

Javascript PdL

Nicolás Cossío Miravalles, Bárbara Rodríguez Ruiz, Huangjue He

Enero 2022

Contents

1	Grupo 127, integrantes	2
2	Código fuente e instrucciones de instalación	2
3	Características específicas del lenguaje	2
3.1	Analizador Léxico	2
3.1.1	Tokens	2
3.1.2	Gramática Regular	3
3.1.3	Automata Finito Determinista	3
3.1.4	Acciones Semánticas	5
3.1.5	Errores que recoge el autómata	5
3.1.6	Tabla de símbolos - Diseño general	6
3.2	Analizador Sintáctico	6
3.2.1	GCL del lenguaje	6
3.2.2	Gramática inicial dada	7
3.2.3	Gramática transformada	7
3.2.4	Reglas	8
3.2.5	Gramática para el árbol sintáctico	9
3.3	Analizador Semántico	11
3.3.1	Funciones semánticas	11
3.3.2	Esquema de Traducción	11
3.4	Tabla de Símbolos	14
3.5	Gestor de Errores	15
4	Anexo - Casos de prueba	15
4.1	Correctos	15
4.1.1	Caso 1	16
4.1.2	Caso 2	44
4.1.3	Caso 3	44
4.1.4	Caso 4	45
4.1.5	Caso 5	45
4.2	Incorrectos:	45
4.2.1	Caso 1	45
4.2.2	Caso 2	47
4.2.3	Caso 3	48
4.2.4	Caso 4	48
4.2.5	Caso 5	49

1 Grupo 127, integrantes

- Nicolás Cossío Miravalles - n.cossio@alumnos.upm.es - b190082
- Bárbara Rodríguez Ruiz - barbara.rodriguez.ruiz@alumnos.upm.es - b190110
- Huangjue He - h.he@alumnos.upm.es - a180022

2 Código fuente e instrucciones de instalación

Disponible en Github, en el repositorio encontrará las instrucciones para instalarlo de forma local.

Formato ejecutable disponible en releases.

3 Características específicas del lenguaje

Se listan la parte asignada, opcional y las elecciones que hemos hecho. El resto del lenguaje sigue las reglas generales del language.

Asignadas :

- **Sentencias:** do-while
- **Operadores especiales:** Post-auto-incremento (++ como sufijo)
- **Comentarios:** de bloque (/coment/)
- **Cadenas:** con comillas dobles (“cadenas”)
- **Técnicas de Análisis Sintáctico:** Descendente recursivo

Opcionales:

- No aceptamos caracteres de escape en cadenas
- No aceptamos declaración e inicialización en la misma sentencia
- Sí se aceptan las constantes lógicas true y false
- No aceptamos operadores unarios

Elecciones de operadores:

- Operadores lógicos: and (&&) y or (||)
- Operadores aritméticos: más (+) y por (*)
- Operadores relacionales: comparación (==) y mayor qué (>)

3.1 Analizador Léxico

3.1.1 Tokens

```
1 let :      < let, - >
2 function : < function, - >
3 return:    < return, - >
4 if:        < if, - >
5 else :     < else, - >
6 input :    < input, - >
7 print :    < print, - >
8 while :    < while, - >
9 do:        < do, - >
10 true :    < boolT, - >
11 false :   < boolF, - >
12 int :     < int, - >
13 boolean : < boolean, - >
14 string :  < string, - >
```

```

1 Operadores Aritméticos:
2 + : < mas, - >
3 * : < por, - >
4
5 Operadores Lógicos:
6 && : < and, - >
7 || : < or, - >
8
9 Operadores relacionales:
10 == : < equals, - >
11 > : < mayor, - >

```

```

1 Identificador : < id, ptoTS >
2 Asignacion = : < asig, - >
3 Cadena      : < cadena, laCadena >
4 Entero      : < cteEnt, valor >
5 ++         : < postIncrement, - >
6 ,          : < coma, - >
7 ;          : < puntoComa, - >
8 (          : < parAbierto, - >
9 )          : < parCerrado, - >
10 {         : < llaveAbierto, - >
11 }         : < llaveCerrado, - >
12 eof       : < eof, - >

```

3.1.2 Gramática Regular

```

1 S -> 1A | dC | ( | ) | { | } | delS | =E | &D | > | +G | * | , | ; | eof | /B | "J
2 A -> 1A | dA | _A | lambda
3 B -> *H
4 C -> dC | lambda
5 D -> &
6 E -> = | lambda
7 G -> + | lambda
8 H -> lambda H | *I
9 I -> /S
10 J -> lambda J | "K
11 K -> lambda
12 otro indica cualquier otro carácter distinto de *

```

3.1.3 Automata Finito Determinista

-
- OC: cualquier carácter distinto de los ya especificados para ese estado

3.1.4 Acciones Semánticas

```
1 - LEER: lee el siguiente carácter del fichero fuente. car := leer()
2   En todas las transiciones menos en las transiciones etiquetadas con O.C,
   excepto en 25->25
3 - CONC: forma una cadena (lexema). Siempre después de leer(), lex:=lex + car
4   En las transiciones: 13->14, 14->15 | 0->1, 1->1, x->x //por hacer, cadenas
5 - VALOR: convierte un carácter a su entero correspondiente, entero:=valor(carácter)
6 - GENTOKEN: genera un token que el A.Léxico pasa al sintáctico
7 0->2: num = valor(car)
8 2->2: num = num*10+valor(car)
9 2->4: if (num < 32768) genToken(cteEnt, num)
10   else error()
11
12 25->26:
13 if (length(lex) < 65) genToken(cadena, lex)
14 else error()
15
16 TEstadosFinales []:=lista de códigos de los tokens de operacion y otros menos los
   no identificadores, cadena o constante que se generan en los estados finales
   del autómata, permiten crear un token al usarse como argumento de genToken( )
17
18 if transicion is not 1->3, 2->4 {
19   a:=TEstadosFinales[numEstadoFinal]
20   if a is not null genToken( a , )
21 }
22
23 buscarTPR(lex): devuelve el código del token que coincide con un lexema dado,
   codigo:=buscarTPR(lex)
24 TPR []:=lista de códigos de tokens de palabras reservadas que permiten crear un
   token al usarse como argumento de genToken( )
25 1->3:
26 zona_decl := boolean que indica si se trata de una declaracion, global = true y
   local = false
27 p:= buscarTPR(lex)
28 if ( p is not null ) genToken( p , - ) //genera token de palabra reservada
29 else { //identificador
30   p:= buscarTS(lex)
31   if ( p is not null ) //ya está declarada
32     genToken(id, p)
33   else { //no está declarada
34     p:=añadirTS_activa(lexema) //AñadirTS devuelve un ptro. al id
35     genToken(id, p)
36   }
37 }
```

