CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

PROCESADOR DE LENGUAJE JavaScript-PL

Autor: Carlos Ismael Ortega Sánchez

Director: Aurora Pérez Pérez

MADRID, JUNIO 2018

1 CONTENIDO

1 INTRODUCCIÓN

Asignaturas como Procesadores de Lenguajes o Compiladores suelen ser complejas para los estudiantes. Su estructura por fases contiene partes basadas en la teoría de lenguajes formales como el análisis léxico y sintáctico que son básicamente autómatas. No obstante, hay otras fases como el análisis semántico y la generación de código intermedio que no se basan tan directamente en la teoría de lenguajes formales, aunque también se construyen a partir de una gramática (en este caso, una gramática de atributo).

Con este Trabajo Fin de Grado se pretende facilitar el paso de los alumnos por las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes de la ETSIInf-UPM, otorgando herramientas que faciliten la programación del trabajo práctico de Procesadores, así como un procesador completo para los alumnos de Traductores de Lenguajes.

La organización de este documento será la siguiente. En el apartado 2 se presentarán trabajos relacionados, en el apartado 3 se hablará sobre la motivación de este trabajo más en profundidad, en el apartado 4 se expondrán las soluciones pensadas y el desarrollo del trabajo, en el 5 se indicarán unas conclusiones alcanzadas durante el desarrollo del proyecto, en la sexta sección se proporcionarán unas nuevas líneas de trabajo que se pueden seguir en un futuro y por último en la séptima sección se incluirá la bibliografía usada en este documento.

2 ESTADO DEL ARTE

Para facilitar el trabajo a los estudiantes y creadores de compiladores, se han desarrollado a lo largo de los años distintas herramientas, tanto para la creación automática de compiladores o partes de estos, como herramientas de apoyo que no generan nada automáticamente. Algunas de las herramientas serán expuestas a continuación.

"Coco / R es un generador de compiladores, que toma una gramática origen y genera un escáner y un analizador sintáctico para el lenguaje de la gramática. El escáner funciona como un autómata finito determinista. El analizador usa descenso recursivo. Los conflictos de LL (1) pueden resolverse mediante un análisis de símbolos múltiples o mediante comprobaciones semánticas. Por lo tanto, la clase de gramáticas aceptadas es LL (k) para una k arbitraria." (Mössenböck & Markus, 2018)

"Lex es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa.

La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado.

Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla." (Viloria Lanero, 2003)

"Bison es un generador de analizadores sintácticos de propósito general que convierte una gramática anotada sin contexto en un LR (Left to right) determinista o un analizador LR generalizado (GLR) que emplea tablas de analizador LALR (Lookahead LR). Como característica experimental, Bison también puede generar IELR o tablas de analizador canónico LR. Una vez que domine Bison, puede usarlo para desarrollar una amplia gama de analizadores de lenguaje, desde los utilizados en calculadoras de escritorio simples hasta lenguajes de programación complejos." (GNU Bison, 2014)

"Yacc (Siglas correspondientes a solamente otro compilador de compiladores en inglés) es un analizador gramatical y analizador sintáctico. Es decir, es un programa que lee una especificación de gramática y genera un código que puede organizar tokens de entrada en un árbol sintáctico de acuerdo con la gramática. Además, la especificación de la gramática tiene un contenido semántico (en forma de acciones asociadas con reglas gramaticales) que también se organizan para ejecutarse cuando se construyen nodos de árbol ("reducido" es el término real)." (Wiki**3, 2018)

"VAST, una herramienta educativa que se ha diseñado para ser utilizada en clases de compiladores y procesadores del lenguaje. La versión actual permite generar y visualizar los árboles sintácticos y su proceso de construcción. Las principales ventajas de VAST son: independencia del generador de parsers utilizado, permite que los estudiantes visualicen el comportamiento de los parsers que desarrollan y consta de una interfaz diseñada para manejar cómodamente árboles sintácticos muy grandes" (Almeida-Martínez & Urquiza-Fuentes, 2009)

La herramienta SDGLL1 es una herramienta usada para la comprobación de que una gramática de un analizador sintáctico es LL1 además de la generación de la tabla sintáctica. (Moreno Gomez, 2018)

3 MOTIVACIÓN DEL PROYECTO

La asignatura de Traductores de Lenguajes de la ETSIInf UPM es continuación de la asignatura de Procesadores de lenguajes y los proyectos prácticos de ambas están íntimamente ligados.

El proyecto de Procesadores de Lenguajes trata del desarrollo de un analizador de código fuente en tres fases (léxica, sintáctica y semántica) además del uso de un gestor de errores y la tabla de símbolos. En el caso de Traductores de Lenguajes, el proyecto es la continuación del de Procesadores y consiste en una ampliación de este para incorporar las últimas fases necesarias de un compilador.

Algunos alumnos se matriculan de Traductores de Lenguajes antes de tener aprobada Procesadores de Lenguajes, y en la mayoría de los casos el hecho de que no hayan aprobado se debe a que no han terminado aún el proyecto de Procesadores de Lenguajes. Otros la tienen ya aprobada pero no necesariamente realizaron un buen proyecto. Incluso existe la posibilidad de que algún alumno se matricule sin haber cursado antes Procesadores, ya que no hay cierres de asignaturas definidos en el plan de estudios. Por todo ello, es frecuente que un alumno de Traductores de Lenguajes no tenga un buen proyecto (un procesador que realice la fase de análisis) sobre el que trabajar. Por eso se decidió realizar este Trabajo de Fin de Grado, cuyo

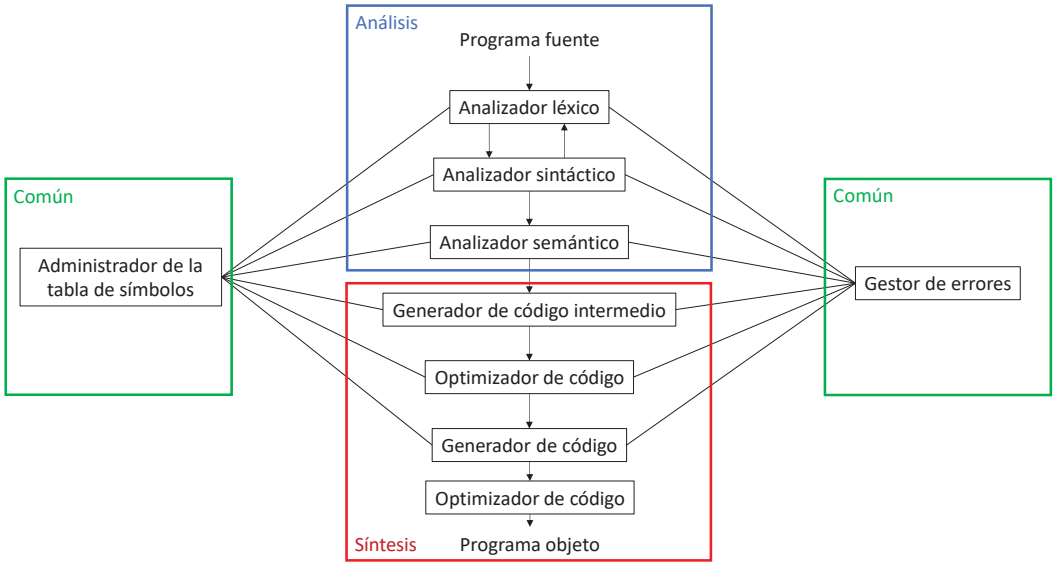
objetivo es implementar un Procesador bien documentado y fácilmente entendible que ceder a estos alumnos para su trabajo.

A lo largo de la realización de este Trabajo de Fin de Grado surgió la idea de ayudar también a los alumnos de Procesadores de Lenguajes otorgándoles código que sea difícil en su creación, pero no tenga ningún objetivo didáctico. Con esto surgió la idea de hacer que la tabla de transiciones de analizador léxico pueda ser creada con dos ficheros aliviando así la carga de trabajo de los alumnos.

Con esto intentamos hacer la estancia de los alumnos de Procesadores de Lenguajes y Traductores de Lenguajes más llevadera para que se puedan concentrar en la parte importante de la asignatura, la parte conceptual, y no en los problemas que pueden surgir al realizar la implementación debidos a la falta de práctica a la hora de programar, lo cual, realmente, no tiene demasiada relación con el objetivo de aprendizaje de estas asignaturas.

4 SOLUCIÓN

El objetivo de los proyectos de Procesadores de Lenguajes y de Compiladores es obtener un compilador que contiene distintas fases mostradas en el siguiente diagrama:



El proyecto (Ortega Sánchez, 2018) solo se va a centrar en la parte de análisis del código fuente desarrollándose las fases de los analizadores léxico, sintáctico y semántico, además de las tablas de símbolos y el gestor de errores.

4.1 ANALIZADOR LÉXICO

El analizador léxico es la fase de análisis que se encarga de procesar el programa fuente sacando los tokens (unidades mínimas de información) del lenguaje, que usará el analizador sintáctico posteriormente. Su función es comprobar que no haya en el fichero fuente "palabras" no pertenecientes al lenguaje.

A continuación, comentaré el analizador léxico creado para el proyecto y la última subsección del analizador léxico tratará de la herramienta que se ha realizado para facilitar el trabajo de los alumnos de Procesadores de Lenguajes.

4.1.1 Tokens

	Patrón	Token
Operadores aritméticos	+	<1, >
	-	<2, >
	*	<3, >
	/	<4, >
	%	<5, >
Operadores relacionales	==	<6, >
	!=	<7, >
	<	<8, >
	>	<9, >
	<=	<10, >
	>=	<11, >
Operadores lógicos	&&	<12, >
	||	<13, >
	!	<14, >
Operadores inc y dec	++	<15, >
	--	<16, >
Operadores de asignación	=	<17, >
	+=	<18, >
	-=	<19, >
	*=	<20, >
	/=	<21, >
	%=	<22, >
	&=	<23, >
	|=	<24, >
Entero	d_{19} d*	<25, valor>
	0 d_O^+	
	0 x d_H^+	

	Patrón	Token
Cadena	" c_1 [\z] "	<26, cadena>
	' c_2 [\z] '	
Palabras reservadas	true	<27, >
	false	<28, >
	var	<29, >
	int	<30, >
	bool	<31, >
	chars	<32, >
	write	<33, >
	prompt	<34, >
	return	<35, >
	if	<36, >
	else	<37, >
	while	<38, >
	do	<39, >
	for	<40, >
	switch	<41, >
	case	<42, >
	break	<43, >
	default	<44, >
	function	<45, >
Caracteres especiales	(	<46, >
	)	<47, >
	{	<48, >
	}	<49, >
	,	<50, >
	;	<51, >
	:	<52, >
	EOF	<53, >
Identificador	l (l | d | _)*	<54,posi_TS>

	Descripción	Conjunto de caracteres representado
d	Dígitos (0-9)	{0-9}
d_{19}	Dígitos de 1 a 9	{1-9}
d_O	Dígitos octales (0-7)	{0-7}
d_H	Dígitos hexadecimales (0-9, A-F)	{0-9, A-F}
c_1	Cualquier carácter excepto " y \	T - {",\}
z	Uno de los siguientes 5 caracteres: n, t, ', ", \	{n, t, ', ", \}
c_2	Cualquier carácter excepto ' y \	T - {', \}
l	Letras	{a-z, A-Z}

Como el atributo del token no es un tipo único de dato, internamente los tokens se han implementado como un objeto con 3 atributos:

- El primero de ellos contiene el código del token (un número entero, del 1 al 54, que indica el tipo de token).
- El atributo del token se almacena bien en el segundo atributo del objeto o bien en el tercero, dependiendo del caso: cuando el atributo del token es entero (el caso de las constantes enteras y los identificadores) se usa el segundo atributo del objeto, que será un entero, mientras que cuando el atributo del token es una cadena (el caso de las cadenas), se usa el tercer atributo del objeto.

