto de acciones

```
1 reporter.emit(StrayChar)
```

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     reporter.emit(FloatOverflow)
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
```

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         reporter.warn(InvEscSeq)
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     reporter.emit(StrOverflow)
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 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 la 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 Análisis Sintáctico

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 $G = (T, N, P, R)$, donde:

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

$$N = \{E, R, RR, U, UU, EE, V, S, L, Q, X, B, T, M, F, FF, F1, F2, F3, H, A, K, C, P\}$$

R se compone de:

$$P \rightarrow BP \mid FP \mid \lambda$$

$$C \rightarrow BC \mid \lambda$$

$$F \rightarrow \text{Func } FF \text{ LBrack } C \text{ RBrack}$$

$$FF \rightarrow F1 F2 F3$$

$$F1 \rightarrow H$$

$$F2 \rightarrow \text{Id}$$

$$F3 \rightarrow \text{LParen } A \text{ RParen}$$

$$H \rightarrow T \mid \text{Void}$$

$$A \rightarrow T \text{ Id } K \mid \text{Void}$$

$$K \rightarrow \text{Comma } T \text{ Id } K \mid \lambda$$

$$B \rightarrow \text{If LParen } E \text{ RParen } S \mid \text{Do LBrack } C \text{ RBrack While LParen } E \text{ RParen Semi}$$

$$B \rightarrow S \mid \text{Let } MT \text{ Id Semi} \mid \text{Let } MT \text{ Id Assign } E \text{ Semi}$$

$$M \rightarrow \lambda$$

$$T \rightarrow \text{Int} \mid \text{Float} \mid \text{Bool} \mid \text{Str}$$

$$S \rightarrow \text{Write } E \text{ Semi} \mid \text{Read } \text{Id } \text{Semi} \mid \text{Ret } X \text{ Semi}$$

$$S \rightarrow \text{Id Assign } E \text{ Semi} \mid \text{Id AndAssign } E \text{ Semi} \mid \text{Id LParen } L \text{ RParen Semi}$$

$$L \rightarrow EQ \mid \lambda$$

$$Q \rightarrow \text{Comma } EQ \mid \lambda$$

$$X \rightarrow E \mid \lambda$$

$$E \rightarrow E \text{ And } R \mid R$$

$$R \rightarrow R \text{ Eq } RR \mid RR$$

$$RR \rightarrow RR \text{ Lt } U \mid U$$

$$U \rightarrow U \text{ Sum } UU \mid UU$$

$$UU \rightarrow UU \text{ Mul } EE \mid EE$$

$$EE \rightarrow \text{Not } EE \mid \text{Sub } EE \mid \text{Sum } EE \mid V$$

$$V \rightarrow \text{Id LParen } L \text{ RParen} \mid \text{LParen } E \text{ RParen} \mid \text{IntLit} \mid \text{FloatLit} \mid \text{StrLit} \mid \text{True} \mid \text{False} \mid \text{Id}$$

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																														s24							
6				r18	r18	r18	r18																														
7																														s26							
8	r27	r27				r27	r27	r27	r27	r27									r27											r27							
9												s31	s30	s33	s34	s32	s29				r16	s35					s43	s44		s37							
10																					s45																
11												s31	s30	s33	s34	s32	s29						s35				s43	s44		s37							
12																				s49	s48																
13																															r2						
14																															r3						
15																				r10																	
16																				r11																	
17																				r14																	
18																														s50							
19																				r19																	
20																				r20																	
21																				r21																	
22																				r22																	
23																				s51																	
24	s7	s5							s6		s9	s10	s11						s12											r4							
25				s22	s21	s19	s20																														
26												s31	s30	s33	s34	s32	s29						s35				s43	s44		s37							
27																				r17											s57						
28																				s58																	
29																				r38	r38	s59	r38				r38	r38	r38	r38	r38						
30																				r39	r39	r39	r39				r39	r39	r39	r39	r39						
31																				r40	r40	r40	r40				r40	r40	r40	r40	r40						
32																				r41	r41	r41	r41				r41	r41	r41	r41	r41						
33																				r42	r42	r42	r42				r42	r42	r42	r42	r42						
34																				r43	r43	r43	r43				r43	r43	r43	r43	r43						
35												s31	s30	s33	s34	s32	s29						s35				s43	s44		s37							
36																				r46	r46	r46	r46				r46	r46	r46	r46	r46						
37												s31	s30	s33	s34	s32	s29						s35				s43	s44		s37							
38																				r48	r48	r48	r48				r48	r48	r48	r48	r48						
39																				r50	r50	r50	r50				r50	s62	r50	r50	r50						
40																				r52	r52	r52	r52				s63	r52	r52	r52	r52						
41																				r54	r54	r54	r54						r54	s64	r54						
42																				r56	r56	r56	r56						r56	s65							
43																				s31	s30	s33	s34	s32	s29												
44																				s31	s30	s33	s34	s32	s29												
45																																					
46																																					
47																																					
48																																					
49																																					
50	s7	s5							s6		s9	s10	s11																				r4				
51																																		r13			
52																																		s75			
53	s7	s5							s6		s9	s10	s11																				r4				
54																																			s78		
55																																					

56																		s80					s57			
57									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
58	r32	r32			r32	r32	r32	r32						r32						r32					r32	
59									s31	s30	s33	s34	s32	s29				s35	r30		s43	s44		s37		
60																		s83				s57				
61															r47	r47	r47		r47	r47	r47	r47	r47	r47		
62									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
63									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
64									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
65									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
66															r59	r59	r59		r59	r59	r59	r59	r59	r59		
67															r60	r60	r60		r60	r60	r60	r60	r60	r60		
68	r33	r33			r33	r33	r33	r33						r33					r33					r33		
69	r34	r34			r34	r34	r34	r34						r34					r34					r34		
70																s88		r28			s57					
71																	s90									
72																	s91				s57					
73																	s92				s57					
74																		s93								
75		s22	s21	s19	s20	s94																				
76																		r58								
77																		r5								
78		s97																								
79																	s98		s99							
80									s9	s10	s11						s12									
81																	r57	r57	r57			r57		s65		
82																		s101								
83																	r44	r44	r44		r44	r44	r44	r44	r44	
84																	r49	r49	r49		r49	r49	r49	r49	r49	
85																	r51	r51	r51		r51	s62	r51	r51	r51	
86																	r53	r53	r53		s63		r53	r53	r53	
87																	r55	r55	r55			r55	s64	r55		
88									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
89																		r31								
90																		s103								
91	r36	r36			r36	r36	r36	r36						r36					r36					r36		
92	r37	r37			r37	r37	r37	r37						r37					r37					r37		
93	r15	r15			r15	r15	r15	r15						r15										r15		
94																		r8								
95																	s104									
96																		s105								
97																		s106								
98									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
99	r25	r25			r25	r25	r25	r25						r25					r25					r25		
100	r26	r26			r26	r26	r26	r26						r26					r26					r26		
101																	r45	r45	r45		r45	r45	r45	r45	r45	
102																	s88		r28			s57				
103	r35	r35			r35	r35	r35	r35						r35						r35					r35	
104																	s109		r6							
105																		r12								
106									s31	s30	s33	s34	s32	s29				s35			s43	s44		s37		
107																		s112						s57		
108																			r29							
109		s22	s21	s19	s20																					
110																			r9							
111																			s114			s57				
112	r24	r24			r24	r24	r24	r24						r24						r24					r24	
113																	s115									
114																		s116								
115																	s109		r6							
116	r23	r23			r23	r23	r23	r23						r23						r23					r23	
117																			r7							

Cuadro 4: Tabla de Goto

	E	R	RR	U	UU	EE	V	S	L	Q	X	B	T	M	F	FF	F1	F2	F3	H	A	K	C	P
0								8				3			2									1
1																								
2										8			3			2								13
3											8			3			2							14
4														16			18	23			17			
5																								
6															25									
7																								
8																								
9	27	42	41	40	39	38	36						28											
10																								
11	46	42	41	40	39	38	36																	
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								
21																								
22																								
23																			52					
24													8			53								54
25																55								
26	56	42	41	40	39	38	36																	
27																								
28																								
29																								
30																								
31																								
32																								
33																								
34																								
35	60	42	41	40	39	38	36																	
36																								
37								61	36															
38																								
39																								
40																								
41																								
42																								
43										66	36													
44										67	36													
45																								
46																								
47	70	42	41	40	39	38	36					71												
48	72	42	41	40	39	38	36																	
49	73	42	41	40	39	38	36																	
50												8			53									74
51																								
52																				76				
53												8			53									77
54																								
55																								
56																								
57		81	41	40	39	38	36																	
58																								

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 recucción-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

El Parser emite 10 excepciones diferentes, donde *Token inesperado* sirve de *fallback*.

Cabe destacar que *Token inesperado*, a pesar de ser la excepción genérica del Parser, es dinámica, es decir, su mensaje varía según la celda en la que se lance, tomando el formato:

"expected <opciones> before <token inesperado>"

La lista de *tokens* esperados se obtiene realizando una búsqueda en anchura 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 o a un error.

El Parser también es capaz de recuperarse de cualquier error, aunque este proceso es más complejo que en el *Lexer*. En la sección del Gestor de Errores se explicará en mayor detalle pero, a alto nivel, al llegar a una celda de error se intentará resincronizar la pila de estados a través de inserciones, sustituciones o eliminaciones.

No obstante, aunque en la práctica nunca lo hará, la resincronización puede llegar a fallar, en cuyo caso se termina el análisis. De hecho, el Parser es el único módulo que puede interrumpir la ejecución del programa prematuramente, es decir, antes de procesar el fichero fuente por completo.

Cuadro 5: Listado de errores del Parser

Error	Severidad
Token inesperado	error
Delimitador desparejado	error
Delimitador sin cerrar	error
Palabra reservadada usada como identificador	error
Punto y coma ausente	error
Coma sobrante en la lista de parámetros de una función	error
Coma sobrante en la lista de argumentos en una llamada	error
Tipo ausente en declaración	error
Tipo de retorno ausente	error
Lista de parámetros ausente	error
Lista de parámetros vacía	error

7 Análisis Semántico

El último módulo del procesador es el Analizador Semántico, que se encarga de asegurar que el programa sea coherente más allá de las reglas sintácticas. Hace comprobaciones de tipo, se asegura de que no hay redeclaraciones, comprueba que se llama a las funciones con el número correcto de parámetros, etc.

En realidad, en este procesador, el Analizador Semántico es un submódulo del *Parser*, ya que sigue la estrategia de *Traducción dirigida por la sintaxis* o *TDS*. Esta consiste en asociar una serie de acciones semánticas a cada regla de la gramática y ciertos atributos a algunos de sus símbolos, construyendo lo que se llama una *Gramática de Atributos*.

El producto final del Analizador Semántico será la Tabla de Símbolos, que irá llenando con las variables y funciones que encuentre, así como sus respectivos atributos, a medida que se analice el fichero fuente.

7.1 Errores

El *Analizador Semántico* puede producir 6 excepciones distintas.

La excepción *Tipo inesperado* sirve la función de error de tipo genérico, que se emitirá siempre que se espere un tipo concreto y se encuentre otro. De nuevo, su mensaje es dinámico y cambia según el tipo esperado y el encontrado.

Como en los módulos anteriores, todas los errores son recuperables. Esta recuperación se hace a través de los tipos especiales *type_error* y *type_ok*, que se propagan a través del *AST*.

Cuadro 6: Listado de errores del Analizador Semántico

Error	Severidad
Tipo de retorno incorrecto	error
Identificador redeclarado	error
Tipo inesperado	error
Función sin declarar	error
Retorno fuera de función	error
Llamada incorrecta	error

7.2 Acciones Semánticas

Como el *Parser* de este procesador es de tipo *LR(1)*, es decir, construye el *AST* desde las hojas hasta la raíz, las acciones semánticas se ejecutarán cada vez que el *Parser* realice una reducción. De este modo cada regla de la gramática tendrá una única regla asociada a ella.

A continuación se muestran las acciones semánticas del Analizador Semántico. Cabe destacar que esto es una simplificación, ya que muchas acciones, en particular aquellas relacionadas con la gestión de errores, son mucho más complejas en realidad.

En cuanto a la notación se ha optado por un *Esquema de Traducción* (*EdT*). Por motivos de legibilidad, las acciones no se escriben intercaladas en la producción, sino como un bloque de código inmediatamente posterior.

Esto supondría un problema de no ser porque en este trabajo todas las acciones semánticas se ejecutan al reducir, y ninguna producción contiene acciones intercaladas. Por tanto toda producción se ajusta al siguiente esquema:

$$X \rightarrow \alpha \{acción\}$$

Y, en consecuencia, cada bloque de código que aparece tras una producción debe interpretarse de este modo, es decir, como una acción sintetizada al final de la producción.

La primera acción que se ejecuta es la creación de la tabla de símbolos. Esta es la única acción que no se puede asociar a ninguna regla, ya que por ser el *Parser* ascendente, no se sabe cuál será la primera regla en reducirse:

```

1 symtable = symtable_make()
2
3 scope_global = scope_make()
4 scope_global.despl = 0
5
6 symtable.scopes.push(scope_global)

```

El resto de acciones serán:

1. $PP \rightarrow P$

```
1 symtable_free(symtable)
```

2. $P \rightarrow BP$

```
1 if B.ret_type != null
2     then error()
```

3. $P \rightarrow FP$

4. $P \rightarrow \lambda$

5. $C \rightarrow BC_1$

```
1 C.ret_type = if B.ret_type == C1.ret_type
2     then B.ret_type
3 else if B.ret_type == null
4     then C1.ret_type
5 else if C1.ret_type == null
6     then B.ret_type
7 else type_error
```

6. $C \rightarrow \lambda$

```
1 C.ret_type = null
```

7. $F \rightarrow \text{Func } FF \text{ LBrack } C \text{ RBrack}$

```
1 scope_local = symtable.scopes.pop()
2 scope_free(scope_local)
3
4 F.type = if C.ret_type == null
5     || (C.ret_type != type_error && C.ret_type == FF.ret_type)
6     then type_ok
7 else type_error
```

8. $FF \rightarrow F1 F2 F3$

