13 de diciembre de 2025

Procesador de MyJS: *jsp*

Memoria del Grupo 59

Andrés Súnico

Índice

1 Introducción

El desarrollo del procesador *jsp* se ha centrado en la experiencia del usuario (*UX*), priorizando tres aspectos clave: una gestión de errores sólida y clara, una interfaz de línea de comandos (*CLI*) intuitiva, y un rendimiento eficiente.

Por ello, se ha elegido *Rust* como el lenguaje de desarrollo. Ofrece una gestión de memoria eficiente, además de integrar *clap*, una de las mejores bibliotecas para desarollar aplicaciones *CLI*.

Gracias al uso del patrón de *inyección de dependencias* en todo el proyecto, el código fuente es altamente extensible y modular.

2 Información Adicional

El código fuente del procesador se puede encontrar en github.com/suuniqo, así como los tests y las dependencias del proyecto.

3 Opciones de la Práctica

Además de las opciones comunes a todos los grupos, se han implementado las opciones:

3.1 Específicas del grupo

- Comentarios de bloque (/* */)
- Cadenas con comillas dobles (" ")
- Sentencia repetitiva do-while
- Asignación con *y* lógico (*&=*)
- Análisis Sintáctico Ascendente

3.2 Adicionales

Para que el procesador esté más completo, se han implementado adicionalmente los operadores:

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

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

4 Diseño del Lexer

El Analizador Léxico o *Lexer* es uno de los 3 módulos principales del procesador.

Al ser la primera capa de procesamiento, es el encargado de manejar el fichero fuente y convertirlo en una lista de *tokens* para el Analizador Sintáctico.

4.1 Tokens

Con el fin de lograr un procesamiento eficiente, tanto en memoria como en complejidad, se han minimizado el número de *tokens* con atributos (tan sólo 4 de los 33 *tokens* usarán un atributo).

Cabe notar, además, que se ha decidido no hacer uso del *token* fin de fichero (*EOF*). Esto es porque el *Lexer* se ha implementado como un iterador de *tokens*, de modo que el final del flujo se detecta naturalmente cuando se consume el iterador.

Cuadro 1: Listado de *tokens*

Elemento	Código	Atributo
boolean	Bool	-
do	Do	-
float	Float	-
function	Func	-
if	If	-
int	Int	-
let	Let	-
read	Read	-
return	Ret	-
string	Str	-
void	Void	-
while	While	-
write	Write	-
constante real	FloatLit	Número
constante entera	IntLit	Número
Cadena	StrLit	Cadena
Identificador	Id	Posición
&=	AndAssign	-
=	Assign	-
,	Comma	-
;	Semi	-
(	LParen	-
)	RParen	-
{	LBrack	-
}	RBrack	-
Suma ($+$)	Sum	-
Por ($*$)	Mul	-
Y lógico (&&)	And	-
Negación (!)	Not	-
Menor ($<$)	Lt	-
Igual ($==$)	Eq	-
Menos ($-$)	Sub	-
Más ($+$)	Sum	-
false	False	-
true	True	-

4.2 Errores

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

El procesador genera mensajes claros con número de línea y columna, muestra la línea afectada y subraya en color la parte errónea.

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

El *Lexer* sólo genera un warning, *Invalid Escape Sequence*. Como se muestra en Acciones Semánticas, al detectar una secuencia de escape inválida no se descartara el *token* cadena, sino que se conserva literalmente (por ejemplo, la secuencia \q, se sustituye por esos dos mismos caracteres)[1].

Cuadro 2: Listado de errores del *Lexer*

Error	Severidad
Carácter inválido	*error*
Comentario inacabado	*error*
Cadena inacabada	*error*
Cadena malformada	*error*
Overflow de Cadena	*error*
Overflow de Entero	*error*
Overflow de Real	*error*
Formato de Real Inválido	*error*
Secuencia de Escape Inválida	*warning*

4.3 Gramática

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

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

P se compone de la regla del axioma:

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

Y del resto de reglas:

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

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

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

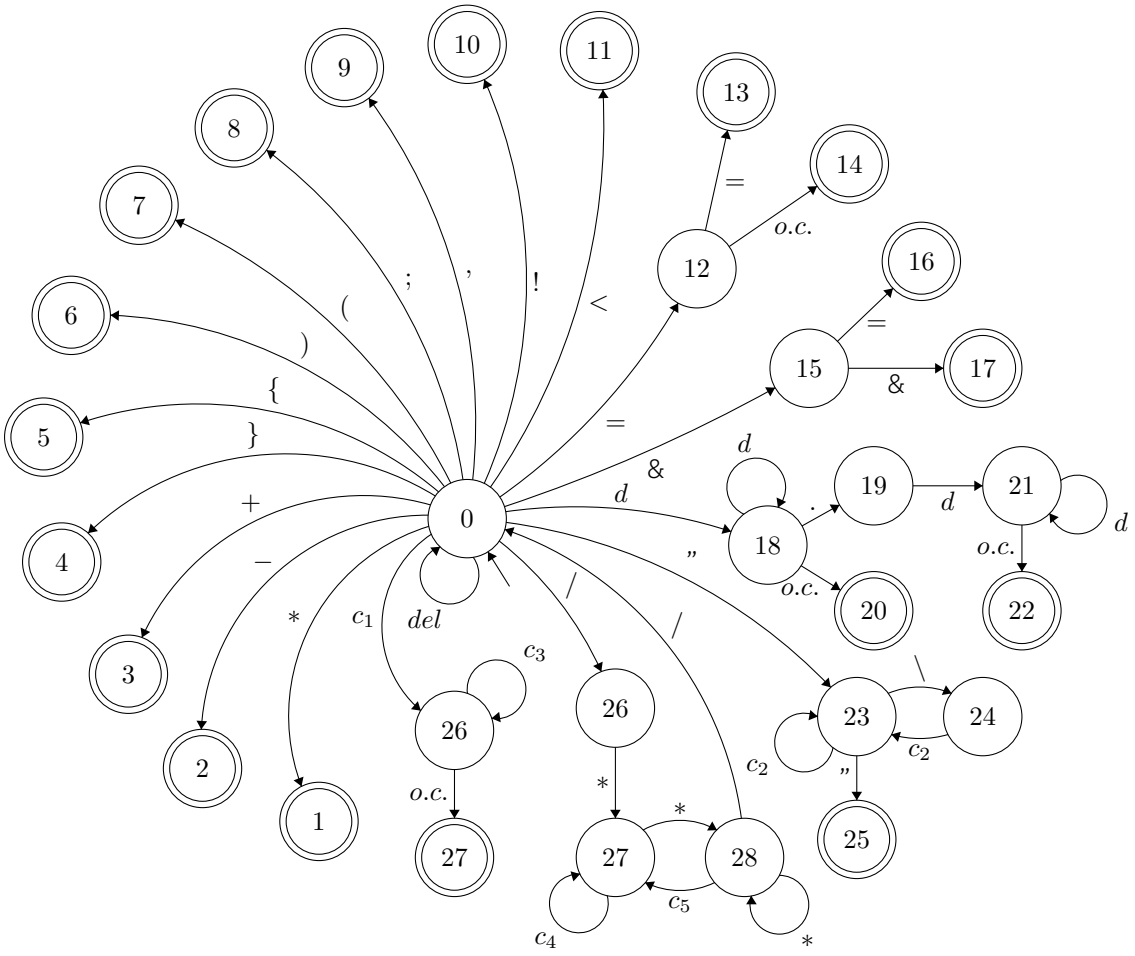
[1] Se ha elegido este comportamiento para que el procesador sea fiel a la documentación oficial de *ECMAScript*.
[2] La definición de delimitador se toma de la documentación oficial de *ECMAScript*.

