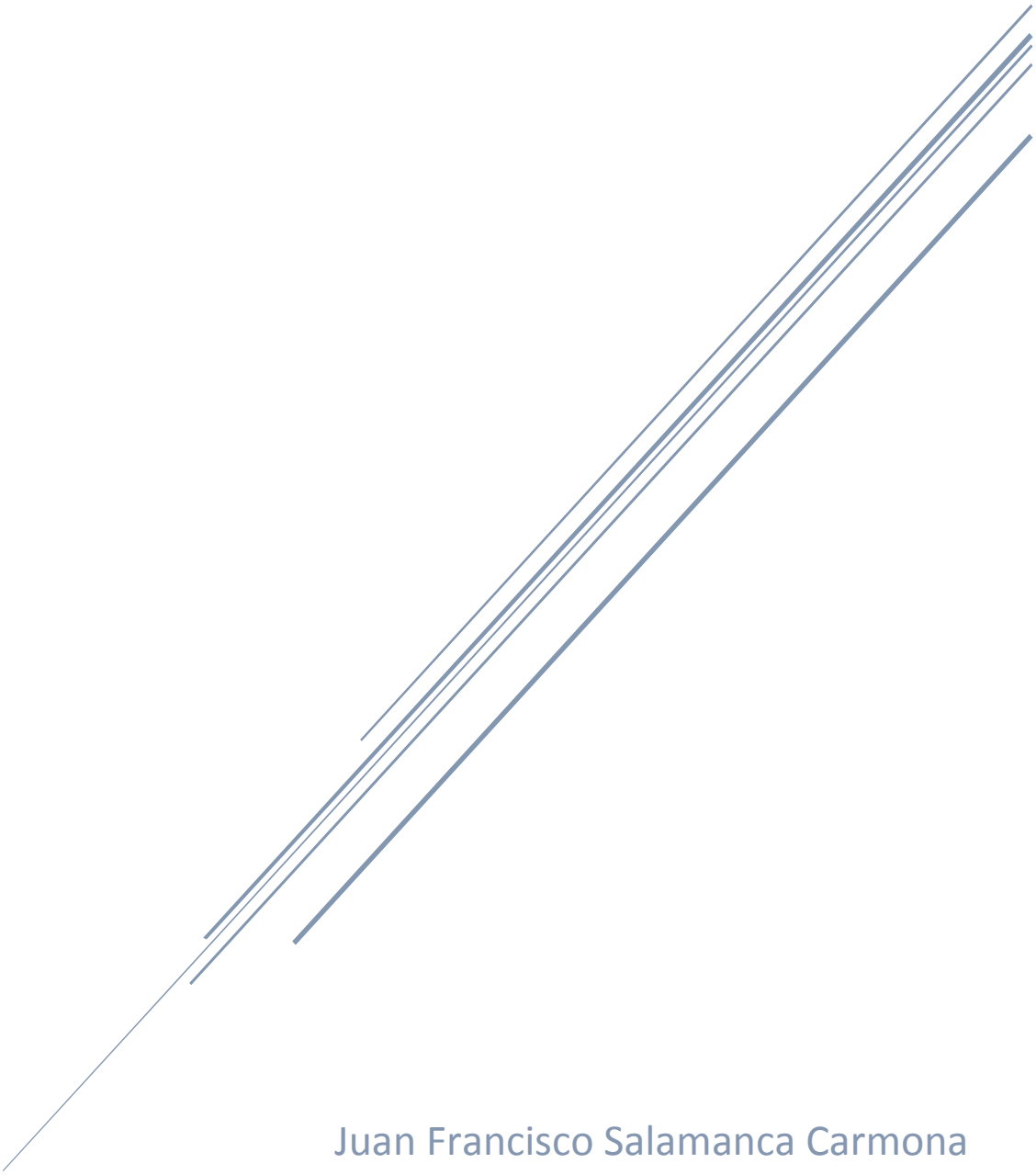
COMPILADOR PARA LENGUAJE JAVASCRIPT

Memoria



Juan Francisco Salamanca Carmona
S100143
50228529W

El procesador se ha desarrollado en el lenguaje Python. Es un procesador realizado con una gramática descendente LL(1). Se han implementado las partes obligatorias del enunciado general y las específicas del grupo: como son la asignación con operación y la sentencia repetitiva do-while.
No se han implementado las partes opcionales como son los Vectores.
Solo se incluyen en el léxico los operadores +, == y ||. El procesador analiza toda la entrada y saca la información de los distintos errores que encuentra.
El procesador no se para en el momento en que encuentra un error, sino que trata de sacar todos los errores posibles para corregir todos los que sean necesarios de una sola vez, esto por lo general ocasiona que se muestren errores que no son pero que al haber ocurrido el error primero, este se expande en forma de cascada sobre el resto del código.

Analizador Léxico.

Tokens:

Numeros -> (DEC, elemento)
Cadenas -> (CAD, elemento)
Operadores -> (OP, elemento)
Separadores -> (SEP, elemento)
Identificadores -> (ID, elemento)
Palabras reservadas -> (PR, elemento)
Saltos de línea -> (SL,)

Gramática:

S -> delS | cI | dN | "C | /O | +R | =R | |A | ; | { | } | (|) | , | \n |
I -> cI | dI | _I | .I'| lambda
I' -> cI |lambda
N -> dN | lambda
C -> ccC | "
O -> /F
F -> c'F | \n
R -> = | lambda
A - > |

d = {0,1,2,3,4,5,6,7,8,9}
c = {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,z,
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,Z}
cc = {todos los caracteres} - {"}
c' = cualquier caracter - {\n}

Los errores que el léxico permite reconocer son aquellos que forman un identificador que contiene caracteres no válidos (como puntos donde no debe), o cuando recibe un símbolo que no está recogido en ninguno de los conjuntos que se especifican en la gramática. Los demás errores serán reconocidos en el analizador sintáctico.
Las acciones semánticas más destacadas son:
· Generar el token correspondiente (operador, identificador, palabra reservada, etc) cuando, tras un determinado número de símbolos válidos, se alcanza un separador (espacio en blanco o tabulador).

· Además, cuando se genera el token de identificador, el léxico lo introduce en la Tabla de Símbolos para que posteriormente el semántico introduzca el tipo y la demás información que sea necesaria del identificador en cuestión.

Analizador Sintáctico.

Gramatica:

Terminales = { -> \n lambda var id = += entero true false function ( ) { } , == + || ; do while return if document.write prompt cadena }

NoTerminales = { P Pprima S D Z I Sprima Sdosprima R F E Eprima T Tprima X Xprima G Gprima L Lprima W Wprima M Mprima Dprima }

Axioma = P

Producciones = {
P -> Pprima
Pprima -> F Pprima
Pprima -> S Pprima
Pprima -> D Pprima
Pprima -> lambda
S -> if (E) S
S -> do { \n S Sdosprima \n } while (E) \n
S -> document.write (M) ; \n
S -> prompt (id) ; \n
S -> return R ; \n
S -> id Sprima
D -> var id Z ; \n
Z -> = I
Z -> lambda
I -> entero
I -> true
I -> false
Sprima -> = E ; \n
Sprima -> + = E ; \n
Sdosprima -> S Sdosprima
Sdosprima -> lambda
R -> E
R -> lambda
F -> function id (W) { \n Dprima S Sdosprima } \n
E -> T Eprima
Eprima -> + T Eprima
Eprima -> lambda
T -> X Tprima
Tprima -> == X Tprima
Tprima -> lambda
X -> G Xprima
Xprima -> || G Xprima
Xprima -> lambda
G -> (E)
G -> id Gprima
G -> entero

Gprima -> (L)
Gprima -> lambda
L -> E Lprima
L -> lambda
Lprima -> , L
Lprima -> lambda
W -> id Wprima
W -> lambda
Wprima -> , W
Wprima -> lambda
M -> E Mprima
M -> cadena Mprima
Mprima -> , M
Mprima -> lambda
Dprima -> D Dprima
Dprima -> lambda
}

Una gramática es recursiva si un símbolo Terminal tiene dos o más consecuentes que comienzan con el mismo símbolo terminal, es decir, el FIRST de ambos es el mismo.
Como se puede comprobar en la gramática especificada arriba, esta situación no se da con ninguno de los símbolos terminales que empleamos, puesto que ninguno tiene dos reglas que comiencen con el mismo símbolo terminal en el consecuente.

