# PROCESADORES DE LENGUAJES

Memoria


## Grupo 87

Gracia Estévez Peña (18M056)

Víctor Montesdeoca Fenoy (18M019)

Alberto Sánchez Pérez (18M039)

# ÍNDICE

# OPCIONES DEL PROCESADOR

- Sentencia condicional compuesta (if, if-else)
- Asignación con suma (+=)
- Comentario de línea (//)
- Con comillas simples (' ')
- Analizador descendente LL1
- Analizador con recursividad

# ANALIZADOR LÉXICO

El analizador léxico recibe el archivo fuente y genera tokens para posterior análisis sintáctico y semántico. Para su realización, hemos definido la lista de tokens, creado una gramática lineal (con su correspondiente autómata), agregándole acciones semánticas para la correcta evaluación de tokens.

# TOKENS

## Palabras reservadas

- <Let, ->
- <If, ->
- <Else, ->
- <Function, ->
- <TipoNumber, ->
- <TipoBoolean, ->
- <TipoCadena, ->
- <Return, ->
- <Alert, ->
- <Input, ->

## Símbolos

- <AbrirPar, ->  (
- <CerrarPar, ->  )
- <AbrirLlave, ->  {
- <CerrarLlave, ->  }
- <PuntoComa, ->  ;
- <Coma, ->  ,
- <Asignación, ->  =
- <AsignaciónSuma, ->  +=
- <Id, puntero_TS>

- <Entero, valor>
- <Cadena, lexema>

## Operadores

- <Suma, -> +
- <Not, -> !
- <Igualdad, -> ==

# GRAMÁTICA ANALIZADOR LÉXICO

S -> delS | =A | dB | 'C | +D | /E | lF | { | } | ( | ) | ; | , | !

A -> = | λ

B -> dB | λ

C -> c1C | '

