

Javascript PdL

Nicolás Cossío Miravalles, Bárbara Rodríguez Ruiz, Huangjue He

Enero 2022

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

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 lenguaje](#).

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:

- Sí 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

3.1.1.1 Palabras reservadas

let : < let, - >

```

function : < function, - >
return:   < return, - >
if:       < if, - >
else :    < else, - >
input :   < input, - >
print :   < print, - >
while :   < while, - >
do:       < do, - >
true :    < boolT, - >
false :   < boolF, - >
int :     < int, - >
boolean : < boolean, - >
string :  < string, - >

```

3.1.1.2 Operadores

Operadores Aritméticos:

```

+ : < mas, - >
* : < por, - >

```

Operadores Lógicos:

```

&& : < and, - >
|| : < or, - >

```

Operadores relacionales:

```

== : < equals, - >
> : < mayor, - >

```

3.1.1.3 Resto de tokens

```

Identificador : < id, ptoTS >
Asignacion =  : < asig, - >
Cadena       : < cadena, laCadena >
Entero       : < cteEnt, valor >
++          : < postIncr, - >
,           : < coma, - >
;           : < puntoComa, - >
(           : < parAbierto, - >
)           : < parCerrado, - >
{           : < llaveAbierto, - >
}           : < llaveCerrado, - >
eof         : < eof, - >

```

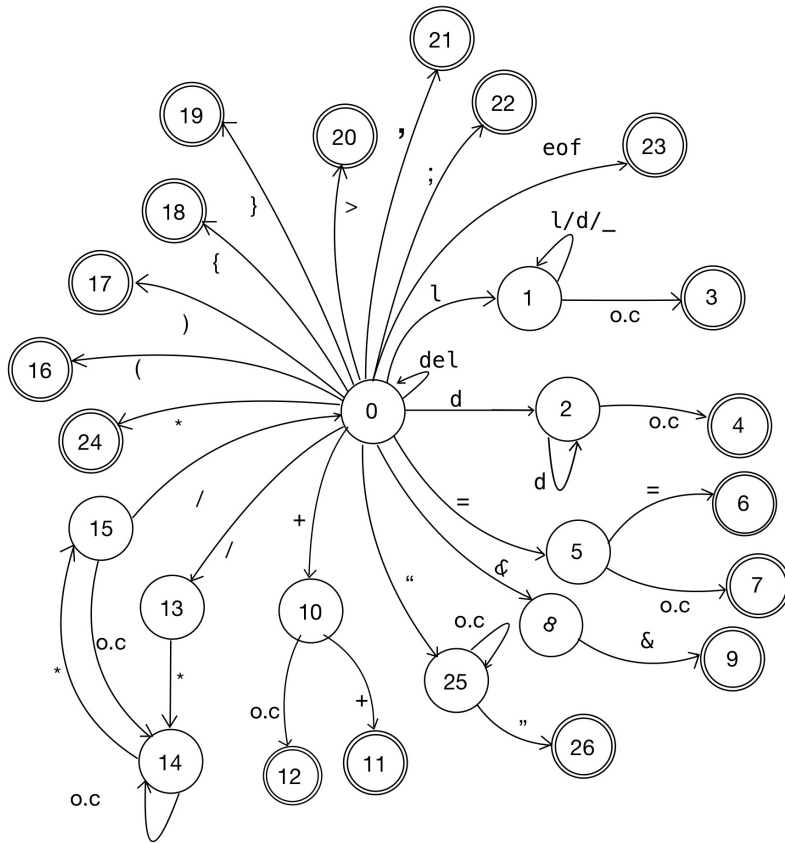
3.1.2 Gramática Regular

```

S -> lA | dC | ( | ) | { | } | delS | =E | &D | > | +G | * | , | ; | eof | /B | "J
A -> lA | dA | _A | λ
B -> *H
C -> dC | λ
D -> &
E -> = | λ
G -> + | λ
H -> λH | *I
I -> /S
J -> λJ | "K
K -> λ
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

```
- LEER: lee el siguiente carácter del fichero fuente. car := leer()
    En todas las transiciones menos en las transiciones etiquetadas con O.C, excepto en
25->25
- CONC: forma una cadena (lexema). Siempre después de leer(), lex:=lex⊕car
    En las transiciones: 13->14, 14->15 | 0->1, 1->1, x->x //por hacer, cadenas
- VALOR: convierte un carácter a su entero correspondiente, entero:=valor(carácter)
- GENTOKEN: genera un token que el A.Léxico pasa al sintáctico
0->2: num = valor(car)
2->2: num = num*10+valor(car)
2->4: if (num < 32768) genToken(cteEnt, num)
    else error()

25->26:
if (length(lex) < 65) genToken(cadena, lex)
else error()
```

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()