Analizador Semántico

Gramatica:

Se va a considerar un tipo enterológico que engloba a los tipos entero y lógico. Cuando se comprueba que un elemento es de dicho tipo; esta comprobación será positiva si el elemento es de tipo lógico o de tipo entero. Las reglas son las siguientes:

```
P -> {TSG = CreaTabla(); DesplG = 0, Activa = TSG}
        P'
        {imprimeTS(); DestruyeTSG()} (1)

P' -> FP'1 {If F.tipo == tipo_ok then
                P'.tipo = P'1.tipo
            Else
                P'.tipo = tipo_error} (2)

P' -> SP'1 {If S.tipo == tipo_ok then
                P'.tipo = P'1.tipo
            else
                P'.tipo = tipo_error} (3)

P' -> DP'1 {P'.tipo = P'1.tipo} (4)

P' -> LAMBDA {P'.tipo = tipo_ok} (5)
```

S -> if(E) S1 {if E.tipo == lógico then
S.tipo = S1.tipo
Else
S.tipo = tipo_error} (6)

S -> do {\n S1S'' \n}while (E)\n {If E.tipo == lógico AND S''.tipo == tipo_ok then
S.tipo = S''.tipo
else
S.tipo = tipo_error} (7)

S -> doc.write(M); \n {if E.tipo != tipo_error then
S.tipo = tipo_ok
Else
S.tipo = tipo_error} (8)

S -> prompt(id); \n {if buscarTS(activa, id) == true then
S.tipo = tipo_ok
else
S.tipo = tipo_error} (9)

S -> return R; \n {If zona_funcion == true then
if R.tipo = enterológico then S.tipo = tipo_ok;
else
S.tipo = tipo_error} (10)

S -> idS' {If BuscarTipoTS(activa, id.entrada) == enterologico and E.tipo == enterologico then
S.tipo = tipo_ok
Else
S.tipo = tipo_error} (11)

D -> {zona_declaracion = true} var id Z ;\n
{if Z.tipo != tipo_error then
AñadeTipoTS(TSG, id.entrada, Z.tipo, DesplG, ambito);
DesplG = DesplG + Z.ancho
D.tipo = tipo_ok
Else
D.tipo = tipo_error
zona_declaracion = false} (12)

Z -> = I {Z.tipo = I.tipo; Z.ancho = I.ancho} (13)

Z -> lambda {Z.tipo = entero} (14)

I = entero {I.tipo = entero; I.ancho = 2} (15)

I = true {I.tipo = logico; I.ancho = 1} (16)

I = false {I.tipo = lógico; I.ancho = 1} (17)

S' -> =E; \n {S'.tipo = E.tipo} (18)

S' -> +=E; \n {S'.tipo = E.tipo} (19)

S'' -> SS''1 {If S.tipo = tipo_ok then
S''.tipo = S''1.tipo
Else
S''.tipo = tipo_error} (20)

S'' -> LAMBDA {S''.tipo = tipo_ok} (21)

R -> E {R.tipo = E.tipo} (22)

R -> LAMBDA {R.tipo = enterológico} (23)

F -> function {zona_function = true}
id(W) {AñadeTipoTS(TSG, id.entrada, W.tipo)}
{\n D'SS''}\n {if S.tipo = tipo_ok then
F.tipo = S''.tipo
else
S.tipo = tipo_error;
zona_funcion = false;
Activa = TSG} (24)

E -> TE' {If T.tipo == enterológico and E'.tipo == entlog then
E.tipo = T.tipo
else if (T.tipo AND E'.tipo) == enterológico then
E.tipo = entero
Else
E.tipo = tipo_error} (25)

E' -> +TE'1 {If T.tipo == enterologico and E'1.tipo == entlog then
E'.tipo = T.tipo
else if (T.tipo AND E'1.tipo) == enterológico then
E'.tipo = entero
else E'.tipo = tipo_error} (26)

E' -> LAMBDA {E'.tipo = entlog} (27)