4.4 Autómata

A continuación se muestra el autómata finito determinista 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

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

4.5.2 Errores

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

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

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

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

INV_FLOAT_FMT Si en el estado 19 no se puede transitar

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

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

```
error("Unterminated␣comment")
```

INV_CHAR Ante cualquier error no manejado en el resto de acciones

```
error("Illegal␣character")
```

4.5.3 Generación Directa

GEN_MUL En la transición 0:1

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

GEN_SUB En la transición 0:2

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

GEN_SUM En la transición 0:3

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

GEN_RBRACK En la transición 0:4

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

GEN_LBRACK En la transición 0:5

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

GEN_RPAREN En la transición 0:6

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

GEN_LPAREN En la transición 0:7

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

GEN_SEMI En la transición 0:8

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

GEN_COMMA En la transición 0:9

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

GEN_NOT En la transición 0:10

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

GEN_LT En la transición 0:11

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

GEN_EQ En la transición 12:13

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

GEN_ASSIGN En la transición 12:14

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

GEN_ANDASSIGN En la transición 15:16

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

GEN_AND En la transición 15:17

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

4.5.4 Generación de Números

INIT_NUM En la transición 0:18

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

INIT_DEC En la transición 20:21

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

GEN_DEC En la transición 21:22

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

ADD_INTDIG En la transición 18:18

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

ADD_DECDIG En las transiciones 21:21

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

GEN_INT En la transición 18:20

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

4.5.5 Generación de Cadenas e Identificadores

INIT_STR En la transición 0:23

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

ADD_CHAR_STR En la transición 23:23

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

ADD_CHAR_ID En la transición 0:26, 26:26

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

ADD_ESCSEQ En la transición 24:23

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

INIT_ID En la transición 0:26

```
lex := ""
```

GEN_STR En la transición 23:25

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

GEN_ID En la transición 26:27

```
code := search_keyword(lex)

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

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

5 Diseño de la Tabla de Símbolos

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

5.1 Estructura y Organización

5.1.1 Entradas

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

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

Lexema: Nombre de la variable.
Tipo: Tipo de la variable.
Desplazamiento: Desplazamiento en memoria relativo a su ámbito.

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

Lexema: Nombre de la función.
Tipo Retorno: Tipo que devuelve la función, pudiendo ser *Void*.
Tipos Argumentos: Lista de los tipos de los parámetros en orden.
Etiqueta: Etiqueta que se usará para navegar a la función en el código ensamblador.

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

5.1.2 Ámbitos

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

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

6 Diseño del Parser

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 raiz, 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:

$T = \{$Todo *token* definido por el *Lexer*$\}$
$N = \{E, R, RR, U, UU, EE, V, S, L, Q, X, B, T, M, F, F1, F2, F3, H, A, K, C, P\}$

R se compone de:

$P \rightarrow BP \mid FP \mid \lambda$
$C \rightarrow BC \mid \lambda$
$F \rightarrow$ Func $F1F2F3$ LBrack C RBrack
$F1 \rightarrow H$
$F2 \rightarrow$ Id
$F3 \rightarrow$ LParen A RParen
$H \rightarrow T \mid$ Void
$A \rightarrow T$ Id $K \mid$ Void
$K \rightarrow$ Comma T Id $K \mid \lambda$
$B \rightarrow$ If LParen E RParen $S \mid$ Do LBrack C RBrack While LParen E RParen Semi
$B \rightarrow S \mid$ Let MT Id Semi $\mid$ Let MT Id Assign E Semi
$M \rightarrow \lambda$
$T \rightarrow$ Int $\mid$ Float $\mid$ Bool $\mid$ Str
$S \rightarrow$ Write E Semi $\mid$ Read Id Semi $\mid$ Ret X Semi
$S \rightarrow$ Id Assign E Semi $\mid$ Id AndAssign E Semi $\mid$ Id LParen L RParen Semi
$L \rightarrow EQ \mid \lambda$
$Q \rightarrow$ Comma $EQ \mid \lambda$
$X \rightarrow E \mid \lambda$
$E \rightarrow E$ And $R \mid R$
$R \rightarrow R$ Eq $RR \mid RR$
$RR \rightarrow RR$ Lt $U \mid U$
$U \rightarrow U$ Sum $UU \mid UU$
$UU \rightarrow UU$ Mul $EE \mid EE$
$EE \rightarrow$ Not $EE \mid$ Sub $EE \mid$ Sum $EE \mid V$
$V \rightarrow$ Id LParen L RParen $\mid$ LParen E RParep $\mid$ IntLit $\mid$ FloatLit $\mid$ StrLit $\mid$ True $\mid$ False $\mid$ 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.

Está 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	r27	r27							r27	r27	r27	r27	r27						r27							r27									r27
9														s30	s29	s32	s33	s31	s28				r16	s34				s42	s43			s36			
10																			s44																
11														s30	s29	s32	s33	s31	s28					s34				s42	s43			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				s42	s43			s36			
26																							r17								s55				
27																							s56												
28																						r38	r38	s57	r38			r38		r38	r38		r38	r38	
29																						r39	r39		r39			r39		r39	r39		r39	r39	
30																						r40	r40		r40			r40		r40	r40		r40	r40	
31																						r41	r41		r41			r41		r41	r41		r41	r41	
32																						r42	r42		r42			r42		r42	r42		r42	r42	
33																						r43	r43		r43			r43		r43	r43		r43	r43	
34														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
35																						r46	r46		r46			r46		r46	r46		r46	r46	
36														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
37																						r48	r48		r48			r48		r48	r48		r48	r48	
38																						r50	r50		r50			r50		s60	r50		r50	r50	
39																						r52	r52		r52			s61			r52		r52	r52	
40																						r54	r54		r54						r54		s62	r54	
41																						r56	r56		r56						r56			s63	
42														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
43														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
44																							s66												
45																							s67								s55				
46														s30	s29	s32	s33	s31	s28					s34	r30			s42	s43			s36			
47														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
48														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			
49																								r13											
50																								s72											
51	s7	s5							s6		s9	s10	s11						s12							r4									
52																										s75									
53																			s76																
54																									s77						s55				
55														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36			

56	r32	r32							r32	r32	r32	r32	r32						r32								r32											r32
57														s30	s29	s32	s33	s31	s28					s34	r30			s42	s43			s36					
58																									s80						s55						
59																						r47	r47		r47			r47		r47	r47		r47	r47		
60														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
61														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
62														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
63														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
64																						r58	r58		r58			r58		r58	r58		r58	r58		
65																						r59	r59		r59			r59		r59	r59		r59	r59		
66	r33	r33							r33	r33	r33	r33	r33						r33								r33											r33
67	r34	r34							r34	r34	r34	r34	r34						r34								r34											r34
68																						s85			r28						s55						
69																									s87												
70																							s88								s55						
71																							s89								s55						
72				s22	s21	s19	s20	s90																														
73																										s93											
74																											r5										
75			s94																																		
76																				s95			s96														
77											s9	s10	s11						s12																		
78																						r57	r57		r57					r57			s63			
79																									s98												
80																						r44	r44		r44			r44		r44	r44		r44	r44		
81																						r49	r49		r49			r49		r49	r49		r49	r49		
82																						r51	r51		r51			r51		s60	r51		r51	r51		
83																						r53	r53		r53			s61			r53		r53	r53		
84																						r55	r55		r55						r55		s62	r55		
85														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
86																									r31												
87																							s100														
88	r36	r36							r36	r36	r36	r36	r36						r36								r36											r36
89	r37	r37							r37	r37	r37	r37	r37						r37								r37											r37
90																									r8												
91																			s101																		
92																									s102												
93	s7	s5							s6		s9	s10	s11						s12								r4										
94																								s104													
95														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
96	r25	r25							r25	r25	r25	r25	r25						r25								r25											r25
97	r26	r26							r26	r26	r26	r26	r26						r26								r26											r26
98																						r45	r45		r45			r45		r45	r45		r45	r45		
99																						s85			r28						s55						
100	r35	r35							r35	r35	r35	r35	r35						r35								r35											r35
101																						s107			r6												
102																										r12											
103																											s109										
104														s30	s29	s32	s33	s31	s28					s34				s42	s43			s36					
105																							s111								s55						
106																									r29												
107				s22	s21	s19	s20																															
108																									r9												
109	r15	r15							r15	r15	r15	r15	r15						r15																			r15
110																									s113						s55						
111	r24	r24							r24	r24	r24	r24	r24						r24								r24											r24
112																			s114																		
113																							s115														
114																						s107			r6												
115	r23	r23							r23	r23	r23	r23	r23						r23								r23											r23
116																									r7												

Cuadro 4: Tabla de Goto

	E	R	RR	U	UU	EE	V	S	L	Q	X	B	T	M	F	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			17				
5																							
6														24									
7																							
8																							
9	26	41	40	39	38	37	35				27												
10																							
11	45	41	40	39	38	37	35																
12																							
13																							
14																							
15																							
16																							
17																							
18																	50						
19																							
20																							
21																							
22																							
23								8				51										52	
24													53										
25	54	41	40	39	38	37	35																
26																							
27																							
28																							
29																							
30																							
31																							
32																							
33																							
34	58	41	40	39	38	37	35																
35																							
36						59	35																
37																							
38																							
39																							
40																							
41																							
42						64	35																
43						65	35																
44																							
45																							
46	68	41	40	39	38	37	35		69														
47	70	41	40	39	38	37	35																
48	71	41	40	39	38	37	35																
49																							
50																		73					
51								8				51										74	
52																							
53																							
54																							
55		78	40	39	38	37	35																
56																							
57	68	41	40	39	38	37	35		79														
58																							

59																							
60						81	35																
61					82	37	35																
62				83	38	37	35																
63			84	39	38	37	35																
64																							
65																							
66																							
67																							
68										86													
69																							
70																							
71																							
72													91							92			
73																							
74																							
75																							
76																							
77								97															
78																							
79																							
80																							
81																							
82																							
83																							
84																							
85	99	41	40	39	38	37	35																
86																							
87																							
88																							
89																							
90																							
91																							
92																							
93								8				51										103	
94																							
95	105	41	40	39	38	37	35																
96																							
97																							
98																							
99										106													
100																							
101																					108		
102																							
103																							
104	110	41	40	39	38	37	35																
105																							
106																							
107													112										
108																							
109																							
110																							
111																							
112																							
113																							
114																					116		
115																							
116																							

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

Por ahora sólo se ha lanza 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 5: Listado de errores del *Parser*

Error	Severidad
Token Inesperado	*error*

La lista de *tokens* esperados se obtiene realizando una busqueda 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

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.

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

    if (n == 1)
        return 1;

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

    do {
        let int c = a + b;
        a = b; b = c;

        n = n + -1;
    } while (!(n < 2));

    return b;
}
```