```

if transicion is not 1->3, 2->4 {
    a:=TEstadosFinales[numEstadoFinal]
    if a is not null genToken( a , )
}

```

```
buscarTPR(lex): devuelve el código del token que coincide con un lexema dado,  
codigo:=buscarTPR(lex)
```

```

TPR []:=lista de códigos de tokens de palabras reservadas que permiten crear un token
al usarse como argumento de genToken( )
1->3:
zona_decl := boolean que indica si se trata de una declaracion, global = true y local =
false
p:= buscarTPR(lex)
if ( p is not null ) genToken( p , - ) //genera token de palabra reservada
else { //identificador
    p:= buscarTS(lex)
    if ( p is not null ) //ya está declarada
        genToken(id, p)
    else { //no está declarada
        p:=añadirTS_activa(lexema) //AñadirTS devuelve un pto. al id
        genToken(id, p)
    }
}
}

```

3.1.5 Errores que recoge el autómeta

ERROR: cualquier transición no recogida en el autómeta corresponde a un caso de error. También son errores todos los lanzados desde las acciones semánticas, que son los siguientes:

- Entero sobrepasa valor máximo permitido
- String sobrepasa longitud máxima permitida

3.1.6 Tabla de símbolos - Diseño general

Todas las tablas tendrán lexema y tipo, pero el resto de los Tributos de la tabla dependerán del tipo.

Para enteros, reales, cadenas, lógicos... tendremos una tabla con el siguiente formato:

TS 1##:

```

* LEXEMA : 'a'
Atributos:
+ tipo: 'entero'
+ despl : 0

```

Para un array:

TS 2##:

```

* LEXEMA : 'a'
Atributos:
+ tipo: 'entero'
+ despl : 0
? núm de dimensiones, límite inf y límte sup de cada dimensión,...

```

Para funciones la tabla seguirá el formato:

TABLA FUNCION SUMA ##3:

```

*LEXEMA : 'suma'
Atributos:
+ tipo: 'funcion'
+ numParam: 2
+ TipoParam01: 'ent'
+ TipoParam2: 'real'
+ TipoRetorno: 'ent'
+ EtiquFuncion: '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

```
P -> B P | F P | eof
B -> let T id ; | if ( E ) S | S | while ( E ) { C }
T -> int | boolean | string
S -> id = E ; | return X ; | id ( L ) | print ( E ) ; | input ( id ) ;
X -> E | λ
C -> B C | λ
L -> E Q | λ
Q -> , E Q | λ
F -> function id H ( A ) { C }
H -> T | λ
A -> T id K | λ
K -> , T id K | λ
E -> E && R | R
R -> R > U | U
U -> U + V | V
V -> id | ( E ) | id (L) | entero | cadena
```

3.2.3 Gramática transformada

```
## indica comentario dentro de la definición de la gramática
P -> B P | F P | eof
B -> let T id ; | if ( E ) S | S | do { C } while ( E ) ;
T -> int | boolean | string
S -> id S' | return X ; | print ( E ) ; | input ( id ) ;
S' -> = E | ( L ) | ++
X -> E | λ
C -> B C | λ
L -> E Q | λ
Q -> , E Q | λ
F -> function id H ( A ) { C }
H -> T | λ
A -> T id K | λ
K -> , T id K | λ
## precedencia menos a más: {||} -> {&&} -> {==} -> {>} -> {+} -> {*} -> {++}
E -> N O1 ##operadores aritméticos
N -> Z O2 ##operadores relacionales
Z -> R O3 ##operadores lógicos
O1 -> || N O1 | && N O1 | λ
O2 -> == N O2 | > N O2 | λ
O3 -> + R O3 | * R O3 | λ
R -> id R' | ( E ) | entero | cadena | true | false
R' -> ( L ) | ++ | λ
```

Justificación de que es gramática LL(1):

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

Para los consecuentes que pueden derivar a λ :

- $O1 \rightarrow \text{First}(O1) \cap \text{Follow}(O1) = \emptyset$
 - $\text{First}(O1) = \{ +, *, \lambda \}$
 - $\text{Follow}(O1) = \text{Follow}(E) = \text{First}(Q) + \text{Follow}(X) + \{ ;,) \} = \{ ;,), \text{coma} \}$