```
1 scope_local = scope_make()
2 scope_local.despl = 0
3
4 scope_local.add_type(F2.pos, F3.type -> F1.type)
5 scope_local.add_label(F2.pos, label_make())
6
7 if F3.params != null {
8     for (type, pos) in (F3.type, F3.params) {
9         scope_local.add_type(pos, type)
10        scope_local.add_despl(pos, scope_local.despl)
11
12        scope_local.despl += type.size
13    }
14}
15
16 symtable.scopes.push(scope_local)
```

9. $F1 \rightarrow H$

```
1 F1.type = H.type
```

10. $F2 \rightarrow Id$

```
1 F2.pos = Id.pos
```

11. $F3 \rightarrow LParen A RParen$

```
1 F3.type = A.type  
2 F3.params = A.params
```

12. $H \rightarrow T$

```
1 H.type = T.type
```

13. $H \rightarrow Void$

```
1 H.type = type_void
```

14. $A \rightarrow T Id K$

```
1 A.type = T.type x K.type  
2 A.params = Id.pos x K.params
```

15. $A \rightarrow Void$

```
1 A.type = type_void  
2 A.params = null
```

16. $K \rightarrow Comma T Id K_1$

```
1 K.type = T.type x K1.type  
2 K.params = Id.pos x K1.params
```

17. $K \rightarrow \lambda$

18. $B \rightarrow If LParen E RParen S$

```
1 B.ret_type = S.ret_type  
2  
3 B.type = if E.type == type_bool  
4     then type_ok  
5 else type_error
```

19. $B \rightarrow Do LBrack C RBrack While LParen E RParen Semi$

```
1 B.ret_type = C.ret_type  
2  
3 B.type = if E.type == type_bool  
4     then type_ok  
5 else type_error
```

20. $B \rightarrow S$

```
1 B.type = S.type  
2 B.ret_type = S.ret_type
```

21. $B \rightarrow \text{Let } MT \text{ Id } \text{ Semi}$

```
1 scope = symtable.scopes.peek()
2
3 scope.add_type(Id.pos, T.type)
4 scope.add_despl(Id.pos, T.type.size)
5
6 scope.despl += T.type.size
```

22. $B \rightarrow \text{Let } MT \text{ Id } \text{ Assign } E \text{ Semi}$

```
1 scope = symtable.scopes.peek()
2
3 scope.add_type(Id.pos, T.type)
4 scope.add_despl(Id.pos, T.type.size)
5
6 scope.despl += T.type.size
7
8 B.type = if E.type == T.type
9     then type_ok
10 else type_error
```

23. $M \rightarrow \lambda$

24. $T \rightarrow \text{Int}$

```
1 T.type = type_int
2 T.type.size = 1
```

25. $T \rightarrow \text{Float}$

```
1 T.type = type_float
2 T.type.size = 2
```

26. $T \rightarrow \text{Bool}$

```
1 T.type = type_bool
2 T.type.size = 1
```

27. $T \rightarrow \text{Str}$

```
1 T.type = type_str
2 T.type.size = 64
```

28. $S \rightarrow \text{Write } E \text{ Semi}$

```
1 S.type = if E.type in {type_int, type_float, type_str}
2     then type_ok
3 else type_error
```

29. $S \rightarrow \text{Read Id Semi}$

```

1 scope = symtable.scopes.peek()
2 type = scope.search_type(Id.pos)
3
4 if type == null {
5     scope.add_type(Id.pos, type_int)
6     scope.add_despl(Id.pos, scope.despl)
7
8     scope.despl += 1
9     type = type_int
10}
11
12 S.type = if type in {type_int, type_float, type_str}
13     then type_ok
14 else type_error

```

30. $S \rightarrow \text{Ret X Semi}$

```

1 S.ret_type = X.type

```

31. $S \rightarrow \text{Id Assign E Semi}$

```

1 scope = symtable.scopes.peek()
2 type = scope.search_type(Id.pos)
3
4 if type == null {
5     scope.add_type(Id.pos, type_int)
6     scope.add_despl(Id.pos, scope.despl)
7
8     scope.despl += 1
9     type = type_int
10}
11
12 S.type = if type == E.type
13     then type_ok
14 else type_error

```

32. $S \rightarrow \text{Id AndAssign E Semi}$

```

1 scope = symtable.scopes.peek()
2 type = scope.search_type(Id.pos)
3
4 if type == null {
5     scope.add_type(Id.pos, type_int)
6     scope.add_despl(Id.pos, scope.despl)
7
8     scope.despl += 1
9     type = type_int
10}
11
12 S.type = if type == E.type && type == type_bool
13     then type_ok
14 else type_error

```

33. $S \rightarrow \text{Id LParen } L \text{ RParen Semi}$

```
1 type = symtable.scopes.peek().search_type(Id.pos)
2
3 S.type = if type == null
4     then type_error
5 else if type == L.type -> ret_type
6     then type_ok
7 else type_error
```

34. $L \rightarrow EQ$

```
1 L.type = E.type x Q.type
```

35. $L \rightarrow \lambda$

```
1 L.type = type_void
```

36. $Q \rightarrow \text{Comma } EQ_1$

```
1 Q.type = E.type x Q1.type
```

37. $Q \rightarrow \lambda$

```
1 Q.type = null
```

38. $X \rightarrow E$

```
1 X.type = E.type
```

39. $X \rightarrow \lambda$

```
1 X.type = type_void
```

40. $E \rightarrow E_1 \text{ And } R$

```
1 E.type = if E1.type == R.type && E1.type == type_bool
2     then type_bool
3 else type_error
```

41. $E \rightarrow R$

```
1 E.type = R.type
```

42. $R \rightarrow R_1 \text{ Eq } RR$

```
1 R.type = if R1.type in {type_int, type_float} && R1.type == RR.type
2     then type_bool
3 else type_error
```

43. $R \rightarrow RR$

```
1 R.type = RR.type
```

44. $RR \rightarrow RR_1 \text{ Lt } U$

```
1 RR.type = if RR1.type in {type_int, type_float} && RR1.type == U.type  
2     then type_bool  
3 else type_error
```

45. $RR \rightarrow U$

```
1 RR.type = U.type
```

46. $U \rightarrow U_1 \text{ Sum } UU$

```
1 U.type = if U1.type in {type_int, type_float} && U1.type == UU.type  
2     then U1.type  
3 else type_error
```

47. $U \rightarrow UU$

```
1 U.type = UU.type
```

48. $UU \rightarrow UU_1 \text{ Mul } EE$

```
1 UU.type = if UU1.type in {type_int, type_float} && UU1.type == EE.type  
2     then UU1.type  
3 else type_error
```

49. $UU \rightarrow EE$

```
1 UU.type = EE.type
```

50. $EE \rightarrow \text{Not } EE_1$

```
1 EE.type = if EE1.type == type_bool  
2     then type_bool  
3 else type_error
```

51. $EE \rightarrow \text{Sub } EE_1$

```
1 EE.type = if EE1.type in {type_int, type_float}  
2     then EE1.type  
3 else type_error
```

52. $EE \rightarrow \text{Sum } EE_1$

```
1 EE.type = if EE1.type in {type_int, type_float}  
2     then EE1.type  
3 else type_error
```

53. $EE \rightarrow V$

```
1 EE.type = V.type
```

54. $V \rightarrow \text{Id LParen } L \text{ RParen}$

```
1 type = symtable.scopes.peek().search_type(Id.pos)
2
3 V.type = if type == null
4     then type_error
5 else if type == L.type -> ret_type
6     then ret_type
7 else type_error
```

55. $V \rightarrow \text{LParen } E \text{ RParen}$

```
1 V.type = E.type
```

56. $V \rightarrow \text{IntLit}$

```
1 V.type = type_int
2 V.type.size = 1
```

57. $V \rightarrow \text{FloatLit}$

```
1 V.type = type_float
2 V.type.size = 2
```

58. $V \rightarrow \text{StrLit}$

```
1 V.type = type_str
2 V.type.size = 64
```

59. $V \rightarrow \text{True}$

```
1 V.type = type_bool
2 V.type.size = 1
```

60. $V \rightarrow \text{False}$

```
1 V.type = type_bool
2 V.type.size = 1
```

61. $V \rightarrow \text{Id}$

```
1 scope = symtable.scopes.peek()
2 type = scope.search_type(Id.pos)
3
4 if type == null {
5     scope.add_type(Id.pos, type_int)
6     scope.add_despl(Id.pos, scope.despl)
7
8     scope.despl += 1
9     type = type_int
10 }
11
12 V.type = type
```

8 Gestión de Errores

El Gestor de Errores, es el componente más importante de un procesador de cara a la UX. Es fundamental emitir errores que sean claros y se entiendan con facilidad.

Es por esto que se han decidido seguir estas pautas durante el diseño del Gestor de Errores:

- Todo error debe entenderse por sí mismo, es decir, en su mensaje debe estar toda la información necesaria para saber porque ha ocurrido, sin necesidad de consultar el fichero fuente.
- Los errores deben ser localizables en el fichero fuente, es decir, se debe proporcionar como mínimo el número de línea y columna de cada error, así como de su contexto.
- El gestor de errores debe ser conservador en la clasificación de los errores. Solo se emitirán diagnósticos específicos cuando el tipo de error pueda determinarse de forma inequívoca. Se intentará minimizar el uso de heurísticas y suposiciones para refinar el diagnóstico, ya que un diagnóstico incorrecto es inaceptable, incluso si solo falla 1 de cada 100 veces.

Además, el Gestor de Errores va a ser capaz de generar sugerencias en determinados errores. Es decir, como corregirlos a través de sustituciones, inserciones o eliminaciones de texto.

8.1 Estructura de un Diagnóstico

Cada diagnóstico va a estar formado por varios componentes, que juntos sirven la función de maximizar la claridad de cada error:

- Mensaje principal: Descripción general del error, que incluye el nombre del fichero, así como el número de línea y columna dónde se produce.
- Rango del error: Fragmento del fichero en el que se localiza el error, subrayado y resaltado de color rojo en la línea correspondiente, y acompañado de una nota con información más específica sobre el problema detectado.
- Rangos de notas: Otros intervalos del fichero, subrayados y resaltados de color azul, que aportan contexto adicional para la comprensión del error.
- Sugerencia: Cuando es posible determinar con certeza como corregir el error, se emite un breve mensaje que indica como modificar el fichero para resolver el fallo.

Para ilustrar la importancia de todos estos elementos, se considera un error del *Parser*: *Delimitador Desemparejado*. Este error ocurre cuando un paréntesis abierto se cierra con una llave o viceversa.

Usando sólo el mensaje principal y el rango del error, el diagnóstico quedaría así:

```
tests/challenge/parser.txt:27:1: error: mismatched closing delimiter `}`  
27 | }  
    ^ mismatched closing delimiter
```

Este diagnóstico está bastante incompleto. Permite localizar el error pero sólo parcialmente, ya que muestra el delimitador cerrado desemparejado pero no el abierto. Esto también significa que el diagnóstico no se puede entender por sí mismo, el usuario tendría que abrir el fichero y ver porqué la llave está desemparejada y con qué.

Para remediar esto, el *Parser* mantiene una pila con todos los delimitadores abiertos sin cerrar, por lo que se puede acceder fácilmente a la información que falta:

```
tests/challenge/parser.txt:27:1: error: mismatched closing delimiter `}`
25 |     write(var
|         ^ unclosed delimiter
|
27 | }
| ^ mismatched closing delimiter
```

Además, este error casi siempre ocurre porque el usuario se ha olvidado de cerrar el delimitador abierto. Por ello, el delimitador cerrado desemparejado suele corresponder a otro delimitador abierto anterior que, en caso de encontrarse, puede proporcionar aún más contexto:

```
tests/challenge/parser.txt:27:1: error: mismatched closing delimiter `}`
21 |     function int operation(void) {
|             ^ closing delimiter possibly meant for this
|
25 |     write(var
|         ^ unclosed delimiter
|
27 | }
| ^ mismatched closing delimiter
```

Ahora el error es mucho más claro. Es localizable y también comprensible sin necesidad de consultar el fichero fuente. Además, con todo este contexto, es fácil ver que el error se puede arreglar cerrando el paréntesis abierto de la línea 25, por lo que es posible añadir una sugerencia:

```
tests/challenge/parser.txt:27:1: error: mismatched closing delimiter `}`
21 |     function int operation(void) {
|             ^ closing delimiter possibly meant for this
|
25 |     write(var
|         ^ unclosed delimiter
|
27 | }
| ^ mismatched closing delimiter
--> help: insert `)` and `;` after `var`
25 |     write(var);
|         ++
```

Todo este proceso se lleva a cabo de forma automática durante la generación de los diagnósticos del procesador. Gracias a la información mantenida por el *Parser*, se crean diagnósticos dinámicos y enriquecidos por su contexto. Ésto permite emitir errores más informativos, robustos y accionables, capaces de guiar al usuario directamente hacia la causa del problema y su posible corrección.

8.2 Recuperación de Errores

Como se ha comentado a lo largo de la memoria, todos los módulos del procesador implementan recuperación de errores con el fin de reportar la mayor cantidad de errores por ejecución, minimizando el número de ejecuciones del procesador necesarias para corregir por completo el fichero fuente.

La estrategia de recuperación varía según el módulo:

8.2.1 Recuperación del Lexer

El *Lexer* tiene la recuperación de errores más sencilla de todas. Al encontrar un error, el Analizador Léxico intenta emitir un *token* ficticio que ayuda a que tanto el *Parser* como el Analizador Semántico generen diagnósticos más coherentes.

Por ejemplo, si ocurre un error de *Overflow de Entero*, como se sabe que el usuario quería escribir una constante de tipo entero, se emite un *token* constante entera sin valor. Se emite un *token* ficticio en todos los diagnósticos del *Lexer* menos en *Carácter inválido*, en la que simplemente se salta el carácter y se continúa analizando.

Las ventajas de esta estrategia frente a no emitir ningún *token* ficticio y continuar el análisis sin intervención pueden observarse en el siguiente fragmento:

```
1 let int a = 3;
2
3 let string real = 92389233892;
```

Que produce los siguientes diagnósticos:

```
lexer_text.txt:3:30: error: integer literal out of range for 16-byte type
  3 | let string real = 92389233892;
    | ^^^^^^^^^^ maximum is 32767

lexer_text.txt:3:30: error: mismatched types
  3 | let string real = 92389233892;
    | ----- ^^^^^^^^^^ expected `string`, found `int`
    | |
    | expected `string` because of this
```

Gracias a que se genera el *token* fantasma, a pesar de producir un error, se puede detectar el error semántico en la declaración, ya que el usuario, hubiese *overflow* o no, estaba intentando asignar a una variable de tipo *string* una constante entera. Además si no se hubiese generado este *token*, el procesador habría emitido también errores sintácticos aparentemente sin sentido, ya que al omitir el *token* constante entera el *Parser* habría encontrado un *token* = seguido de un punto y coma.

8.2.2 Recuperación del Parser

Como se comentó brevemente en la sección de errores del *Parser*, cuando el Analizador Sintáctico encuentra un error, intenta encontrar una secuencia de cambios que permita resincronizar la pila de estados con el *token* que lo provocó.

Para ello se realiza una búsqueda heurística en anchura por el autómata *LR(1)*. Cada error se intenta corregir mediante una secuencia de inserciones, una secuencia de eliminaciones o una sustitución. Cada arreglo se puntuá según el número de inserciones y/o eliminaciones que conlleva, y se selecciona la solución con la puntuación más baja.

Si no se encuentra una corrección adecuada, o si el arreglo resultara demasiado costoso o inconsistente, se emite el error y se interrumpe la ejecución del procesador.

La clasificación de los errores se realiza analizando los contenidos de la pila de estados junto al *token* que produjo el error. Esta estrategia se ha escogido frente a una clasificación manual basada en las celdas de las tablas de acción y *goto* por su flexibilidad. Una clasificación manual resulta demasiado laboriosa, ya que no solo hay cientos de celdas en las tablas, sino que cualquier modificación de la gramática obligaría a rehacer por completo el manejo de los errores.

A continuación se muestra un ejemplo de cada tipo de recuperación: Por inserción, eliminación y sustitución.