Fichero de *tokens*:

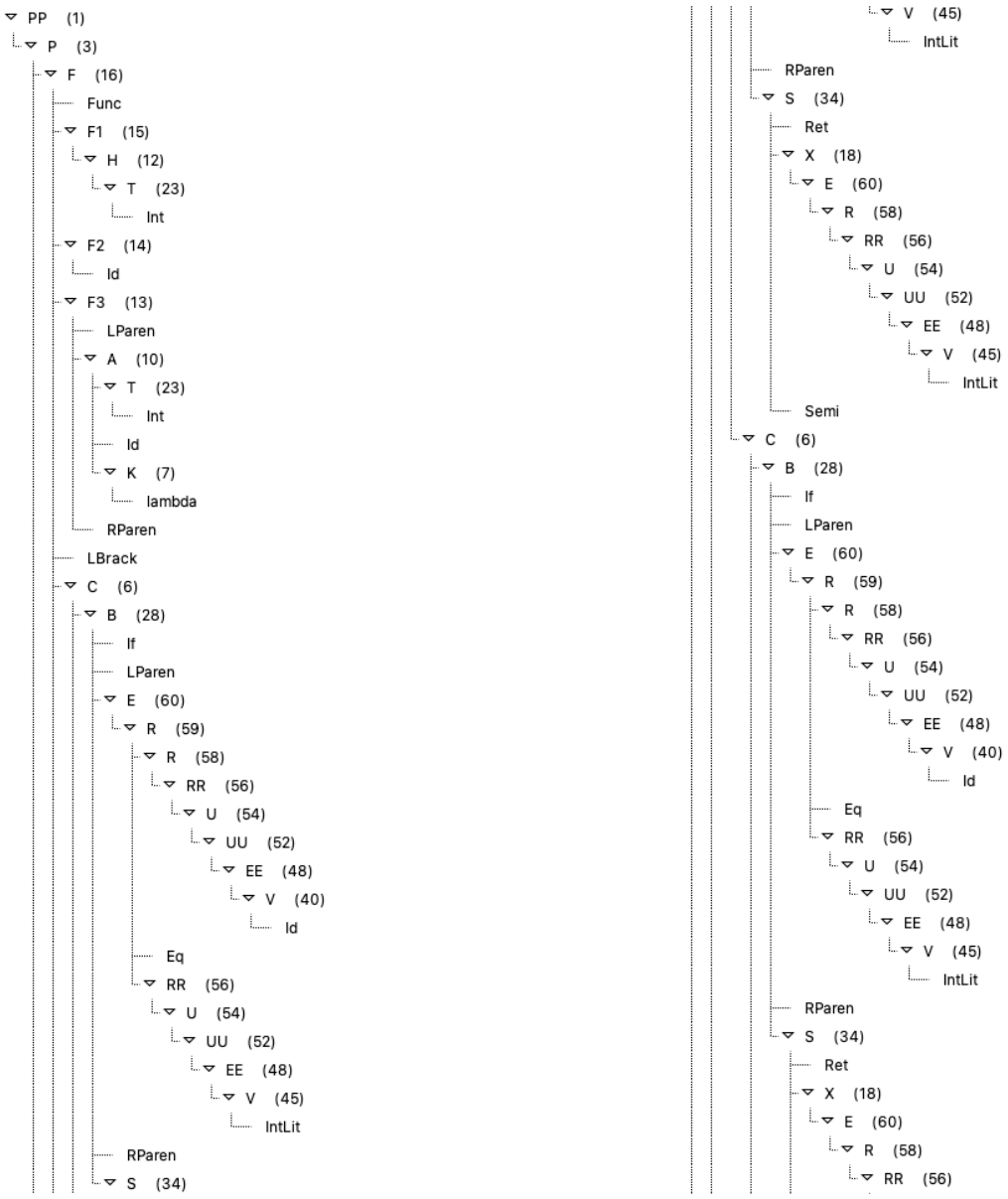
```
<Func, >
<Int, >
<Id, 0>
<LParen, >
<Int, >
<Id, 1>
<RParen, >
<LBrack, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 0>
<RParen, >
<Ret, >
<IntLit, 0>
<Semi, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 1>
<RParen, >
<Ret, >
<IntLit, 1>
<Semi, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<IntLit, 0>
<Semi, >
<Let, >
<Int, >
<Id, 3>
<Assign, >
<IntLit, 1>
<Semi, >
<Do, >
<LBrack, >
<Let, >
<Int, >
<Id, 4>
<Assign, >
<Id, 2>
<Sum, >
<Id, 3>
<Semi, >
<Id, 2>
<Assign, >
<Id, 3>
<Semi, >
<Id, 3>
<Assign, >
<Id, 4>
<Semi, >
<Id, 1>
<Assign, >
<Id, 1>
<Sum, >
<Sub, >
<IntLit, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Not, >
<LParen, >
<Id, 1>
<Lt, >
<IntLit, 2>
<RParen, >
<RParen, >
<Semi, >
<Ret, >
<Id, 3>
<Semi, >
<RBrack, >
```