- $O2 \rightarrow \text{First}(O2) \cap \text{Follow}(O2) = \emptyset$
 - $\text{First}(O2) = \{ ==, >, \lambda \}$
 - $\text{Follow}(O2) = \text{Follow}(N) = \text{First}(O1) + \text{First}(O2) + \text{Follow}(E) = \{ +, *, ==, >, ;,), coma \}$
- $O3 \rightarrow \text{First}(O3) \cap \text{Follow}(O3) = \emptyset$
 - $\text{First}(O3) = \{ \&\&, ||, \lambda \}$
 - $\text{Follow}(O3) = \text{Follow}(Z) = \text{First}(O2) + \text{Follow}(N) = \{ ==, >, +, *, ;,), coma \}$
- $X \rightarrow E \mid \lambda \rightarrow \text{First}(X) \cap \text{Follow}(X) = \emptyset$
 - $\text{First}(X) = \{ id, (, cteEnt, cadena, boolT, boolF, \lambda \}$
 - $\text{Follow}(X) = \{ ; \}$

$$\text{First}(V) \subseteq \text{First}(U) \subseteq \text{First}(R) \subseteq \text{First}(E) \subseteq \text{First}(X)$$

- $C \rightarrow B C \mid \lambda \rightarrow \text{First}(B C) \cap \text{Follow}(C) = \emptyset$
 - $\text{First}(C) = \text{First}(B) = \{ let, if, id, return, print, input, do \}$
 - $\text{Follow}(C) = \{ llaveAbierto \}$
- $L \rightarrow E Q \mid \lambda \rightarrow \text{First}(EQ) \cap \text{Follow}(L) = \emptyset$
 - $\text{First}(L) = \text{First}(E) = \{ id, (, cteEnt, cadena, boolT, boolF \}$
 - $\text{Follow}(L) = \{) \}$

$$\text{First}(V) \subseteq \text{First}(U) \subseteq \text{First}(R) \subseteq \text{First}(E) \subseteq \text{First}(EQ)$$

- $Q \rightarrow , E Q \mid \lambda \rightarrow \text{First}(, E Q) \cap \text{Follow}(Q) = \emptyset$
 - $\text{First}(Q) = \{ , \}$
 - $\text{Follow}(Q) = \text{Follow}(L) = \{) \}$
- $H \rightarrow T \mid \lambda \rightarrow \text{First}(T) \cap \text{Follow}(H) = \emptyset$
 - $\text{Follow}(H) = \{ (\}$
- $A \rightarrow T id K \mid \lambda \rightarrow \text{First}(T id K) \cap \text{Follow}(A) = \emptyset$
 - $\text{First}(A) = \text{First}(T) = \{ int, boolean, string \}$
 - $\text{Follow}(A) = \{) \}$

$$\text{First}(T) \subseteq \text{First}(T id K)$$

- $K \rightarrow , T id K \mid \lambda \rightarrow \text{First}(, T id K) \cap \text{Follow}(K) = \emptyset$
 - $\text{First}(K) = \{ , \}$
 - $\text{Follow}(K) = \text{Follow}(A) = \{) \}$
- $R' \rightarrow \text{First}(R) \cap \text{Follow}(R') = \emptyset$
 - $\text{First}(R') = \{ (, ++, \lambda \}$
 - $\text{Follow}(R') = \text{Follow}(R) = \text{First}(O) + \text{Follow}(E) = \{ \&\&, +, *, ==, >, \lambda \} + \{ coma, puntoComa,) \} = \{) \}$

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

3.2.4 Reglas

```
1 - P -> B P
2 - P -> F P
3 - P -> eof
4 - B -> let T id ;
```

```

5 - B -> if ( E ) S
6 - B -> S
7 - B -> do { C } while ( E );
8 - T -> int
9 - T -> boolean
10 - T -> string
11 - S -> id S' ;
12 - S -> return X ;
13 - S -> print ( E ) ;
14 - S -> input ( id ) ;
15 - S' -> asig E
16 - S' -> ( L )
17 - S' -> ++
18 - X -> E
19 - X -> λ
20 - C -> B C
21 - C -> λ
22 - L -> E Q
23 - L -> λ
24 - Q -> , E Q
25 - Q -> λ
26 - F -> function id H ( A ) { C }
27 - H -> T
28 - H -> λ
29 - A -> T id K
30 - A -> λ
31 - K -> , T id K
32 - K -> λ
33 - E -> N O1
34 - N -> Z O2
35 - Z -> R O3
36 - O1 -> || N O1
37 - O1 -> && N O1
38 - O1 -> λ
39 - O2 -> == Z O2
40 - O2 -> > Z O2
41 - O2 -> λ
42 - O3 -> + R O3
43 - O3 -> * R O3
44 - O3 -> λ
45 - R -> id R'
46 - R -> ( E )
47 - R -> entero
48 - R -> cadena
49 - R -> true
50 - R -> false
51 - R' -> ( L )
52 - R' -> ++
53 - R' -> λ

```

3.2.5 Gramática para el árbol sintáctico

```

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 }
NoTerminales = { A B C E F H K L N O1 O2 O3 P Q R Rp S Sp T X Z }
Axioma = P

```

```

Producciones = {
P -> B P
P -> F P
P -> eof
B -> let T id puntoComa
B -> if parAbierto E parCerrado S
B -> S
B -> do llaveAbierto C llaveCerrado while parAbierto E parCerrado puntoComa
T -> int
T -> boolean