D -> = | λ

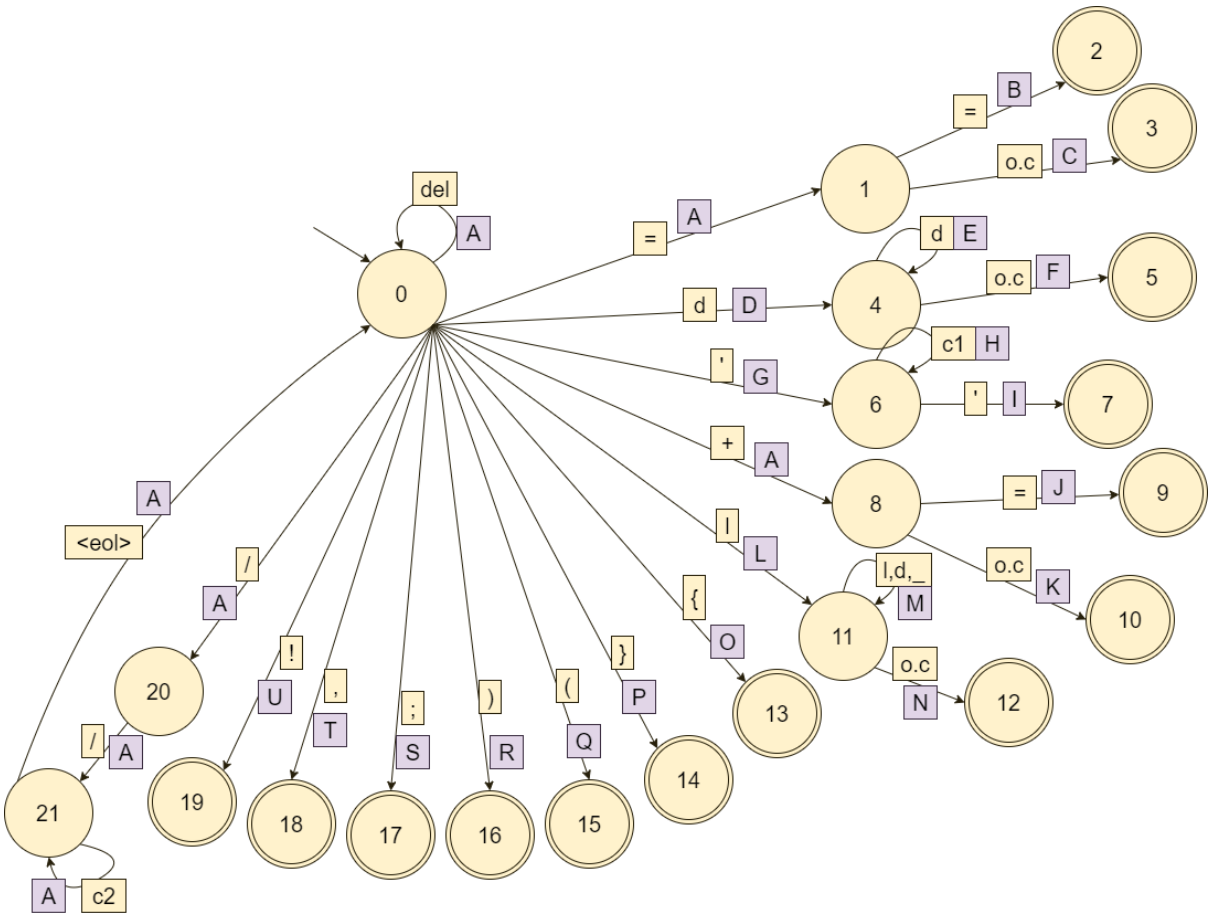
E -> /G

F -> lF | dF | _F | λ

G - > c2 | <eol>S

- ➔ *d = {0,1, ..., 8, 9}*
- ➔ *l = {a, b, ..., y, z, A, B, ..., Y, Z}*
- ➔ *c1 = Cualquier carácter ascii menos ‘*
- ➔ *c2 = Cualquier carácter ascii menos <eol>*
- ➔ *o.c = Otro carácter*

# AUTÓMATA


del
A
0
=
A
1
=
B
2
o.c
C
3
d
D
4
d
E
o.c
F
5
c1
H
'
G
6
'
I
7
+
A
8
=
J
9
I
L
o.c
K
10
l,d,_
M
11
o.c
N
12
{
O
13
}
P
14
(
Q
15
)
R
16
;
S
17
,
T
18
!
U
19
/
A
20
/
A
21
A
c2
A
<eol>

# ACCIONES SEMÁNTICAS DEL ANALIZADOR LÉXICO

A: Leer;
B: Leer; GenerarToken(<Igualdad, ->);
C: GenerarToken(<Asignacion, ->);
D: num = valor A(d);
E: num = num*10 + valor A(d);
F: Leer; if(num>32767 || num<-32767)
then ERROR('entero fuera de rango');
else GenerarToken(Entero, numero);
G: lexema = NULL; cont=0; Leer;
H: lexema = lexema ⊕ c1; cont++; Leer;
I: Leer; if(cont>64)
then ERROR('cadena fuera de rango');
else GenerarToken(<Cadena, lexema>);
J: GenerarToken(<AsignacionSuma->); Leer;
K: GenerarToken(<Suma, ->);
L: lexema = l; Leer;
M: lexema = lexema ⊕ l,d,_; Leer;
N: Leer; if(PalabraReservada(lexema))
then tok=codigoToken(lexema); GenerarToken(<tok,->);
else p=buscarTS(lexema);
if(p=NULL) then
p=insertarTS(lexema); GenerarToken(<Id,p>);
else ERROR('Identificador ya declarado')
O: Leer; GenerarToken(<AbrirLlave, ->);
P: Leer; GenerarToken(<CerrarLlave, ->);
Q: Leer; GenerarToken(<AbrirPar, ->);
R: Leer; GenerarToken(<CerrarPar, ->);
S: Leer; GenerarToken(<PuntoComa, ->);
T: Leer; GenerarToken(<Coma, ->);
U: Leer; GenerarToken(<Not, ->);

# TABLA DE SÍMBOLOS

La tabla de símbolos almacena la información principal del archivo fuente. Para su realización hemos empleado una clase de java que almacena los atributos de los elementos de la tabla de símbolos como parámetros mediante el uso del analizador semántico. Los atributos definidos son:

- ➔ Lexema: El nombre del elemento.
- ➔ Tipo: El tipo del elemento (entero, cadena, funcion).
- ➔ Desplazamiento: El desplazamiento de la pila.
- ➔ Número de Parámetros: Nº de parámetros de la función.
- ➔ Tipo de Parámetros: Lista con el tipo de cada parámetro de la función.
- ➔ Modo de Paso: El modo de retorno (en este caso es siempre por valor) de la función.
- ➔ Tipo Retorno: El tipo que devuelve la función.
- ➔ Etiqueta: Etiqueta que define la función.