Tabla de símbolos:

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

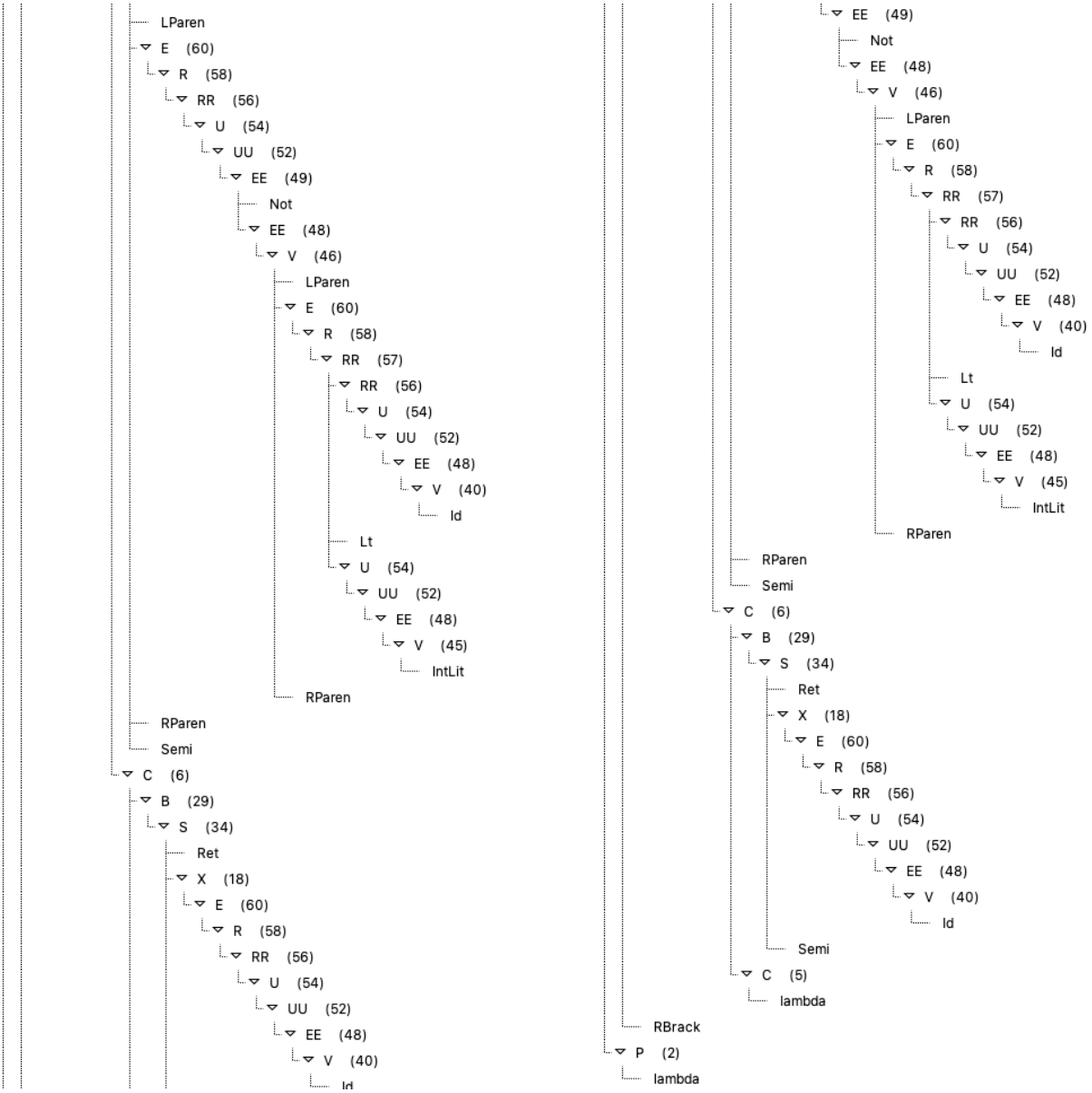
Parse:

```
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
```


PP (1)
P (3)
F (16)
Func
F1 (15)
H (12)
T (23)
Int
F2 (14)
Id
F3 (13)
LParen
A (10)
T (23)
Int
Id
K (7)
lambda
RParen
LBrack
C (6)
B (28)
If
LParen
E (60)
R (59)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Eq
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)
V (45)
IntLit
RParen
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (28)
If
LParen
E (60)
R (59)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Eq
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)


R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Eq
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (24)
Do
LBrack
C (6)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (55)


R (58)
RR (56)
U (55)
U (54)
UU (52)
EE (48)
V (40)
Id
Sum
UU (52)
EE (48)
V (40)
Id
Semi
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
EE (48)
V (40)
Id
Semi
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (55)
U (54)
UU (52)
EE (48)
V (40)
Id
Sum
UU (52)
EE (51)
Sub
EE (48)
V (45)
IntLit
Semi
C (5)
lambda
RBrack
While
LParen
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (49)
Not
EE (48)
V (46)
LParen
E (60)


LParen
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (49)
Not
EE (48)
V (46)
LParen
E (60)
R (58)
RR (57)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Lt
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
RParen
Semi
C (6)
B (29)
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
EE (49)
Not
EE (48)
V (46)
LParen
E (60)
R (58)
RR (57)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Lt
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
RParen
Semi
C (6)
B (29)
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
C (5)
lambda
RBrack
P (2)
lambda

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

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

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

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

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

Fichero de *tokens*:

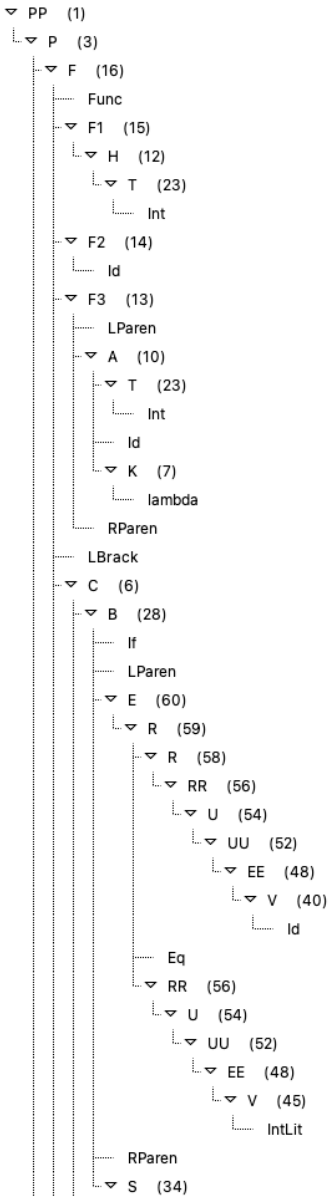
```
<Func, >
<Int, >
<Id, 0>
<LParen, >
<Int, >
<Id, 1>
<RParen, >
<LBrack, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 0>
<RParen, >
<Ret, >
<IntLit, 1>
<Semi, >
<If, >
<LParen, >
<Id, 1>
<Lt, >
<IntLit, 2>
<RParen, >
<Ret, >
<Id, 1>
<Semi, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<Id, 1>
<Semi, >
<Do, >
<LBrack, >
<Id, 1>
<Assign, >
<Id, 1>
<Sum, >
<Sub, >
<IntLit, 1>
<Semi, >
<Id, 2>
<Assign, >
<Id, 2>
<Mul, >
<Id, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Not, >
<LParen, >
<Id, 1>
<Lt, >
<IntLit, 2>
<RParen, >
<RParen, >
<Semi, >
<Ret, >
<Id, 2>
<Semi, >
<RBrack, >
<Let, >
<Int, >
<Id, 1>
<Assign, >
<IntLit, 0>
<Semi, >
<Read, >
<Id, 1>
<Semi, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<Id, 0>
<LParen, >
<Id, 1>
<RParen, >
<Semi, >
<Write, >
<LParen, >
<Id, 2>
<RParen, >
<Semi, >
```