```



```

T -> string
S -> id Sp puntoComa
S -> return X puntoComa
S -> print parAbierto E parCerrado puntoComa
S -> input parAbierto id parCerrado puntoComa
Sp -> asig E
Sp -> parAbierto L parCerrado
Sp -> postIncrement
X -> E
X -> lambda
C -> B C
C -> lambda
L -> E Q
L -> lambda
Q -> coma E Q
Q -> lambda
F -> function id H parAbierto A parCerrado llaveAbierto C llaveCerrado
H -> T
H -> lambda
A -> T id K
A -> lambda
K -> coma T id K
K -> lambda
E -> N O1
N -> Z O2
Z -> R O3
O1 -> or N O1
O1 -> and N O1
O1 -> lambda
O2 -> equals Z O2
O2 -> mayor Z O2
O2 -> lambda
O3 -> mas R O3
O3 -> por R O3
O3 -> lambda
R -> id Rp
R -> parAbierto E parCerrado
R -> cteEnt
R -> cadena
R -> true
R -> false
Rp -> parAbierto L parCerrado
Rp -> postIncrement
Rp -> lambda
}

```

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

```

tipo TS:
    .crear() -> crea una tabla de símbolos vacía
    .destruir( tabla ) -> destruye la tabla de símbolos "tabla"
    .desp = desplazamiento actual de la tabla, última posición libre
    .insertatId(id) -> se inserta en la última posición el identificador, creando una
nueva entrada y poniendo el desplazamiento
        de la entrada como el valor actual de TS.desp
    .insertarTipoId ( pos, tipo ) -> inserta el tipo de la variable (id.pos) en la TS

    .insertarTipoParam ( tipo1 x tipo2 x ... ) -> inserta un producto cartesiano de los
tipos de los
    parámetros de los argumentos de una función
    .insertarTipoDev ( tipo ) -> inserta el tipo que devuelve una función en la tabla
general

    .buscarId( id.pos ) -> busca un identificador en la tabla, devuelve true si existe,
false si no
    .getTipoParam( id.pos ) -> devuelve el valor (producto cartesiano de tipos) que
identifica los tipos
        de los argumentos de la función id
tipo id:
    .pos = posición en la TS que corresponda, adquiere el valor de TS.pos al insertarse
con
    TS.insertarId(id)

tipo reglas: son todas las reglas que contiene la gramática
    .tipo = tipo que devuelve la regla (boolean, string, entero o int, vacio, function)
    puede ser una producto cartesiano de tipos o solo uno
    .tipoDev= devolución de una regla

```

3.3.2 Esquema de Traducción

```

P' -> { TSG = TS.crear() TSactual = TSG } P { TS.destruir(TSG) }
P -> B P
P -> F P
P -> eof
B -> let T id puntoComa
    { if TSactual.buscarId(id) == false )
      then
        id.pos = TSactual.insertarId( id )
        TSactual.insertarTipoId( id.pos, T.tipo )
        TS.despl = displ + T.ancho
    }
B -> if parAbierto E parCerrado S
    { if (E.tipo != boolean)
      then error("El tipo de E tiene que ser boolean ya que nos encontramos en la
condición de if")
    }
B -> S
B -> do llaveAbierto C llaveCerrado while parAbierto E parCerrado puntoComa
    { if (E.tipo != boolean)
      then error("La condición del while debe ser de tipo booleano")
    }
T -> int { T.tipo:= int, T.ancho:= 1 }
T -> boolean { T.tipo:= boolean, T.ancho:= 1 }
T -> string { T.tipo:= string, T.ancho:= 64}
S -> id S' puntoComa
    { if ( TSactual.buscarId( id ) == false ) ## no está en tabla local
      then if (TSG.buscarId( id ) == true ) ## sí está en global-> llamada a función o
asignación a variable
        then if (S'.tipo != TSG.getTipoParam(id) ) ## argumentos no coinciden con los
de la función
          then if (id.tipo != S'.tipo ) ## asignación
            then error( "Tipos en la asignación no coinciden")
            ## funcion
          else error("Argumentos no coinciden con los de la función")
        else if ( S'.tipo == postIncrem and id.tipo != cteE )
          then error("El operador post incremento solo es aplicable a variables del tipo