T -> XT' {If X.tipo == enterologico and T'.tipo == entlog then
T.tipo = X.tipo
else if (X.tipo AND T'.tipo) == enterológico then
T.tipo = entero
else T.tipo = tipo_error} (28)

T' -> ==XT1' {if X.tipo == enterologico and T'1.tipo == entlog then
T'.tipo = X.tipo
else if (X.tipo AND T'1.tipo) == enterológico then
T.tipo = lógico
Else
T.tipo = tipo_error} (29)

T' -> LAMBDA {T'.tipo = entlog} (30)

X -> GX' {If G.tipo == enterologico and X'.tipo == entlog then
X.tipo = G.tipo
else if (G.tipo AND X'.tipo) = enterológico then
X.tipo = lógico

Else
X.tipo = tipo_error} (31)

X' -> ||GX'1 {If G.tipo == enterologico and X'1.tipo == entlog then
X'.tipo = G.tipo
else if (G.tipo AND X'1.tipo == enterológico) then
X'.tipo = lógico
Else
X'.tipo = tipo_error} (32)
X' -> LAMBDA {X'.tipo = entlog} (33)

G -> (E) {G.tipo = E.tipo} (34)

G -> idG' {tipo = BuscarTipoTS(activa, id.entrada);
if tipo == enterologico AND G'.tipo == entlog then
G.tipo = tipo;
else if tipo == enterologico AND G'.tipo != entlog then
if comparaTipoArgumentos(id) == G'.tipo then
G.tipo = entlog // Funcion
else G.tipo = tipo_error
else
G.tipo = tipo_error} (35)

G -> ent {G.tipo = entero} (36)

G' -> (L) {G'.tipo = L.tipo} (37)

G' -> LAMBDA {G'.tipo = entlog} (38)

L -> EL' {If E.tipo OR L'.tipo == tipo_error then
L.tipo = tipo_error
else if L'.tipo == tipo_ok then
L.tipo = E.tipo
Else
L.tipo == E.tipo ProductoCartesiano L'.tipo} (39)

L -> LAMBDA {L.tipo = tipo_ok} (40)

L' -> ,L {L'.tipo = L.tipo} (41)

L' -> LAMBDA {L'.tipo = tipo_ok} (42)

W -> idW' { If W'.tipo == tipo_ok then
W.tipo = enterológico
else
W.tipo = enterológico ProductoCartesiano W'.tipo} (43)

W -> LAMBDA {W.tipo -> tipo_ok} (44)

W' -> ,W {W'.tipo = W.tipo} (45)

W' -> LAMBDA {W'.tipo -> tipo_ok} (46)

M -> EM' {If E.tipo == enterologico AND M'.tipo != tipo_error then

M.tipo=tipo_ok
Else
M.tipo=tipo_error}(47)

M -> cadenaM' {M.tipo = M'.tipo} (48)

M' -> ,M {M'.tipo = M.tipo} (49)

M' -> lambda {M.tipo = tipo_ok} (50)

D' -> DD'1 {If D.tipo == tipo_ok then
D'.tipo = D'.tipo
Else
D'.tipo = tipo_error} (51)

D' -> lambda {D'.tiH.tipo == enterológico and B'1.tipo == entlog then
B'.tipo = H.tipo
Else
D'.tipo = tipo_ok} (52)

Nota: El "ProductoCartesiano" hace referencia a un objeto tupla de Python que incluye todos los tipos que son parámetros de las funciones, tanto en la declaración como en la llamada. Se obtiene de convertir la lista a una tupla.

Diseño de la tabla de símbolos.

La tabla de símbolos se ha implementado por medio de la clase tabla_simbolo. Esta está implementa por medio de un diccionario de objetos de la clase donde la clave es el lexema a guardar, y el valor un objeto del tipo "Info" creado para ello. Dicho objeto está formado por los elementos que interesan a la tabla de símbolos, como son el desplazamiento, el tipo, el número de parámetros y su tipo, etc.
La clase ofrece una serie de métodos para acceder a los campos de cada una de las entradas y para añadir nuevas entradas a la tabla de símbolos. Gestiona el desplazamiento de los identificadores en cada ámbito y se encarga de imprimir la tabla de símbolos al fichero de tablas y vaciar la tabla cuando corresponde, proporcionando dos métodos para tal fin.