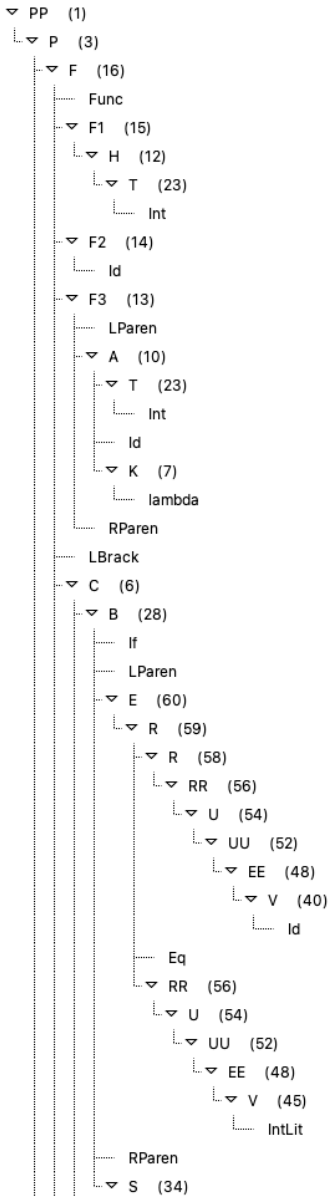
Tabla de símbolos:

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

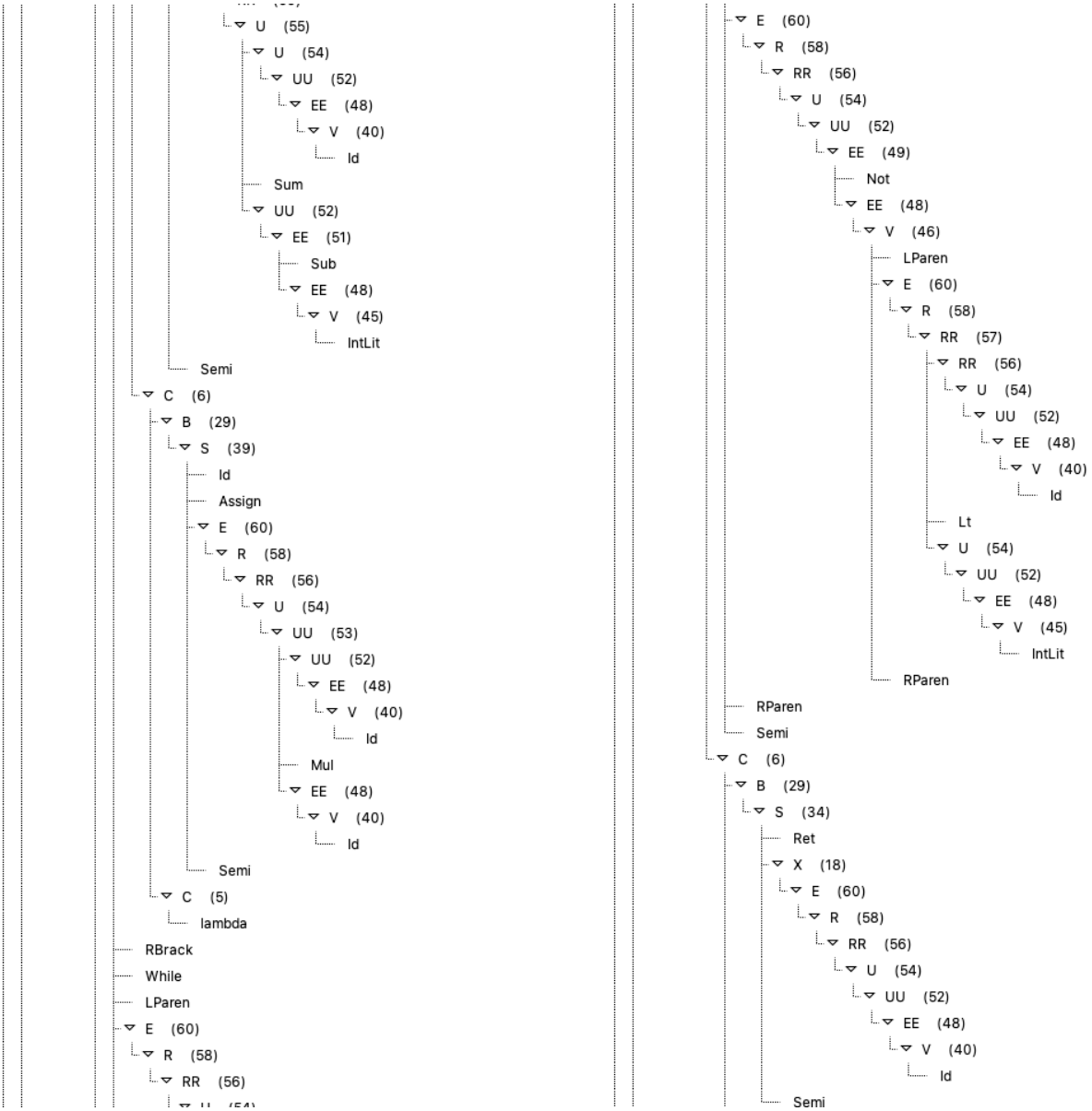
Parse:

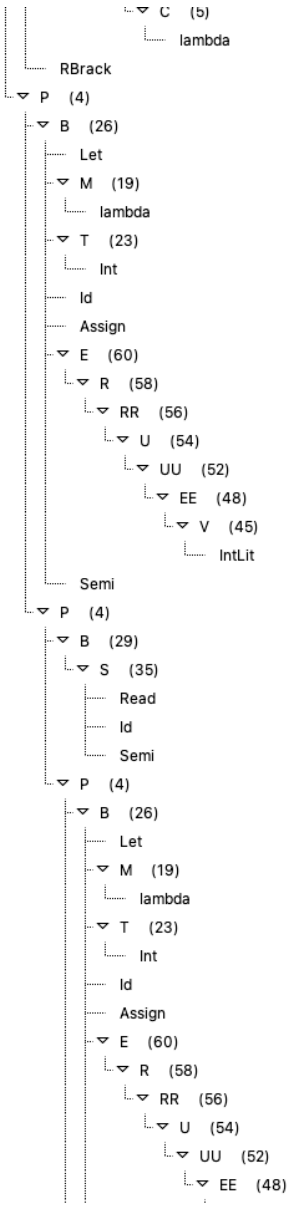
```
ascending 23 12 15 14 23 7 10 13 40 48 52 54 56 58 45 48 52 54 56 59 60 45 48 52 54 56 58 60 18 34
      28 40 48 52 54 56 45 48 52 54 57 58 60 40 48 52 54 56 58 60 18 34 28 19 23 40 48 52 54 56 58
     60 26 40 48 52 54 45 48 51 52 55 56 58 60 39 29 40 48 52 40 48 53 54 56 58 60 39 29 5 6 6 40
     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
```