```

```

entero")
    else ## es una declaracion e inicialización de una variable global i.e (a = 5)
        id.pos = TSG.insertarId( id )
        TSG.insertarTipo ( id.pos, S'.tipo )
        ancho = if (S'.tipo == string ) else 1
        TSG.pos = TSG.pos + ancho
    }
S -> return X puntoComa
{
    S.tipo = tipo_ok
    S.tipoRet = X.tipo
}
S -> print parAbierto E
{
    S.tipoRet = vacio
    S.tipo = tipo_ok if (E.tipo == string ) else error("La función print solo acepta
parámetros de tipo string")
}
parCerrado puntoComa
S -> input parAbierto id
{ if (TSactual.buscarId(id) == true )
    then if TSactual.buscarTipo(id) not in (string, cteEnt)
        then error("La función input debe recibir una variable de tipo string o entero")
    else if (TSG.buscarId(id) == true )
        then if TSG.buscarTipo(id) not in (string, cteEnt)
            then error("La función input debe recibir una variable de tipo string o entero")
        else error("Variable no ha sido previamente declarada")
    } parCerrado puntoComa
S' -> asig E puntoComa { S'.tipo = E.tipo }
S' -> parAbierto L parCerrado { S'.tipo = L.tipo }
S' -> postIncrement { S'.tipo = postIncrement }
X -> E { X.tipo = E.tdefipo }
X -> lambda { X.tipo = vacio }
C -> B C
C -> lambda { C.tipo = vacio }
L -> E Q { L.tipo = L.tipo x Q.tipo } ## tipo1 x tipo2 x tipo3 o vacio
L -> lambda { L.tipo = vacio }
Q -> coma E {if E.tipo != vacio)
    then Q.tipo = Q.tipo x E.tipo }
Q
Q -> lambda { Q.tipo = vacio }
F -> function id
{ tabla = crearTS()
  TSactual = tabla
  Desp_tabla1 = 0
  TSG.insertarId( id ) }
H
{ TSactual.insertartipoTS (H.tipo);
  TSG.insertarTipoDev( id, H.tipo )}
parAbierto A parCerrado
{ TSG.insertarTipoParam( id.pos, A.tipo )} ## sintáctico solo acepta boolean string o
int, si no es ninguno dará error
llaveAbierto C llaveCerrado
{ tabla.destruir()
  TSactual = TSG }
H -> T { H.tipo = T.tipo }
H -> lambda { H.tipo = vacio }
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 ... )
A -> lambda { A.tipo = vacio }
K -> coma T { K.tipo = T.tipo x K.tipo } id K
K -> lambda { K.tipo = vacio }
E -> N O1 { E.tipo = "cteEnt" }
N -> Z O2 { N.tipo = "cteEnt" }
Z -> R O3 { Z.tipo = "boolean" }
O1 -> mas N { if R.tipo != cteEnt
    then error("Operador + solo acepta datos enteros")
    } O1
O1 -> por N { if R.tipo != cteEnt
    then error("Operador * solo acepta datos enteros")

```

```

    } O1
O1 -> lambda { O1.tipo = tipo_ok}
O2 -> equals Z { if Z.tipo != cteEnt
    then error("Operador > solo acepta datos enteros")
    } O2
O2 -> mayor Z { if Z.tipo != cteEnt
    then error("Operador > solo acepta datos enteros")
    } O2
O2 -> lambda { O1.tipo = "boolean"}
O3 -> or R { if R.tipo != boolean
    then error("Operador || solo acepta datos lógicos")
    } O3
O3 -> and R { if R.tipo != boolean
    then error("Operador || solo acepta datos lógicos")
    else if O1.tipo == vacio then O3.tipo = tipo_ok } O3
O3 -> lambda { O3.tipo = tipo_ok }
R -> id R' { if (R'.tipo == postIncrement and id.tipo != cteEnt )
    then error("El operador post incremento solo es aplicable a variables del tipo
entero")
    else if (R'.tipo != vacio ) ## se trata de una llamada a una función
    then if ( TSG.buscarId( id ) == false )
        then error("Error la función no ha sido declarada previamente")
        else if (R'.tipo != TSG.getParam( id ) )
            then error("Tipos de los atributos incorrectos en llamada a función")
            else R.tipo = TSG.getTipoDev( id )
    else:
        R.tipo = id.tipo ## habria que buscarlo en ambas tablas para ver en
        ## en cual esta y coger el tipo de la tabla
    }
R -> parAbierto E parCerrado { R.tipo:= E.tipo }
R -> cteEnt { R.tipo:= int, R.ancho:= 1 }
R -> cadena { R.tipo:= string R.ancho:= 64 }
R -> boolT { R.tipo:= boolean R.ancho:= 1 }
R -> boolF { R.tipo:= boolean R.ancho:= 1 }
R' -> lambda { R'.tipo = vacio }
R' -> parAbierto L parCerrado
    { R'.tipo = L.tipo }
R' -> postIncrement { R'.tipo = postIncrement }