En un principio me decidí por utilizar una tabla general y una para cada uno de los ámbitos locales detectados en el código procesado, pero debido al modo en que funcionaba el analizador léxico este enfoque no fue posible, y es que el analizador funciona de forma secuencial y se encarga de analizar todo el fichero de una vez, por lo que no es capaz de manejar los diferentes ámbitos, añadiendo todos los identificadores encontrados a la tabla que se le pasa como parámetro (la tabla global). Por todo esto, decidí que la solución más sencilla para este inconveniente era utilizar una única tabla de símbolos, y llevar control del ámbito por medio de un campo adicional en cada entrada, dentro del objeto Info de cada entrada de la tabla.

De esta forma, el analizador léxico añade los identificadores a la tabla de símbolos, y el analizador semántico es el encargado de asignar el ámbito a

cada identificador. El ámbito de un identificador puede ser global o local, y si es de este último tipo, recibe el valor del procedimiento asociado a ese ámbito, para poder gestionar los múltiples ámbitos locales que pueden existir. Para esto, utilizo un nuevo atributo dentro de la tabla de símbolos para almacenar el desplazamiento del ámbito local, y modificar varios métodos de la misma clase para gestionar correctamente los dos ámbitos.

Este nuevo enfoque es posible debido al subconjunto que se debe procesar de javascript, que no contempla la declaración de una función dentro de otra, lo que provoca que haya en un instante del análisis dos ámbitos como máximo (uno global y otro local).

Casos de prueba.

Para probar el funcionamiento del procesador, he diseñado cinco casos de éxito a partir de las estructuras válidas que es capaz de procesar. El objetivo de estas pruebas es comprobar el parse correcto de las reglas y la construcción de las tablas de símbolos. Las pruebas han sido diseñadas con una dificultad incremental.
Para cada caso de prueba se indica el código correspondiente, una descripción del mismo y el volcado de las tablas de símbolos generadas. Los árboles sintácticos asociados a cada uno de los casos se encuentran incluidos en la carpeta “Arboles” que se encuentra en el CD que se entrega adjunto.

Nota: El orden de las entradas en la tabla de símbolos no corresponde con el orden de inserción en la misma. Para ver el orden de inserción de los identificadores de tipo entero, lógico o enterológico, será necesario guiarse por el desplazamiento de cada entrada de la tabla de símbolos correspondiente.

Casos de éxito:

Caso 1(codigo_sencillo.js):

En esta primera prueba, se declararán tres variables de ámbito global y se utilizarán las sentencias *if* (condicional simple) y el bucle do-*while*

```
var identi = 1;
var accesible = true;
var res = 0;
do{
        identi = identi + 2;
        if (accesible) res = (identi + 6);
 }while(identi)
```

Tabla de símbolos:

```
----------------> Tabla de simbolos de: codigo_sencillo.js <----------------
----> Tabla General <----
lexema: res, tipo: entero, desplazamiento: 3, ambito: global
lexema: identi, tipo: entero, desplazamiento: 0, ambito: global
lexema: accesible, tipo: logico, desplazamiento: 2, ambito: global
```

Caso 2(codigo_do_while.js):

En la siguiente prueba se realiza la declaración de 4 variables en el ámbito global, seguidas por un bucle *do-while* con una sentencia condicional simple, expresión, y sentencia *prompt* anidados.
Finalmente, fuera de la sentencia de bucle, se utiliza una sentencia *document.write*.

```
// Prueba con sentencias
var identi = 1;
var fin = true;
var a = false;
var b = true;
do{
        if (a || b) a = b;
        identi = identi + 2;
        prompt(fin);
 }while(fin)
document.write("El valor de indenti es: ", identi);
```

Tabla de símbolos:

```
----------------> Tabla de simbolos de: codigo_do_while.js <----------------
----> Tabla General <----
lexema: a, tipo: logico, desplazamiento: 3, ambito: global
lexema: b, tipo: logico, desplazamiento: 4, ambito: global
lexema: fin, tipo: logico, desplazamiento: 2, ambito: global
lexema: identi, tipo: entero, desplazamiento: 0, ambito: global
```