# ANALIZADOR SINTÁCTICO

El analizador sintáctico se encarga de la generación de un parse a través de estructuras sintácticas definidas sobre los tokens generados por el analizador léxico. Para ello, hemos empleado un analizador sintáctico descendente recursivo que emplea una gramática inicial con condición LL1. Hemos utilizado la herramienta SGDLL1 para la evaluación de condición LL1 y la generación de la tabla para la gramática.

# GRAMÁTICA INICIAL ANALIZADOR SINTÁCTICO

**Terminales:** { let if else function tipoNumber tipoBoolean tipoCadena return alert input ( ) { } ; , = += id entero cadena + ! == }

**No Terminales:** {S, A, B, C, D, E, F, G, H, I, J, K, L}

**Axioma:** S

**Producciones:** {

P -> B P

P -> F P

P -> λ

F -> function H id ( A ) { C }

H -> T

H -> λ

A -> T id K

A -> λ

K -> ,T id K

K -> λ

C -> BC

C -> λ

B -> if ( E ) S

B -> if ( E ) { C }

B -> if ( E ) { C } else { C }

B ->  let T id ;

B ->  S

T -> tipoNumber

T -> tipoBoolean

T -> tipoString

S -> id = E ;

S -> id += E ;

S -> id ( L ) ;

S -> alert ( E ) ;

S -> input ( id ) ;

S -> return X ;

L -> E Q

L -> λ

Q -> , E Q

Q -> λ

X -> E

X -> λ

E -> E == R

E -> R

R -> R + U

R -> U

U -> !U

U -> V

V -> id

V -> ( E )

V -> id ( L )

V -> entero

V -> cadena

# GRAMÁTICA EN SGDLL(1)

Además de factorizar y eliminar recursividad, para evitar problemas con SGDLL(1) convertimos los símbolos en palabras. A continuación dejamos el cambio (cada letra es la inicial del símbolo):

| | |
|---|---|
| ( | ap (abir paréntesis) |
| ) | cp (cerrar paréntesis) |
| { | al (abril llave) |
| } | cl (cerrar llave) |
| ; | pc (punto y coma) |
| , | c (coma) |
| = | i (igual) |
| += | mi (más igual) |
| + | m (más) |
| ! | n (not) |
| == | ii (igual igual) |

```
Axioma = P

NoTerminales = { P F H A K C B B1 B2 T S S1 L Q X E E1 R R1 U
V V1 }

Terminales = { let if else function tipoNumber tipoBoolean
tipoCadena return alert input ap cp al cl pc c i mi id entero
cadena m n ii }

Producciones = {

        1. P ->  B P
        2. P ->  F P
        3. P -> lambda
```

4. F ->  function H id ap A cp al C cl

5. H -> T
6. H -> lambda

7. A -> T id K
8. A -> lambda

9. K -> c T id K
10. K -> lambda

11. C -> B C
12. C -> lambda

13. B -> if ap E cp B1
14. B -> let T id pc
15. B -> S

16. B1 -> S
17. B1 -> al C cl B2

18. B2 -> lambda
19. B2 -> else al C cl

20. T -> tipoNumber
21. T -> tipoBoolean
22. T -> tipoCadena

23. S -> id S1
24. S -> alert ap E cp pc
25. S -> input ap id cp pc
26. S -> return X pc

27. S1 -> i E pc
28. S1 -> mi E pc
29. S1 -> ap L cp pc

30. L -> E Q
31. L -> lambda

32. Q -> c E Q
33. Q -> lambda

34. X -> E

```
35.     X -> lambda

36.     E -> R E1

37.     E1 -> ii R E1
38.     E1 -> lambda

39.     R -> U R1

40.     R1 -> m U R1
41.     R1 -> lambda

42.     U -> n U
43.     U -> V

44.     V ->  id V1
45.     V -> ap E cp
46.     V -> entero
47.     V -> cadena

48.     V1 -> ap L cp
49.     V1 -> lambda

}
```