3.1.5 Errores que recoge el autómata

```
1 ERROR: cualquier transición no recogida en el autómata corresponde a un caso de
   error.
2 También son errores todos los lanzados desde las acciones semánticas, que son los
   siguientes:
3 - Entero sobrepasa valor máximo permitido
```

4 - String sobrepasa longitud máxima permitida

3.1.6 Tabla de símbolos - Diseño general

```
1 Todas las tablas tendrán lexema y tipo, pero el resto de los Tributos de la tabla
  dependeran del tipo.
2
3 Para enteros, reales, cadenas, lógicos... tendremos una tabla con el siguiente
  formato:
4
5 TS 1##:
6
7 * LEXEMA : 'a'
8 Atributos:
9 + tipo: 'entero'
10 + despl : 0
11
12 Para un array:
13
14 TS 2##:
15
16 * LEXEMA : 'a'
17 Atributos:
18 + tipo: 'entero'
19 + despl : 0
20 ? núm de dimensiones, límite inf y límte sup de cada dimensión,...
21
22 Para funciones la tabla seguirá el formato:
23
24 TABLA FUNCION SUMA ##3:
25
26 *LEXEMA : 'suma'
27 Atributos:
28 + tipo: 'funcion'
29 + numParam: 2
30 + TipoParam01: 'ent'
31 + TipoParam2: 'real'
32 + TipoRetorno: 'ent'
33 + EtiqFuncion: ''Etsuma01
```

3.2 Analizador Sintáctico

3.2.1 GCL del lenguaje

ACLARACIÓN: usamos los símbolos en vez de los nombres propios de los tokens en las gramáticas debido a la mejor legibilidad que dan. Somos conscientes que el símbolo es la representación del token en el lenguaje y el viceversa en la gramática. En el apartado “Gramática para el árbol sintáctico” se encuentra la gramática final propiamente dada.

3.2.2 Gramática inicial dada

```
1 P -> B P | F P | eof
2 B -> let T id ; | if ( E ) S | S | while ( E ) { C }
3 T -> int | boolean | string
4 S -> id = E ; | return X ; | id ( L ) | print ( E ) ; | input ( id ) ;
5 X -> E | lambda
6 C -> B C | lambda
7 L -> E Q | lambda
8 Q -> , E Q | lambda
9 F -> function id H ( A ) { C }
10 H -> T | lambda
11 A -> T id K | lambda
12 K -> , T id K | lambda
13 E -> E && R | R
14 R -> R > U | U
15 U -> U + V | V
16 V -> id | ( E ) | id ( L ) | entero | cadena
```

3.2.3 Gramática transformada

```
1 ## indica comentario dentro de la definición de la gramática
2 P -> B P | F P | eof
3 B -> let T id ; | if ( E ) S | S | do { C } while ( E ) ;
4 T -> int | boolean | string
5 S -> id S' | return X ; | print ( E ) ; | input ( id ) ;
6 S' -> = E | ( L ) | ++
7 X -> E | lambda
8 C -> B C | lambda
9 L -> E Q | lambda
10 Q -> , E Q | lambda
11 F -> function id H ( A ) { C }
12 H -> T | lambda
13 A -> T id K | lambda
14 K -> , T id K | lambda
15 ## precedencia menos a más: {||} -> {&&} -> {==} -> {>} -> {+} -> {*} -> {++}
16 E -> N 01 ##operadores aritméticos
17 N -> Z 02 ##operadores relacionales
18 Z -> R 03 ##operadores lógicos
19 01 -> || N 01 | && N 01 | lambda
20 02 -> == N 02 | > N 02 | lambda
21 03 -> + R 03 | * R 03 | lambda
22 R -> id R' | ( E ) | entero | cadena | true | false
23 R' -> ( L ) | ++ | lambda
```

Justificación de que es gramática LL(1):

Como la gramática está factorizada no existe ninguna producción: $A \rightarrow a / B / \dots$ donde $\text{First}(a) \cap \text{First}(B) \neq \emptyset$

Para los consecuentes que pueden derivar a lambda :

- $01 \rightarrow \text{First}(01) \cap \text{Follow}(01) = \emptyset$

- First(O1) = { +, *, lambda }
- Follow(O1) = Follow(E) = First(Q) + Follow(X) + { ;,) } = { ;,), coma }
- O2 → First(O2) interseccion Follow(O2) = \emptyset
 - First(O2) = { ==, >, lambda }
 - Follow(O2) = Follow(N) = First(O1) + First(O2) + Follow(E) = { +, *, ==, >, ;,), coma }
- O3 → First(O3) interseccion Follow(O3) = \emptyset
 - First(O3) = { &&, ||, lambda }
 - Follow(O3) = Follow(Z) = First(O2) + Follow(N) = { ==, >, +, *, ;,), coma }
- X → E | lambda → First(X) interseccion Follow(X) = \emptyset
 - First(X) = { id, (, cteEnt, cadena, boolT, boolF, lambda }
 - Follow(X) = { ; ; }
 - First(V)contenido en First(U)contenido en First(R)contenido en First(E)contenido en First(X)
- C → B C | lambda → First(B C) interseccion Follow(C) = \emptyset
 - First(C) = First(B) = { let, if, id, return, print, input, do }
 - Follow(C) = { llaveAbierto }
- L → E Q | lambda → First(EQ) interseccion Follow(L) = \emptyset
 - First(L) = First(E) = { id, (, cteEnt, cadena, boolT, boolF }
 - Follow(L) = {) }
 - First(V)contenido en First(U)contenido en First(R)contenido en First(E)contenido en First(EQ)
- Q → , E Q | lambda → First(, E Q) interseccion Follow(Q) = \emptyset
 - First(Q) = { , }
 - Follow(Q) = Follow(L) = {) }
- H → T | lambda → First(T) interseccion Follow(H) = \emptyset
 - Follow(H) = { (}
- A → T id K | lambda → First(T id K) interseccion Follow(A) = \emptyset
 - First(A) = First(T) = { int, boolean, string }
 - Follow(A) = {) }
 - First(T)contenido en First(T id K)
- K → , T id K | lambda → First(, T id K) interseccion Follow(K) = \emptyset
 - First(K) = { , }
 - Follow(K) = Follow(A) = {) }
- R' → First(R) interseccion Follow(R') = \emptyset
 - First(R') = { (, ++, lambda }
 - Follow(R') = Follow(R) = First(O) + Follow(E) = { &&, +, *, ==, >, lambda } + { coma, puntoComa,) } = {) }

```
function hola int (){
return "cadena";
}
```