4.1.2 Gramática

La Gramática regular que genera el lenguaje reconocido por el analizador léxico es la que se describe a continuación.

G = (N, T, S, P)

No Terminales

N = {S, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y}

Terminales

Letras	Digit	;	,	{	}	(	)	/	\	%	*	+	-	=
<	>	!	'	"	_	&	|	\n	\t	\r	:	esp	EOF	o.c. imprimible

Letras: Todas las letras del alfabeto tanto mayúsculas como minúsculas.
Digit: Dígitos del 1 al 9
o.c. imprimible: Cualquier otro carácter imprimible.

Reglas de producción

S→ del S | / A | - E | d_{19} F | 0 G | " K | ' M | + O | * P | % Q | = R | & T | "|" U | I V | < W | < X | ! Y | (|) | { | } | , | ; | : | EOF
A→ / B | * C | = | λ
B→ \n S | c_3 B
C→ * D | c_4 C
D→ * D | c_5 C | / S
E→ - | = | λ
F→ d F | λ
G→ d_O H | x I | λ
H → d_O H | λ
I→ d_H J
J→ d_H J | λ
K → c_1 K | \ L | "
L→ n K | t K | " K | \ K
M→ \ N | c_2 L | '
N→ n M | t M | ' M | \ M
O→ + | = | λ
P→ = | λ
Q→ = | λ
R→ = | λ
T→ & | =
U→ "|" | =

	Descripción	Conjunto de caracteres representado
l	Letras	{a-z, A-Z}
d	Dígitos (0-9)	{0-9}
d_{19}	Dígitos de 1 a 9	{1-9}
d_O	Dígitos octales (0-7)	{0-7}
d_H	Dígitos hexadecimales (0-9, A-F, a-f)	{0-9, A-F, a-f}
c_1	Cualquier carácter excepto ", \, salto de línea y fin de fichero	T - {", \, \n, EOF}
c_2	Cualquier carácter excepto ', \, salto de línea y fin de fichero	T - {', \, \n, EOF}
del	{espacio en blanco, \r, \t, \n}	{espacio, \r, \t, \n}
c_3	Cualquier carácter excepto salto de línea y fin de fichero	T – {\n, EOF}
c_4	Cualquier carácter excepto * y fin de fichero	T – {*, EOF}
c_5	Cualquier carácter excepto *, / y fin de fichero	T – {*, /, EOF}

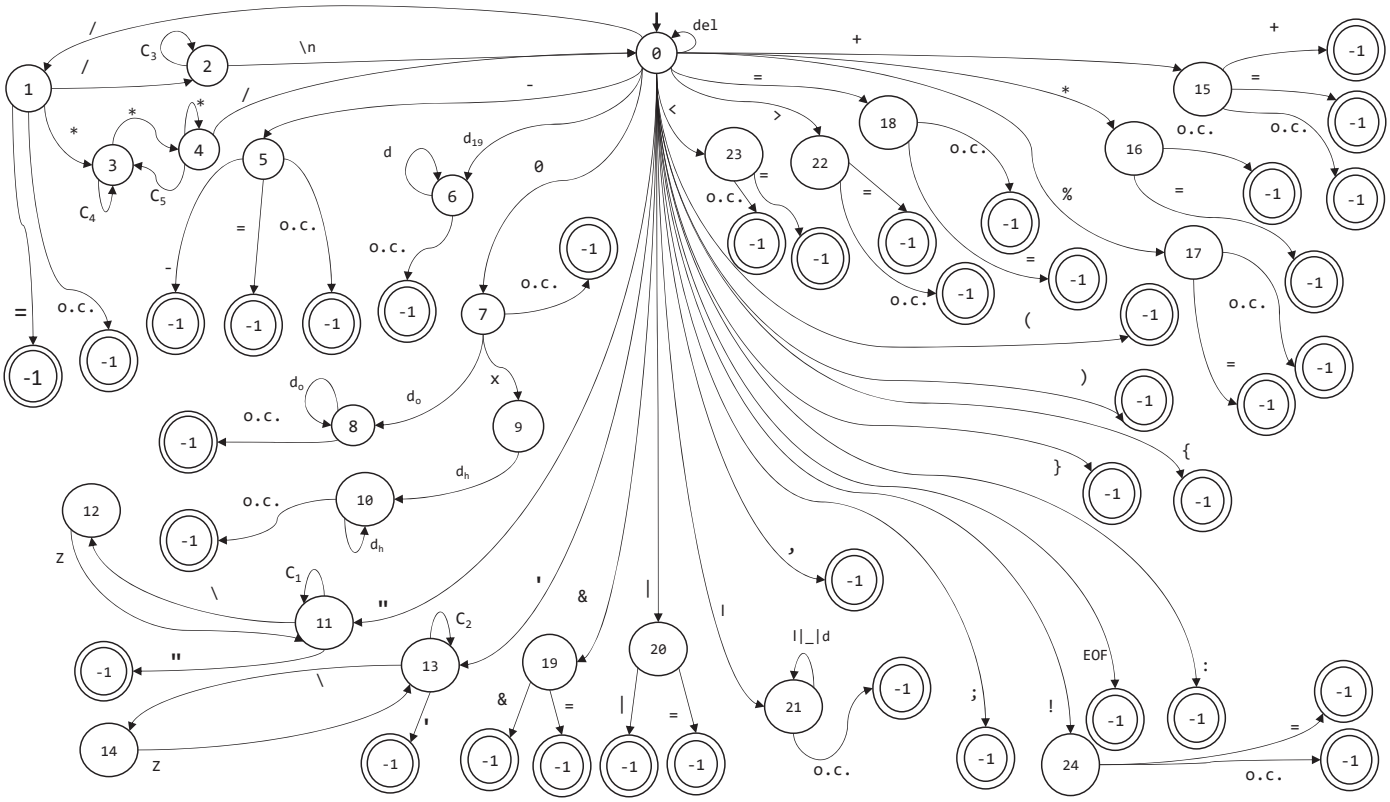
Debido a que el carácter "|" del lenguaje fuente coincide con el carácter que se usa para separar los distintos consecuentes de las reglas de un mismo símbolo no terminal, para referirnos al carácter del lenguaje fuente lo representamos entre comillas en la gramática: "|"

V→ l V | _ V | d V | λ

W→ = | λ
X→ = | λ
Y→ = | λ

4.1.3 Autómata finito determinista

Diagrama del autómata



	Descripción	Conjunto de caracteres representado
l	Letras	{a-z, A-Z}
d	Dígitos (0-9)	{0-9}
d_{19}	Dígitos de 1 a 9	{1-9}
d_O	Dígitos octales (0-7)	{0-7}
d_H	Dígitos hexadecimales (0-9, A-F,a-f)	{0-9, A-F,a-f}
c_1	Cualquier carácter excepto " y \, salto de línea y fin de fichero	T - {", \, \n, EOF}
z	Uno de los siguientes 2 caracteres: n, t	{n, t}
c_2	Cualquier carácter excepto ' y \, salto de línea y fin de fichero	T - {', \, \n, EOF}
del	{espacio en blanco, \r, \t, \n}	{blanco, \r, \t, \n}
c_3	Cualquier carácter excepto salto de línea	T – {\n}
c_4	Cualquier carácter excepto * y fin de fichero	T – {*, EOF}
c_5	Cualquier carácter excepto *, / y fin de fichero	T – {*, /, EOF}

Tabla de transiciones del autómata

	0	1-7	8-9	A-F a-f	x	n t	G-Z g-m o-s u-w y-z	_	/	\	"	'	&	|	!	:	+
0	7 L	6 N	6 N	21 R	21 R	21 R	21 R	-	1 L	-	11 S	13 S	19 L	20 L	24 L	-1 T_{52}	15 L
1	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	2 L	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4
2	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L
3	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L
4	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	0 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L
5	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2
6	6 D	6 D	6 D	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
7	8 N	8 N	-1 T_{25}	-1 T_{25}	9 L	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
8	8 O	8 O	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
9	10 F	10 F	10 F	10 F	-	-	-	-	-	-	-	-	-	-	-	-	-
10	10 H	10 H	10 H	10 H	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
11	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	12 C	-1 T_{26}	11 C	11 C	11 C	11 C	11 C	11 C
12	-	-	-	-	-	11 C	-	-	-	-	-	-	-	-	-	-	-
13	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	14 C	13 C	-1 T_{26}	13 C	13 C	13 C	13 C	13 C
14	-	-	-	-	-	13 C	-	-	-	-	-	-	-	-	-	-	-
15	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_{15}
16	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3
17	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5
18	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}
19	-	-	-	-	-	-	-	-	-	-	-	-	-1 T_{12}	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-1 T_{13}	-	-	-
21	21 C	21 C	21 C	21 C	21 C	21 C	21 C	21 C	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P
22	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9
23	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8
24	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}

	-	*	%	=	<	>	,	;	{	}	(	)	EOF	\n	Del-{\n}	o.c. imprimible
0	5 L	16 L	17 L	18 L	23 L	22 L	-1 T_{50}	-1 T_{51}	-1 T_{48}	-1 T_{49}	-1 T_{46}	-1 T_{47}	-1 T_{53}	0 L	0 L	-
1	-1 T_4	3 L	-1 T_4	-1 T_{21}	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4	-1 T_4
2	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	2 L	-	0 L	2 L	2 L
3	3 L	4 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	-	3 L	3 L	3 L
4	3 L	4 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	3 L	-	3 L	3 L	3 L
5	-1 T_{16}	-1 T_2	-1 T_2	-1 T_{19}	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2	-1 T_2
6	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
7	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
8	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}	-1 T_{25}
11	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	11 C	-	-	11 C	11 C
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	13 C	-	-	13 C	13 C
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-1 T_1	-1 T_1	-1 T_1	-1 T_{18}	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1	-1 T_1
16	-1 T_3	-1 T_3	-1 T_3	-1 T_{20}	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3	-1 T_3
17	-1 T_5	-1 T_5	-1 T_5	-1 T_{22}	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5	-1 T_5
18	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_6	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}	-1 T_{17}
19	-	-	-	-1 T_{23}	-	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-1 T_{24}	-	-	-	-	-	-	-	-	-	-	-	-
21	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-1 P	-	-1 P	-1 P
22	-1 T_9	-1 T_9	-1 T_9	-1 T_{11}	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9	-1 T_9
23	-1 T_8	-1 T_8	-1 T_8	-1 T_{10}	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8	-1 T_8
24	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_7	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}	-1 T_{14}

* En la transformación del estado 21 la acción semántica T no tiene subíndice porque dependiendo de las anteriores acciones tendrá un valor del 27 al 45 o el valor 53, dependiendo de si es una palabra reservada o una id

Acciones Semánticas

1. L: Leer del fichero fuente el siguiente carácter
2. N: La variable `valor`, en la cual se va a ir progresivamente calculando el valor del número que se reconozca, se inicializa con el valor del dígito que se acaba de leer. Y a continuación se lee el siguiente carácter del fichero fuente
3. F: La variable `valor`, en la cual se va a ir progresivamente calculando el valor del número que se reconozca, se inicializa con el valor del dígito hexadecimal que se acaba de leer (transformado la letra hexadecimal en su valor decimal: A=10, B=11,…). A continuación, se lee el siguiente carácter del fichero fuente
4. D: Continuar calculando el valor del número, lo cual se hace multiplicando la variable `valor` por 10 y sumándole el nuevo dígito leído. A continuación, se lee el siguiente carácter del fichero fuente
5. O: Continuar calculando el valor del número octal, multiplicando por 8 la variable `valor` y sumando el nuevo dígito leído. A continuación, se lee el siguiente carácter del fichero fuente
6. H: Continuar calculando el valor del número hexadecimal, multiplicando la variable `valor` por 16 y sumando el valor del nuevo carácter leído (haciendo una previa transformación si es una letra de la 'A' a la 'F'). Y a continuación se lee el siguiente carácter
7. S: Inicializar la variable `lexema` con la cadena vacía. Y leer el siguiente carácter del fichero fuente
8. R: Inicializar la variable `lexema` con la letra leída. Y leer el siguiente carácter del fichero fuente
9. C: Concatenar a la variable `lexema` el carácter que se acaba de leer. A continuación, leer el siguiente carácter del fichero fuente
10. P: Comprobar si el lexema leído es una palabra reservada y genera otra acción semántica de creación de token
11. $T_{nº}$: Crear token del tipo nº, por ejemplo, la acción semántica T_4 corresponderá a `la creación de un token <4, >`.
    En la creación de todos los tokens excepto los: 1,2,3,4,5,14 y 25, los de palabra reservada e id se habrá realizado una lectura de un carácter.
    En el token id se realizarán además las siguientes comprobaciones: 1. Si estamos en zona de declaración, que no exista el id en la tabla de símbolos activa (si existe lanza un error), y la creación de la entrada en la tabla para el id. 2. Si no estamos en zona de declaración se comprobará que el id ha sido declarado antes, tanto en la tabla de símbolos activa como en la global, si no lo ha sido se crea en la tabla de símbolos global.
    En el token 25 correspondiente a los números se comprobará que el número creado no exceda el valor máximo (32767) y se añadirá como atributo el valor numérico que se ha formado
    En el token 26 correspondiente con las cadenas, se añadirá como atributo del token la cadena que se ha formado

4.1.4 Errores

El analizador léxico detecta un error cuando no ha llegado a un estado final y no hay ninguna transición posible en el autómata para el carácter leído. En tal caso, no se reconoce ningún token y se envía un error.