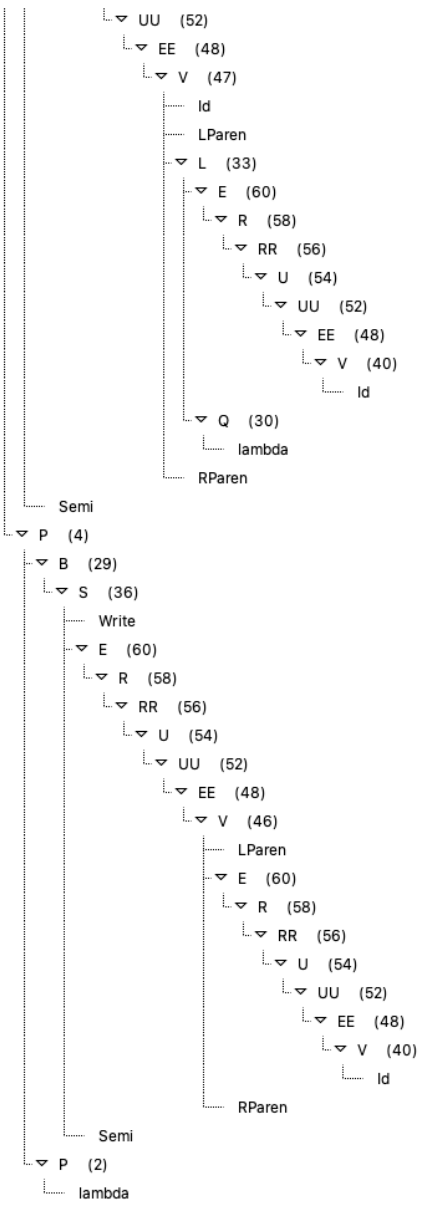

PP (1)
P (3)
F (16)
Func
F1 (15)
H (12)
T (23)
Int
F2 (14)
Id
F3 (13)
LParen
A (10)
T (23)
Int
Id
K (7)
lambda
RParen
LBrack
C (6)
B (28)
If
LParen
E (60)
R (59)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Eq
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)


PP (1)
P (3)
F (16)
Func
F1 (15)
H (12)
T (23)
Int
F2 (14)
Id
F3 (13)
LParen
A (10)
T (23)
Int
Id
K (7)
lambda
RParen
LBrack
C (6)
B (28)
If
LParen
E (60)
R (59)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Eq
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)


V (45)
IntLit
RParen
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (28)
If
LParen
E (60)
R (58)
RR (57)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Lt
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
C (6)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
C (6)
B (24)
Do
LBrack
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (55)
U (54)
UU (52)


U (55)
U (54)
UU (52)
EE (48)
V (40)
Id
Sum
UU (52)
EE (51)
Sub
EE (48)
V (45)
IntLit
Semi
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (53)
UU (52)
EE (48)
V (40)
Id
Mul
EE (48)
V (40)
Id
Semi
C (5)
lambda
RBrack
While
LParen
E (60)
R (58)
RR (56)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (49)
Not
EE (48)
V (46)
LParen
E (60)
R (58)
RR (57)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Lt
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
RParen
Semi
C (6)
B (29)
S (34)
Ret
X (18)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi


C (5)
lambda
RBrack
P (4)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
P (4)
B (29)
S (35)
Read
Id
Semi
P (4)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)


UU (52)
EE (48)
V (47)
Id
LParen
L (33)
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Q (30)
lambda
RParen
Semi
P (4)
B (29)
S (36)
Write
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (46)
LParen
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
RParen
Semi
P (2)
lambda

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

```
let int global=13;
function void nothing(void) { read global; do{global=global+-1;}while(!(global<0));}
function int main(float b, string d){let string aux="this is a str";
/*** this is a comment ***/ let float foo=1203.123; let int bar=2398;
/* semicolons */let int weird &= 3+-+-+-+-+-+-+-+-+5; let float oper=2.0000; let boolean bool=
    false;
let boolean george=true;george&=bool;}/*** / eof /***/
```

Fichero de *tokens*:

```
<Let, >
<Int, >
<Id, 0>
<Assign, >
<IntLit, 13>
<Semi, >
<Func, >
<Void, >
<Id, 1>
<LParen, >
<Void, >
<RParen, >
<LBrack, >
<Read, >
<Id, 0>
<Semi, >
<Do, >
<LBrack, >
<Id, 0>
<Assign, >
<Id, 0>
<Sum, >
<Sub, >
<IntLit, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Not, >
<LParen, >
<Id, 0>
<Lt, >
<IntLit, 0>
<RParen, >
<RParen, >
<Semi, >
<RBrack, >
<Func, >
<Int, >
<Id, 2>
<LParen, >
<Float, >
<Id, 3>
<Comma, >
<Str, >
<Id, 4>
<RParen, >
<LBrack, >
<Let, >
<Str, >
<Id, 5>
<Assign, >
<StrLit, "this is a str">
<Semi, >
<Let, >
<Float, >
<Id, 6>
<Assign, >
<FloatLit, 1203.23>
<Semi, >
<Let, >
<Int, >
<Id, 7>
<Assign, >
<IntLit, 2398>
<Semi, >
<Let, >
<Int, >
<Id, 8>
<AndAssign, >
<IntLit, 3>
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sub, >
<Sum, >
<Sum, >
<IntLit, 5>
<Semi, >
<Let, >
<Float, >
<Id, 9>
<Assign, >
<FloatLit, 2>
<Semi, >
<Let, >
<Bool, >
<Id, 10>
<Assign, >
<False, >
<Semi, >
<Let, >
<Bool, >
<Id, 11>
<Assign, >
<True, >
<Semi, >
<Id, 11>
<AndAssign, >
<Id, 10>
<Semi, >
<RBrack, >
```

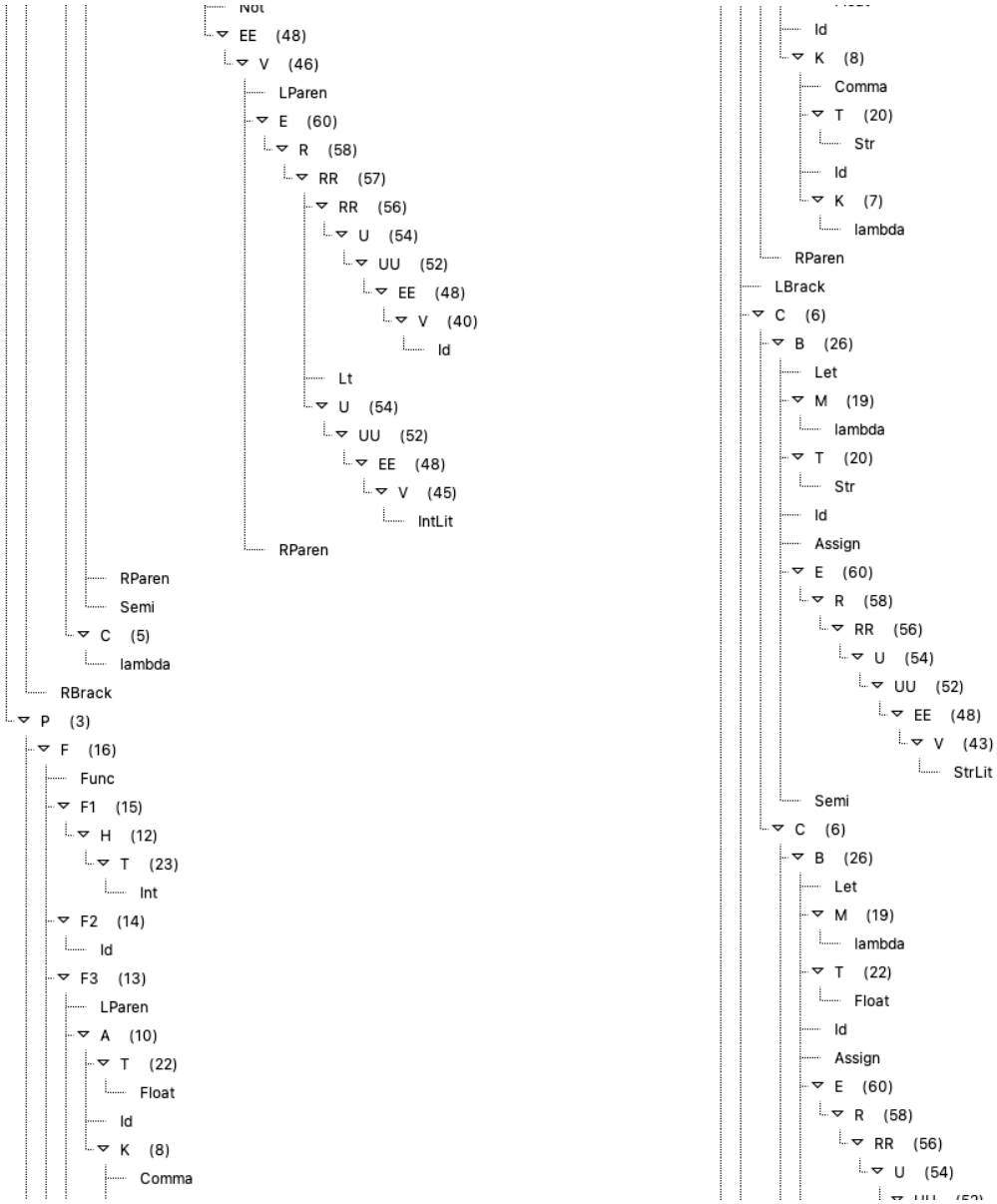
Tabla de símbolos:

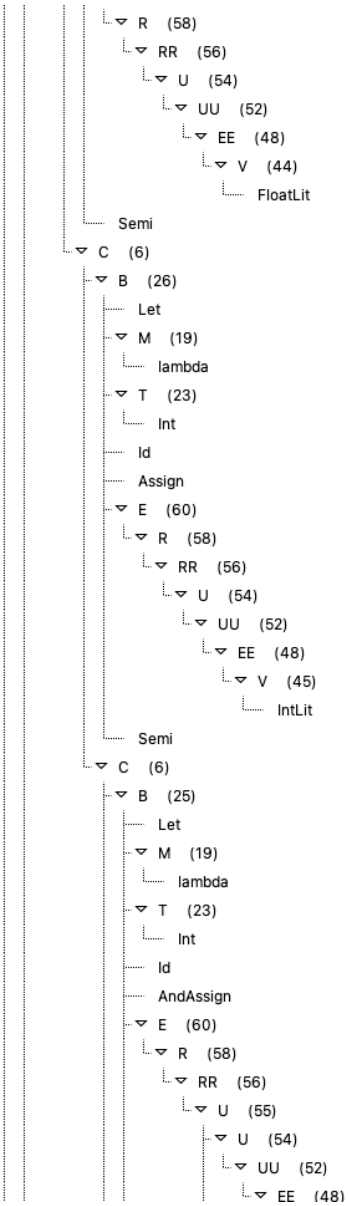
```
table #0:
* 'global'
* 'nothing'
* 'main'
* 'b'
* 'd'
* 'aux'
* 'foo'
* 'bar'
* 'oper'
* 'bool'
* 'george'
```

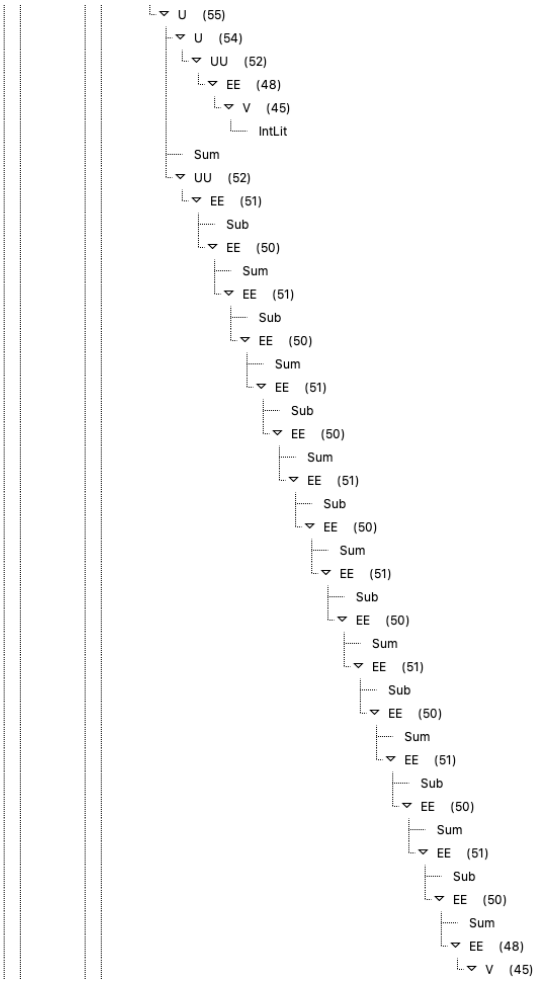
Parse:

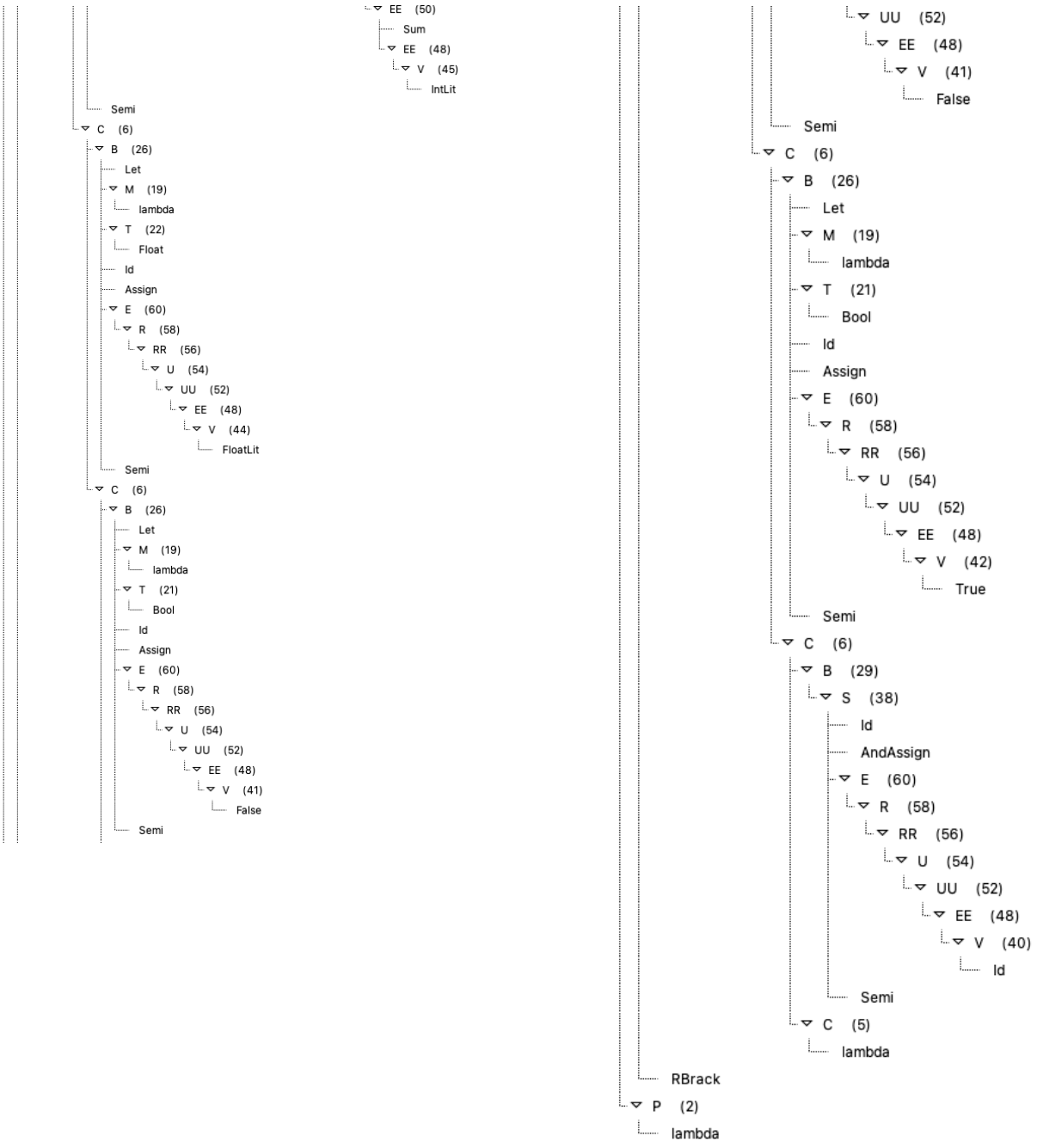
```
ascending 19 23 45 48 52 54 56 58 60 26 11 15 14 9 13 35 29 40 48 52 54 45 48 51 52 55 56 58 60 39 29 5
    6 40 48 52 54 56 45 48 52 54 57 58 60 46 48 49 52 54 56 58 60 24 5 6 6 16 23 12 15 14 22 20 7 8 10
    13 19 20 43 48 52 54 56 58 60 26 19 22 44 48 52 54 56 58 60 26 19 23 45 48 52 54 56 58 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 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
```