3.2.4 Reglas

```
1 1 - P -> B P
2 2 - P -> F P
3 3 - P -> eof
```



```

4 4 - B -> let T id ;
5 5 - B -> if ( E ) S
6 6 - B -> S
7 7 - B -> do { C } while ( E );
8 8 - T -> int
9 9 - T -> boolean
10 10 - T -> string
11 11 - S -> id S' ;
12 12 - S -> return X ;
13 13 - S -> print ( E ) ;
14 14 - S -> input ( id ) ;
15 15 - S' -> asig E
16 16 - S' -> ( L )
17 17 - S' -> ++
18 18 - X -> E
19 19 - X -> lambda
20 20 - C -> B C
21 21 - C -> lambda
22 22 - L -> E Q
23 23 - L -> lambda
24 24 - Q -> , E Q
25 25 - Q -> lambda
26 26 - F -> function id H ( A ) { C }
27 27 - H -> T
28 28 - H -> lambda
29 29 - A -> T id K
30 30 - A -> lambda
31 31 - K -> , T id K
32 32 - K -> lambda
33 33 - E -> N 01
34 34 - N -> Z 02
35 35 - Z -> R 03
36 36 - 01 -> || N 01
37 37 - 01 -> && N 01
38 38 - 01 -> lambda
39 39 - 02 -> == Z 02
40 40 - 02 -> > Z 02
41 41 - 02 -> lambda
42 42 - 03 -> + R 03
43 43 - 03 -> * R 03
44 44 - 03 -> lambda
45 45 - R -> id R'
46 46 - R -> ( E )
47 47 - R -> entero
48 48 - R -> cadena
49 49 - R -> true
50 50 - R -> false
51 51 - R' -> ( L )
52 52 - R' -> ++
53 53 - R' -> lambda

```

3.2.5 Gramática para el árbol sintáctico

```

1 Terminales = { eof let id puntoComa if parAbierto parCerrado llaveAbierto
  llaveCerrado while do else function return input print true false int boolean
  string mas por and equals mayor asig cadena cteEnt postIncrem coma or cteEnt }

```

```

2 NoTerminales = { A B C E F H K L N 01 02 03 P Q R Rp S Sp T X
  Z }
3 Axioma = P
4
5 Producciones = {
6 P -> B P
7 P -> F P
8 P -> eof
9 B -> let T id puntoComa
10 B -> if parAbierto E parCerrado S
11 B -> S
12 B -> do llaveAbierto C llaveCerrado while parAbierto E parCerrado puntoComa
13 T -> int
14 T -> boolean
15 T -> string
16 S -> id Sp puntoComa
17 S -> return X puntoComa
18 S -> print parAbierto E parCerrado puntoComa
19 S -> input parAbierto id parCerrado puntoComa
20 Sp -> asig E
21 Sp -> parAbierto L parCerrado
22 Sp -> postIncrement
23 X -> E
24 X -> lambda
25 C -> B C
26 C -> lambda
27 L -> E Q
28 L -> lambda
29 Q -> coma E Q
30 Q -> lambda
31 F -> function id H parAbierto A parCerrado llaveAbierto C llaveCerrado
32 H -> T
33 H -> lambda
34 A -> T id K
35 A -> lambda
36 K -> coma T id K
37 K -> lambda
38 E -> N 01
39 N -> Z 02
40 Z -> R 03
41 01 -> or N 01
42 01 -> and N 01
43 01 -> lambda
44 02 -> equals Z 02
45 02 -> mayor Z 02
46 02 -> lambda
47 03 -> mas R 03
48 03 -> por R 03
49 03 -> lambda
50 R -> id Rp
51 R -> parAbierto E parCerrado
52 R -> cteEnt
53 R -> cadena
54 R -> true
55 R -> false
56 Rp -> parAbierto L parCerrado

```

```

57 Rp -> postIncrement
58 Rp -> lambda
59 }

```

3.3 Analizador Semántico

3.3.0.1 Tipos de Datos El lenguaje dispone de los siguientes tipos de datos básicos:

- El tipo **entero** se refiere a un número entero que debe representarse con un tamaño de 1 palabra (16 bits). Se representa con la palabra **int**.
- El tipo **lógico** permite representar valores lógicos. Se representa también con un tamaño de 1 palabra (16 bits). Las expresiones relacionales y lógicas devuelven un valor lógico. Se representa con la palabra **boolean**.
- El tipo **cadena** permite representar secuencias de caracteres. Se representa con la palabra **string** y una variable de tipo cadena ocupa 64 palabras (128 bytes).

No hay conversión de tipos automática en el lenguaje.

3.3.1 Funciones semánticas

```

1 tipo TS:
2   .crear() -> crea una tabla de símbolos vacía
3   .destruir( tabla ) -> destruye la tabla de símbolos "tabla"
4   .desp = desplazamiento actual de la tabla, última posición libre
5   .insertatId(id) -> se inserta en la última posición el identificador, creando
        una nueva entrada y poniendo el desplazamiento
6   de la entrada como el valor actual de TS.desp
7   .insertarTipoId ( pos, tipo ) -> inserta el tipo de la variable (id.pos) en la TS
8
9   .insertarTipoParam ( tipo1 x tipo2 x ... ) -> inserta un producto cartesiano de
        los tipos de los
10  parámetros de los argumentos de una función
11  .insertarTipoDev ( tipo ) -> inserta el tipo que devuelve una función en la
        tabla general
12
13  .buscarId( id.pos ) -> busca un identificador en la tabla, devuelve true si
        existe, false si no
14  .getTipoParam( id.pos ) -> devuelve el valor (producto cartesiano de tipos) que
        identifica los tipos
15  de los argumentos de la función id
16 tipo id:
17  .pos = posición en la TS que corresponda, adquiere el valor de TS.pos al
        insertarse con
18  TS.insertarId(id)
19
20 tipo reglas: son todas las reglas que contiene la gramática
21  .tipo = tipo que devuelve la regla (boolean, string, entero o int, vacío,
        function)
22  puede ser una producto cartesiano de tipos o solo uno
23  .tipoDev= devolución de una regla

```

3.3.2 Esquema de Traducción

```

1 P' -> { TSG = TS.crear() TSactual = TSG } P { TS.destruir(TSG) }
2 P -> B P
3 P -> F P
4 P -> eof
5 B -> let T id puntoComa