Los errores que detecta el analizador léxico son:

1. Que se lea un carácter que no pertenece al conjunto de símbolos terminales de la gramática.
2. Que se lea un carácter para el que no hay transición en el estado actual, aunque ese carácter sí que pertenezca al conjunto de símbolos terminales de la gramática
3. Que se intente declarar una variable con un nombre ya usado.
4. Que el valor de un número entero sea superior a 32767.
5. Que una cadena se quede sin cerrar, o sea, que no esté la comilla de cierre.

4.1.5 Herramienta para la implementación

Para facilitar el desarrollo del analizador léxico en el proyecto de Procesadores de Lenguajes se ha decidido usar ficheros para definir la tabla de transiciones del autómata del analizador. Con esto los alumnos no tendrán que programar el array que representa la tabla, ni la lógica de su lectura quitando muchísima dificultad innecesaria al programa permitiendo a los alumnos concentrarse en lo importante, el diseño del autómata.

La implementación se basa en dos ficheros: TablaDeTransiciones.csv y agrupaciones.txt

TablaDeTransiciones.csv

Se trata de un fichero separado por comas (traducción de las siglas csv, coma separated values) fácilmente editable con Excel. Este fichero contendrá la tabla de transiciones del autómata codificada de la siguiente manera:

- Se reservan dos celdas contiguas para cada transición del autómata
- En la primera fila, la primera columna de cada pareja que forma una transición tendrá el carácter que "disparará" la transición. Si se quiere poner más de un carácter, por ejemplo, agrupar las letras de la a hasta la z para que cualquiera de ellas dispare la transición, se deberá definir la agrupación en agrupaciones.txt, fichero que se explicará más adelante.
- En el resto de las filas, la primera columna de cada pareja que forma la transición contendrá el estado destino o estará vacía si se quiere representar un caso de error. El estado final es el estado -1.
- Cada fila excepto la cabecera (primera fila) representara un estado, siendo la fila 2 el estado 0, la fila 3 el estado 1, etc. (estado = fila – 2)
- En las filas que no son la cabecera en la segunda columna, de cada pareja que forma una transición se tendrá la acción semántica a realizar o el código de error que se quiere enviar (siempre que el estado destino esté vacío). Como los alumnos programaran sus propias acciones semánticas y su gestor de errores, en este campo se podrá poner lo que se crea necesario.

Agrupaciones.txt

En este fichero se agruparán los caracteres para su utilización en la tabla de transiciones que se definirán de la siguiente manera:

nombre=caracter1,caracter2,caracter3

Los espacios pueden formar parte del lenguaje por eso hay que evitar usarlos a no ser que se quieran definir.

Como el igual y la coma se usan para definir las agrupaciones, si se quisiesen poner como carácter en una agrupación se debería usar el carácter de escape '\' quedando así \, y \= .

Por último, se permiten comentarios de línea en este fichero empezando la línea comentada con //.

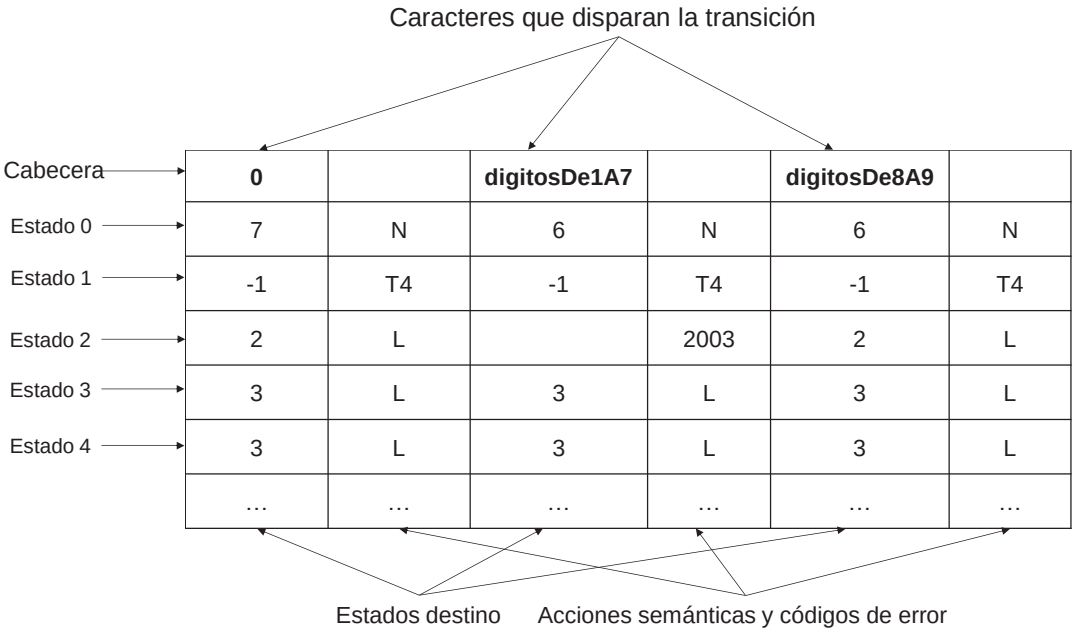
Ejemplo

A continuación, pondré una pequeña parte del fichero TablaDeTransiciones.csv, modificada para que incluya un error aunque no debería de ir ahí, y el fichero de agrupaciones usados en el proyecto a modo de ejemplo

- Agrupaciones.txt:

```
// En este archivo se colocaran las agrupaciones de la siguiente forma
// nombre=caracter1,caracter2,caracter3
// Las agrupaciones se separaran por saltos de linea
// ATENCION: NO PONER ESPACIOS A NO SER QUE SE CONSIDEREN PARTE DEL LENGUAJE
// ATENCION: NO PONER CARACTERES NO ALFANUMERICOS COMO NOMBRE DE AGRUPACION
// Los comentarios iran precedidos de // al inicio de la linea y no hay comentarios de bloque
// Como las comas y el igual pertenecen a la codificacion de este archivo iran precedidos de \
// siendo \, y \=
// Por ultimo como ayuda extra el programa acepta o.c. y EOF sin necesidad de escribirlo en
este documento

// Digitos del 1 al 7
digitosDe1A7=1,2,3,4,5,6,7

// Digitos 8 y 9
digitosDe8A9=8,9

// Letras hexadecimales
letrasHexadecimales=A,B,C,D,E,F,a,b,c,d,e,f

// Caracteres pertenecientes al salto de linea y la tabulacion
saltoDeLineaYTabulacion=n,t

// Resto de letras que no tienen asignaciones especiales
letrasRestantes=G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,g,h,i,j,k,l,m,o,p,q,r,s,u,v,w,y,z

// Caracteres omitibles
del= ,\r,\t,\n
```

- TablaDeTransiciones.csv:



4.2 ANALIZADOR SINTÁCTICO

El analizador sintáctico se encarga de pedir tokens al analizador léxico y comprobar que las sentencias están correctamente formadas de acuerdo con una gramática de contexto libre.

Se ha decidido usar un analizador recursivo para el cual se necesitara una gramática LL(1), para la comprobación de que la gramática es LL(1) se ha usado SDGLL1 (ETSIINF, 2018), en la siguiente sección (3.2.1) en la que se muestra la gramática se ha usado el formato necesario para el funcionamiento de dicho programa.

4.2.1 Gramática LL(1)

Axioma = P

NoTerminales = { P B T E E2 V L L2 Q S V2 X S2 F G G2 D D2 I I2 H C
A K CASE B2 ID THEN ELSE INICIALIZACION ACTUALIZACION
ACTUALIZACION2 ASIGNACION ASIGNACION_OP J J2 M M2 INCDEC
}

Terminales = {
EOF var identificador int bool chars if ( ) || && > >= < <= == != + - * / % entero
cadena = += -= *= /= %= &= |= , write prompt return function iniBloq endBloq do
switch case : break ; ++ -- default true false ! else while for
}

```
//// Reglas de la gramática
Producciones = {

        P -> B P                    //// Programa, bloques de codigo, funciones y fin de fichero
        P -> F P
        P -> EOF

        B -> var T identificador B2 ID ;                    //// Bloques de codigo
        B -> if ( E ) THEN
        B -> S ;
        B -> switch ( E ) iniBloq CASE endBloq
        B -> while ( E ) iniBloq C endBloq
        B -> do iniBloq C endBloq while ( E ) ;
        B -> for ( INICIALIZACION ; E ; ACTUALIZACION ) iniBloq  C endBloq

        B2 -> ASIGNACION E
        B2 -> lambda

        INICIALIZACION -> identificador ASIGNACION E            ////Bloque FOR
        INICIALIZACION -> lambda

        ACTUALIZACION -> identificador ACTUALIZACION2
        ACTUALIZACION -> INCDEC identificador
        ACTUALIZACION -> lambda

        ACTUALIZACION2 -> ASIGNACION E
        ACTUALIZACION2 -> INCDEC

        INCDEC -> ++
        INCDEC -> --

        ASIGNACION -> =                 //// Todos los terminales de asignacion
        ASIGNACION -> ASIGNACION_OP

        ASIGNACION_OP -> +=             //// Los terminales de asignacion con operacion
        ASIGNACION_OP -> -=
        ASIGNACION_OP -> *=
        ASIGNACION_OP -> /=
        ASIGNACION_OP -> %=
        ASIGNACION_OP -> &=
        ASIGNACION_OP -> |=

        THEN -> iniBloq C endBloq ELSE                  //// Bloque THEN
        THEN -> S ;

        ELSE -> else iniBloq C endBloq                  //// Bloque ELSE
        ELSE -> lambda
```

```
ID -> , identificador B2 ID                    //// Concatenacion de identificadores
ID -> lambda

CASE -> case entero : C CASE    //// Bloque CASE del Switch
CASE -> default : C             //// Default del switch
CASE -> lambda

T -> int                        //// Tipos
T -> bool
T -> chars

E -> G N2

E2 -> || G E2                          //// Or lógico
E2 -> lambda

G -> D G2

G2 -> && D G2                          //// And lógico
G2 -> lambda

D -> I D2

D2 -> == I D2                          //// Comparación de igualdad
D2 -> != I D2
D2 -> lambda

I -> J I2

I2 -> > J I2                           //// Comparación de tamaño
I2 -> >= J I2
I2 -> < J I2
I2 -> <= J I2
I2 -> lambda

J -> M J2

J2 -> + M J2                           //// Suma y resta
J2 -> - M J2
J2 -> lambda

M -> V M2

M2 -> * V M2                           //// Multiplicacion división y porcentaje
M2 -> / V M2
M2 -> % V M2
M2 -> lambda
```

```
V -> identificador V2                    //// Valores (Es decir variables, funciones...)
V -> entero
V -> cadena
V -> true
V -> false
V -> ( E )
V -> INCDEC identificador
V -> + V
V -> - V
V -> ! V

V2 -> ( L )
V2 -> lambda
V2 -> INCDEC

L -> E Q                                 //// Parametros
L -> lambda

L2 -> E Q

Q -> , E Q                               //// Concatenacion de parametros
Q -> lambda

S -> identificador S2                    //// Sentencias
S -> INCDEC identificador
S -> return X
S -> write ( L2 )
S -> prompt ( identificador )
S -> break

S2 -> ASIGNACION E
S2 -> ( L )
S2 -> INCDEC

X -> E                                   //// Valores para el return de una funcion
X -> lambda

F -> function H identificador ( A )  iniBloq  C endBloq          //// Bloque de funcion

A -> T identificador K          //// Parametros de la declaracion de una funcion
A -> lambda

K -> , T identificador K        //// Parametros en declaracion de funcion
K -> lambda

C -> B C                        //// Codigo que puede ir dentro de cualquier bloque
C -> lambda
```

```
        H -> T                          //// Tipo de una funcion
        H -> lambda
}
```

4.2.2 Errores

El analizador sintáctico detecta un error cuando el programa no sigue el lenguaje correctamente.