```
tests/challenge/parser.txt:37:21: error: expected a statement or a function, found `int`
36 | int a = 4;
   |         - after this
37 | int wellwell(void) {}
   | ^^^ expected a statement or a function
--> help: insert `function` before `int`
37 | function int wellwell(void) {}
   | +++++++
```

```
tests/challenge/parser.txt:52:14: error: trailing comma in a function call
52 | chachi(hola,);
   |         ^ here
--> help: remove the trailing comma
52 - chachi(hola,);
52 + chachi(hola);
```

```
tests/challenge/parser.txt:45:6: error: expected `;` or a binary operator, found `,`
45 | b = 4,
   |         ^ expected `;` or a binary operator
--> help: replace `,` by `;`
45 - b = 4,
45 + b = 4;
```

8.2.3 Recuperación del Analizador Semántico

La recuperación de este módulo ya se vio en parte en sus acciones semánticas. Al encontrar un error se propaga el tipo especial *tipo_error*, que permite continuar el análisis y la emisión de más excepciones cuando se reducen conjuntos de símbolos sin errores.

Es importante también como se comporta cuando ocurren errores sintácticos. Se considera el siguiente fragmento:

```
1 let var = ;
2
3 var = "hola";
```

El analizador sintáctico para recuperar los errores de ese fichero insertará un tipo tras el *let* y una expresión tras el *=* en la primera línea. Sin embargo, estas inserciones no tienen porque ser coherentes semánticamente y desde luego no tienen porque ser compatibles con la tercera línea. Por ello, si el Analizador Semántico ve que los *tokens* que produjeron el error son ficticios, no emite ningún diagnóstico y simplemente asigna *tipo_error* y continúa. Habrá que esperar a que se corrijan los errores sintácticos para emitir los semánticos:

```
sem_test.txt:1:5: error: missing type in a variable declaration
1 | let var = ;
|     ^^^ expected type
--> help: add the missing type
1 | let type var = ;
|     +++

```

Sin embargo, si el fragmento fuese el siguiente:

```
1 int var = 3;
2
3 var = "hola";
```

Ahora sí que se emitirán diagnósticos ya que se sabe que el tipo de *var* es *int*, aunque la sentencia genere un error:

```
sem_test.txt:1:1: error: expected a function or a statement, found `int`
1 | int var = 3;
|     ^^^ expected a function or a statement
--> help: insert `let` before `int`
1 | let int var = 3;
|     +++
sem_test.txt:3:13: error: mismatched types
1 | int var = 3;
|     --- expected `int` due to it's declaration
:
3 | var = "hola";
|     ^^^^^^ expected `int`, found `string`
```

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.java 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:

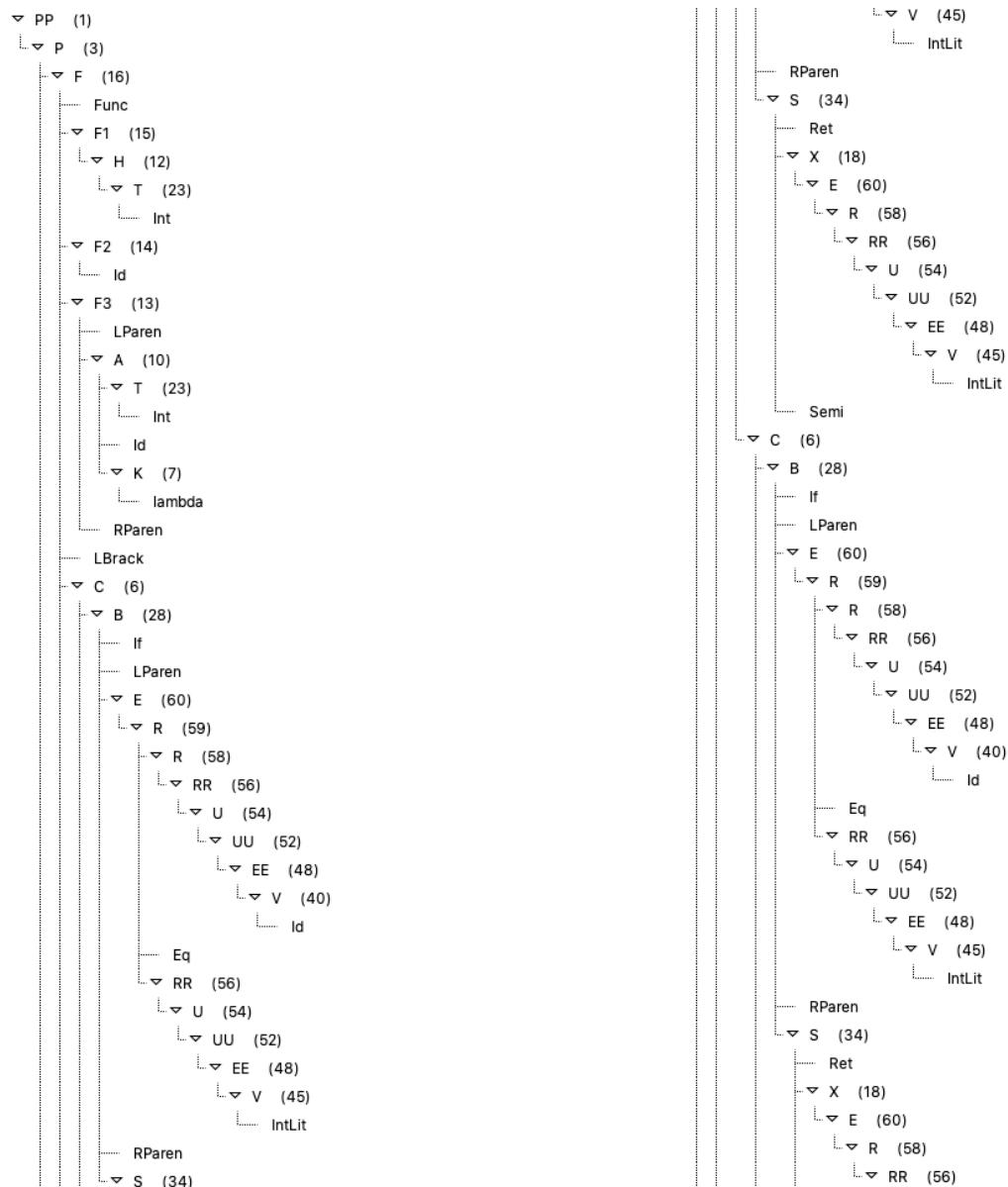
1 <Func, >	20 <Id, 1>	40 <LBrack, >	60 <Sum, >
2 <Int, >	21 <Eq, >	41 <Let, >	61 <Sub, >
3 <Id, 0>	22 <IntLit, 1>	42 <Int, >	62 <IntLit, 1>
4 <LParen, >	23 <RParen, >	43 <Id, 4>	63 <Semi, >
5 <Int, >	24 <Ret, >	44 <Assign, >	64 <RBrack, >
6 <Id, 1>	25 <IntLit, 1>	45 <Id, 2>	65 <While, >
7 <RParen, >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

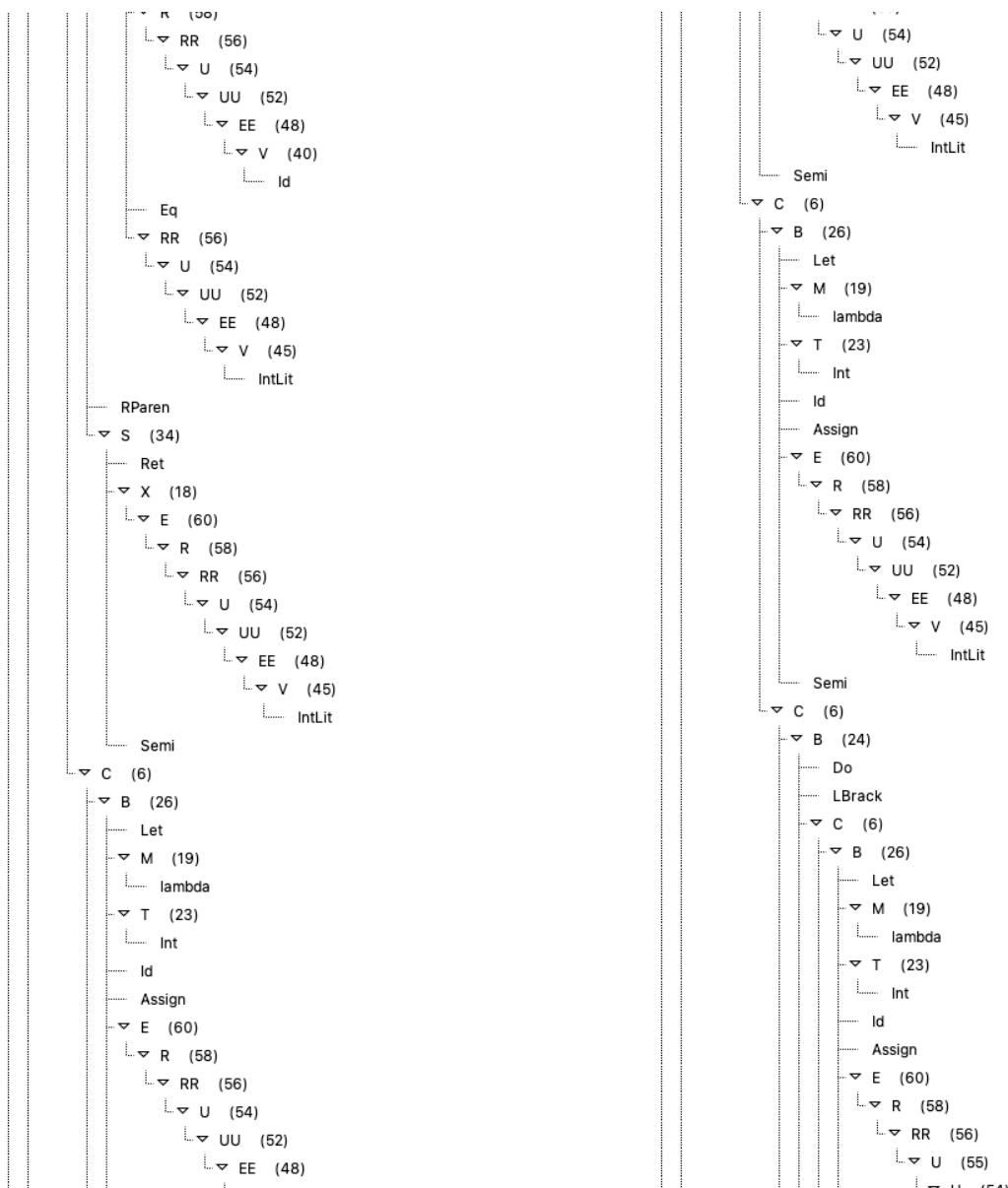
Parse:

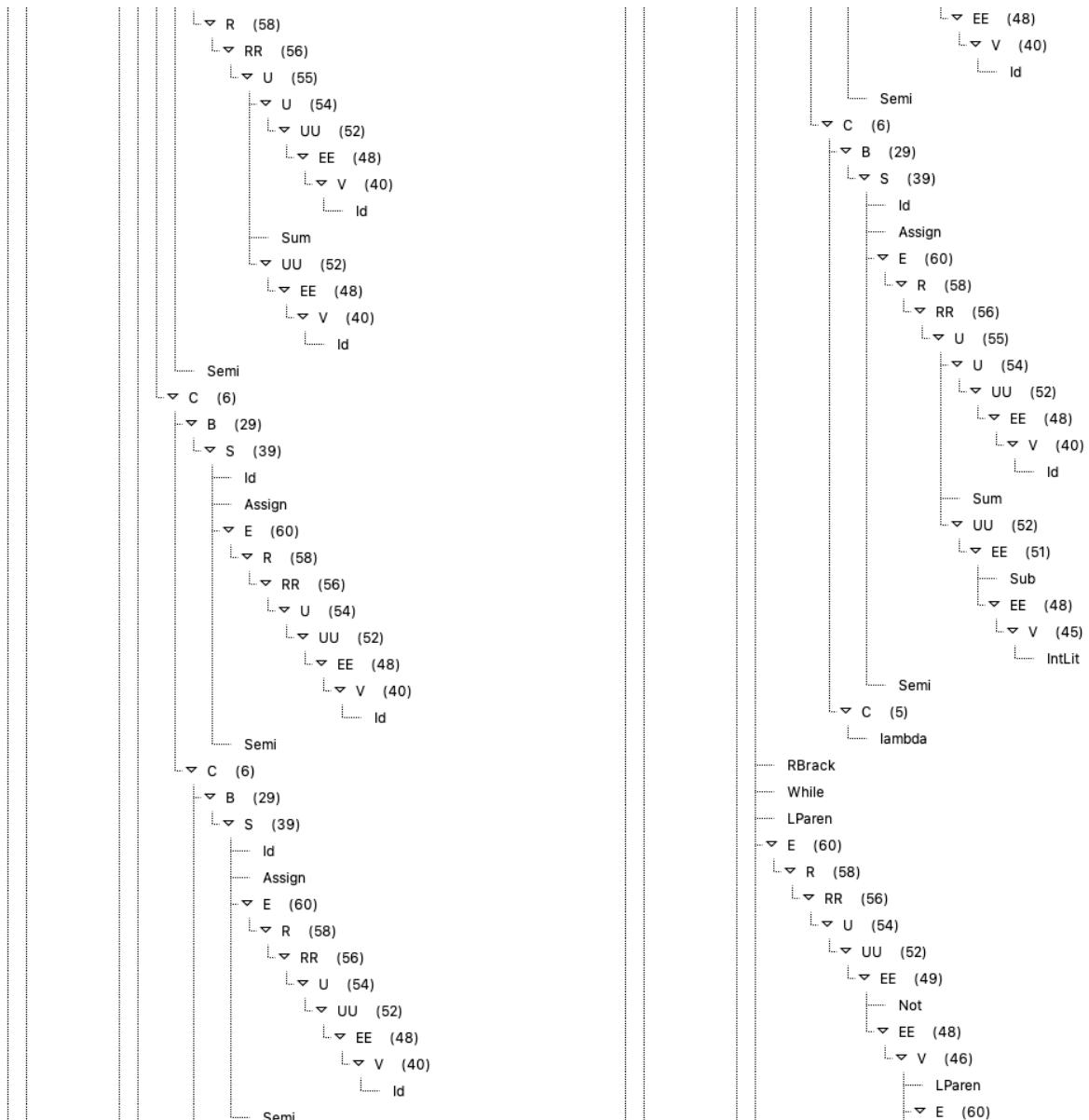
```
1 ascending 23 12 15 14 23 7 10 13 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48  
52 54 56 58 60 18 34 28 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48 52 54  
56 58 60 18 34 28 19 23 45 48 52 54 56 58 60 26 19 23 45 48 52 54 56 58 60  
26 19 23 40 48 52 54 40 48 52 55 56 58 60 26 40 48 52 54 56 58 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 16 2 3 1
```

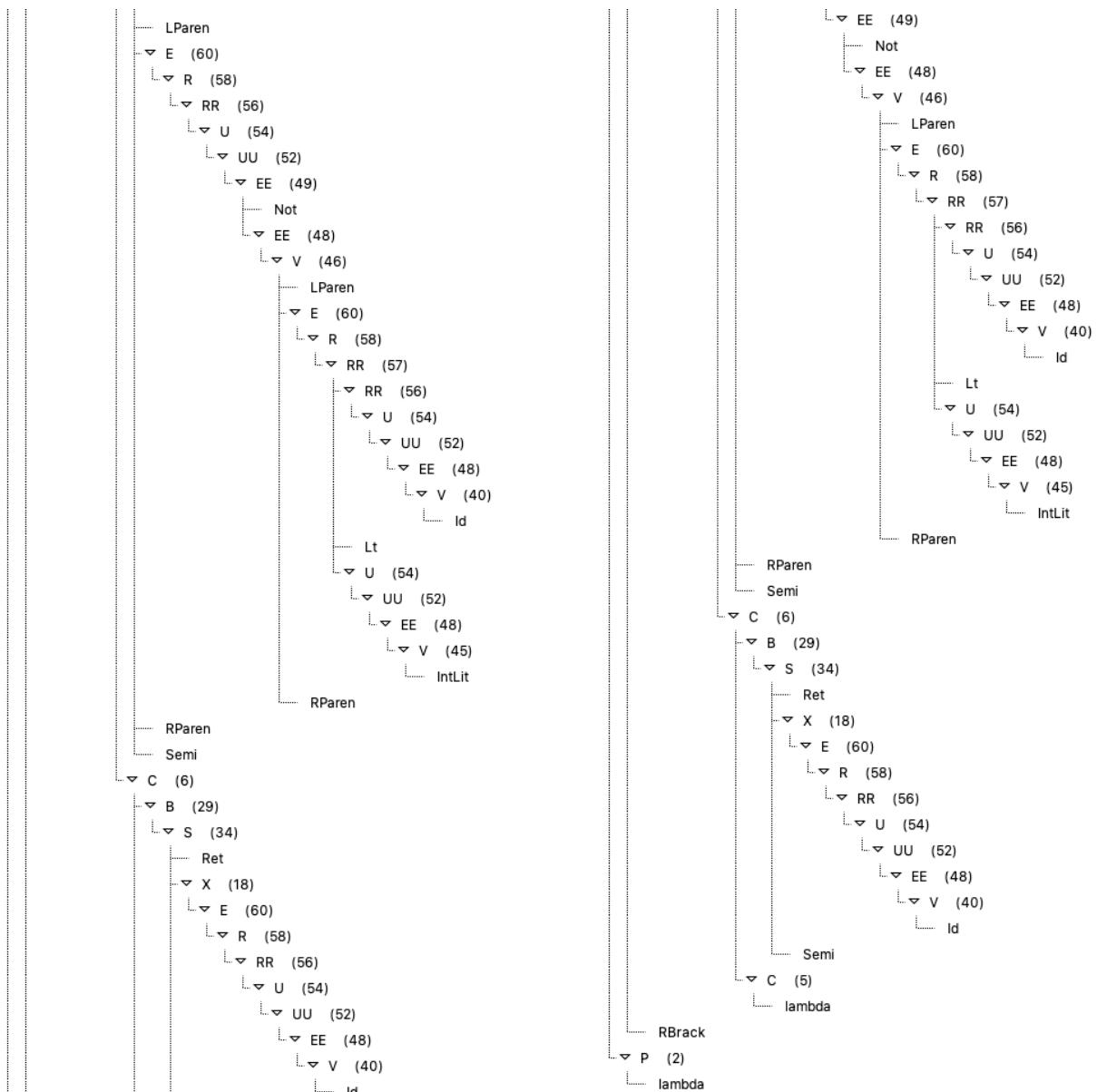
Tabla de símbolos:

```
1 fib table #1:  
2 * lexema: 'n'  
3 + tipo: 'int'  
4 + despl: 0  
5 * lexema: 'a'  
6 + tipo: 'int'  
7 + despl: 1  
8 * lexema: 'b'  
9 + tipo: 'int'  
10 + despl: 2  
11 * lexema: 'c'  
12 + tipo: 'int'  
13 + despl: 3  
14  
15 global table #0:  
16 * lexema: 'fib'  
17 + etiqFuncion: '__tag_fib__'  
18 + tipo: 'function'  
19 + tipoRetorno: 'int'  
20 + numParam: 1  
21     + tipoParam01: 'int'
```









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, >	29 <Id , 2>	58 <Semi, >
2 <Int , >	30 <Assign , >	59 <Ret , >
3 <Id , 0>	31 <Id , 1>	60 <Id , 2>
4 <LParen , >	32 <Semi, >	61 <Semi, >
5 <Int , >	33 <Do, >	62 <RBrack , >
6 <Id , 1>	34 <LBrack , >	63 <Let , >
7 <RParen , >	35 <Id , 1>	64 <Int , >
8 <LBrack , >	36 <Assign , >	65 <Id , 1>
9 <If , >	37 <Id , 1>	66 <Assign , >
10 <LParen , >	38 <Sum, >	67 <IntLit , 0>
11 <Id , 1>	39 <Sub , >	68 <Semi, >
12 <Eq , >	40 <IntLit , 1>	69 <Read , >
13 <IntLit , 0>	41 <Semi, >	70 <Id , 1>
14 <RParen , >	42 <Id , 2>	71 <Semi, >
15 <Ret , >	43 <Assign , >	72 <Let , >
16 <IntLit , 1>	44 <Id , 2>	73 <Int , >
17 <Semi , >	45 <Mul , >	74 <Id , 2>
18 <If , >	46 <Id , 1>	75 <Assign , >
19 <LParen , >	47 <Semi , >	76 <Id , 0>
20 <Id , 1>	48 <RBrack , >	77 <LParen , >
21 <Lt , >	49 <While , >	78 <Id , 1>
22 <IntLit , 2>	50 <LParen , >	79 <RParen , >
23 <RParen , >	51 <Not , >	80 <Semi , >
24 <Ret , >	52 <LParen , >	81 <Write , >
25 <Id , 1>	53 <Id , 1>	82 <LParen , >
26 <Semi , >	54 <Lt , >	83 <Id , 2>
27 <Let , >	55 <IntLit , 2>	84 <RParen , >
28 <Int , >	56 <RParen , >	85 <Semi , >
	57 <RParen , >	

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  
2 52 54 56 58 60 18 34 28 40 48 52 54 56 45 48 52 54 57 58 60 40 48 52 54 56  
3 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  
4 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  
5 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  
7 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  
8 29 2 4 4 4 4 3 1
```

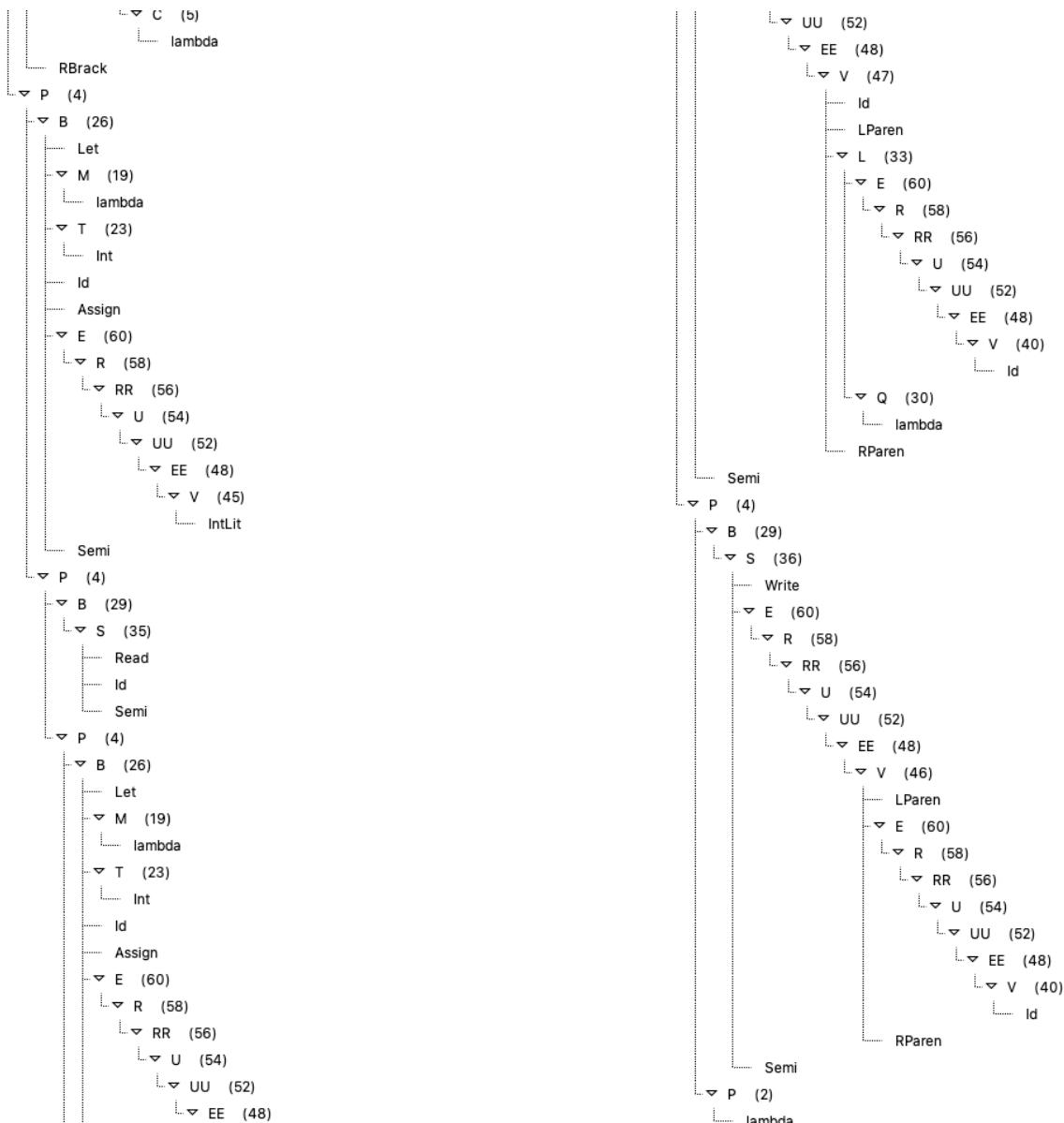
Tabla de símbolos:

```
1 factorial table #1:  
2 * lexema: 'n'  
3 + tipo: 'int'  
4 + despl: 0  
5 * lexema: 'res'  
6 + tipo: 'int'  
7 + despl: 1  
8  
9 global table #0:  
10 * lexema: 'factorial'  
11 + etiqFuncion: '__tag_factorial__'  
12 + tipo: 'function'  
13 + tipoRetorno: 'int'  
14 + numParam: 1  
15 + tipoParam01: 'int'  
16 * lexema: 'n'  
17 + tipo: 'int'  
18 + despl: 0  
19 * lexema: 'res'  
20 + tipo: 'int'  
21 + despl: 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=
6 false;
7 let boolean george=true;george&=bool;}/***/ / eof /***/

```

Fichero de tokens:

1 <Let, >	38 <Func, >	76 <Sum, >
2 <Int, >	39 <Int, >	77 <Sub, >
3 <Id, 0>	40 <Id, 2>	78 <Sum, >
4 <Assign, >	41 <LParen, >	79 <Sub, >
5 <IntLit, 13>	42 <Float, >	80 <Sum, >
6 <Semi, >	43 <Id, 3>	81 <Sub, >
7 <Func, >	44 <Comma, >	82 <Sum, >
8 <Void, >	45 <Str, >	83 <Sub, >
9 <Id, 1>	46 <Id, 4>	84 <Sum, >
10 <LParen, >	47 <RParen, >	85 <Sub, >
11 <Void, >	48 <LBrack, >	86 <Sum, >
12 <RParen, >	49 <Let, >	87 <Sub, >
13 <LBrack, >	50 <Str, >	88 <Sum, >
14 <Read, >	51 <Id, 5>	89 <IntLit, 5>
15 <Id, 0>	52 <Assign, >	90 <Semi, >
16 <Semi, >	53 <StrLit, "this is a str	91 <Let, >
17 <Do, >	">	92 <Float, >
18 <LBrack, >	54 <Semi, >	93 <Id, 9>
19 <Id, 0>	55 <Let, >	94 <Assign, >
20 <Assign, >	56 <Float, >	95 <FloatLit, 2>
21 <Id, 0>	57 <Id, 6>	96 <Semi, >
22 <Sum, >	58 <Assign, >	97 <Let, >
23 <Sub, >	59 <FloatLit, 1203.23>	98 <Bool, >
24 <IntLit, 1>	60 <Semi, >	99 <Id, 10>
25 <Semi, >	61 <Let, >	100 <Assign, >
26 <RBrack, >	62 <Int, >	101 <False, >
27 <While, >	63 <Id, 7>	102 <Semi, >
28 <LParen, >	64 <Assign, >	103 <Let, >
29 <Not, >	65 <IntLit, 2398>	104 <Bool, >
30 <LParen, >	66 <Semi, >	105 <Id, 11>
31 <Id, 0>	67 <Let, >	106 <Assign, >
32 <Lt, >	68 <Int, >	107 <True, >
33 <IntLit, 0>	69 <Id, 8>	108 <Semi, >
34 <RParen, >	70 <AndAssign, >	109 <Id, 11>
35 <RParen, >	71 <IntLit, 3>	110 <AndAssign, >
36 <Semi, >	72 <Sum, >	111 <Id, 10>
37 <RBrack, >	73 <Sub, >	112 <Semi, >
	74 <Sum, >	113 <RBrack, >
	75 <Sub, >	

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 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 6 16 2 3
    3 4 1

```

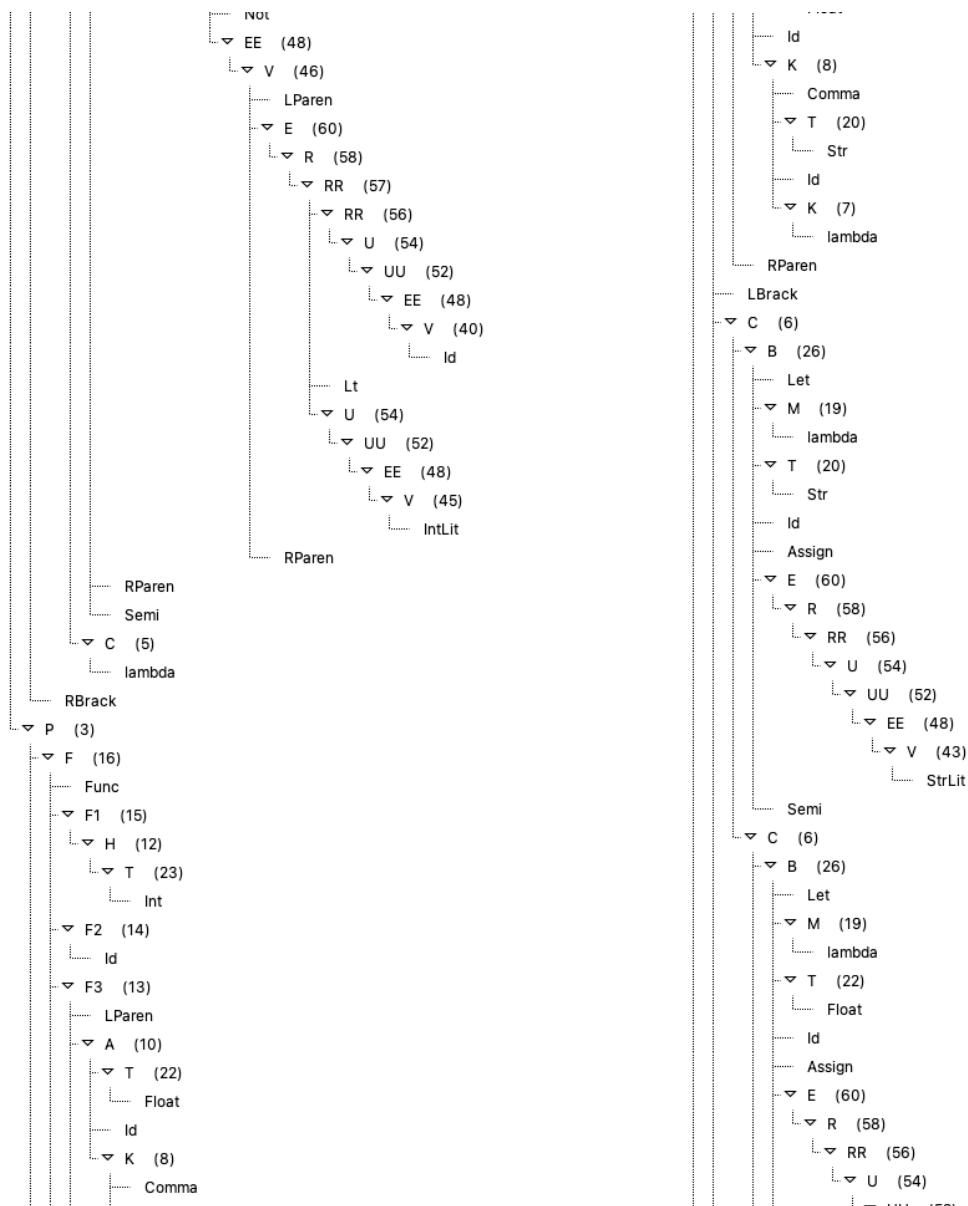
Tabla de símbolos:

```

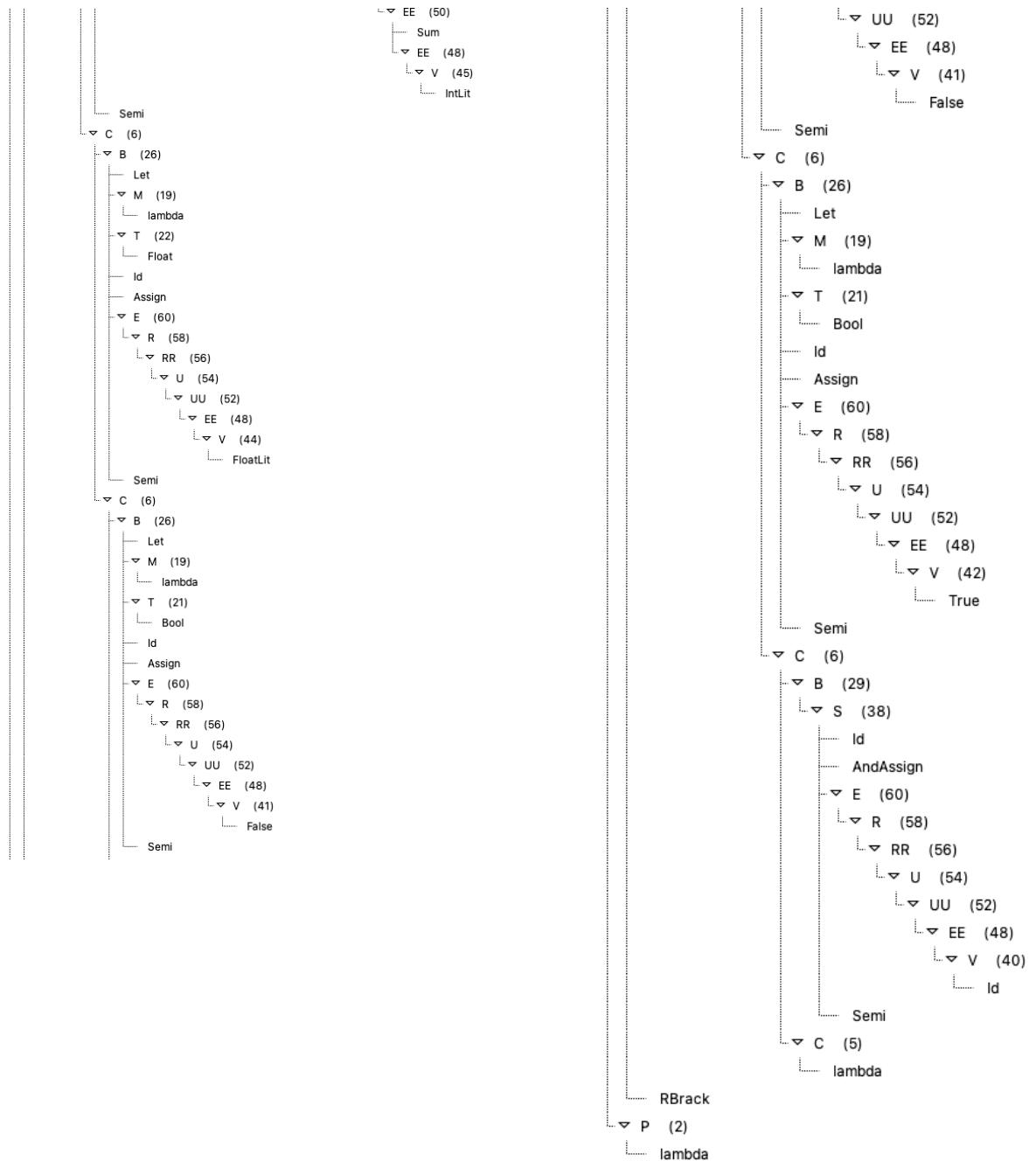
1 nothing table #1:
2
3 main table #2:
4 * lexema: 'b'
  + tipo: 'float'
  + despl: 0
5 * lexema: 'd'
  + tipo: 'string'
  + despl: 2
6 * lexema: 'aux'
  + tipo: 'string'
  + despl: 66
7 * lexema: 'foo'
  + tipo: 'float'
  + despl: 130
8 * lexema: 'bar'
  + tipo: 'int'
  + despl: 132
9 * lexema: 'weird'
  + tipo: 'int'
  + despl: 133
10 * lexema: 'oper'
  + tipo: 'float'
  + despl: 134
11 * lexema: 'bool'
  + tipo: 'boolean'
  + despl: 136
12 * lexema: 'george'
  + tipo: 'boolean'
  + despl: 137
13
14 global table #0:
15 * lexema: 'global'
  + tipo: 'int'
  + despl: 0
16 * lexema: 'nothing'
  + etiqFuncion: '__tag_nothing__'
  + tipo: 'function'
  + tipoRetorno: 'void'
  + numParam: 0
17 * lexema: 'main'
  + etiqFuncion: '__tag_main__'
  + tipo: 'function'
  + tipoRetorno: 'int'
  + numParam: 2
  + tipoParam01: 'float'
  + tipoParam02: 'string'

```









syntax.javascript Fichero con construcciones sintácticas enrevesadas y recursividad.