```

```

6   { if TSactual.buscarId(id) == false )
7       then
8           id.pos = TSActual.insertarId( id )
9           TSActual.insertarTipoId( id.pos, T.tipo )
10          TS.despl = displ + T.ancho
11      }
12 B  -> if parAbierto E parCerrado S
13  { if (E.tipo != boolean)
14      then error("El tipo de E tiene que ser boolean ya que nos encontramos en la
15          condición de if")
16  }
16 B  -> S
17 B  -> do llaveAbierto C llaveCerrado while parAbierto E parCerrado puntoComa
18  { if (E.tipo != boolean)
19      then error("La condición del while debe ser de tipo booleano")
20  }
21 T  -> int { T.tipo:= int, T.ancho:= 1 }
22 T  -> boolean { T.tipo:= boolean, T.ancho:= 1 }
23 T  -> string { T.tipo:= string, T.ancho:= 64}
24 S  -> id S' puntoComa
25  { if ( TSActual.buscarId( id ) == false ) ## no está en tabla local
26      then if (TSG.buscarId( id ) == true ) ## sí está en global-> llamada a función
27          o asignación a variable
28              then if (S'.tipo != TSG.getTipoParam(id) ) ## argumentos no coinciden con
29                  los de la función
30                      then if (id.tipo != S'.tipo ) ## asignación
31                          then error( "Tipos en la asignación no coinciden")
32                          ## funcion
33                          else error("Argumentos no coinciden con los de la función")
34      else if ( S'.tipo == postIncrement and id.tipo != cteE )
35          then error("El operador post incremento solo es aplicable a variables del
36              tipo entero")
37      else ## es una declaracion e inicialización de una variable global i.e (a = 5)
38          id.pos = TSG.insertarId( id )
39          TSG.insertarTipo ( id.pos, S'.tipo )
40          ancho = if (S'.tipo == string ) else 1
41          TSG.pos = TSG.pos + ancho
42      }
43 S  -> return X puntoComa
44  {
45      S.tipo = tipo_ok
46      S.tipoRet = X.tipo
47  }
48 S  -> print parAbierto E
49  {
50      S.tipoRet = vacio
51      S.tipo = tipo_ok if (E.tipo == string ) else error("La función print solo acepta
52          parámetros de tipo string")
53  }
54 parCerrado puntoComa
55 S  -> input parAbierto id
56  { if (TSActual.buscarId(id) == true )
57      then if TSActual.buscarTipo(id) not in (string, cteEnt)
58          then error("La función input debe recibir una variable de tipo string o
59              entero")
60      else if (TSG.buscarId(id) == true )

```

```

56     then if TSG.buscarTipo(id) not in (string, cteEnt)
57         then error("La función input debe recibir una variable de tipo string o
           entero")
58     else error("Variable no ha sido previamente declarada")
59 } parCerrado puntoComa
60 S' -> asig E puntoComa { S'.tipo = E.tipo }
61 S' -> parAbierto L parCerrado { S'.tipo = L.tipo }
62 S' -> postIncrement { S'.tipo = postIncrement }
63 X -> E { X.tipo = E.tdefipo }
64 X -> lambda { X.tipo = vacio }
65 C -> B C
66 C -> lambda { C.tipo = vacio }
67 L -> E Q { L.tipo = L.tipo x Q.tipo }    ## tipo1 x tipo2 x tipo3 o vacio
68 L -> lambda { L.tipo = vacio }
69 Q -> coma E {if E.tipo != vacio)
70     then Q.tipo = Q.tipo x E.tipo }
71 Q
72 Q -> lambda { Q.tipo = vacio }
73 F -> function id
74 { tabla = crearTS()
75   TSactual = tabla
76   Desp_tabla1 = 0
77   TSG.insertarId( id ) }
78 H
79 { TSactual.insertartipoTS (H.tipo);
80   TSG.insertarTipoDev( id, H.tipo )}
81 parAbierto A parCerrado
82 { TSG.insertarTipoParam( id.pos, A.tipo )} ## sintáctico solo acepta boolean
      string o int, si no es ninguno dará error
83 llaveAbierto C llaveCerrado
84 { tabla.destruir()
85   TSactual = TSG }
86 H -> T { H.tipo = T.tipo }
87 H -> lambda { H.tipo = vacio }
88 A -> T id K { if ( K.tipo != vacio) then A.tipo = T.tipo x K.tipo} ##
      concatenamiento de ids (tipo1 x tipo2 x tipo3 x ... )
89 A -> lambda { A.tipo = vacio }
90 K -> coma T { K.tipo = T.tipo x K.tipo } id K
91 K -> lambda { K.tipo = vacio }
92 E -> N 01 { E.tipo = "cteEnt" }
93 N -> Z 02 { N.tipo = "cteEnt" }
94 Z -> R 03 { Z.tipo = "boolean" }
95 01 -> mas N { if R.tipo != cteEnt
96     then error("Operador + solo acepta datos enteros")
97     } 01
98 01 -> por N { if R.tipo != cteEnt
99     then error("Operador * solo acepta datos enteros")
100    } 01
101 01 -> lambda { 01.tipo = tipo_ok}
102 02 -> equals Z { if Z.tipo != cteEnt
103     then error("Operador > solo acepta datos enteros")
104     } 02
105 02 -> mayor Z { if Z.tipo != cteEnt
106     then error("Operador > solo acepta datos enteros")
107     } 02
108 02 -> lambda { 01.tipo = "boolean"}

```

```

109 03 -> or R { if R.tipo != boolean
110     then error("Operador || solo acepta datos lógicos")
111     } 03
112 03 -> and R { if R.tipo != boolean
113     then error("Operador || solo acepta datos lógicos")
114     else if 01.tipo == vacio then 03.tipo = tipo_ok } 03
115 03 -> lambda { 03.tipo = tipo_ok }
116 R -> id R' { if (R'.tipo == postIncrem and id.tipo != cteEnt )
117     then error("El operador post incremento solo es aplicable a variables del
118         tipo entero")
119     else if (R'.tipo != vacio ) ## se trata de una llamada a una función
120     then if ( TSG.buscarId( id ) == false )
121         then error("Error la función no ha sido declarada previamente")
122         else if (R'.tipo != TSG.getParam( id ) )
123             then error("Tipos de los atributos incorrectos en llamada a función")
124             else R.tipo = TSG.getTipoDev( id )
125     else:
126         R.tipo = id.tipo ## habria que buscarlo en ambas tablas para ver en
127         ## en cual esta y coger el tipo de la tabla
128     }
129 R -> parAbierto E parCerrado { R.tipo:= E.tipo }
130 R -> cteEnt { R.tipo:= int, R.ancho:= 1 }
131 R -> cadena { R.tipo:= string R.ancho:= 64 }
132 R -> boolT { R.tipo:= boolean R.ancho:= 1 }
133 R -> boolF { R.tipo:= boolean R.ancho:= 1 }
134 R' -> lambda { R'.tipo = vacio }
135 R' -> parAbierto L parCerrado
136     { R'.tipo = L.tipo }
137 R' -> postIncrem { R'.tipo = postIncrem }

```

3.4 Tabla de Símbolos

Para las tablas de simbolos hemos seguido un formato como el que se muestra a continuación:

```

1 -----
2 TABLA PRINCIPAL #0
3
4 * LEXEMA : "demo"
5 ATRIBUTOS :
6 + Tipo: funcion
7 +numParam: 0
8 +TipoRetorno: string
9 -----
10 -----
11 TABLA de funcion "demo" #1
12
13 * LEXEMA : "v1"
14 ATRIBUTOS :
15 + Tipo: int
16 + Despl: 0
17
18 * LEXEMA : "v2"
19 ATRIBUTOS :
20 + Tipo: int
21 + Despl: 1
22
23 * LEXEMA : "v3"
24 ATRIBUTOS :

```

```

25 + Tipo: int
26 + Despl: 2
27 -----

```

3.5 Gestor de Errores

Este apartado lo hemos manejado según de dónde provenía el error. Para los errores léxicos hemos hecho que en vez de parar la ejecución siga produciendo tokens y buscando errores léxicos, para sí poder dar la mayor información posible pese a que no se pueda hacer un análisis sintáctico o gramatical.