```

3.4 Tabla de Símbolos

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

```

-----
TABLA PRINCIPAL #0

* LEXEMA : "demo"
  ATRIBUTOS :
+ Tipo: funcion
+ numParam: 0
+ TipoRetorno: string
-----
-----
TABLA de funcion "demo" #1

* LEXEMA : "v1"
  ATRIBUTOS :
+ Tipo: int
+ Despl: 0

* LEXEMA : "v2"
  ATRIBUTOS :
+ Tipo: int
+ Despl: 1

* LEXEMA : "v3"
  ATRIBUTOS :
+ Tipo: int

```

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 autocorrigan 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 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:

```
*****
NonDeclaredError at line 2:
  estafuncionnoexiste();
                        ^~~~~~
Error la función estafuncionnoexiste no ha sido declarada previamente
*****

*****
Lexical error at line 34:
  print ('Es bisiesto?');
        ^~~~~~
Cadena se debe especificar entre " ", no con ' '. Corregido
*****

*****
TypeError at line 7:
  input (v1);
        ^~~~~~
Variable a es de tipo boolean, input() debe recibir una variable de tipo string o
entero
*****

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:

```
let string cadena;
input(cadena);
let boolean logico1;
let boolean logico2;
let int int2;
int1 = 000000378;

int2 = int1++;
cadena = "string";
logico1 = true;
logico1 = false;
function ff string(string ss)
{
    logico2 = logico1;
    if (logico2) cadena = ff (ss);
    varglobal = 78;
    return cadena;
}

function funcion string (string logico2)
{
    let int var;
    do {
        logico1 = int1 == int2; print(0); logico2 = "";
    } while (logico1);
    return logico2;
}

cadena = (ff(funcion(cadena)));
print(cadena);
let boolean booleano;
function bisiesto boolean (int a)
{ let string bis;
  print ("Es bisiesto?");
  input(bis);
  return ((a + 4 == 0));
}
function dias int (int m, int a)
{
    let int dd;
    print ("di cuantos dias tiene el mes ");
    print (m);
    input(dd);
    if (bisiesto(a)) dd = dd + 1;
    return dd;
}
```

```

}
function esFechaCorrecta boolean (int d, int m, int a)
{
    return (d == dias (m, a));
}

```

Listado de tokens:

```

< let , None >
< string , None >
< id , cadena >
< puntoComa , None >
< input , None >
< parAbierto , None >
< id , cadena >
< parCerrado , None >
< puntoComa , None >
< let , None >
< boolean , None >
< id , logico1 >
< puntoComa , None >
< let , None >
< boolean , None >
< id , logico2 >
< puntoComa , None >
< let , None >
< int , None >
< id , int2 >
< puntoComa , None >
< id , int1 >
< asig , None >
< cteEnt , 0 >
< puntoComa , None >
< id , int2 >
< asig , None >
< id , int1 >
< postIncrement , None >
< puntoComa , None >
< id , cadena >
< asig , None >
< cadena , string >
< puntoComa , None >
< id , logico1 >
< asig , None >
< true , None >
< puntoComa , None >
< id , logico1 >
< asig , None >
< false , None >
< puntoComa , None >
< if , None >
< parAbierto , None >
< id , int2 >
< mas , None >
< id , int2 >
< equals , None >
< id , int1 >
< or , None >
< id , logico1 >
< and , None >
< id , logico2 >
< parCerrado , None >
< print , None >
< parAbierto , None >
< cadena , Como narices se ha evaluado esto a true >
< parCerrado , None >
< puntoComa , None >
< function , None >
< id , ff >
< string , None >
< parAbierto , None >

```