Los errores que detecta el analizador sintáctico son:

1. Que en la raíz de un programa no se encuentre una sentencia valida ya sean declaración o modificación de variables, llamadas o declaración de funciones o iniciación de algún bloque de código if, for, switch, while o do-while.
2. Que la parte de actualización de la declaración de un bucle for no sea correcta, tiene que ser una asignación, un pre o pos incremento o un pre o pos decremento.
3. Se ha recibido un tipo distinto a int, bool o chars.
4. La expresión que se ha leído es inválida.
5. La sentencia que se ha leído es inválida.
6. La declaración de una función es incorrecta.
7. Se ha recibido un token no esperado.
8. No se han añadido parámetros a un write.

4.3 ANALIZADOR SEMÁNTICO

El analizador semántico se encarga de comprobar que no haya errores en el manejo de los tipos, activar y desactivar un flag para indicarle al léxico si se está o no en una zona de declaración de variables, que no se puede acceder a una variable donde no es visible y que las sentencias break y deafult estén correctamente colocadas.

Para el diseño del analizador semántico se utiliza una Gramática de Atributo. En este Trabajo Fin de Grado, se ha usado la notación de Esquema de Traducción (EdT); por lo tanto, las acciones semánticas se incluirán en los consecuentes de las reglas de la gramática de contexto libre del analizador sintáctico (se representan encerradas entre llaves).

4.3.1 Acciones en la gramática

Consideraciones previas

Una curiosidad de JavaScript-PL es que no es obligatorio declarar todos los identificadores antes de que se utilicen; en este caso, un uso de un identificador no declarado se considera una variable global entera.

Por cómo se ha implementado este Procesador, la entrada del identificador en la tabla de símbolos (el atributo del token identificador) es negativa si se encuentra en la tabla local y positiva si se encuentra en la global. Al llamar a métodos como aniadirTS() el método, por el signo de la entrada, discernirá la tabla de símbolos que tiene que editar.

Se usará el valor null para indicar que un atributo se le da un valor vacío a un atributo.

Existen funciones especiales para la gramática de atributos:

- aniadirTS(índice, tipo, desp, param): Completa los datos de un identificador en la tabla de símbolos que corresponda según el índice.
- actualizarFuncionTS(índice, tipoRetorno, param, modo, etiquetaFuncion): Actualiza los datos del identificador de una función en la tabla de símbolos que corresponda. Como el lenguaje no permite anidamiento siempre será en la global.
- tipoId(índice): Devuelve el tipo del identificador correspondiente al índice.

Gramática de atributos

P' -> **{ crearTS() }** P

P -> **{ B.return = false; B.break = false }** B P

P -> F P

P -> eof **{ cerrarTSActiva() }**

B -> var **{zonaDeclaracion = true }** T identificador **{ aniadirTS(identificador.entrada, T.tipo, desp, false); desp += T.ancho; }** B2 **{ if(T.tipo == B2.tipo || B2.tipo == Tipo_OK) then: ID.tipo = T.tipo; ID.ancho = T.ancho; else: error; }** ID ; **{ B.tipoReturn = Tipo_OK; zonaDeclaracion = false; }**

B -> if (E **{ if (E.tipo != bool) then: error; }**) **{ THEN.return = B.return; THEN.break = B.break; }** THEN **{ B.tipoReturn = THEN.tipoReturn }**

B -> **{ S.return = B.return; S.break = B.break; }** S **{ B.tipoReturn = S.tipoReturn }** ;

B -> switch (E **{ if (E.tipo != int) then: error }**) iniBloq **{ CASE.return = B.return }** CASE **{ B.tipoReturn = CASE.tipoReturn }** endBloq

B -> while (E **{ if (E.tipo != bool) then: error }**) iniBloq **{ C.return = B.return; C.break = B.break; }** C **{ B.tipoReturn = C.tipoReturn }** endBloq

B -> do iniBloq **{ C.return = B.return; C.break = B.break; }** C endBloq while (E) ; **{ if (E.tipo != bool) then: error; else B.tipoReturn = C.tipoReturn }**

B -> for (INICIALIZACION ; E **{ if (E.tipo != bool) then: error }** ; ACTUALIZACION) iniBloq **{ C.return = B.return: C.break = B.break; }** C **{ B.tipoReturn = C.tipoReturn }**endBloq **{ B.tipoReturn = C.tipoReturn }**

B2 -> ASIGNACION E **{ if (ASIGNACION.tipo == E.tipo || ASINGACION.tipo == Tipo_OK) then: B2.tipo = E.tipo; else: error; }**

B2 -> lambda **{ B2.tipo = Tipo_OK }**

INICIALIZACION -> identificador ASIGNACION E **{ if (ASIGNACION.tipo == E.tipo || ASINGACION.tipo == Tipo_OK) then: if (tipoId(identificador.entrada) == Tipo_Vacio) then: aniadirTS(identificador.entrada, int, 8, false); desp += 8; }{ if (tipoId(identificador.entrada) != E.tipo) then: error; }**

INICIALIZACION -> lambda

ACTUALIZACION -> identificador ACTUALIZACION2 **{ if (tipoId(identificador.entrada) == Tipo_Vacio) then: then: aniadirTS(identificador.entrada, int, 8, false); desp += 8; } { if (ACTUALIZACION2.tipo != tipoId(identificador.entrada)) then: error; }**

ACTUALIZACION -> INCDEC identificador **{ if (tipoId(identificador.entrada) != int) then : if (tipoId(identificador.entrada) == Tipo_Vacio) then: aniadirTS(identificador.entrada, int, 8, false); desp += 8; else: error }**

ACTUALIZACION -> lambda

ACTUALIZACION2 -> ASIGNACION E **{ if (ASIGNACION.tipo == E.tipo || ASINGACION.tipo == Tipo_OK) then: ACTUALIZACION2.tipo = E.tipo; }**

ACTUALIZACION2 -> INCDEC **{ ACTUALIZACION2.tipo = int }**

INCDEC -> ++

INCDEC -> --

ASIGNACION -> = **{ ASIGNACION.tipo = Tipo_OK }**

ASIGNACION -> ASIGNACION_OP **{ ASIGNACION.tipo = ASIGNACION_OP.tipo }**

ASIGNACION_OP -> += **{ ASIGNACION_OP.tipo = int }**

ASIGNACION_OP -> -= **{ ASIGNACION_OP.tipo = int }**

ASIGNACION_OP -> *= **{ ASIGNACION_OP.tipo = int }**

ASIGNACION_OP -> /= **{ ASIGNACION_OP.tipo = int }**

ASIGNACION_OP -> %= **{ ASIGNACION_OP.tipo = int }**

ASIGNACION_OP -> &= **{ ASIGNACION_OP.tipo = bool }**

ASIGNACION_OP -> |= **{ ASIGNACION_OP.tipo = bool }**

THEN -> iniBloq **{ C.return = THEN.return; C.break = THEN.break; }** C endBloq **{ ELSE.return = THEN.return; ELSE.break = THEN.break; }** ELSE **{ if (C.tipoReturn == ELSE.tipoReturn || ELSE.tipoReturn == Tipo_OK) then: THEN.tipoReturn = C.tipoReturn; else: if (C.tipoReturn == Tipo_OK) then: THEN.tipoReturn = ELSE.tipoReturn; else: error; }**

THEN -> **{ S.return = THEN.return; S.break = THEN.break; }** S **{ THEN.tipoReturn = S.tipoReturn }** ;

ELSE -> else iniBloq **{ C.return = ELSE.return; C.break = ELSE.break; }** C endBloq **{ ELSE.tipoReturn = C.tipoReturn; }**

ELSE -> lambda **{ ELSE.tipoReturn = Tipo_OK }**

ID -> , identificador B2 **{ if(ID.tipo == B2.tipo || B2.tipo == Tipo_OK) then: ID'.tipo = ID.tipo; ID'.ancho = ID.ancho; aniadirTS(identificador.entrada, ID.tipo, desp, false); desp += ID.ancho; else: error; }** ID'

ID -> lambda

CASE -> case entero : **{ C.return = CASE.return; C.break = true; }** C **{ CASE'.return = CASE.return }** CASE' **{ if (C.tipoReturn == CASE'.tipoReturn || CASE'.tipoReturn == Tipo_OK) then: CASE.tipoReturn = C.tipoReturn; else: error; }**

CASE -> default : **{ C.return = CASE.return; C.break = true }** C **{ CASE.tipoReturn = C.tipoReturn; }**

CASE -> lambda **{ CASE.tipoReturn = Tipo_OK; }**

T -> int **{ T.tipo = int; T.ancho = 8 }**

T -> bool **{ T.tipo = bool; T.ancho = 1 }**

T -> chars **{ T.tipo = chars; T.ancho = 8 }**

E -> G E2 **{ if(E2.tipo == bool && G.tipo != bool) then: error; else: E.tipo = G.tipo; E.ancho = G.ancho }**

E2 -> || G **{ if(G.tipo != bool) then: error; }** E2' **{ E2.tipo = bool; E2.ancho = 1; }**

E2 -> lambda **{ E2.tipo = Tipo_OK }**

G -> D G2 **{ if(G2.tipo == bool && D.tipo != bool) then: error; else: G.tipo = D.tipo; G.ancho = D.ancho }**

G2 -> && D **{ if(D.tipo != bool) then: error; } G2' { G2.tipo = bool; G2.ancho = 1; }**

G2 -> lambda **{ G2.tipo = Tipo_OK }**

D -> I D2 **{ if (D2.tipo == bool) then: D.tipo = D2.tipo; D.ancho = D2.ancho; else: D.tipo = I.tipo; D.ancho = I.ancho; }**

D2 -> == I D2' **{ D2.tipo = bool; D2.ancho = 1 }**

D2 -> != I D2' **{ D2.tipo = bool: D2.ancho = 1 }**

D2 -> lambda **{ D2.tipo = Tipo_OK }**

I -> J **{ I2.tipo = J.tipo }** I2 **{ if (I2.tipo == bool) then: I.tipo = I2.tipo; I.ancho = I2.ancho; else: I.tipo = J.tipo; I.ancho = J.ancho }**

I2 -> > J **{ if(J.tipo != I2.tipo || J.tipo != int) then: error; then: I2'.tipo = J.tipo; }** I2' **{ I2.tipo = bool; I2.ancho = 1; }**