Hemos añadido la funcionalidad de que se autocorrijan los comentarios que no estén puestos con el formato pedido de " " en vez del de `.`.

Sobre los errores sintácticos o semánticos, detenemos completamente el análisis, ya que al no recibir el token que esperamos se rompe el árbol sintáctico y es imposible continuar.

Para los mensajes de error hemos creado una clase que implementan todas las partes del analizador con su propio método para crear una instancia de este tipo de error. Al crearse un error este automáticamente crea un string del error diciendo el tipo de error, la línea donde ha ocurrido, además obteniendo dicha línea y mostrándola, para ser más visual.

Debajo de la línea usamos un indicador para mostrar en qué columna está el error, por lo que así el usuario puede saber exactamente dónde está el error, no solo a nivel de línea sino de carácter dentro de esta. Estos son algunos ejemplos:

```

1 *****
2 NonDeclaredError at line 2:
3   estafuncionnoexiste();
4   ~~~~~~
5 Error la función estafuncionnoexiste no ha sido declarada previamente
6 *****
7
8 *****
9 Lexical error at line 34:
10   print ('Es bisiesto?');
11   ~~~~~~
12 Cadena se debe especificar entre " ", no con ' '. Corregido
13 *****
14
15 *****
16 TypeError at line 7:
17   input (v1);
18   ~~~~~~
19 Variable a es de tipo boolean, input() debe recibir una variable de tipo string o
   entero
20 *****
21
22
23 Error fatal, saliendo ...

```

4 Anexo - Casos de prueba

4.1 Correctos

Formato:

- Breve explicación del código y los elementos del lenguaje que queremos demostrar que se funcionan
- Código del caso escrito en Javascript PdL

Para el primer caso además se mostrará la siguiente información:

- Listado de tokens
- Imágen del árbol de análisis sintáctico generado mediante VAST
- Volcado de la Tabla de Símbolos

4.1.1 Caso 1

Esto es una demostración de todo lo que se podría hacer con el lenguaje

- Declaraciones con todos los tipos posibles.
- Declaraciones de funciones con varios parámetros, algunas de ellas en sus códigos de bloque son recursivas. Se llaman a las funciones con los parámetros esperados.
- Se utilizan todas las operaciones posibles
- Se hacen returns con valores directamente o de otros resultados de funciones.
- Se utiliza el bucle do while, junto con condiciones if simples
- Asignaciones
- Se utilizan funciones predeterminadas como input o print

Código:

```

1 let string cadena;
2 input(cadena);
3 let boolean logico1;
4 let boolean logico2;
5 let int int2;
6 int1 = 000000378;
7
8 int2 = int1++;
9 cadena = "string";
10 logico1 = true;
11 logico1 = false;
12 function ff string(string ss)
13 {
14   logico2 = logico1;
15   if (logico2) cadena = ff (ss);
16   varglobal = 78;
17   return cadena;
18 }
19
20 function funcion string (string logico2)
21 {
22   let int var;
23   do {
24     logico1 = int1 == int2; print(0); logico2 = "";
25     } while (logico1);
26   return logico2;
27 }
28
29 cadena = (ff(funcion(cadena)));
30 print(cadena);
31 let boolean booleano;
32 function bisiesto boolean (int a)
33 { let string bis;
34   print ("Es bisiesto?");
35   input(bis);
36   return ((a + 4 == 0));
37 }
38 function dias int (int m, int a)
39 {

```



```

40 let int dd;
41 print ("di cuantos dias tiene el mes ");
42 print (m);
43 input(dd);
44 if (bisiestro(a)) dd = dd + 1;
45 return dd;
46 }
47 function esFechaCorrecta boolean (int d, int m, int a)
48 {
49 return (d == dias (m, a));
50 }

```

```

1 Listado de tokens:
2 < let , None >
3 < string , None >
4 < id , cadena >
5 < puntoComa , None >
6 < input , None >
7 < parAbierto , None >
8 < id , cadena >
9 < parCerrado , None >
10 < puntoComa , None >
11 < let , None >
12 < boolean , None >
13 < id , logico1 >
14 < puntoComa , None >
15 < let , None >
16 < boolean , None >
17 < id , logico2 >
18 < puntoComa , None >
19 < let , None >
20 < int , None >
21 < id , int2 >
22 < puntoComa , None >
23 < id , int1 >
24 < asig , None >
25 < cteEnt , 0 >
26 < puntoComa , None >
27 < id , int2 >
28 < asig , None >
29 < id , int1 >
30 < postIncrem , None >
31 < puntoComa , None >
32 < id , cadena >
33 < asig , None >
34 < cadena , string >
35 < puntoComa , None >
36 < id , logico1 >
37 < asig , None >
38 < true , None >
39 < puntoComa , None >
40 < id , logico1 >
41 < asig , None >
42 < false , None >
43 < puntoComa , None >
44 < if , None >
45 < parAbierto , None >

```

```

46 < id , int2 >
47 < mas , None >
48 < id , int2 >
49 < equals , None >
50 < id , int1 >
51 < or , None >
52 < id , logico1 >
53 < and , None >
54 < id , logico2 >
55 < parCerrado , None >
56 < print , None >
57 < parAbierto , None >
58 < cadena , Como narices se ha evaluado esto a true >
59 < parCerrado , None >
60 < puntoComa , None >
61 < function , None >
62 < id , ff >
63 < string , None >
64 < parAbierto , None >
65 < string , None >
66 < id , ss >
67 < parCerrado , None >
68 < llaveAbierto , None >
69 < id , logico2 >
70 < asig , None >
71 < id , logico1 >
72 < puntoComa , None >
73 < if , None >
74 < parAbierto , None >
75 < id , logico2 >
76 < parCerrado , None >
77 < id , cadena >
78 < asig , None >
79 < id , ff >
80 < parAbierto , None >
81 < id , ss >
82 < parCerrado , None >
83 < puntoComa , None >
84 < id , varglobal >
85 < asig , None >
86 < cteEnt , 0 >
87 < puntoComa , None >
88 < return , None >
89 < id , cadena >
90 < puntoComa , None >
91 < llaveCerrado , None >
92 < function , None >
93 < id , funcion >
94 < string , None >
95 < parAbierto , None >
96 < string , None >
97 < id , logico2 >
98 < parCerrado , None >
99 < llaveAbierto , None >
100 < let , None >
101 < int , None >