PP (1)
P (4)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
P (3)
F (16)
Func
F1 (15)
H (11)
Void
F2 (14)
Id
F3 (13)
LParen
A (9)
Void
RParen
LBrack
C (6)
B (29)
S (35)
Read
Id
Semi
C (6)
B (24)
Do
Semi
C (6)
B (24)
Do
LBrack
C (6)
B (29)
S (39)
Id
Assign
E (60)
R (58)
RR (56)
U (55)
U (54)
UU (52)
EE (48)
V (40)
Id
Sum
UU (52)
EE (51)
Sub
EE (48)
V (45)
IntLit
Semi
C (5)
lambda
RBrack
While
LParen
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (49)
Not
EE (48)
V (46)
LParen


Not
EE (48)
V (46)
LParen
E (60)
R (58)
RR (57)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Lt
U (54)
UU (52)
EE (48)
V (45)
IntLit
RParen
RParen
Semi
C (5)
lambda
RBrack
P (3)
F (16)
Func
F1 (15)
H (12)
T (23)
Int
F2 (14)
Id
F3 (13)
LParen
A (10)
T (22)
Float
Id
K (8)
Comma
Id
K (8)
Comma
T (20)
Str
Id
K (7)
lambda
RParen
LBrack
C (6)
B (26)
Let
M (19)
lambda
T (20)
Str
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (43)
StrLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (22)
Float
Id
Assign
E (60)
R (58)
RR (56)
U (54)


R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (44)
FloatLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (23)
Int
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Semi
C (6)
B (25)
Let
M (19)
lambda
T (23)
Int
Id
AndAssign
E (60)
R (58)
RR (56)
U (55)
U (54)
UU (52)
EE (48)


U (55)
U (54)
UU (52)
EE (48)
V (45)
IntLit
Sum
UU (52)
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (51)
Sub
EE (50)
Sum
EE (48)
V (45)


EE (50)
Sum
EE (48)
V (45)
IntLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (22)
Float
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (44)
FloatLit
Semi
C (6)
B (26)
Let
M (19)
lambda
T (21)
Bool
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (41)
False
Semi
UU (52)
EE (48)
V (41)
False
Semi
C (6)
B (26)
Let
M (19)
lambda
T (21)
Bool
Id
Assign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (42)
True
Semi
C (6)
B (29)
S (38)
Id
AndAssign
E (60)
R (58)
RR (56)
U (54)
UU (52)
EE (48)
V (40)
Id
Semi
C (5)
lambda
RBrack
P (2)
lambda

A.2 Casos Incorrectos

unterm.javascript Fichero con errores de sentencias incompletas.

```
/* unterminated declaration */
let int = 2;

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

/* unfinished float */
let float x = 478234.

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

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

Diagnósticos:

```
unterm.javascript:5:18: error: missing terminating character '"' on string literal
    |
  5 | let string foo = "im not finishing this string
    |                  ^ started here
    |
unterm.javascript:6:18: error: missing terminating character '"' on string literal
    |
  6 | let string bar = "im not finishing this one either\
    |                  ^ started here
    |
unterm.javascript:7:27: warning: unknown escape sequence '\q'
    |
  7 | let string rep = "last one\q
    |                           ^^ interpreted as \\q
    |
unterm.javascript:7:18: error: missing terminating character '"' on string literal
    |
  7 | let string rep = "last one\q
    |                  ^ started here
    |
unterm.javascript:10:15: error: expected digit after '.' in float literal
    |
 10 | let float x = 478234.
    |               ^^^^^^^ perhaps you meant '478234.0'
    |
unterm.javascript:13:1: error: unterminated block comment
    |
 13 | /*** this one is not ** ****** *** /*
    | ^^ started here
    |
unterm.javascript:2:9: error: expected 'identifier' before '='
    |
  2 | let int = 2;
    |         ^ before this token
    |
```

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

```
/* parse error, missing type */
function hello() {}

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

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

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

/* max float */
let float fmax = 340282346638528859811704183484516925440.0;
let float fmax_plus_more = 3402823466385288598117041834845169254382923892341.0;
let float lots_of_decimals = 0.23470283498209348092184092384529038092 8;
```

Diagnósticos:

```
overflow.javascript:8:18: error: string literal is too long
   |
 8 | let string bar = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
   |                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ length is 65 but maximum is 64
   |
overflow.javascript:12:25: error: integer literal out of range for 16-byte type
   |
12 | let int imax_plus_one = 32768;
   |                         ^^^^^ maximum is 32767
   |
overflow.javascript:16:28: error: float literal out of range for 32-byte type
   |
16 | let float fmax_plus_more = 3402823466385288598117041834845169254382923892341.0;
   |                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ maximum is 3.4028235e38
   |
overflow.javascript:2:10: error: expected 'int', 'float', 'string', 'boolean' or 'void' before 'hello'
   |
 2 | function hello() {}
   |          ^^^^^ before this token
   |
```

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

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

Diagnósticos:

```
noise.javascript:2:7: error: illegal character '.' in program
    |
  2 | 123.45.67 12abc 123456. 78
    |       ^ here
    |
noise.javascript:2:17: error: expected digit after '.' in float literal
    |
  2 | 123.45.67 12abc 123456. 78
    |                 ^^^^^^^ perhaps you meant '123456.0'
    |
noise.javascript:3:5: warning: unknown escape sequence '\e'
    |
  3 | "bad\escape" "no closing quote
    |     ^^ interpreted as \\e
    |
noise.javascript:3:14: error: missing terminating character '"' on string literal
    |
  3 | "bad\escape" "no closing quote
    |              ^ started here
    |
noise.javascript:5:14: error: illegal character '>' in program
    |
  5 | =!= &== != <=> &&|!!
    |              ^ here
    |
noise.javascript:5:18: error: illegal character '|' in program
    |
  5 | =!= &== != <=> &&|!!
    |                  ^ here
    |
noise.javascript:6:1: error: missing terminating character '"' on string literal
    |
  6 | "unterminated string again /* unclosed comment
    | ^ started here
    |
noise.javascript:7:18: warning: unknown escape sequence '\x'
    |
  7 | "bad string with \x illegal escape" "" ""empty""
    |                  ^^ interpreted as \\x
    |
noise.javascript:8:7: error: malformed string literal, contains control character '\u{1b}'
    |
  8 | "Hello\u{1b}World"
    |       ^^^^^^ remove this character
    |
noise.javascript:9:21: error: unterminated block comment
    |
  9 | /* final comment */ /* another unclosed
    |                     ^^ started here
    |
noise.javascript:1:4: error: expected '(' before 'condition'
    |
  1 | if condition write "no parenthesis";
    |    ^^^^^^^^^ before this token
    |
```