```
< string , None >
< id , ss >
< parCerrado , None >
< llaveAbierto , None >
< id , logico2 >
< asig , None >
< id , logico1 >
< puntoComa , None >
< if , None >
< parAbierto , None >
< id , logico2 >
< parCerrado , None >
< id , cadena >
< asig , None >
< id , ff >
< parAbierto , None >
< id , ss >
< parCerrado , None >
< puntoComa , None >
< id , varglobal >
< asig , None >
< cteEnt , 0 >
< puntoComa , None >
< return , None >
< id , cadena >
< puntoComa , None >
< llaveCerrado , None >
< function , None >
< id , funcion >
< string , None >
< parAbierto , None >
< string , None >
< id , logico2 >
< parCerrado , None >
< llaveAbierto , None >
< let , None >
< int , None >
< id , var >
< puntoComa , None >
< do , None >
< llaveAbierto , None >
< id , logico1 >
< asig , None >
< id , int1 >
< equals , None >
< id , int2 >
< puntoComa , None >
< print , None >
< parAbierto , None >
< cteEnt , 0 >
< parCerrado , None >
< puntoComa , None >
< id , logico2 >
< asig , None >
< cadena , >
< puntoComa , None >
< llaveCerrado , None >
< while , None >
< parAbierto , None >
< id , logico1 >
< parCerrado , None >
< puntoComa , None >
< return , None >
< id , logico2 >
< puntoComa , None >
< llaveCerrado , None >
< id , cadena >
< asig , None >
< parAbierto , None >
< id , ff >
```



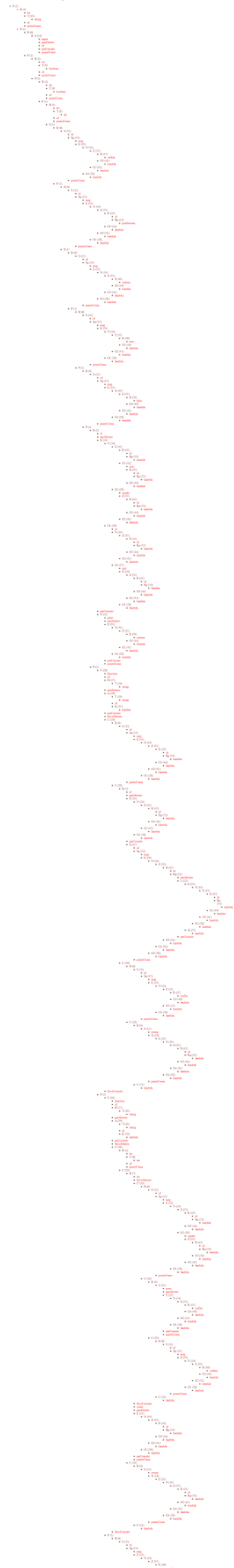
```
< parAbierto , None >
< id , funcion >
< parAbierto , None >
< id , cadena >
< parCerrado , None >
< parCerrado , None >
< parCerrado , None >
< puntoComa , None >
< print , None >
< parAbierto , None >
< id , cadena >
< parCerrado , None >
< puntoComa , None >
< let , None >
< boolean , None >
< id , booleano >
< puntoComa , None >
< function , None >
< id , bisiesto >
< boolean , None >
< parAbierto , None >
< int , None >
< id , a >
< parCerrado , None >
< llaveAbierto , None >
< let , None >
< string , None >
< id , bis >
< puntoComa , None >
< print , None >
< parAbierto , None >
< cadena , Es bisiesto? >
< parCerrado , None >
< puntoComa , None >
< input , None >
< parAbierto , None >
< id , bis >
< parCerrado , None >
< puntoComa , None >
< return , None >
< parAbierto , None >
< parAbierto , None >
< id , a >
< mas , None >
< cteEnt , 0 >
< equals , None >
< cteEnt , 0 >
< parCerrado , None >
< parCerrado , None >
< puntoComa , None >
< llaveCerrado , None >
< function , None >
< id , dias >
< int , None >
< parAbierto , None >
< int , None >
< id , m >
< coma , None >
< int , None >
< id , a >
< parCerrado , None >
< llaveAbierto , None >
< let , None >
< int , None >
< id , dd >
< puntoComa , None >
< print , None >
< parAbierto , None >
< cadena , di cuantos dias tiene el mes >
< parCerrado , None >
```

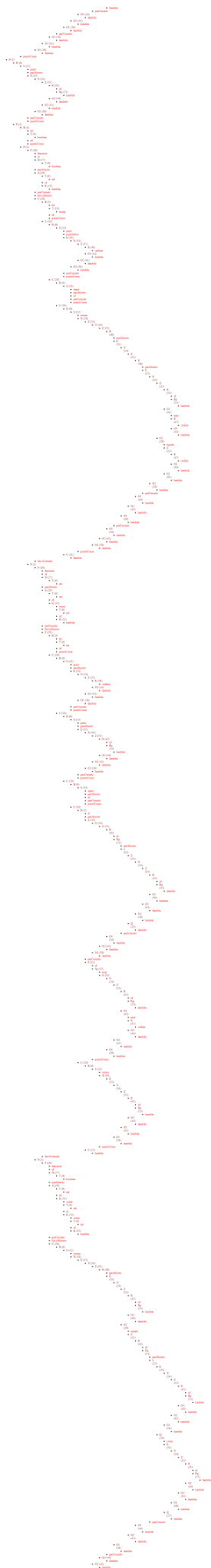
```

< puntoComa , None >
< print , None >
< parAbierto , None >
< id , m >
< parCerrado , None >
< puntoComa , None >
< input , None >
< parAbierto , None >
< id , dd >
< parCerrado , None >
< puntoComa , None >
< if , None >
< parAbierto , None >
< id , bisiesto >
< parAbierto , None >
< id , a >
< parCerrado , None >
< parCerrado , None >
< id , dd >
< asig , None >
< id , dd >
< mas , None >
< cteEnt , 0 >
< puntoComa , None >
< return , None >
< id , dd >
< puntoComa , None >
< llaveCerrado , None >
< function , None >
< id , esFechaCorrecta >
< boolean , None >
< parAbierto , None >
< int , None >
< id , d >
< coma , None >
< int , None >
< id , m >
< coma , None >
< int , None >
< id , a >
< parCerrado , None >
< llaveAbierto , None >
< return , None >
< parAbierto , None >
< id , d >
< equals , None >
< id , dias >
< parAbierto , None >
< id , m >
< coma , None >
< id , a >
< parCerrado , None >
< parCerrado , None >
< puntoComa , None >
< llaveCerrado , None >
< eof , None >

```

Árbol de análisis sintáctico generado mediante VAST





Volcado de la Tabla de Símbolos