```

```

102 < id , var >
103 < puntoComa , None >
104 < do , None >
105 < llaveAbierto , None >
106 < id , logico1 >
107 < asig , None >
108 < id , int1 >
109 < equals , None >
110 < id , int2 >
111 < puntoComa , None >
112 < print , None >
113 < parAbierto , None >
114 < cteEnt , 0 >
115 < parCerrado , None >
116 < puntoComa , None >
117 < id , logico2 >
118 < asig , None >
119 < cadena , >
120 < puntoComa , None >
121 < llaveCerrado , None >
122 < while , None >
123 < parAbierto , None >
124 < id , logico1 >
125 < parCerrado , None >
126 < puntoComa , None >
127 < return , None >
128 < id , logico2 >
129 < puntoComa , None >
130 < llaveCerrado , None >
131 < id , cadena >
132 < asig , None >
133 < parAbierto , None >
134 < id , ff >
135 < parAbierto , None >
136 < id , funcion >
137 < parAbierto , None >
138 < id , cadena >
139 < parCerrado , None >
140 < parCerrado , None >
141 < parCerrado , None >
142 < puntoComa , None >
143 < print , None >
144 < parAbierto , None >
145 < id , cadena >
146 < parCerrado , None >
147 < puntoComa , None >
148 < let , None >
149 < boolean , None >
150 < id , booleano >
151 < puntoComa , None >
152 < function , None >
153 < id , bisiestro >
154 < boolean , None >
155 < parAbierto , None >
156 < int , None >
157 < id , a >

```

```

158 < parCerrado , None >
159 < llaveAbierto , None >
160 < let , None >
161 < string , None >
162 < id , bis >
163 < puntoComa , None >
164 < print , None >
165 < parAbierto , None >
166 < cadena , Es bisiestro? >
167 < parCerrado , None >
168 < puntoComa , None >
169 < input , None >
170 < parAbierto , None >
171 < id , bis >
172 < parCerrado , None >
173 < puntoComa , None >
174 < return , None >
175 < parAbierto , None >
176 < parAbierto , None >
177 < id , a >
178 < mas , None >
179 < cteEnt , 0 >
180 < equals , None >
181 < cteEnt , 0 >
182 < parCerrado , None >
183 < parCerrado , None >
184 < puntoComa , None >
185 < llaveCerrado , None >
186 < function , None >
187 < id , dias >
188 < int , None >
189 < parAbierto , None >
190 < int , None >
191 < id , m >
192 < coma , None >
193 < int , None >
194 < id , a >
195 < parCerrado , None >
196 < llaveAbierto , None >
197 < let , None >
198 < int , None >
199 < id , dd >
200 < puntoComa , None >
201 < print , None >
202 < parAbierto , None >
203 < cadena , di cuantos dias tiene el mes >
204 < parCerrado , None >
205 < puntoComa , None >
206 < print , None >
207 < parAbierto , None >
208 < id , m >
209 < parCerrado , None >
210 < puntoComa , None >
211 < input , None >
212 < parAbierto , None >
213 < id , dd >

```

```

214 < parCerrado , None >
215 < puntoComa , None >
216 < if , None >
217 < parAbierto , None >
218 < id , bisiestro >
219 < parAbierto , None >
220 < id , a >
221 < parCerrado , None >
222 < parCerrado , None >
223 < id , dd >
224 < asig , None >
225 < id , dd >
226 < mas , None >
227 < cteEnt , 0 >
228 < puntoComa , None >
229 < return , None >
230 < id , dd >
231 < puntoComa , None >
232 < llaveCerrado , None >
233 < function , None >
234 < id , esFechaCorrecta >
235 < boolean , None >
236 < parAbierto , None >
237 < int , None >
238 < id , d >
239 < coma , None >
240 < int , None >
241 < id , m >
242 < coma , None >
243 < int , None >
244 < id , a >
245 < parCerrado , None >
246 < llaveAbierto , None >
247 < return , None >
248 < parAbierto , None >
249 < id , d >
250 < equals , None >
251 < id , dias >
252 < parAbierto , None >
253 < id , m >
254 < coma , None >
255 < id , a >
256 < parCerrado , None >
257 < parCerrado , None >
258 < puntoComa , None >
259 < llaveCerrado , None >
260 < eof , None >