Caso 3(codigo_funcion1.js):

En esta prueba comenzamos el testeo de la estructura de funciones. Se realizan 3 declaraciones (una de ellas sin inicialización), y seguidamente se declara una función. Dentro de la función se prueba una sentencia condicional simple, una sentencia *document.write*, y finalmente una sentencia de retorno de una función.

```
var a = 16;
var b;
var c = false;
function prueba(d){
        e = 3;
        b = a + 8;
        if (d || c) document.write("c is false");
        return b;
}
```

Tabla de símbolos:

----------> Tabla de simbolos de: codigo_funcion1.js <----------
----> Tabla Local: 'prueba' <----
lexema: e, tipo: entlog, desplazamiento: 2, ambito: prueba
lexema: d, tipo: entlog, desplazamiento: 0, ambito: prueba

----> Tabla General <----
lexema: a, tipo: entero, desplazamiento: 0, ambito: global
lexema: c, tipo: logico, desplazamiento: 4, ambito: global
lexema: b, tipo: entero, desplazamiento: 2, ambito: global
lexema: prueba, tipo: function, num_par: 1, tipo_par: [entlog], ambito: global

Caso 4(codigo_funcion2.js):

El objetivo de esta prueba es comprobar la correcta gestión de los ámbitos locales. Para ello, se declaran dos funciones, realizando dos declaraciones de variables en cada una.

```
function prueba (a, b){
        var c = 7;
        var d = 4;
        return a + b;
}
function lla_ma_da(a, b, c){
        var e = true;
        var res;
        res = prueba(a,c);
        if (a || b) return res;
}
```

Tabla de símbolos:

----------> Tabla de simbolos de: codigo_funcion2.js <----------
----> Tabla Local: 'prueba' <----
lexema: a, tipo: entlog, desplazamiento: 0, ambito: prueba
lexema: c, tipo: entero, desplazamiento: 4, ambito: prueba
lexema: b, tipo: entlog, desplazamiento: 2, ambito: prueba
lexema: d, tipo: entero, desplazamiento: 6, ambito: prueba

----> Tabla Local: 'lla_ma_da' <----
lexema: a, tipo: entlog, desplazamiento: 0, ambito: lla_ma_da
lexema: c, tipo: entlog, desplazamiento: 4, ambito: lla_ma_da
lexema: b, tipo: entlog, desplazamiento: 2, ambito: lla_ma_da
lexema: e, tipo: logico, desplazamiento: 6, ambito: lla_ma_da
lexema: res, tipo: entero, desplazamiento: 7, ambito: lla_ma_da

----> Tabla General <----
lexema: prueba, tipo: function, num_par: 2, tipo_par: [entlog entlog], ambito: global
lexema: lla_ma_da, tipo: function, num_par: 3, tipo_par: [entlog entlog entlog], ambito: global

Caso 5(codigo_funcion3.js):

En este caso de prueba decidimos presentar un código más complejo, además de probar el comportamiento del procesador frente a las variables no declaradas previamente. El resultado esperado de nuestro procesador frente a esta situación es declarar las variables no declaradas explícitamente como locales por encontrarse dentro de una función y de tipo entero.
Se han probado distintos tipos de sentencias anidadas a la sentencia de bucle *do-while*, como son la sentencia condicional simple, asignaciones de variables declaradas explícitamente y sin declarar y sentencias *prompt*.

```
function funcion3 (var_logica, var_entera1, var_entera2){
        var a = false;
        var b = true;
        var fin = false;
        do{
                if (a || b ) var_logica = b;
                var_entera2 +=  var_entera1 + 2 ;
                var_entera1 = var_entera1 + 1;
                var_no_declarada1 = var_entera1 + var_entera2;
                prompt(fin);
                prompt(var_no_declarada2);
         }while(fin)
        document.write("El valor de la variable lógica es: ",var_logica,"El
resultado de la función es ", var_entera2);
        document.write(var_no_declarada3);
        return var_entera2;
}
```