```
TS GLOBAL #1
*Lexema: 'contador'
*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:

```
function potencia int (int z, int dim) {
  let int s;
  s = 0;
  do{
    z = z*z;
    print(z);
    s++;
  } while(dim>s);
  return z;
} /* 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:

```
function demo string() { /* definición de la función demo, sin argumentos y devuelve un string */
  let int v1;
  let int v2;
  let int v3;
  print ("Escriba tres números: ");
  input (v1);
  input (v2);
  input (v3);
  if(v1==v2) return "Primer y segundo valor son idénticos";
  if(v2==v3) return "Segundo y tercer valor son idénticos";
  if(v1==v3) return "Primer y tercer valor son idénticos";
}
```

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:

```
function factorial int (int n){ /* n: parámetro formal de la función entera */
  let int result;
  let int aux;
  result = 1;
  aux = 2;

  if(n == 0) return 1;
  do {
    result = result * aux;
    aux++;
  }while(n>aux);

  return result;
} /* funcion representativa */

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:

```
function compara boolean (string input1, string input2, string input3){
  let boolean result;
  result = false;

  if(input1 == input2 && input2 == input3) result = true;
  if(input1 == input2 && input2 == input3) print("Los 3 inputs recibidos son iguales");

  return result;
} /* 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:

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

Mensajes de error:

Lexical error at line 2:

```
return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
      ^~~~~~
```

Simbolo: "%" no permitido.

No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados

Lexical error at line 2:

```
return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
      ^~~~~~
```

Simbolo: "%" no permitido.

No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados

Lexical error at line 2:

```
return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
      ^~~~~~
```

Simbolo: "!" no permitido.

No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados

Lexical error at line 2:

```
return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);
      ^~~~~~
```

Simbolo: "%" no permitido.

No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados

Lexical error at line 3:

```
} // fin de bisiestos: función lógica
^~
```

Comentarios de tipo '//comentario' no estan permitidos

NonSupportedOperationError at line 3:

```
} // fin de bisiestos: función lógica
^~
```

Esperaba uno de los siguientes símbolos['mayor', 'equals', 'parCerrado', 'coma', 'and', 'or', 'puntoComa', 'lambda']

Error fatal, saliendo ...

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:

```
let int num = 1;

function factorial int (int x) {
  if (x > 1)
    return x * factorial (x - 1);
  return 1;
}

function Suma boolean (int aux, int fin){
  /* se define la función Suma que recibe dos enteros por valor */
  /* usa la variable global x */
  do{
    aux = aux + factorial(aux-1)
    x = x + 2
  }while(fin>x)

  return aux > 10000;
} // la función devuelve un lógico

function imprime (int a){
  print (a);
}

imprime (factorial (Suma (5, 6)));
```

Mensajes de error:

```
*****
Lexical error at line 5:
    return x * factorial (x - 1);
                        ^~~~~~

Simbolo: "-" no permitido.
No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
*****

Lexical error at line 13:
    aux = aux + factorial(aux-1)
                        ^~~~~~

Simbolo: "-" no permitido.
No pertenece al lenguaje, consulte la documentacion para ver caracteres aceptados
*****

Lexical error at line 18:
} // la función devuelve un lógico
^~

Comentarios de tipo '//comentario' no estan permitidos
*****
```

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:

```
/*
Imprimir en salida estándar con formateo de variables entre la cadena de caracteres
*/

let int number; number = 32768;
let int ten = 10;
print('Fifteen is ' + five '+' ten 'and not ' + 2 * five '+' ten'.');
print("An int variable is lower than " + number);
let string test =
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz";
print(test);
```

Mensajes de error:

```
*****
Lexical error at line 3:
let int number; number = 32768;
                        ^~~~~~
Digito con valor mayor al permitido (32768) en el sistema
*****
```

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:

```
function comparacion void (*) {
  do{
    print("ERROR")
  }while(input>1)
}
```

Mensajes de error:

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

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:

```
function FactorialFor int ( int n){
  let int i;
  let int factorial; factorial = 1;
  for (i = 1; i <= n; i++)
  {
```

```
    factorial *= i;
  }
  return factorial;
}
```

Mensajes de error:

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