```

1 let int a = 9;
2
3 /* el lenguaje es sensible a las mayúsculas */
4 function int Void(int a) {
5     do {
6         if (a == a)
7             return a;
8     } while (!!(a < a));
9
10    return a + -a;
11 }
12
13 /* el lenguaje admite recursividad */
14 function int recursive(int a, float b, string c) {
15     return 3 + recursive(
16         3 * (implicit + recursive(6,6.0,"6")),
17         0.0000000000 * 1.1 * 2.2,
18         "."
19     );
20 }
```

Fichero de tokens:

1 <Let, >	22 <RParen, >	45 <RBrack, >	68 <Id , 5>
2 <Int, >	23 <Ret, >	46 <Func, >	69 <Sum, >
3 <Id, 0>	24 <Id, 0>	47 <Int, >	70 <Id, 2>
4 <Assign, >	25 <Semi, >	48 <Id, 2>	71 <LParen, >
5 <IntLit, 9>	26 <RBrack, >	49 <LParen, >	72 <IntLit, 6>
6 <Semi, >	27 <While, >	50 <Int, >	73 <Comma, >
7 <Func, >	28 <LParen, >	51 <Id, 0>	74 <FloatLit, 6>
8 <Int, >	29 <Not, >	52 <Comma, >	75 <Comma, >
9 <Id, 1>	30 <Not, >	53 <Float, >	76 <StrLit, "6">
10 <LParen, >	31 <Not, >	54 <Id, 3>	77 <RParen, >
11 <Int, >	32 <LParen, >	55 <Comma, >	78 <RParen, >
12 <Id, 0>	33 <Id, 0>	56 <Str , >	79 <Comma, >
13 <RParen, >	34 <Lt , >	57 <Id, 4>	80 <FloatLit, 0>
14 <LBrack, >	35 <Id, 0>	58 <RParen, >	81 <Mul, >
15 <Do, >	36 <RParen, >	59 <LBrack, >	82 <FloatLit, 1>
16 <LBrack, >	37 <RParen, >	60 <Ret, >	83 <Mul, >
17 <If, >	38 <Semi, >	61 <IntLit, 3>	84 <FloatLit, 2>
18 <LParen, >	39 <Ret, >	62 <Sum, >	85 <Comma, >
19 <Id, 0>	40 <Id, 0>	63 <Id, 2>	86 <StrLit, ".">
20 <Eq, >	41 <Sum, >	64 <LParen, >	87 <RParen, >
21 <Id, 0>	42 <Sub, >	65 <IntLit, 3>	88 <Semi, >
	43 <Id, 0>	66 <Mul, >	89 <RBrack, >
	44 <Semi, >	67 <LParen, >	

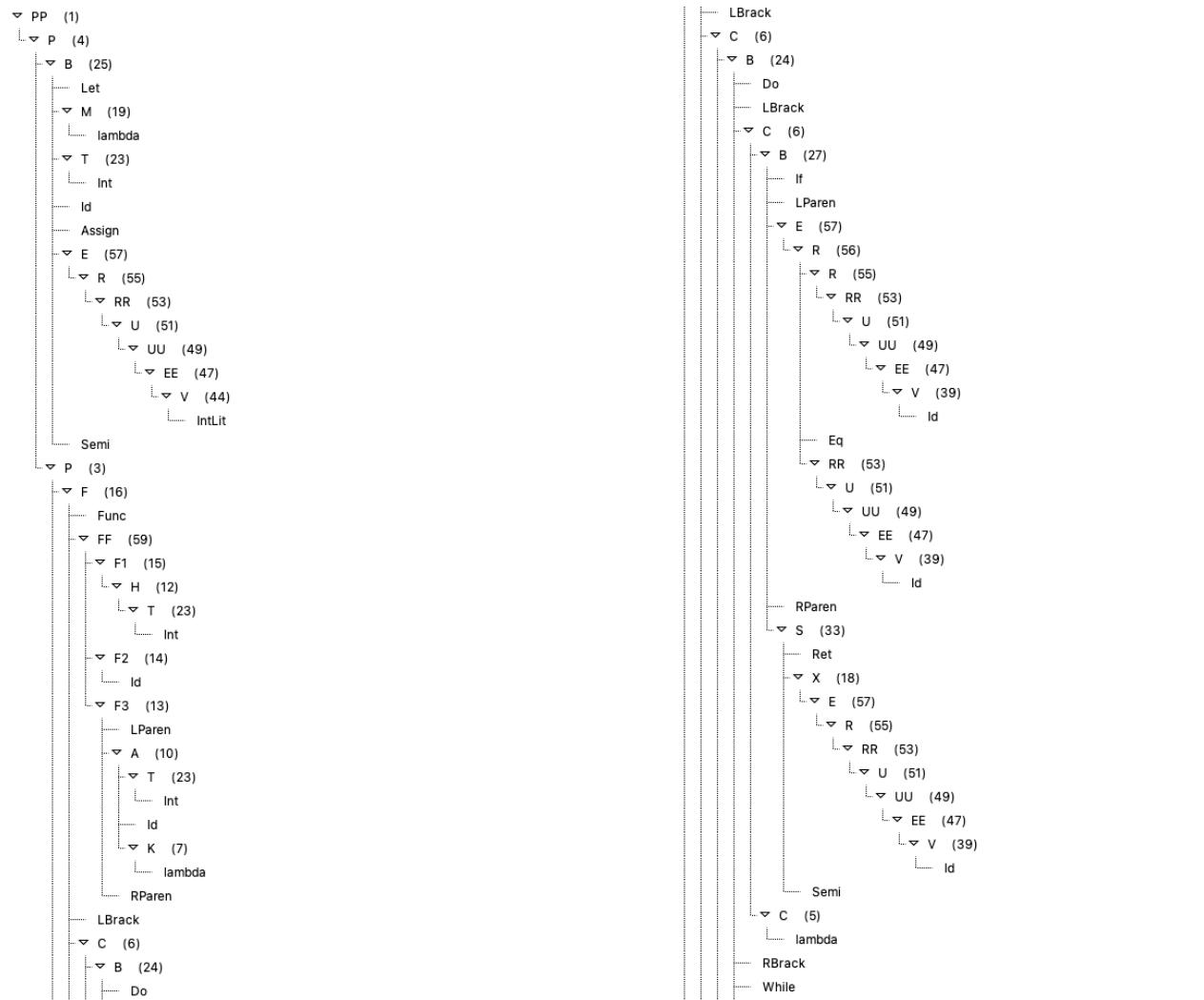
Parse:

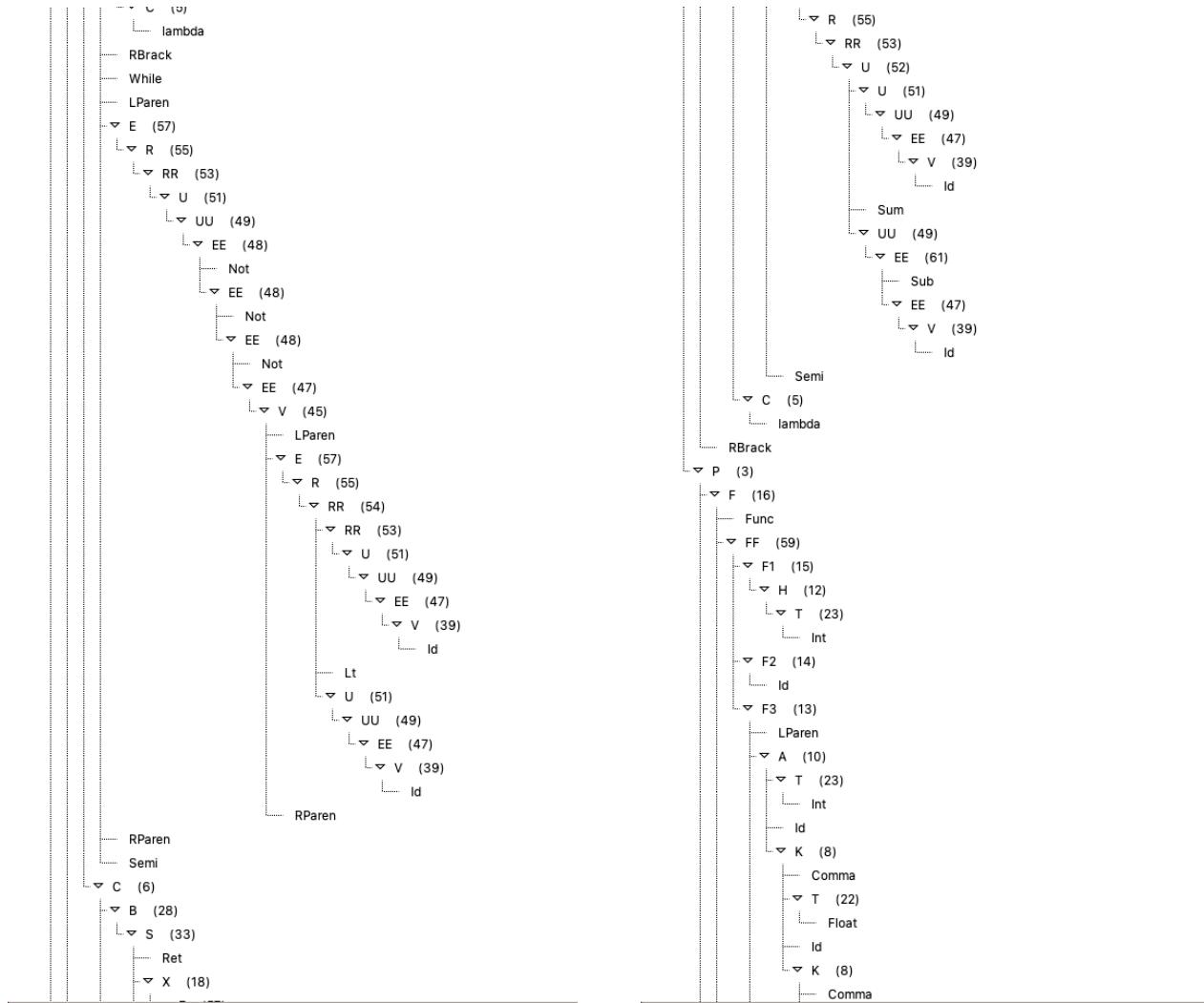
Tabla de símbolos:

```

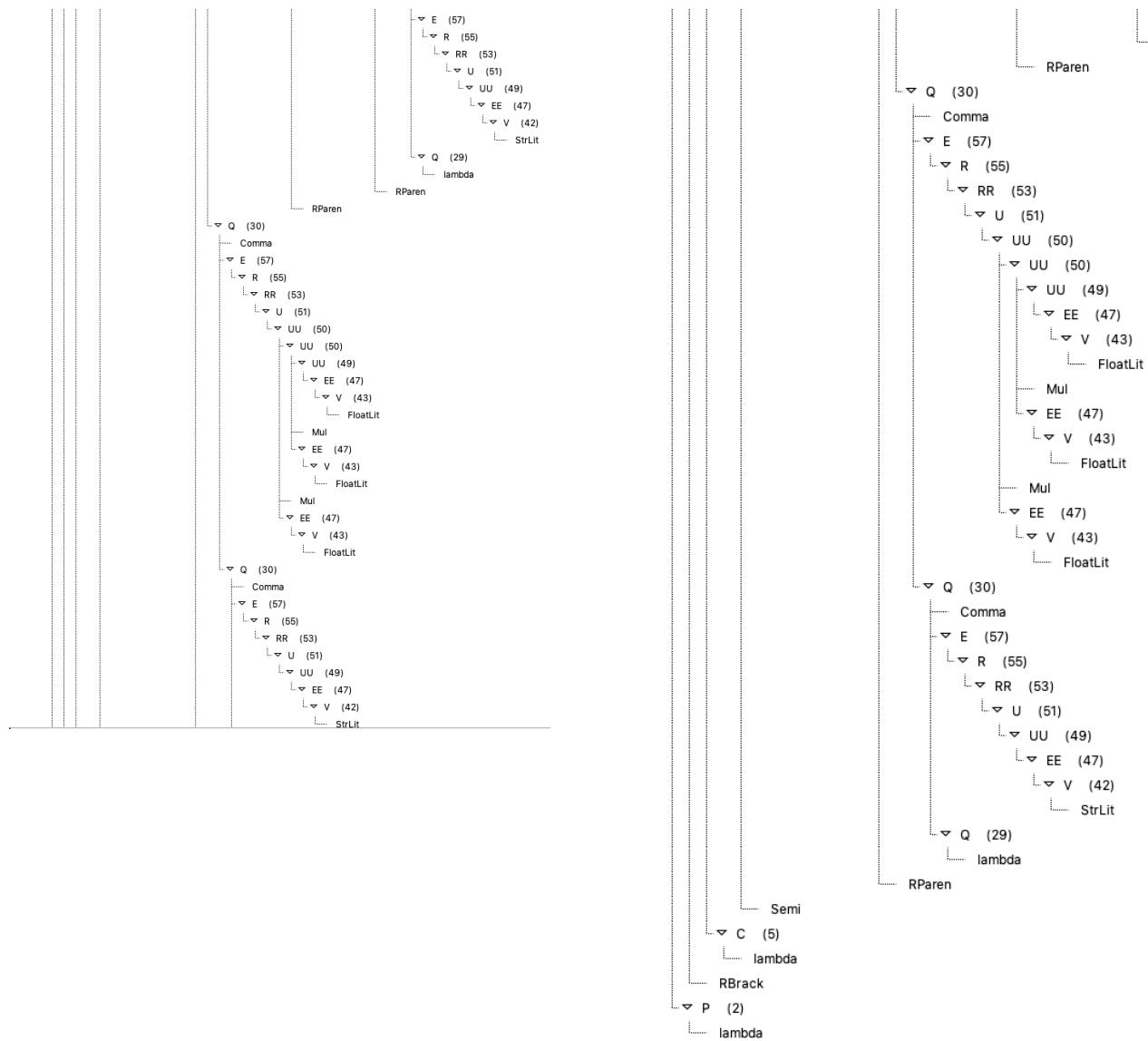
1 Void table #1:
2 * lexema: 'a'
3   + tipo: 'int'
4   + despl: 0
5
6 recursive table #2:
7 * lexema: 'a'
8   + tipo: 'int'
9   + despl: 0
10 * lexema: 'b'
11   + tipo: 'float'
12   + despl: 1
13 * lexema: 'c'
14   + tipo: 'string'
15   + despl: 3
16
17 global table #0:
18 * lexema: 'a'
19   + tipo: 'int'
20   + despl: 0
21 * lexema: 'Void'
22   + etiqFuncion: '__tag_Void__'
23   + tipo: 'function'
24     + tipoRetorno: 'int'
25     + numParam: 1
26       + tipoParam01: 'int'
27 * lexema: 'recursive'
28   + etiqFuncion: '__tag_recursive__'
29   + tipo: 'function'
30     + tipoRetorno: 'int'
31     + numParam: 3
32       + tipoParam01: 'int'
33       + tipoParam02: 'float'
34       + tipoParam03: 'string'
35 * lexema: 'implicit'
36   + tipo: 'int'
37   + despl: 1

```









hard.javascript Fichero con múltiples *edge cases* tanto sintácticos como semánticos.

```

1 read var;
2
3 let int a = var + ((23));
4
5 function int noreturn(void) {}
6
7 function void noop(void) {return;return;return;}
8
9 write implicit;
10
11 implicit = var + - + - implicit;
12
13 function int other_scope(float var) {
14     var = 5.0;
15     var = 6.0;
16 }
17
18 let string mystr = "mystr";
19
20 function int last_try(void) {
21     implicit = 0;
22
23     let float implicit;
24
25     implicit = 0.0;
26 }
```

Fichero de tokens:

1 <Read, >	23 <RBrack, >	47 <Sub, >	71 <StrLit , "mystr
2 <Id , 0>	24 <Func, >	48 <Id , 4>	72 <Semi, >
3 <Semi, >	25 <Void, >	49 <Semi, >	73 <Func, >
4 <Let, >	26 <Id , 3>	50 <Func, >	74 <Int , >
5 <Int , >	27 <LParen, >	51 <Int , >	75 <Id , 7>
6 <Id , 1>	28 <Void, >	52 <Id , 5>	76 <LParen, >
7 <Assign , >	29 <RParen, >	53 <LParen, >	77 <Void , >
8 <Id , 0>	30 <LBrack, >	54 <Float , >	78 <RParen , >
9 <Sum, >	31 <Ret , >	55 <Id , 0>	79 <LBrack , >
10 <LParen , >	32 <Semi, >	56 <RParen , >	80 <Id , 4>
11 <LParen , >	33 <Ret , >	57 <LBrack , >	81 <Assign , >
12 <IntLit , 23>	34 <Semi, >	58 <Id , 0>	82 <IntLit , 0>
13 <RParen , >	35 <Ret , >	59 <Assign , >	83 <Semi , >
14 <RParen , >	36 <Semi, >	60 <FloatLit , 5>	84 <Let , >
15 <Semi , >	37 <RBrack, >	61 <Semi , >	85 <Float , >
16 <Func , >	38 <Write , >	62 <Id , 0>	86 <Id , 4>
17 <Int , >	39 <Id , 4>	63 <Assign , >	87 <Semi , >
18 <Id , 2>	40 <Semi, >	64 <FloatLit , 6>	88 <Id , 4>
19 <LParen , >	41 <Id , 4>	65 <Semi , >	89 <Assign , >
20 <Void , >	42 <Assign , >	66 <RBrack , >	90 <FloatLit , 0>
21 <RParen , >	43 <Id , 0>	67 <Let , >	91 <Semi , >
22 <LBrack , >	44 <Sum, >	68 <Str , >	92 <RBrack , >
	45 <Sub , >	69 <Id , 6>	
	46 <Sum, >	70 <Assign , >	

Parse:

```

1 ascending 34 28 19 23 39 47 49 51 44 47 49 51 53 55 57 45 47 49 51 53 55 57 45
    47 49 52 53 55 57 25 23 12 15 14 9 13 59 5 16 11 15 14 9 13 59 17 33 28 17
    33 28 17 33 28 5 6 6 6 16 39 47 49 51 53 55 57 35 28 39 47 49 51 39 47 61
    60 61 49 52 53 55 57 38 28 23 12 15 14 22 7 10 13 59 43 47 49 51 53 55 57
    38 28 43 47 49 51 53 55 57 38 28 5 6 6 16 19 20 42 47 49 51 53 55 57 25 23
    12 15 14 9 13 59 44 47 49 51 53 55 57 38 28 19 22 26 43 47 49 51 53 55 57
    38 28 5 6 6 6 16 2 3 4 3 4 4 3 3 4 4 1

```

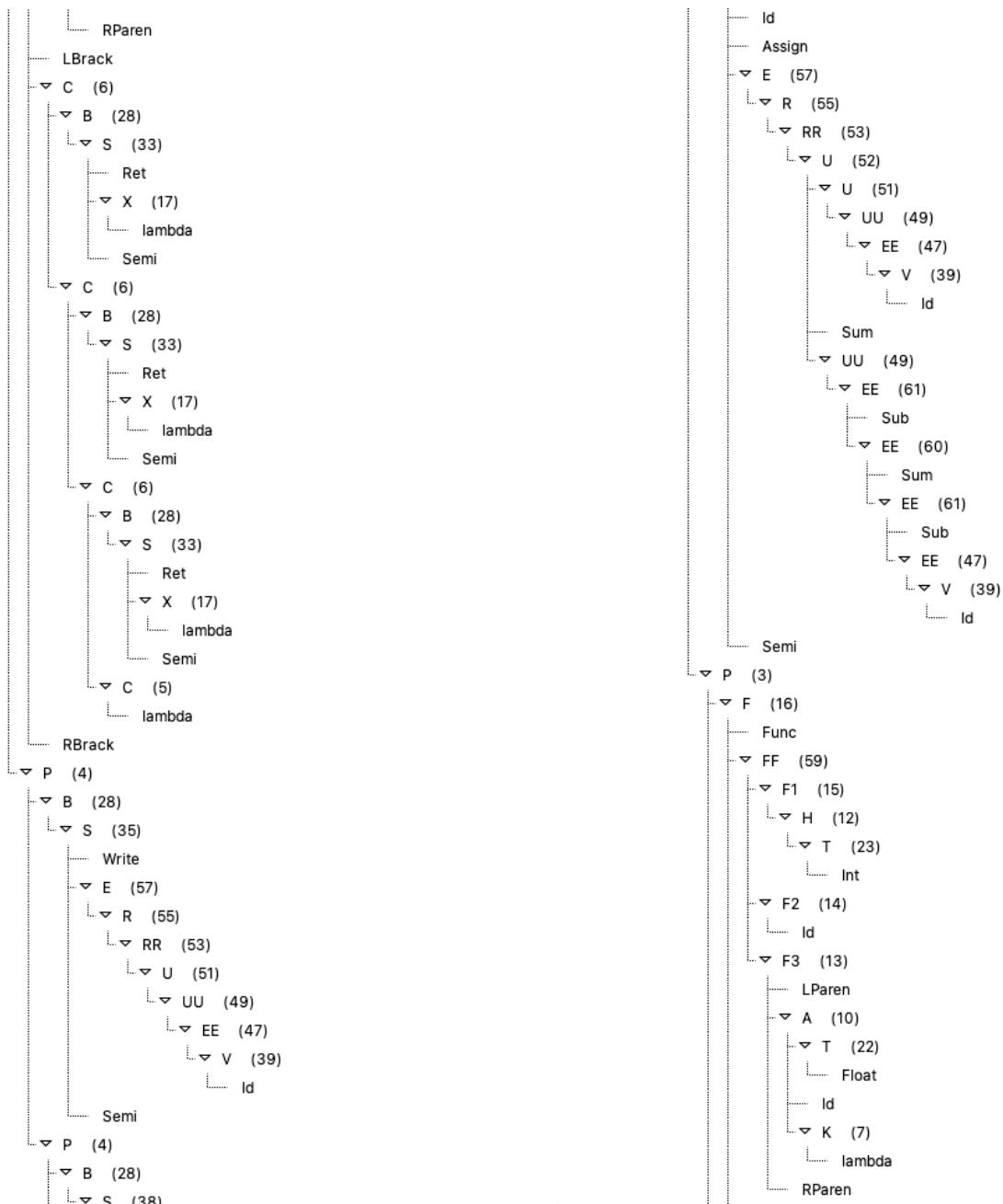
Tabla de símbolos:

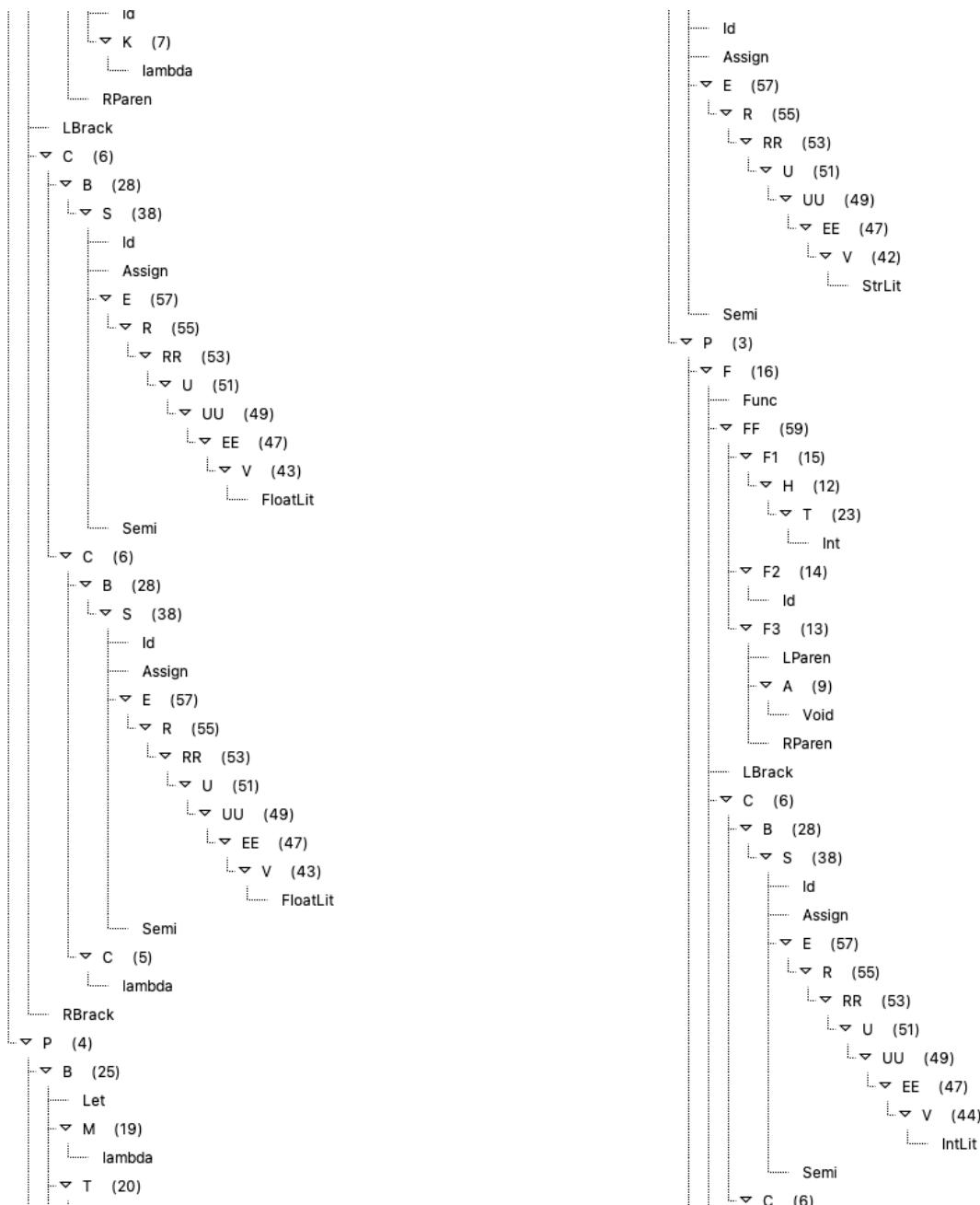
```

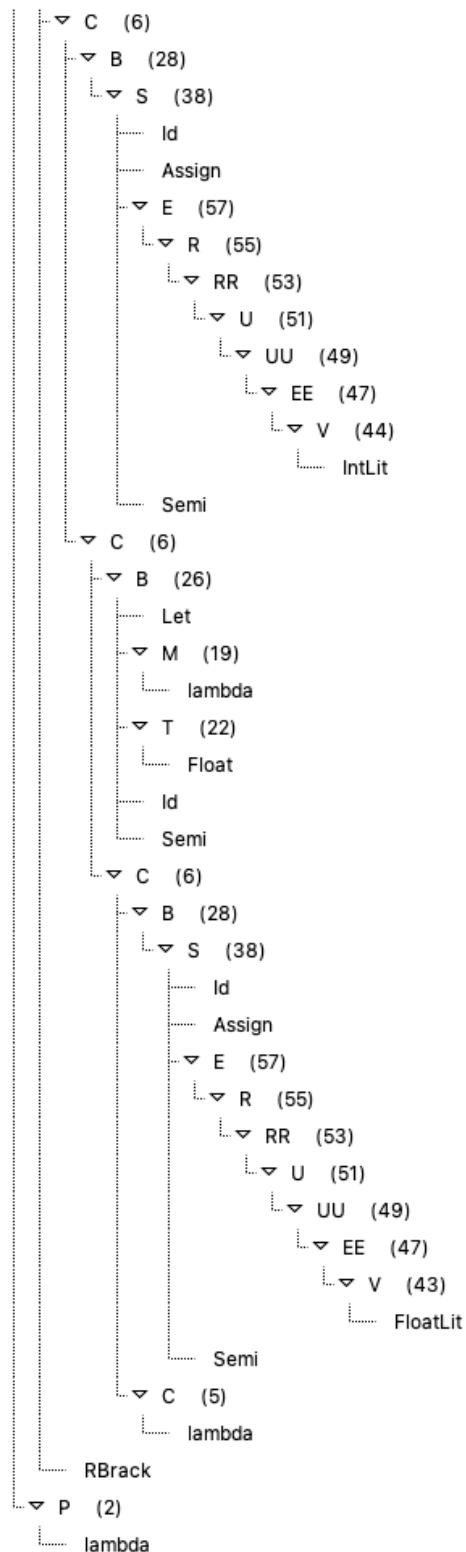
1 noreturn table #1:
2
3 noop table #2:
4
5 other_scope table #3:
6 * lexema: 'var'
7     + tipo: 'float'
8     + despl: 0
9
10 last_try table #4:
11 * lexema: 'implicit'
12     + tipo: 'float'
13     + despl: 0
14
15 global table #0:
16 * lexema: 'var'
17     + tipo: 'int'
18     + despl: 0
19 * lexema: 'a'
20     + tipo: 'int'
21     + despl: 1
22 * lexema: 'noreturn'
23     + etiqFuncion: '__tag_noreturn__'
24     + tipo: 'function'
25     + tipoRetorno: 'int'
26     + numParam: 0
27 * lexema: 'noop'
28     + etiqFuncion: '__tag_noop__'
29     + tipo: 'function'
30     + tipoRetorno: 'void'
31     + numParam: 0
32 * lexema: 'implicit'
33     + tipo: 'int'
34     + despl: 2
35 * lexema: 'other_scope'
36     + etiqFuncion: '__tag_other_scope__'
37     + tipo: 'function'
38     + tipoRetorno: 'int'
39     + numParam: 1
40     + tipoParam01: 'float'
41 * lexema: 'mystr'
42     + tipo: 'string'
43     + despl: 3
44 * lexema: 'last_try'
45     + etiqFuncion: '__tag_last_try__'
46     + tipo: 'function'
47     + tipoRetorno: 'int'
48     + numParam: 0

```









A.2 Casos Incorrectos

lexer.js Fichero con errores mayoritariamente léxicos

```

1 /* unterminated declaration */
2 let int = 2;
3
4 $ /* illegal */
5
6 let int max_plus_one = 32768;
7 /* unfinished variables */
8 let float x = 478234.;
9 let string rep = "last one\q
10
11 let int a = 3;
12 /*** this one is not ** ***** ***/

```

Diagnósticos:

```

doc-tests/incorrect/lexer.js:4:15: error: illegal character '$' in program
  4 | $ /* illegal */
    | ^ here

doc-tests/incorrect/lexer.js:2:12: error: expected an identifier, found '='
  2 | let int = 2;
    |   ^ expected an identifier
  -> help: insert an identifier after 'int'
  2 | let int foo = 2;
    |   +++
    |
doc-tests/incorrect/lexer.js:6:29: error: integer literal out of range for 16-byte type
  6 | let int max_plus_one = 32768;
    |   ^^^^^^ maximum is 32767

doc-tests/incorrect/lexer.js:8:22: error: missing decimal part after '.' in float literal
  8 | let float x = 478234.;
    |   ^^^^^^ expected digit after '.'
  -> help: add a decimal part
  8 | let float x = 478234.0;
    |   +
    |
doc-tests/incorrect/lexer.js:9:28: warning: unknown escape sequence '\q'
  9 | let string rep = "last one\q
    |   ^ interpreted as \\q
    |
doc-tests/incorrect/lexer.js:9:28: error: missing terminating character '"' on string literal
  9 | let string rep = "last one\q
    |   ^^^^^^^^^ here
  -> help: close the string literal
  9 | let string rep = "last one\q"
    |   +
    |
doc-tests/incorrect/lexer.js:11:14: error: missing ';' at the end of a statement
  9 | let string rep = "last one\q
    |   ----- after this
    |
  11 | let int a = 3;
    |   ^^^ expected ;
  -> help: insert ';' after '"last one\q"'
  9 | let string rep = "last one\q";
    |   +
    |
doc-tests/incorrect/lexer.js:12:37: error: unterminated block comment
  12 | /*** this one is not ** ***** ***/
    |   ^ started here

doc-tests/incorrect/lexer.js: warning: file generated 1 warning
doc-tests/incorrect/lexer.js: error: couldn't process file due to 7 previous errors

```

parser.javascript Fichero con errores mayoritariamente sintácticos