I2 -> >= J **{ if(J.tipo != I2.tipo || J.tipo != int) then: error; then: I2'.tipo = J.tipo; }** I2' **{ I2.tipo = bool; I2.ancho = 1; }**

I2 -> < J **{ if(J.tipo != I2.tipo || J.tipo != int) then: error; then: I2'.tipo = J.tipo; }** I2' **{ I2.tipo = bool; I2.ancho = 1; }**

I2 -> <= J **{ if(J.tipo != I2.tipo || J.tipo != int) then: error; then: I2'.tipo = J.tipo; }}** I2' **{ I2.tipo = bool; I2.ancho = 1; }**

I2 -> lambda **{ I2.tipo = Tipo_OK }**

J -> M J2 **{ if (M.tipo == J2.tipo || J2.tipo == Tipo_OK) then: J.tipo = M.tipo; J.ancho = M.ancho; }**

J2 -> + M J2' **{ if (M.tipo == int && (J2'.tipo == int || J2'.tipo == Tipo_OK)) then: J2.tipo = M.tipo; J2.ancho = M.ancho; else: error; }**

J2 -> - M J2' **{ if (M.tipo == int && (J2'.tipo == int || J2'.tipo == Tipo_OK)) then: J2.tipo = M.tipo; J2.ancho = M.ancho; else: error; }**

J2 -> lambda **{ J2.tipo = Tipo_OK }**

M -> V M2 **{ if (V.tipo == M2.tipo || M2.tipo == Tipo_OK) then: M.tipo = V.tipo; M.ancho = V.ancho; }**

M2 -> * V M2' **{ if (V.tipo == int && (M2'.tipo == int || M2'.tipo == Tipo_OK)) then: M2.tipo = V.tipo; M2.ancho = V.ancho; else: error; }**

M2 -> / V M2' **{ if (V.tipo == int && (M2'.tipo == int || M2'.tipo == Tipo_OK)) then: M2.tipo = V.tipo; M2.ancho = V.ancho; else: error; }**

M2 -> % V M2' **{ if (V.tipo == int && (M2'.tipo == int || M2'.tipo == Tipo_OK)) then: M2.tipo = V.tipo; M2.ancho = V.ancho; else: error; }**

M2 -> lambda **{ M2.tipo = Tipo_OK }**

V -> identificador **{ V2.entrada = identificador.entrada }** V2 **{ V.tipo = tipoID(identificador.entrada) }**

V -> entero **{ V.tipo = int; V.ancho = 8; }**

V -> cadena **{ V.tipo = char; V.ancho = 8 }**

V -> true **{ V.tipo = bool; V.ancho = 1 }**

V -> false **{ V.tipo = bool; V.ancho = 1 }**

V -> (E) **{ V.tipo = E.tipo; V.ancho = E.ancho }**

V -> INCDEC identificador **{ if (tipoId(identificador.entrada) != int) then: error; else: V.tipo = int;}**

V -> + V' **{ if (V'.tipo == int) then: V.tipo = int; V.ancho = 8; else: error }**

V -> - V' **{ if (V'.tipo == int) then: V.tipo = int; V.ancho = 8; else: error }**

V -> ! V' **{ if (V'.tipo == bool) then: V.tipo = bool; V.ancho = 1; else: error }**

V2 -> (L **{ if(!paramValidos(V2.entrada, L.param)) then: error; }**) **{ V2.tipo = tipoRetorno(V2.entrada); if(V2.tipo == int || V2.tipo == chars) then: V2.ancho = 8; else: if(V2.tipo==bool) then: V2.ancho = 1; }**

V2 -> lambda **{ V2.tipo = Tipo_OK; }**

V2 -> INCDEC **{ V2.tipo = int; V2.ancho = 8; }**

L -> E Q **{ L.param = E.tipo x Q.param }**

L -> lambda **{ L.param = [] }**

L2 -> E Q **{ L.param = E.tipo x Q.param }**

Q -> , E Q' **{ Q.param = E.tipo x Q'.param }**

Q -> lambda **{ Q.param = [] }**

S -> identificador **{ if (tipoId(identificador.entrada) == Tipo_Vacio) then: aniadirTS(identificador.entrada, int, 8, false); desp += 8; } { S2.entrada = identificador.entrada }** S2 **{ S.tipoReturn = Tipo_OK }**

S -> INCDEC identificador **{ if (tipoId(identificador.entrada) != int) then: if (tipoId(identificador.entrada) == Tipo_Vacio) then:**

aniadirTS(identificador.entrada, int, 8, false); else: error; } { S.tipoReturn = Tipo_OK }

S -> **{ if (!S.return) then: error; }** return X **{ S.tipoReturn = X.tipoReturn }**

S -> write (L2) **{ S.tipoReturn = Tipo_OK }**

S -> prompt (identificador) **{ if (tipoId(identificador.entrada) != int && tipoId(identificador.entrada) != bool) then: error; else: S.tipoReturn = Tipo_OK }**

S -> **{ if (!break) then: error; }** break **{ S.tipoReturn = Tipo_OK }**

S2 -> ASIGNACION E **{ if ((ASIGNACION.tipo != E.tipo || ASIGNACION.tipo != tipoID(S2.entrada)) && (ASIGNACION.tipo != tipoID(S2.entrada) || ASINGACION.tipo != Tipo_OK)) then: error; }**

S2 -> (L **{ if(!paramValidos(S2.entrada, L.param)) then: error; }**)

S2 -> INCDEC **{ if(tipoID(S2.entrada) != int) then: error; }**

X -> E **{ X.tipoReturn = E.tipo }**

X -> lambda **{ X.tipoReturn = Tipo_Vacio }**

F -> function **{ zonaDeclaracion = true; C.return = true; C.break = false; }** H identificador **{ crearTS(); }** (A) **{ zonaDeclaracion = false; actualizarFuncionTS(identificador.entrada, H.tipo, A.param, A.modo, etiquetaFuncion()); }** iniBloq C **{ if (C.tipoReturn != H.tipo && C.tipoReturn != Tipo_OK) then: error; }** endBloq **{ cerrarTSActiva(); }**

A -> T identificador **{ aniadirTS(identificador.entrada, T.tipo, desp, false); desp += T.ancho; }** K **{ A.param = T.tipo x K.param; A.modo = false x K.modo; }**

A -> lambda **{ A.param = null; A.modo = null; }**

K -> , T identificador **{ aniadirParam(identificador.entrada, T.tipo, desp); desp += T.ancho; }** K' **{ K.param = T.tipo x K'.param; K.modo = false x K'.modo; }**

K -> lambda **{ K.param = null; K.modo = null; }**

C -> **{ B.return = C.return; B.break = C.break; }** B **{ C'.return = C.return; C'.break = C.break; }** C' **{ if (B.tipoReturn == C'.tipoReturn || C'.tipoReturn == Tipo_OK) then: C.tipoReturn=B.tipoReturn; else: error; }**

C -> lambda **{ C.tipo = Tipo_OK; }**

H -> T **{ H.tipo = T.tipo; }**

H -> lambda **{ H.tipo = Tipo_Vacio }**

4.3.2 Errores

El analizador semántico detecta errores cuando el programa fuente no respeta las reglas semánticas del lenguaje.

Los errores detectados por este analizador son:

1. Los tipos no son consistentes, por ejemplo, cuando a una variable de tipo int se le asigna un valor booleano. Otro ejemplo puede ser cuando los tipos de los parámetros pasados a una función no corresponden con lo que espera la función.
2. Se pone un break fuera de un switch.
3. Se pone un return fuera de una función.

4.4 TABLA DE SÍMBOLOS

La tabla de símbolos es una estructura de datos en la que se almacena toda la información sobre los identificadores que aparecen en el código fuente, para su uso en las distintas fases del compilador.

El lenguaje JavaScript-PL no tiene anidamiento, es decir, no permite que se declaren unas funciones dentro de otras. Por simplicidad de la implementación, se ha considerado que hay dos tipos de tabla de símbolos: uno para el ámbito global y otro para el local. La tabla de símbolos del ámbito global existe durante toda la ejecución del analizador, mientras que las tablas de ámbito local se van creando y destruyendo según se necesite.

En esta implementación solo puede haber como máximo 2 tablas creadas al mismo tiempo, para identificar a que tabla pertenece un identificador los índices de identificador positivos pertenecen a la tabla global y los negativos a una tabla local.

4.4.1 Tabla global

Se crea al comienzo del análisis y se va rellenando con la información correspondiente a todas las variables globales y a todas las funciones que se declaran. A esta tabla de símbolos se puede acceder desde cualquier punto del programa incluso si se está dentro de una función.

4.4.2 Tabla local

Se crea cuando se define una función y se va rellenando con los parámetros y las variables de la función. Como el lenguaje no permite la declaración de funciones anidadas, esta tabla de símbolos no puede contener entradas correspondientes a identificadores de tipo función. A una tabla local no se podrá acceder desde zonas de código exteriores a la función en sí y será eliminada al acabar de ser usada (es decir, al alcanzar el final de la declaración de la función).

4.4.3 Organización de la tabla de símbolos

La tabla de símbolos contendrá una entrada por cada identificador encontrado en el programa. Para cada entrada, se almacenarán diversos atributos. Dependiendo del tipo del identificador se tendrá un número distinto de atributos con distintos valores.

Tipos de identificadores

1. Variables y parámetros de función

   Los identificadores de tipo variable y parámetro contendrán 3 campos

   1. Lexema: Lexema del identificador
   2. Tipo: Tipo del dato, puede ser chars, int o bool
   3. Despl: Desplazamiento en memoria
   4. Parámetro: Tendrá un 0 si la entrada corresponde a una variable del programa o a un parámetro que se pasa por valor; tendrá un 1 si la entrada corresponde a un parámetro que se pasa por referencia (aunque el lenguaje fuente, JavaSricpt-PL, no tiene paso por referencia, se ha añadido este atributo en el diseño de la tabla de símbolos para futuros cambios en el lenguaje fuente)

2. Funciones

   Las entradas correspondientes a identificadores de tipo función contendrán un número variable de atributo dependiendo del número de parámetros que necesite.

   1. Lexema: Lexema de la función
   2. Tipo: Tipo del lexema que en este caso será función
   3. NumParam: Numero de parámetros que tiene la función
   4. TipoParam: Un campo por cada parámetro el cual contendrá su tipo
   5. ModParam: Un campo por cada parámetro el cual contendrá si se va a pasar por referencia o no.
   6. TipoRetorno: Tipo de retorno de la función pudiendo ser void además del resto de tipos
   7. EtiqFuncion: Etiqueta de la primera instrucción del código ejecutable de la función

4.4.4 Errores

La tabla de símbolos detecta cuando se intenta asignar atributos inválidos para el tipo de identificador que se está modificando. Aunque estos errores son identificados por el analizador semántico, estas últimas comprobaciones se realizan por posibles fallos en el analizador dados por una mala modificación de los alumnos de Compiladores. Esto es por lo que los errores serán codificados con un código menor que 2000 reservados para los errores del analizador.

4.5 GESTOR DE ERRORES

El gestor de errores es llamado por cualquiera de los tres analizadores en caso de que detecten algún error, también será llamado si hay algún error en la ejecución del compilador. Le pasan como parámetro el código asignado al error. El gestor de errores se encarga de sacar un mensaje de error y de parar la compilación.