Tabla de símbolos:

```
----------------> Tabla de simbolos de: codigo_funcion3.js <----------------
----> Tabla Local: 'funcion3' <----
lexema: a, tipo: logico, desplazamiento: 6, ambito: funcion3
lexema: b, tipo: logico, desplazamiento: 7, ambito: funcion3
lexema: var_no_declarada2, tipo: entero, desplazamiento: 11, ambito: funcion3
lexema: var_no_declarada3, tipo: entero, desplazamiento: 13, ambito: funcion3
lexema: var_logica, tipo: entlog, desplazamiento: 0, ambito: funcion3
lexema: fin, tipo: logico, desplazamiento: 8, ambito: funcion3
lexema: var_entera1, tipo: entlog, desplazamiento: 2, ambito: funcion3
lexema: var_entera2, tipo: entlog, desplazamiento: 4, ambito: funcion3
lexema: var_no_declarada1, tipo: entlog, desplazamiento: 9, ambito: funcion3

----> Tabla General <----
lexema: funcion3, tipo: function, num_par: 3, tipo_par: [ entlog entlog entlog ],
ambito: global
```

Casos de error:

Caso 1(codigo_error_dw.js):

En este caso se prueba a recibir la palabra document.write con un error sintáctico(falta el punto entre medias).

```
var fin = false;
var inicio = true;
if (inicio) document write("Esto es una prueba de fallo");
```

Error generado:

Al no detectar document.write como uno solo, salta un error al recibir un paréntesis después de un id cuando no debería.

```
ERROR: en la línea 3 se ha producido un error en la sentencia con el token (
ERROR: en la línea 3 se ha producido un error en la sentencia con el token (
```

Caso 2(codigo_error_funcion.js):

Se prueba a introducir los argumentos de una función sin coma entre medias. Además de un token que no reconoce el lenguaje “!”.

```
var a = 1;
function prueba(control mayorEdad){
        if (!mayorEdad) control = true;
        return control;
}
```

Error generado:

En la función los argumentos no se declaran bien, por tanto provoca un fallo en cascada.

```
ERROR: ! no es un simbolo valido [3,6]
ERROR: En la línea 1, columna 34 se esperaba: mayorEdad
ERROR: En la línea 1, columna 34 se esperaba: mayorEdad
ERROR: En la línea 1, columna 34 se esperaba: mayorEdad
ERROR: En la línea 1, columna 34 se esperaba: )
ERROR: En la línea 1, columna 34 se esperaba: )
ERROR: en la línea 1 se ha producido un error en la funcion con el token )
```

Caso 3(código_error_id.js):

En este caso se prueba a meter caracteres incorrectos como nombre de un identificador.

```
var a = 1;
var b = false;
var ab.cd = 5;
prompt(e);
document.write("El token erroneo es: ", a);
```

Error generado:
Informa de que recibe un punto cuando no debe y por tanto genera errores asociado de esto.

```
Error en la línea 3, columna 7: se recibe un punto (.) cuando no
correspondeERROR: . no es un simbolo valido [4,7]
ERROR: En la línea 3, columna 10 se esperaba: cd
ERROR: En la línea 3, columna 10 se esperaba: cd
```

Caso 4(código_error_ops.js):

En este código se incluye un operador incorrecto.

```
var a = 15;
var b = 5;
var x = true;
do{
        z = a * b;
} while(x)
```

Error generado:

Informa del error del token no aceptado por la gramática, además de todos los demás errores asociados a esto.

```
ERROR: * no es un simbolo valido [6,8]
ERROR: En la línea 5, columna 11 se esperaba: b
ERROR: En la línea 5, columna 11 se esperaba: b
ERROR: En la línea 5, columna 11 se esperaba: ;
ERROR: En la línea 5, columna 11 se esperaba: ;
ERROR: En la línea 5, columna 11 se esperaba: ;
ERROR: En la línea 5, columna 11 se esperaba: ;
ERROR: En la línea 5, columna 11 se esperaba: ;
ERROR: en la línea 5 se ha producido un error en la sentencia con el token ;
```

Caso 5(código_error_var.js):

En este caso se comprueba el resultado de hacer dos declaraciones diferentes pero usando el mismo identificador.

```
var a = 1;
var x = true;
var z;
var a = false;
if (x) a += 2;
```

Error generado:

ERROR: en la línea 5 se ha producido un error en la inicialización (variable ya inicializada) con el token if