```

1 /* declaraciones incorrectas */
2 let y = 10;
3 let let int x = 100;
4 let int z = 2 + ;
5
6 /* funciones incorrectas */
7 function int func(void) {
8     write(c
9 }
10 function void if() {}
11 function chao(int a) {}
12
13 void wellwell(void) {}
14
15 /* mas sustituciones y eliminaciones */
16 a == 4;
17
18 b = 4,
19
20 function int chachi(int) { }
21 chachi(hola,);
22

```

Diagnósticos:

```

doc-tests/incorrect/parser.javascript:2:11: error: missing type in a variable declaration
2 | let y = 10;
  |   ^ expected type
--> help: add the missing type
2 | let type y = 10;
  |   +++
doc-tests/incorrect/parser.javascript:3:20: error: expected a type, found `let`
3 | let let int x = 100;
  |   ^^^ expected a type
--> help: remove the unnecessary `let`
3 - let let int x = 100;
3 + let int x = 100;
  |
doc-tests/incorrect/parser.javascript:4:18: error: expected an expression, found `;`
4 | let int z = 2 + ;
  |   ^ expected an expression
--> help: insert an expression before `;`
4 | let int z = 2 + expression;
  |   ++++++
doc-tests/incorrect/parser.javascript:9:1: error: mismatched closing delimiter `}`
7 | function int func(void) {
  |   - closing delimiter possibly meant for this
8 |     write(c
  |       ^ unclosed delimiter
9 |   }
  |   ^ mismatched closing delimiter
--> help: insert `)` and `;` after `c`
8 |     write(c);
  |       ++

```

```
doc-tests/incorrect/parser.javascript:10:21: error: keyword `if` used as an identifier
10 |     function void if() {}
      ^ expected an identifier
--> help: change the name to use it as an identifier
10 - function void if() {}
10 + function void my_if() {}

doc-tests/incorrect/parser.javascript:11:23: error: missing return type in a function declaration
11 |     function chao(int a) {}
      ^^^^^ expected return type
--> help: add a return type
11 |     function type chao(int a) {}
      +++
--> help: insert `function` before `void`
11 |     function chao(int a) {}           - after this
:
13 |     void wellwell(void) {}
      ^^^ expected a statement or a function
--> help: insert `function` before `void`
13 |     function void wellwell(void) {}
      ++++++
doc-tests/incorrect/parser.javascript:16:7: error: expected `(` or an assignment, found `==`
16 |     a == 4;
      ^^ expected `(` or an assignment
--> help: replace `==` by an assignment
16 - a == 4;
16 + a = 4;
:

doc-tests/incorrect/parser.javascript:18:6: error: expected `;` or a binary operator, found `,`
18 |     b = 4,
      ^ expected `;` or a binary operator
--> help: replace `,` by `;`
18 - b = 4;
18 + b = 4;
:

doc-tests/incorrect/parser.javascript:22:14: error: trailing comma in a function call
22 |     chachi(hola,);
      ^ here
--> help: remove the trailing comma
22 - chachi(hola,);
22 + chachi(hola);

doc-tests/incorrect/parser.javascript: error: couldn't process file due to 10 previous errors
```

sem.javascript Fichero con errores mayoritariamente semánticos

```

1 asdf &= true;
2
3 extra = "hi";
4 let float asdf = 3 + first;
5 first = "hello";
6
7 read sec;
8 sec = 5.0;
9
10 let boolean hey = true && third;
11 let boolean a = false;
12
13 function int oper(int a, float b, string c) {
14     let int hey = 3;
15     hey = hey + 4;
16
17     a = 5;
18     let int a = 5;
19
20     yo = oper(1, 2, 3, 4);
21     read troll;
22
23     let float oper = 3.0;
24
25     oper(1, 1.0, "1");
26
27     oper = 2.0;
28
29     return oper(c, hey, a);
30 }
```

Diagnósticos:

```

doc-tests/incorrect/sem.javascript:1:1: error: mismatched types
1 | asdf &= true;
| ^^^^ -- expected `boolean` due to this logical operator
| |
| expected `boolean`, found `int`

doc-tests/incorrect/sem.javascript:3:13: error: mismatched types
3 | extra = "hi";
| ^^^^ expected `int`, found `string`
| |
| expected `int` due to this implicit declaration

doc-tests/incorrect/sem.javascript:4:27: error: identifier already defined
1 | asdf &= true;
| ----- previously defined here
|
4 | let float asdf = 3 + first;
|      ^^^^ later redefined
|
--> help: choose a different name
4 - let float asdf = 3 + first;
4 + let float other_asdf = 3 + first;
| |

doc-tests/incorrect/sem.javascript:6:16: error: mismatched types
4 | let float asdf = 3 + first;
|      ----- expected `int` due to this implicit declaration
|
6 | first = "hello";
|      ^^^^^^ expected `int`, found `string`

doc-tests/incorrect/sem.javascript:9:10: error: mismatched types
7 | read sec;
|      --- expected `int` due to this implicit declaration
|
9 | sec = 5.0;
|      ^^^ expected `int`, found `float`
```

```
doc-tests/incorrect/sem.javascript:11:32: error: mismatched types
11 |   let boolean hey = true && third;
      |          ^^^^^^ expected `boolean`, found `int`
      |          |
      |          expected `boolean` due to this logical operator

doc-tests/incorrect/sem.javascript:20:22: error: identifier already defined
15 |   function int oper(int a, float b, string c) {
      |           - previously defined here
      :
20 |     let int a = 5;
      |           ^ later redefined
--> help: choose a different name
20 -     let int a = 5;
20 +     let int other_a = 5;

doc-tests/incorrect/sem.javascript:22:30: error: function called with wrong number of arguments
15 |   function int oper(int a, float b, string c) {
      |           ---- expected due to it's parameter list
      :
22 |     yo = oper(1, 2, 3, 4);
      |           ^^^^^^^^^^^ expected 3 arguments, found 4

doc-tests/incorrect/sem.javascript:27:26: error: mismatched types
25 |     let float oper = 3.0;
      |           ----- found `float` due to this declaration
      :
27 |     oper(1, 1.0, "1");
      |           ^^^^^----- variables can't be called
      |           |
      |           expected `function`, found `float`

doc-tests/incorrect/sem.javascript:31:31: error: mismatched types
25 |     let float oper = 3.0;
      |           ----- found `float` due to this declaration
      :
31 |     return oper(c, hey, a);
      |           ^^^^^----- variables can't be called
      |           |
      |           expected `function`, found `float`
--> help: try removing the call
31 -     return oper(c, hey, a);
31 +     return oper;
      |

doc-tests/incorrect/sem.javascript: error: couldn't process file due to 10 previous errors
```

lex_plus_parse.javascript Fichero con errores mayoritariamente léxicos y sintácticos

```
1 if (condition) write "no parenthesis";
2
3 let int int a = 5.;
4
5 do { }
6
7 let int a = 4;
8
9 function void(void) {return;}
10
11 do {
12     return 5;
13 } while(5);
14
15 let string a = "Hello^["
```

Diagnósticos:

```
doc-tests/incorrect/lex_plus_parse.javascript:3:5: error: mismatched types
  3 | if (condition) write "no parenthesis";
    | ^^^^^^^^^ expected `boolean`, found `int`  

doc-tests/incorrect/lex_plus_parse.javascript:5:19: error: expected an identifier, found `int`
  5 | let int int a = 5.;  

    | ^^^ expected an identifier  

    |--> help: remove the unnecessary `int`  

  5 - let int int a = 5.;  

  5 + let int a = 5.;  

  |  

doc-tests/incorrect/lex_plus_parse.javascript:5:19: error: missing decimal part after `.` in float literal
  5 | let int int a = 5.;  

    | ^^^ expected digit after `.`  

    |--> help: add a decimal part  

  5 | let int int a = 5.0;  

    | +  

doc-tests/incorrect/lex_plus_parse.javascript:5:19: error: mismatched types
  5 | let int int a = 5.;  

    | --- ^^^ expected `int`, found `float`  

    | |  

    | | expected `int` because of this
```

```
doc-tests/incorrect/lex_plus_parse.javascript:9:14: error: expected `while`, found `let`
 7 | do { }
    - after this
  :
 9 | let int a = 4;
  ^^^ expected `while`
--> help: insert `while`, `(`, an expression, `)` and `;` after `}`
 7 | do { } while (expression);
    +++++ ++++++++
doc-tests/incorrect/lex_plus_parse.javascript:9:14: error: identifier already defined
 5 | let int int a = 5;
    - previously defined here
  :
 9 | let int a = 4;
    ^ later redefined
--> help: choose a different name
 9 - let int a = 4;
 9 + let int other_a = 4;
  :
doc-tests/incorrect/lex_plus_parse.javascript:11:29: error: expected an identifier, found `(`
 11 | function void(void) {return;}
    ^ expected an identifier
--> help: insert an identifier after `void`
 11 | function void foo(void) {return;}
    +++
doc-tests/incorrect/lex_plus_parse.javascript:15:11: error: mismatched types
 15 | } while(5);
    ^ expected `boolean`, found `int`
doc-tests/incorrect/lex_plus_parse.javascript:17:22: error: malformed string literal, contains control character `\u{1b}`
 17 | let string a = "Hello\u{1b}
    ^^^^^^ help: remove this character
doc-tests/incorrect/lex_plus_parse.javascript:17:22: error: missing `;` at the end of a statement
 17 | let string a = "Hello
    ^ expected `;`
--> help: insert `;` after `Hello`
 17 | let string a = "Hello\u{1b};
    +
doc-tests/incorrect/lex_plus_parse.javascript: error: couldn't process file due to 10 previous errors
```

challenge.javascript Fichero con todo tipo de errores

```

1 let lol_sum = oper();
2 let string troll = "3o4";
3 a = true?,
4
5 function void call2(void) {
6     return;
7     return false;
8     return;
9     return 1 == blop;
10 }
11 let int a = 0.;
12
13 if (cond) write "hello";
14 function void noop(void) { return; return; return; }
15 doom = noop(pram1, pram2, pram3);
16
17 function int repeat(int x, float x) {
18     x = 5.0;
19     return repeat(x);
20     let int hola = a;
21 }
22
23 do { return doom; } while (!true);
24 /*
```

Diagnósticos:

```

doc-tests/incorrect/challenge.javascript:1:5: error: missing type in a variable declaration
1 | let lol_sum = oper();
| ^^^^^^ expected type
--> help: add the missing type
1 | let type lol_sum = oper();
| +++
doc-tests/incorrect/challenge.javascript:1:15: error: call to undefined function `oper`
1 | let lol_sum = oper();
| ^^^ help: try defining it first
doc-tests/incorrect/challenge.javascript:4:10: error: illegal character `?` in program
4 | a = true?;           ^ here
doc-tests/incorrect/challenge.javascript:4:10: error: expected `;` or a binary operator, found `,`
4 | a = true?;           ^ expected `;` or a binary operator
--> help: replace `,` by `;`
4 - a = true?;
4 + a = true?;
```

```

doc-tests/incorrect/challenge.javascript:8:21: error: unexpected return type in a void function
6 | function void call2(void) {
|     ----- unexpected due to its return type
|     :
8 |     return false;
|         ^^^^^^ unexpected `boolean`
```

```

10 |     return 1 == blop;
|         ^^^^^^^^^^ unexpected `boolean`
--> help: change the return type to `boolean`
6 - function void call2(void) {
6 + function boolean call2(void) {
```

```

doc-tests/incorrect/challenge.javascript:13:15: error: missing decimal part after `.` in float literal
13 | let int a = 0.;
|     ^^^ expected digit after `.`
--> help: add a decimal part
13 | let int a = 0.0;
|     +
```

```
doc-tests/incorrect/challenge.javascript:13:15: error: identifier already defined
  4 | a = true?;
    - previously defined here
  :
13 | let int a = 0.;
    ^ later redefined
--> help: choose a different name
13 - let int a = 0.;
13 + let int other_a = 0.;

doc-tests/incorrect/challenge.javascript:15:24: error: mismatched types
15 | if (cond) write "Hello";
    ^^^^ expected `boolean`, found `int`

doc-tests/incorrect/challenge.javascript:19:33: error: function called with wrong number of arguments
17 | function void noop(void) { return; return; return; }
    ----- expected due to it's parameter list
  :
19 | doom = noop(pram1, pram2, pram3);
    ^^^^^^^^^^^^^^^^^ expected no arguments, found 3

doc-tests/incorrect/challenge.javascript:21:37: error: identifier already defined
21 | function int repeat(int x, float x) {
    -                                     ^ later redefined
    |                                     |
    | previously defined here
--> help: choose a different name
21 - function int repeat(int x, float x) {
21 + function int repeat(int x, float other_x) {

doc-tests/incorrect/challenge.javascript:22:16: error: mismatched types
21 | function int repeat(int x, float x) {
    -                                     --- expected `int` due to it's declaration
22 |     x = 5.0;
    ^^^ expected `int`, found `float`

doc-tests/incorrect/challenge.javascript:23:25: error: function called with wrong number of arguments
21 | function int repeat(int x, float x) {
    ----- expected due to it's parameter list
  :
23 |     return repeat(x);
    ^ expected 2 arguments, found 1

doc-tests/incorrect/challenge.javascript:29:2: error: unterminated block comment
29 | /* started here

doc-tests/incorrect/challenge.javascript:27:34: error: return statement outside of a function body
27 | do { return doom; } while (!true);
    ^^^^^^^^^^^^^ cannot be used outside a function

doc-tests/incorrect/challenge.javascript: error: couldn't process file due to 14 previous errors
```