El gestor de errores recibirá el error encontrado codificado con un número de 1000 a 5000:

4.5.1 Errores del programa.

Los códigos del 1000 al 1999 están reservados para errores del analizador y no errores detectados en el programa fuente, estas errores corresponden a errores de mala utilización del programa y errores que podrían originar los alumnos de Compiladores al modificar el código fuente, por ejemplo hay errores de la tabla de símbolos que deberían de ser detectados por el semántico, pero si alguna modificación rompe el semántico la tabla de símbolos sería capaz de detectarlo y lanzar un error del 1005 al 1009:

1. No se han recibido suficientes argumentos para la ejecución del compilador.
2. No se ha podido abrir el archivo que contiene el programa a analizar.
3. Ha ocurrido un error al leer o escribir archivos.
4. Ha ocurrido un error con el índice de la tabla de símbolos.
5. Se intenta insertar un atributo invalido en la tabla de símbolos (Que no sea tipo, desplazamiento, parámetro, retorno o etiqueta)
6. Se intenta insertar un tipo invalido en la tabla de símbolos (Que no sea int, chars, bool o func)
7. Se ha intentado insertar un desplazamiento a una función en la tabla de símbolos
8. Se ha intentado asignar un atributo invalido de función a una variable en la tabla de símbolos (Parámetro, retorno o etiqueta)
9. El desplazamiento que se intenta añadir a una variable no es un int.

4.5.2 Errores detectados por el analizador léxico.

Los códigos del 2000 al 2999 están reservados para errores léxicos:

1. Entero fuera de rango: Valor numérico máximo de 32767 sobrepasado.
2. Intento de declaración de variable con nombre ya usado en la tabla de símbolos activa (esto es en realidad un error semántico, pero, puesto que nuestro analizador léxico se encarga de insertar los lexemas en la tabla de símbolos, es código insertado dentro del analizador léxico el que hace esta comprobación).
3. Se ha leído un carácter que no pertenece al conjunto de caracteres válidos del lenguaje
4. No se esperaba el carácter en la formación de un token
5. No se esperaba un carácter no imprimible
6. Se esperaba la inicialización de un número hexadecimal, pero se ha encontrado otro carácter.
7. Se esperaba una n o una t para formar \n o \t pero se ha recibido otro carácter
8. Se esperaba un & o un = para formar && o &= pero se ha recibido otro carácter
9. Se esperaba un | o un = para formar || o |= pero se ha recibido otro carácter

4.5.3 Errores detectados por el analizador sintáctico.

Los códigos del 3000 al 3999 están reservados para errores sintácticos:

1. Que en la raíz de un programa no se encuentre una sentencia valida ya sean declaración o modificación de variables, llamadas o declaración de funciones o iniciación de algún bloque de código if, for, switch, while o do-while.
2. Que la parte de actualización de la declaración de un bucle for no sea correcta: tiene que ser una asignación, un pre o post incremento o un pre o post decremento.
3. Se ha recibido un tipo distinto a int, bool o chars.
4. La expresión que se ha leído es inválida.
5. La sentencia que se ha leído es inválida.
6. La declaración de una función es incorrecta.
7. Se ha detectado un token no esperado.
8. No se ponen parámetros en un write

4.5.4 Errores detectados por el analizador semántico.

Los códigos del 4000 al 4999 están reservados para errores semánticos:

1. Los tipos no son consistentes, por ejemplo, cuando a una variable de tipo int se le asigna un valor booleano. Otro ejemplo puede ser cuando los tipos de los parámetros pasados a una función no corresponden con lo que espera la función.
2. Se pone un break fuera de un switch.
3. Se pone un return fuera de una función.

4.6 PRUEBAS

Como parte de este Trabajo de Fin de Grado, se han generado unas Pruebas, usando para ello la librería de Junit 5 de java. Se ha probado cada analizador por separado teniendo en cuenta los errores que puede dar.

4.6.1 Pruebas del analizador léxico

Hay que tener en cuenta que como no se ejecutan el resto de los analizadores el id de la tabla de símbolos de los identificadores no es correcto, es solamente secuencial.

Pruebas con programas correctos

Este conjunto de pruebas se ha realizado con programas que no contienen errores, para comprobar si los tokens se detectan bien, el programa pasara las pruebas si la salida coincide con la esperada.

1. El analizador léxico es capaz de detectar todos los tipos de tokens:

   Este test comprueba que el analizador es capaz de reconocer todos los tipos de tokens

   - Programa que analizar (Escrito en una sola línea):

```
+ - * / % == != < > <= >= && || ! ++ -- = += -= *= /= %= &= |= 123 0123
0x12AF "Cadena\n" 'Cadena\n' true false var int bool chars write prompt
```

```
return if else while do for switch case break default function ( ) { } , ;
: identificador
```

- Salida esperada (En una sola columna):

```
<1, null>
<2, null>
<3, null>
<4, null>
<5, null>
<6, null>
<7, null>
<8, null>
<9, null>
<10, null>
<11, null>
<12, null>
<13, null>
<14, null>
<15, null>
<16, null>
<17, null>
<18, null>
<19, null>
<20, null>
<21, null>
<22, null>
<23, null>
<24, null>
<25, 123>
<25, 83>
<25, 4783>
<26, Cadena\n>
<26, Cadena\n>
<27, null>
<28, null>
<29, null>
<30, null>
<31, null>
<32, null>
<33, null>
<34, null>
<35, null>
<36, null>
<37, null>
<38, null>
<39, null>
<40, null>
<41, null>
<42, null>
<43, null>
<44, null>
<45, null>
<46, null>
<47, null>
<48, null>
<49, null>
<50, null>
<51, null>
<52, null>
<54, 1>
<53, null>
```

2. El analizador léxico es capaz de detectar los tokens de un programa sencillo:

Este test se encarga de comprobar que el analizador léxico es capaz de identificar correctamente los tokens de un programa secuencial.

- Programa que analizar (Los saltos de línea pueden son ser los reales por espacio):

```
/* Programa de ejemplo  */
/******* José Luis Fuertes, 5, enero, 2018 *********/
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que
no hay que implementar */

var chars s; /* variable global cadena */

var int For, Do, While;   // tres variables globales

// Parte del programa principal:
s = "El factorial ";      // Primera sentencia que se ejecutaría

write (s);
write ("\nIntroduce un 'número'.");
prompt (num);       /* se lee un número del teclado y se guarda en la
variable global num */

switch (num);
{
       case 1:
       case 0: write ("El factorial de ", num, " siempre es 1.\n"); break;
       default:
              if (num < 0)
              {
                     write ('No existe el factorial de un negativo.\n');
              }
```

```
            else
            {
                  s = "otro texto";
            }
}
/* esto constituye la llamada a una función sin argumentos.
Es en este instante cuando se llama a esta función y, por tanto,
cuando se ejecuta todo el código de dicha función */
```

- Salida esperada (En una sola columna pueden aparecer salto de línea por espacio):

```
<29, null>
<32, null>
<54, 1>
<51, null>
<29, null>
<30, null>
<54, 2>
<50, null>
<54, 3>
<50, null>
<54, 4>
<51, null>
<54, 1>
<17, null>
<26, El factorial >
<51, null>
<33, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<33, null>
<46, null>
<26, \nIntroduce un
'número'.>
<47, null>
<51, null>
<34, null>
<46, null>
<54, 5>
<47, null>
<51, null>
<41, null>
<46, null>
<54, 5>
<47, null>
<51, null>
<48, null>
<42, null>
<25, 1>
<52, null>
<42, null>
<25, 0>
<52, null>
<33, null>
<46, null>
<26, El factorial de
>
<50, null>
<54, 5>
<50, null>
<26,  siempre es
1.\n>
<47, null>
<51, null>
<43, null>
<51, null>
<44, null>
<52, null>
<36, null>
<46, null>
<54, 5>
<8, null>
<25, 0>
<47, null>
<48, null>
<33, null>
<46, null>
<26, No existe el
factorial de un
negativo.\n>
<47, null>
<51, null>
<49, null>
<37, null>
<48, null>
<54, 1>
<17, null>
<26, otro texto>
<51, null>
<49, null>
<49, null>
<53, null>
```

3. El analizador léxico es capaz de detectar los tokens de un programa con bucles:

Este test se encarga de comprobar que el analizador es capaz de detectar los tokens de un programa con bucles

- Programa que analizar (Los saltos de línea pueden son ser los reales por espacio):

```
/* Programa de ejemplo  */
/******* José Luis Fuertes, 5, enero, 2018 *********/
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que
no hay que implementar */

var chars s; /* variable global cadena */
```

```
var int factorial = 1;    // variable local inicializada a uno
do
{
      factorial *= n--;  // equivale a: factorial = factorial * n; n = n -
1;
} while (n);       // mientras n no sea 0


var int factorial = 1, i; // variables locales: factorial inicializada a 1
e i inicializada a 0 por omisión
while (i < num)                 // num es variable global entera sin
declarar
{
      factorial *= ++i;  // equivale a: i = i + 1; factorial = factorial *
i;
}

var int i, factorial = 1; /* variables locales */
for (i = 1; i <= n; i++)
{
      factorial *= i;
}

var int For, Do, While;   // tres variables globales

// Parte del programa principal:
s = "El factorial ";      // Primera sentencia que se ejecutaría

write (s);
write ("\nIntroduce un 'número'.");
prompt (num);      /* se lee un número del teclado y se guarda en la
variable global num */

switch (num);
{
      case 1:
      case 0: write ("El factorial de ", num, " siempre es 1.\n"); break;
      default:
            if (num < 0)
            {
                  write ('No existe el factorial de un negativo.\n');
            }
            else
            {
                  s = "otro texto"
            }
}
/* esto constituye la llamada a una función sin argumentos.
Es en este instante cuando se llama a esta función y, por tanto,
cuando se ejecuta todo el código de dicha función */
```

- Salida esperada (En una sola columna pueden aparecer salto de línea por espacio):

```
<29, null>                <30, null>                <39, null>
<32, null>                <54, 2>                   <48, null>
<54, 1>                   <17, null>                <54, 2>
<51, null>                <25, 1>                   <20, null>
<29, null>                <51, null>                <54, 3>
```

```
<16, null>
<51, null>
<49, null>
<38, null>
<46, null>
<54, 3>
<47, null>
<51, null>
<29, null>
<30, null>
<54, 2>
<17, null>
<25, 1>
<50, null>
<54, 4>
<51, null>
<38, null>
<46, null>
<54, 4>
<8, null>
<54, 5>
<47, null>
<48, null>
<54, 2>
<20, null>
<15, null>
<54, 4>
<51, null>
<49, null>
<29, null>
<30, null>
<54, 4>
<50, null>
<54, 2>
<17, null>
<25, 1>
<51, null>
<40, null>
<46, null>
<54, 4>
<17, null>
<25, 1>
<51, null>
<54, 4>
<10, null>
<54, 3>
<51, null>
<54, 4>
<15, null>
<47, null>
<48, null>
<54, 2>
<20, null>
<54, 4>
<51, null>
<49, null>
<29, null>
<30, null>
<54, 6>
<50, null>
<54, 7>
<50, null>
<54, 8>
<51, null>
<54, 1>
<17, null>
<26, El factorial >
<51, null>
<33, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<33, null>
<46, null>
<26, \nIntroduce un
'número'.>
<47, null>
<51, null>
<34, null>
<46, null>
<54, 5>
<47, null>
<51, null>
<41, null>
<46, null>
<54, 5>
<47, null>
<51, null>
<48, null>
<42, null>
<25, 1>
<52, null>
<42, null>
<25, 0>
<52, null>
<33, null>
<46, null>
<26, El factorial de
>
<50, null>
<54, 5>
<50, null>
<26,  siempre es
1.\n>
<47, null>
<51, null>
<43, null>
<51, null>
<44, null>
<52, null>
<36, null>
<46, null>
<54, 5>
<8, null>
<25, 0>
<47, null>
<48, null>
<33, null>
<46, null>
<26, No existe el
factorial de un
negativo.\n>
<47, null>
<51, null>
<49, null>
<37, null>
<48, null>
<54, 1>
<17, null>
<26, otro texto>
<49, null>
<49, null>
<53, null>
```

4. El analizador léxico es capaz de detectar los tokens de un programa con funciones:

Este test se encarga de comprobar que el analizador es capaz de detectar los tokens de un programa con bucles