```

Árbol de análisis sintáctico generado mediante VAST

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
```

Arbol

P (1)

B (4)

let
T (10)
string
id
puntoComa
P (1)
B (6)
S (14)
input
parAbierto
id
parCerrado
puntoComa
P (1)
B (4)
let
T (9)
boolean
id
puntoComa
P (1)
B (4)
let
T (9)
boolean
id
puntoComa
P (1)
B (4)
let
T (8)
int
id
puntoComa
P (1)
B (6)
S (11)
id

Sp (15)
 asig
 E (33)
 N (34)
 Z (35)
 R (47)
 cteEnt
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 puntoComa
 P (1)
 B (6)
 S (11)
 id
 Sp (15)
 asig
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (52)
 postIncrem
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 puntoComa
 P (1)
 B (6)
 S (11)
 id

Sp (15)
asig
E (33)
N (34)
Z (35)
R (48)
cadena
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
P (1)
B (6)
S (11)
id
Sp (15)
asig
E (33)
N (34)
Z (35)
R (49)
true
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
P (1)
B (6)
S (11)
id
Sp (15)
asig

E (33)
N (34)
Z (35)
R (50)
false
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
P (2)
F (26)
function
id
H (27)
T (10)
string
parAbierto
A (29)
T (10)
string
id
K (32)
lambda
parCerrado
llaveAbierto
C (20)
B (6)
S (11)
id
Sp (15)
asig
E (33)
N (34)
Z (35)
R (45)

id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 puntoComa
 C (20)
 B (5)
 if
 parAbierto
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 parCerrado
 S (11)
 id
 Sp (15)
 asig
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (51)

parAbierto

L (22)

E (33)

N (34)

Z (35)

R (45)

id

Rp (53)

lambda

O3 (44)

lambda

O2 (41)

lambda

O1 (38)

lambda

Q (25)

lambda

parCerrado

O3 (44)

lambda

O2 (41)

lambda

O1 (38)

lambda

puntoComa

C (20)

B (6)

S (11)

id

Sp (15)

asig

E (33)

N (34)

Z (35)

R (47)

cteEnt

O3 (44)

lambda

O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (20)
B (6)
S (12)
return
X (18)
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (21)
lambda
llaveCerrado
P (2)
F (26)
function
id
H (27)
T (10)
string
parAbierto
A (29)
T (10)
string

id
 K (32)
 lambda
 parCerrado
 llaveAbierto
 C (20)
 B (4)
 let
 T (8)
 int
 id
 puntoComa
 C (20)
 B (7)
 do
 llaveAbierto
 C (20)
 B (6)
 S (11)
 id
 Sp (15)
 asig
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (39)
 equals
 Z (35)
 R (45)
 id
 Rp (53)
 lambda

O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (20)
B (6)
S (13)
print
parAbierto
E (33)
N (34)
Z (35)
R (47)
cteEnt
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
puntoComa
C (20)
B (6)
S (11)
id
Sp (15)
asig
E (33)
N (34)
Z (35)
R (48)
cadena
O3 (44)
lambda

O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (21)
lambda
llaveCerrado
while
parAbierto
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
puntoComa
C (20)
B (6)
S (12)
return
X (18)
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)

lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 puntoComa
 C (21)
 lambda
 llaveCerrado
 P (1)
 B (6)
 S (11)
 id
 Sp (15)
 asig
 E (33)
 N (34)
 Z (35)
 R (46)
 parAbierto
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (51)
 parAbierto
 L (22)
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (51)
 parAbierto
 L (22)
 E (33)
 N (34)

Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 Q (25)
 lambda
 parCerrado
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 Q (25)
 lambda
 parCerrado
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 parCerrado
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 puntoComa
 P (1)

B (6)
S (13)
print
parAbierto
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
puntoComa
P (1)
B (4)
let
T (9)
boolean
id
puntoComa
P (2)
F (26)
function
id
H (27)
T (9)
boolean
parAbierto
A (29)
T (8)
int
id

K (32)
lambda
parCerrado
llaveAbierto
C (20)
B (4)
let
T (10)
string
id
puntoComa
C (20)
B (6)
S (13)
print
parAbierto
E (33)
N (34)
Z (35)
R (48)
cadena
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
puntoComa
C (20)
B (6)
S (14)
input
parAbierto
id
parCerrado
puntoComa
C (20)

B (6)
 S (12)
 return
 X (18)
 E (33)
 N (34)
 Z (35)
 R (46)
 parAbierto
 E (33)
 N (34)
 Z (35)
 R (46)
 parAbierto
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (42)
 mas
 R (47)
 cteEnt
 O3 (44)
 lambda
 O2 (39)
 equals
 Z (35)
 R (47)
 cteEnt
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda

parCerrado
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (21)
lambda
llaveCerrado
P (2)
F (26)
function
id
H (27)
T (8)
int
parAbierto
A (29)
T (8)
int
id
K (31)
coma
T (8)
int
id
K (32)
lambda
parCerrado

llaveAbierto

C (20)

B (4)

let

T (8)

int

id

puntoComa

C (20)

B (6)

S (13)

print

parAbierto

E (33)

N (34)

Z (35)

R (48)

cadena

O3 (44)

lambda

O2 (41)

lambda

O1 (38)

lambda

parCerrado

puntoComa

C (20)

B (6)

S (13)

print

parAbierto

E (33)

N (34)

Z (35)

R (45)

id

Rp (53)

lambda

O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
puntoComa
C (20)
B (6)
S (14)
input
parAbierto
id
parCerrado
puntoComa
C (20)
B (5)
if
parAbierto
E (33)
N (34)
Z (35)
R (45)
id
Rp (51)
parAbierto
L (22)
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)
lambda
O2 (41)

lambda
O1 (38)
lambda
Q (25)
lambda
parCerrado
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
parCerrado
S (11)
id
Sp (15)
asig
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (42)
mas
R (47)
cteEnt
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (20)
B (6)
S (12)

return
X (18)
E (33)
N (34)
Z (35)
R (45)
id
Rp (53)
lambda
O3 (44)
lambda
O2 (41)
lambda
O1 (38)
lambda
puntoComa
C (21)
lambda
llaveCerrado
P (2)
F (26)
function
id
H (27)
T (9)
boolean
parAbierto
A (29)
T (8)
int
id
K (31)
coma
T (8)
int
id
K (31)
coma

T (8)
 int
 id
 K (32)
 lambda
 parCerrado
 llaveAbierto
 C (20)
 B (6)
 S (12)
 return
 X (18)
 E (33)
 N (34)
 Z (35)
 R (46)
 parAbierto
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (39)
 equals
 Z (35)
 R (45)
 id
 Rp (51)
 parAbierto
 L (22)
 E (33)
 N (34)
 Z (35)
 R (45)

id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 Q (24)
 coma
 E (33)
 N (34)
 Z (35)
 R (45)
 id
 Rp (53)
 lambda
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 Q (25)
 lambda
 parCerrado
 O3 (44)
 lambda
 O2 (41)
 lambda
 O1 (38)
 lambda
 parCerrado
 O3 (44)
 lambda
 O2 (41)
 lambda

O1 (38)

lambda

puntoComa

C (21)

lambda

llaveCerrado

P (3)

eof

Volcado de la Tabla de Símbolos

```
1 TS GLOBAL #1
2 *Lexema: 'contador '
3 *Lexema: 'x'
```

4.1.2 Caso 2

Función potencia que devuelve un número a la potencia deseada, este caso correcto demuestra que crea una variable de la manera correcta con let para la inicialización y luego la asignación. Admitimos el do {S} while (E) con los comentarios de bloque `/**/`.

Código:

```
1 function potencia int (int z, int dim) {
2   let int s;
3   s = 0;
4   do{
5     z = z*z;
6     print(z);
7     s++;
8   } while(dim>s);
9   return z;
10 } /* fin de potencia*/
```

4.1.3 Caso 3

En este caso se comprueba con una simple función que devuelve string llamado demo, los operadores relaciones con el `==`, y la palabra reservada input para obtener variables del usuario desde el I/O. También podemos ver que con el return, podemos devolver no solo variables o enteros, si no que también cadenas de caracteres.

Código:

```
1 function demo string() { /* definición de la función demo, sin argumentos y
   devuelve un string */
2   let int v1;
3   let int v2;
4   let int v3;
5   print ("Escriba tres números: ");
6   input (v1);
7   input (v2);
8   input (v3);
9   if(v1==v2) return "Primer y segundo valor son idénticos";
10  if(v2==v3) return "Segundo y tercer valor son idénticos";
11  if(v1==v3) return "Primer y tercer valor son idénticos";
12 }
```

4.1.4 Caso 4

Función básica de calcular el factorial de un número. Comprobamos que sean todos los operadores relacionales y aritméticos aceptados, como son el equals (==), el por (*). La última sentencia hacemos comprobación sobre funciones anidadas, demostrando que el procesador lo admite.

Código:

```
1 function factorial int (int n){ /* n: parámetro formal de la función entera */
2   let int result;
3   let int aux;
4   result = 1;
5   aux = 2;
6
7   if(n == 0) return 1;
8   do {
9     result = result * aux;
10    aux++;
11  }while(n>aux);
12
13  return result;
14 } /* funcion representativa */
15
16 print(factorial(factorial(2)));
```

4.1.5 Caso 5

Este último caso correcto, tenemos una comparación de dos inputs tipo string y comprobamos los tipos booleanos, los cuales son admitidos por el lenguaje y que los “if” solo admiten una sola sentencia despues de su ejecución.

Código:

```
1 function compara boolean (string input1, string input2, string input3){
2   let boolean result;
3   result = false;
4
5   if(input1 == input2 && input2 == input3) result = true;
6   if(input1 == input2 && input2 == input3) print("Los 3 inputs recibidos son
   iguales");
7
8   return result;
9 } /* funcion representativa */
```

4.2 Incorrectos:

Formato:

- Breve explicación del código y los errores que se encuentran en el código que el procesador debería reconocer
- Código del caso escrito en Javascript PdL
- Listado de errores generados (para todos los casos)

4.2.1 Caso 1

Función básica que calcula si un año es bisiesto o no con tipos booleanos. Es un caso incorrecto por un error léxico ya que no se admiten en el lenguaje algunos tokens como son %, !=, al igual que los comentarios con //.

Código:

```
1 function bisiestro boolean (int a, int b, c) {
2   return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
3 } // fin de bisiestro: función lógica
```

Mensajes de error:

```
1 *****
2 Lexical error at line 2:
3   return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
4   ~~~~~
5 Simbolo: "%" no permitido.
6 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
7 *****
8
9
10 *****
11 Lexical error at line 2:
12   return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
13   ~~~~~
14 Simbolo: "%" no permitido.
15 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
16 *****
17
18
19 *****
20 Lexical error at line 2:
21   return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
22   ~~~~~
23 Simbolo: "!" no permitido.
24 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
25 *****
26
27
28 *****
29 Lexical error at line 2:
30   return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
31   ~~~~~
32 Simbolo: "%" no permitido.
33 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
34 *****
35
36
37 *****
38 Lexical error at line 3:
39 } // fin de bisiestro: funciÃ³n lÃ³gica
40   ~
41 Comentarios de tipo '//comentario' no estan permitidos
42 *****
43
44
45 *****
46 NonSupportedException at line 3:
47 } // fin de bisiestro: funciÃ³n lÃ³gica
48 ~
49 Esperaba uno de los siguientes sÃ­mbolos['mayor', 'equals', 'parCerrado', 'coma',
    'and', 'or', 'puntoComa', 'lambda']
```

```

50 *****
51
52
53 Error fatal, saliendo ...
54
55 Process finished with exit code 1

```

4.2.2 Caso 2

Bloque de código de factoriales y booleanos de sumas con un error sintáctico en la primera línea para la asignación conjunta con la inicialización de variables, en los comentarios estilo `/**` y el uso no aceptado de la resta con el menos `'-'`.

Código:

```

1 let int num = 1;
2
3 function factorial int (int x) {
4     if (x > 1)
5         return x * factorial (x - 1);
6     return 1;
7 }
8
9 function Suma boolean (int aux, int fin){
10     /* se define la función Suma que recibe dos enteros por valor */
11     /* usa la variable global x */
12     do{
13         aux = aux + factorial(aux-1)
14         x = x + 2
15     }while(fin>x)
16
17     return aux > 10000;
18 } // la función devuelve un lógico
19
20 function imprime (int a){
21     print (a);
22 }
23
24 imprime (factorial (Suma (5, 6)));

```

Mensajes de error:

```

1 *****
2 Lexical error at line 5:
3     return x * factorial (x - 1);
4                             ^~~~~~
5 Simbolo: "-" no permitido.
6 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
7 *****
8
9
10 *****
11 Lexical error at line 13:
12     aux = aux + factorial(aux-1)
13                             ^~~~~~
14 Simbolo: "-" no permitido.
15 No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
16 *****

```

```

17
18
19 *****
20 Lexical error at line 18:
21 } // la función devuelve un lógico
22   ~
23 Comentarios de tipo '//comentario' no estan permitidos
24 *****

```

4.2.3 Caso 3

Bloque de código que tiene un error léxico por usar una cadena que excede el tamaño máximo permitido además de asignar variables junto a la inicialización de variables.

Código:

```

1 /*
2 Imprimir en salida estándar con formateo de variables entre la cadena de caracteres
3 */
4
5 let int number; number = 32768;
6 let int ten = 10;
7 print('Fifteen is ' + five '+' ten 'and not ' + 2 * five '+' ten.'.');
8 print("An int variable is lower than " + number);
9 let string test =
10   "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz";
11 print(test);

```

Mensajes de error:

```

1 *****
2 Lexical error at line 3:
3 let int number; number = 32768;
4   ~~~~~~
5 Dígito con valor mayor al permitido (32768) en el sistema
6 *****

```

4.2.4 Caso 4

Este código del caso 4 comprueba el tipo de la función (void, que no está reconocida) además del tipo obligatorio dentro del paréntesis de los inputs. Esto provoca un error sintáctico al no usar o poner los tipos de variable adecuados.

Código:

```

1 function comparacion void (*) {
2   do{
3     print("ERROR")
4   }while(input>1)
5 }

```

Mensajes de error:

```

1 *****
2 TypeError at line 1:
3 function comparacion void (*) {
4   ~~~~~~
5 Tipo de función no aceptado. Debe usar ['int', 'boolean', 'string'] o "" (no poner
   nada para void)
6 *****

```


4.2.5 Caso 5

Función de factorial de un número pero con bucle for, la cual no es reconocida en el lenguaje. Error sintáctico en la función con paréntesis faltante.

Código:

```
1 function FactorialFor int ( int n){
2   let int i;
3   let int factorial; factorial = 1;
4   for (i = 1; i <= n; i++)
5   {
6     factorial *= i;
7   }
8   return factorial;
9 }
```

Mensajes de error:

```
1 *****
2 NonSupportedException at line 4:
3   for (i = 1; i <= n; i++)
4       ~~~~
5 Esperaba uno de los siguientes símbolos['mayor', 'equals', 'parCerrado', 'coma',
   'and', 'or', 'puntoComa', 'lambda']
6 *****
```