- Programa que analizar (Los saltos de línea pueden son ser los reales por espacio):

```
/* Programa de ejemplo  */
/******* José Luis Fuertes, 5, enero, 2018 *********/
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que
no hay que implementar */
```

```
var chars s; /* variable global cadena */

function int FactorialRecursivo (int n)      /* n: parámetro formal de la
función entera */
{
      if (n == 0)  return 1;
      return n * FactorialRecursivo (n - 1); /* llamada recursiva */
}

function int FactorialDo (int n)
{
      var int factorial = 1;    // variable local inicializada a uno
      do
      {
            factorial *= n--;  // equivale a: factorial = factorial * n; n
= n - 1;
      } while (n);       // mientras n no sea 0
      return factorial;  // devuelve el valor entero de la variable
factorial
}

function int FactorialWhile ()
{
      var int factorial = 1, i; // variables locales: factorial
inicializada a 1 e i inicializada a 0 por omisión
      while (i < num)                 // num es variable global entera sin
declarar
      {
            factorial *= ++i;  // equivale a: i = i + 1; factorial =
factorial * i;
      }
      return factorial;
}

function int FactorialFor (int n)
{
      var int i, factorial = 1; /* variables locales */
      for (i = 1; i <= n; i++)
      {
            factorial *= i;
      }
      return factorial;
}

var int For, Do, While;   // tres variables globales

function imprime (chars s, chars msg, int f) /* función que recibe 3
argumentos */
{
      write (s); write (msg); write (f);
      write ("\n");      // imprime un salto de línea */
      return;      /* finaliza la ejecución de la función (en este caso,
se podría omitir) */
}

function chars cadena (bool log)
{
      if (!log)
```

```
	{
		return s;
	}
	else
	{
		return "Fin";
	}
}	// fin cadena: función que devuelve una cadena

// Parte del programa principal:
s = "El factorial ";	// Primera sentencia que se ejecutaría

write (s);
write ("\nIntroduce un 'número'.");
prompt (num);	/* se lee un número del teclado y se guarda en la
variable global num */

switch (num);
{
	case 1:
	case 0: write ("El factorial de ", num, " siempre es 1.\n"); break;
	default:
		if (num < 0)
		{
			write ('No existe el factorial de un negativo.\n');
		}
		else
		{
			For = FactorialFor (num);
			While = FactorialWhile ();
			Do = FactorialDo (num);
			imprime (cadena (false), "recursivo es: ",
FactorialRecursivo (num));
			imprime (s, "con do-while es: ", Do);
			imprime (s, "con while es: ", While);
			imprime (cadena (false), "con for es: ", For);
		}
}

function bool bisiesto (int a)
{
	return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);	//se
tienen en cuenta la precedencia de operadores
} // fin de bisiesto: función lógica

function int dias (int m, int a)
{
	switch (m)
	{
		case 1: case 3: case 5: case 7: case 8: case 10: case 12:
			return 31; break;
		case 4: case 6: case 9: case 11:
			return 30;
		case 2: return 1;
			break;
		default: return 0;
	}
} // fin de dias. Todos los return devuelven un entero y la función es
entera
```

```
function bool esFechaCorrecta (int d, int m, int a)
{
	return m>=1 && m<=12 && d>=1 && d <= dias (m, a);
} //fin de esFechaCorrecta

function imprime2 (int v, int w)
{
	write (v + w, "\n");
} //fin de imprime2

function potencia (int z, int dim)
{
	var int s;   // Oculta a la global
	for (s=0; s < dim; s++);
	{
		z *= z;
		imprime ("Potencia:", " ", z);
	}
} // fin de potencia: función que no devuelve nada

function demo ()   /* definición de la función demo, sin argumentos y que
no devuelve nada */
{
	var int i;   // Variables locales
	var int v0, v1, v2, v3, zv;
	var chars s; // Oculta a la s global

	write ('Escriba "tres" números: ');
	prompt (v1); prompt (v2); prompt (v3);

	if (!((v1 == v2) && (v1 != v3)))/* NOT ((v1 igual a v2) AND (v1
distinto de v3))  */
	{
		write ('Escriba su nombre: ');
		prompt (s);
		v0 = v3;     /* si v2<v3, v0=v2; en otro caso v0=v3 */
		write (s);
	}
	s = "El primer valor era ";
	if (v1 != 0)
	{
		write (s, v1, ".\n");
	}
	else
	{
		write (s, 0, ".\n");      // imprime la cadena `El primer
valor era 0.\n´
	}

	potencia (v0, 4);
	for (i=1; i <= 10; ++i);
	{
		zv+=i;
	}
	potencia (zv, 5);
	imprime2 (i, num);
	imprime ("", cadena(true), 666);
}
```

```
demo();
/* esto constituye la llamada a una función sin argumentos.
Es en este instante cuando se llama a esta función y, por tanto,
cuando se ejecuta todo el código de dicha función */
```

- Salida esperada (En una sola columna pueden aparecer salto de línea por espacio):

```
<29, null>
<32, null>
<54, 1>
<51, null>
<45, null>
<30, null>
<54, 2>
<46, null>
<30, null>
<54, 3>
<47, null>
<48, null>
<36, null>
<46, null>
<54, 3>
<6, null>
<25, 0>
<47, null>
<35, null>
<25, 1>
<51, null>
<35, null>
<54, 3>
<3, null>
<54, 2>
<46, null>
<54, 3>
<2, null>
<25, 1>
<47, null>
<51, null>
<49, null>
<45, null>
<30, null>
<54, 4>
<46, null>
<30, null>
<54, 3>
<47, null>
<48, null>
<29, null>
<30, null>
<54, 5>
<17, null>
<25, 1>
<51, null>
<39, null>
<48, null>
<54, 5>
<20, null>
<54, 3>
<16, null>
<51, null>
<49, null>
<38, null>
<46, null>
<54, 3>
<47, null>
<51, null>
<35, null>
<54, 5>
<51, null>
<49, null>
<45, null>
<30, null>
<54, 6>
<46, null>
<47, null>
<48, null>
<29, null>
<30, null>
<54, 5>
<17, null>
<25, 1>
<50, null>
<54, 7>
<8, null>
<54, 8>
<47, null>
<48, null>
<54, 5>
<20, null>
<15, null>
<54, 7>
<51, null>
<49, null>
<35, null>
<54, 5>
<51, null>
<49, null>
<45, null>
<30, null>
<54, 9>
<46, null>
<30, null>
<54, 3>
<47, null>
<48, null>
<29, null>
<30, null>
<54, 7>
<50, null>
<54, 5>
<17, null>
<25, 1>
<51, null>
<40, null>
<46, null>
<54, 7>
<17, null>
<25, 1>
<51, null>
<54, 7>
<10, null>
<54, 3>
<51, null>
<54, 7>
<15, null>
<47, null>
<48, null>
<54, 5>
<20, null>
<54, 7>
<51, null>
<49, null>
<35, null>
<54, 5>
<51, null>
<49, null>
<29, null>
<30, null>
<54, 10>
<50, null>
<54, 11>
<50, null>
<54, 12>
<51, null>
<45, null>
<54, 13>
<46, null>
<32, null>
<54, 1>
<50, null>
<32, null>
<54, 14>
<50, null>
<30, null>
<54, 15>
<47, null>
```

```
<48, null>
<33, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<33, null>
<46, null>
<54, 14>
<47, null>
<51, null>
<33, null>
<46, null>
<54, 15>
<47, null>
<51, null>
<33, null>
<46, null>
<26, \n>
<47, null>
<51, null>
<35, null>
<51, null>
<49, null>
<45, null>
<32, null>
<54, 16>
<46, null>
<31, null>
<54, 17>
<47, null>
<48, null>
<36, null>
<46, null>
<14, null>
<54, 17>
<47, null>
<48, null>
<35, null>
<54, 1>
<51, null>
<49, null>
<37, null>
<48, null>
<35, null>
<26, Fin>
<51, null>
<49, null>
<49, null>
<54, 1>
<17, null>
<26, El factorial >
<51, null>
<33, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<33, null>
<46, null>
<26, \nIntroduce un
'número'.>
<47, null>
<51, null>
<34, null>
<46, null>
<54, 8>
<47, null>
<51, null>
<41, null>
<46, null>
<54, 8>
<47, null>
<51, null>
<48, null>
<42, null>
<25, 1>
<52, null>
<42, null>
<25, 0>
<52, null>
<33, null>
<46, null>
<26, El factorial de
>
<50, null>
<54, 8>
<50, null>
<26,  siempre es
1.\n>
<47, null>
<51, null>
<43, null>
<51, null>
<44, null>
<52, null>
<36, null>
<46, null>
<54, 8>
<8, null>
<25, 0>
<47, null>
<48, null>
<33, null>
<46, null>
<26, No existe el
factorial de un
negativo.\n>
<47, null>
<51, null>
<49, null>
<37, null>
<48, null>
<54, 10>
<17, null>
<54, 9>
<46, null>
<54, 8>
<47, null>
<51, null>
<26, \nIntroduce un
'número'.>
<47, null>
<51, null>
<54, 12>
<17, null>
<54, 6>
<46, null>
<47, null>
<51, null>
<54, 11>
<17, null>
<54, 4>
<46, null>
<54, 8>
<47, null>
<51, null>
<54, 13>
<46, null>
<54, 16>
<46, null>
<28, null>
<47, null>
<50, null>
<26, recursivo es: >
<50, null>
<54, 2>
<46, null>
<54, 8>
<47, null>
<47, null>
<51, null>
<54, 13>
<46, null>
<54, 1>
<50, null>
<26, con do-while es:
>
<50, null>
<54, 11>
<47, null>
<51, null>
<54, 13>
<46, null>
<54, 1>
<50, null>
<26, con while es: >
<50, null>
<54, 12>
<47, null>
<51, null>
<54, 13>
<46, null>
<54, 16>
<46, null>
<28, null>
<47, null>
<50, null>
<26, con for es: >
<50, null>
<54, 10>
```

<47, null>
<51, null>
<49, null>
<49, null>
<45, null>
<31, null>
<54, 18>
<46, null>
<30, null>
<54, 19>
<47, null>
<48, null>
<35, null>
<46, null>
<54, 19>
<5, null>
<25, 4>
<6, null>
<25, 0>
<12, null>
<54, 19>
<5, null>
<25, 100>
<7, null>
<25, 0>
<13, null>
<54, 19>
<5, null>
<25, 400>
<6, null>
<25, 0>
<47, null>
<51, null>
<49, null>
<45, null>
<30, null>
<54, 20>
<46, null>
<30, null>
<54, 21>
<50, null>
<30, null>
<54, 19>
<47, null>
<48, null>
<41, null>
<46, null>
<54, 21>
<47, null>
<48, null>
<42, null>
<25, 1>
<52, null>
<42, null>
<25, 3>
<52, null>
<42, null>
<25, 5>
<52, null>
<42, null>
<25, 7>
<52, null>
<42, null>
<25, 8>
<52, null>
<42, null>
<25, 10>
<52, null>
<42, null>
<25, 12>
<52, null>
<35, null>
<25, 31>
<51, null>
<43, null>
<51, null>
<42, null>
<25, 4>
<52, null>
<42, null>
<25, 6>
<52, null>
<42, null>
<25, 9>
<52, null>
<42, null>
<25, 11>
<52, null>
<35, null>
<25, 30>
<51, null>
<42, null>
<25, 2>
<52, null>
<35, null>
<25, 1>
<51, null>
<43, null>
<51, null>
<44, null>
<52, null>
<35, null>
<25, 0>
<51, null>
<49, null>
<49, null>
<45, null>
<31, null>
<54, 22>
<46, null>
<30, null>
<54, 23>
<50, null>
<30, null>
<54, 21>
<50, null>
<30, null>
<54, 19>
<47, null>
<48, null>
<35, null>
<54, 21>
<11, null>
<25, 1>
<12, null>
<54, 21>
<10, null>
<25, 12>
<12, null>
<54, 23>
<11, null>
<25, 1>
<12, null>
<54, 23>
<10, null>
<54, 20>
<46, null>
<54, 21>
<50, null>
<54, 19>
<47, null>
<51, null>
<49, null>
<45, null>
<54, 24>
<46, null>
<30, null>
<54, 25>
<50, null>
<30, null>
<54, 26>
<47, null>
<48, null>
<33, null>
<46, null>
<54, 25>
<1, null>
<54, 26>
<50, null>
<26, \n>
<47, null>
<51, null>
<49, null>
<45, null>
<54, 27>
<46, null>
<30, null>
<54, 28>
<50, null>
<30, null>
<54, 29>
<47, null>
<48, null>
<29, null>
<30, null>
<54, 1>
<51, null>

```
<40, null>
<46, null>
<54, 1>
<17, null>
<25, 0>
<51, null>
<54, 1>
<8, null>
<54, 29>
<51, null>
<54, 1>
<15, null>
<47, null>
<51, null>
<48, null>
<54, 28>
<20, null>
<54, 28>
<51, null>
<54, 13>
<46, null>
<26, Potencia:>
<50, null>
<26,  >
<50, null>
<54, 28>
<47, null>
<51, null>
<49, null>
<49, null>
<45, null>
<54, 30>
<46, null>
<47, null>
<48, null>
<29, null>
<30, null>
<54, 7>
<51, null>
<29, null>
<30, null>
<54, 31>
<50, null>
<54, 32>
<50, null>
<54, 33>
<50, null>
<54, 34>
<50, null>
<54, 35>
<51, null>
<29, null>
<32, null>
<54, 1>
<51, null>
<33, null>
<46, null>
<26, Escriba "tres"
números: >
<47, null>
<51, null>
<34, null>
<46, null>
<54, 32>
<47, null>
<51, null>
<34, null>
<46, null>
<54, 33>
<47, null>
<51, null>
<34, null>
<46, null>
<54, 34>
<47, null>
<51, null>
<36, null>
<46, null>
<14, null>
<46, null>
<46, null>
<54, 32>
<6, null>
<54, 33>
<47, null>
<12, null>
<46, null>
<54, 32>
<7, null>
<54, 34>
<47, null>
<47, null>
<47, null>
<48, null>
<33, null>
<46, null>
<26, Escriba su
nombre: >
<47, null>
<51, null>
<34, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<54, 31>
<17, null>
<54, 34>
<51, null>
<33, null>
<46, null>
<54, 1>
<47, null>
<51, null>
<49, null>
<54, 1>
<17, null>
<26, El primer valor
era >
<51, null>
<36, null>
<46, null>
<54, 32>
<7, null>
<25, 0>
<47, null>
<48, null>
<33, null>
<46, null>
<54, 1>
<50, null>
<54, 32>
<50, null>
<26, .\n>
<47, null>
<51, null>
<49, null>
<37, null>
<48, null>
<33, null>
<46, null>
<54, 1>
<50, null>
<25, 0>
<50, null>
<26, .\n>
<47, null>
<51, null>
<49, null>
<54, 27>
<46, null>
<54, 31>
<50, null>
<25, 4>
<47, null>
<51, null>
<40, null>
<46, null>
<54, 7>
<17, null>
<25, 1>
<51, null>
<54, 7>
<10, null>
<25, 10>
<51, null>
<15, null>
<54, 7>
<47, null>
<51, null>
<48, null>
<54, 35>
<18, null>
<54, 7>
<51, null>
<49, null>
```

```
<54, 27>
<46, null>
<54, 35>
<50, null>
<25, 5>
<47, null>
<51, null>
<54, 24>
<46, null>
<54, 7>
<50, null>
<54, 8>
<47, null>
<51, null>
<54, 13>
<46, null>
<26, >
<50, null>
<54, 16>
<46, null>
<27, null>
<47, null>
<50, null>
<25, 666>
<47, null>
<51, null>
<49, null>
<54, 30>
<46, null>
<47, null>
<51, null>
<53, null>
```

Pruebas de detección de errores en programa fuente

Este conjunto de pruebas se hará con programas fuente incorrectos para comprobar que se detectan todos los errores léxicos.

1. Carácter no esperado en el programa fuente:

   En esta prueba se comprueba si el analizador léxico detecta un fallo cuando lee un carácter no esperado en un determinado instante (un carácter para el cual no hay una transición en ese estado), es decir comprueba que se detectan correctamente los errores 2004 del gestor de errores.

2. Se supera el valor numérico de 32767 (rango máximo de los enteros en el lenguaje)
3. Carácter no esperado en un programa con solo uno de estos caracteres _, \, á, é, í, ó, ú, se deberían de probar cualquier carácter imprimible, pero el conjunto es demasiado grande como para probar todos los caracteres, con este conjunto creo que se hace una estimación valida.
4. Se encuentra un EOF en un comentario de línea, de bloque o entre medias del */ que cierra el comentario de bloque
5. Se encuentra un salto de línea o un fin de fichero en medio de una cadena (para cadenas encerradas entre dobles comillas o entre comillas simples)
6. Se encuentra un carácter que no es un dígito hexadecimal cuando se espera la inicialización de un número hexadecimal, es decir después de leer un 0x
7. En una cadena se ha recibido un \ pero el siguiente carácter no es ni n ni t
8. El carácter posterior a & o | no es ni = ni otro & o | respectivamente

4.6.2 Pruebas del analizador sintáctico

Pruebas con programas correctos

Este conjunto de pruebas se ha realizado con programas que no contenían errores, para ver que el analizador sintáctico no detecta ninguno. Los programas fuente utilizados han sido el básico, el de con bucles y el de con funciones del analizador léxico.

Pruebas de detección de errores en el programa fuente

Los errores probados son:

1. Se ha encontrado un error en la base del programa.
2. Se ha encontrado un error en la actualización de la declaración de un for
3. Se detecta correctamente la falta de parámetros en un write
4. Se ha recibido un token no esperado

4.6.3 Pruebas del analizador semántico

Pruebas con programas correctos

Este conjunto de pruebas se ha realizado con programas que no contienen errores, para comprobar así que el analizador semántico no detecta ninguno. Los programas fuente que se han utilizado son el básico, el de con bucles y el de con funciones del analizador léxico.

Pruebas de detección de errores en el programa fuente

Los errores probados son:

1. Se detecta la inconsistencia en los tipos ya sea en los parámetros de la función, como en el tipo de retorno de las mismas, como errores en el tipo de asignación y modificación de variables, como que el tipo de las expresiones no sea el indicado en ciertas regiones de código como en la condición de un if el cual tiene que ser bool.
2. Se detecta el incorrecto posicionamiento de un break en distintas partes del código.
3. Se detecta el incorrecto posicionamiento de un return en distintas partes del código.

4.6.4 Pruebas de las tablas de símbolos

Pruebas de funcionamiento correcto

En este conjunto de pruebas se comprueba que la tabla de símbolos funciona correctamente con casos válidos como los siguientes:

1. Se actualiza una variable con los atributos correspondientes válidos
2. Se actualiza una entrada de función con los atributos correspondientes válidos
3. Se crea una tabla de símbolos local.
4. Se insertan variables y funciones con sus respectivos atributos en las dos tablas de símbolos, la local y la global, correctamente.

Pruebas de funcionamiento incorrecto

En este conjunto de pruebas se comprobará que la lógica de la tabla de símbolos detecta los errores que pueden ocurrir por una modificación errónea del analizador semántico:

1. Se intentan asignar atributos inválidos a una variable (Los correspondientes a una función)
2. Se intentan asignar atributos inválidos a una función (Los correspondientes a una variable)
3. Se intenta asignar un valor inválido a un atributo que sería válido (Por ejemplo, asignar al atributo tipo un numero)

Estas pruebas se harán tanto en la tabla global como en una tabla local que se creará.

5 CONCLUSIONES

En un proyecto de esta índole, enfocado a su uso por otros alumnos, la documentación es quizá la parte más importante. La creación de una documentación que sea perfectamente entendible es muy difícil y una de las partes que más tiempo ha llevado es la decisión de la terminología y la forma de comunicar ideas para que el proyecto sea lo más entendible posible, necesitando varias revisiones y modificaciones.

El objetivo del proyecto estaba claro desde el principio, pero a lo largo de su desarrollo se han ideado nuevos objetivos. Esto deja claro que en un proceso de desarrollo de este tamaño puede haber cambios y que la planificación necesita tener la capacidad de manejar imprevistos.

6 POSIBLES TRABAJOS FUTUROS

Posibles líneas de trabajo futuro se pueden centrar en mejorar el programa ya hecho en los siguientes campos:

1. Seguir el ejemplo de la creación del analizador léxico a partir de los ficheros que contienen la matriz de transición del autómata finito determinista, y plantear la creación de un analizador sintáctico a partir de una tabla como la que genera el programa SDGLL1 (Moreno Gomez, 2018). En esta parte habría que tener en cuenta el analizador semántico.
2. Incluir el resto de los tipos de analizador sintáctico para que se pueda ejecutar cualquiera mediante un argumento de programa. Esto podría llegar a ser bastante didáctico para ver los distintos funcionamientos de primera mano. En esta parte también habría que tener en cuenta el analizador semántico porque está muy ligado al sintáctico.
3. La inclusión de una vista grafica para mostrar las tablas de símbolos creadas, el árbol sintáctico, elegir el tipo de analizador sintáctico… En este apartado las posibilidades son muchas.

Como se puede apreciar el programa puede ser mejorado en gran medida y prácticamente sin límite.

7 BIBLIOGRAFÍA

Aho, A. V., Lam, M., Sethi, R., & Ullman, J. D. (2008). *Compiladores: Principios, Técnicas y Herramientas.* Addison-Wesley.

Almeida-Martínez, F. J., & Urquiza-Fuentes, J. (8-10 de Julio de 2009). *Universidad Politecnica de Cataluña.* Obtenido de https://upcommons.upc.edu/bitstream/handle/2099/7894/p199.pdf

Appel, A. W., & Palsberg, J. (2002). *Modern Compiler Implementation in Java.* 2ª ed. Cambridge University Press.

Cooper, K., & Torczon, L. (2002). *Engineering a Compiler.* Morgan Kaufmann.

ETSIINF. (15 de 05 de 2018). *http://www-lt.ls.fi.upm.es*. Obtenido de Procesadores de lenguajes : http://www-lt.ls.fi.upm.es/compiladores/procesadores/Herramientas.html

GNU Bison. (06 de Agosto de 2014). *Gnu Bison*. Obtenido de https://www.gnu.org/software/bison/

Grune, D., Bal, H., Jacobs, C., & Langendoen, K. (2000). *Modern Compiler Design.* John Wiley & Sons.

Kakde, O. G. (2002). *Algorithms for Compiler Design.* Charles River Media.

Moreno Gomez, J. L. (31 de 05 de 2018). *Procesadores de lenguajes - Herramientas*. Obtenido de http://www-lt.ls.fi.upm.es/compiladores/procesadores/Herramientas.html

Mössenböck, H., & Markus, L. (29 de Enero de 2018). *Jhoannes Kepler Universität Linz*. Obtenido de http://www.ssw.uni-linz.ac.at/Coco/

Ortega Sánchez, C. I. (04 de 06 de 2018). *GitHub*. Obtenido de https://github.com/CHA0S14/Analizador-JavaScript-PL.

Viloria Lanero, A. (17 de Marzo de 2003). *Departamento de Informática de la Universidad de Valladolid.* Obtenido de https://www.infor.uva.es/~mluisa/talf/docs/aula/A3-A6.pdf

*Wiki**3*. (13 de Abril de 2018). Obtenido de https://www.l2f.inesc-id.pt/~david/w/pt/The_YACC_Parser_Generator

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Wed Jun 06 21:58:41 CEST 2018
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)