# CONDICIÓN LL1

FIRST de T -> tipoNumber = { tipoNumber }
FIRST de T -> tipoBoolean = { tipoBoolean }
FIRST de T -> tipoCadena = { tipoCadena }
FIRST de T = { tipoBoolean tipoCadena tipoNumber }
FIRST de A -> T id K = { tipoBoolean tipoCadena tipoNumber }
FIRST de A -> lambda = { lambda }
FIRST de A = { tipoBoolean tipoCadena tipoNumber lambda }
FOLLOW de A = { cp }
FIRST de B -> if ap E cp B1 = { if }
FIRST de B -> let T id pc = { let }
FIRST de S -> id S1 = { id }
FIRST de S -> alert ap E cp pc = { alert }
FIRST de S -> input ap id cp pc = { input }
FIRST de S -> return X pc = { return }
FIRST de S = { alert id input return }
FIRST de B -> S = { alert id input return }
FIRST de B = { alert id if input let return }
FIRST de B1 -> S = { alert id input return }
FIRST de B1 -> al C cl B2 = { al }
FIRST de B1 = { al alert id input return }
Analizando producción B2 -> lambda
FIRST de B2 -> else al C cl = { else }
FIRST de B2 = { else lambda }
FIRST de P -> B P = { alert id if input let return }
FIRST de F -> function H id ap A cp al C cl = { function }
FIRST de F = { function }
FIRST de P -> F P = { function }
FIRST de P -> lambda = { lambda }
FIRST de P = { alert function id if input let return lambda }
FIRST de C -> B C = { alert id if input let return }
FIRST de C -> lambda = { lambda }
FIRST de C = { alert id if input let return lambda }
FOLLOW de C = { cl }
FOLLOW de B = { alert cl function id if input let return $ (final de cadena) }
FOLLOW de B1 = { alert cl function id if input let return $ (final de cadena) }
FOLLOW de B2 = { alert cl function id if input let return $ (final de cadena) }
FIRST de U -> n U = { n }
FIRST de V -> id V1 = { id }
FIRST de V -> ap E cp = { ap }
FIRST de V -> entero = { entero }
FIRST de V -> cadena = { cadena }
FIRST de V = { ap cadena entero id }

FIRST de U -> V = { ap cadena entero id }
FIRST de U = { ap cadena entero id n }
FIRST de R -> U R1 = { ap cadena entero id n }
FIRST de R = { ap cadena entero id n }
FIRST de E -> R E1 = { ap cadena entero id n }
FIRST de E = { ap cadena entero id n }
FIRST de E1 -> ii R E1 = { ii }
FIRST de E1 -> lambda = { lambda }
FIRST de E1 = { ii lambda }
FIRST de Q -> c E Q = { c }
FIRST de Q -> lambda = { lambda }
FIRST de Q = { c lambda }
FOLLOW de L = { cp }
FOLLOW de Q = { cp }
FOLLOW de X = { pc }
FOLLOW de E = { c cp pc }
FOLLOW de E1 = { c cp pc }
FIRST de H -> T = { tipoBoolean tipoCadena tipoNumber }
FIRST de H -> lambda = { lambda }
FIRST de H = { tipoBoolean tipoCadena tipoNumber lambda }
FOLLOW de H = { id }
FIRST de K -> c T id K = { c }
FIRST de K -> lambda = { lambda }
FIRST de K = { c lambda }
FOLLOW de K = { cp }
FIRST de L -> E Q = { ap cadena entero id n }
FIRST de L -> lambda = { lambda }
FIRST de L = { ap cadena entero id n lambda }
FIRST de R1 -> m U R1 = { m }
FIRST de R1 -> lambda = { lambda }
FIRST de R1 = { m lambda }
FOLLOW de R = { c cp ii pc }
FOLLOW de R1 = { c cp ii pc }
FIRST de S1 -> i E pc = { i }
FIRST de S1 -> mi E pc = { mi }
FIRST de S1 -> ap L cp pc = { ap }
FIRST de S1 = { ap i mi }
FIRST de V1 -> ap L cp = { ap }
FIRST de V1 -> lambda = { lambda }
FIRST de V1 = { ap lambda }
FOLLOW de U = { c cp ii m pc }
FOLLOW de V = { c cp ii m pc }
FOLLOW de V1 = { c cp ii m pc }
FIRST de X -> E = { ap cadena entero id n }
FIRST de X -> lambda = { lambda }
FIRST de X = { ap cadena entero id n lambda }

# TABLA

| | al | alert | ap | c | cadena | cl | cp | else | entero | function | i | id | if | ii | input | let | m | mi | n | pc | return | tipoBoolean | tipoCadena | tipoNumber | $ (final de cadena) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | -- | -- | -- | -- | -- | -- | A → lambda | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | A → T id K | A → T id K | A → T id K | -- |
| B | -- | B → S | -- | -- | -- | -- | -- | -- | -- | -- | -- | B → S | B → if ap E cp B1 | -- | B → S | B → let T id pc | -- | -- | -- | -- | B → S | -- | -- | -- | -- |
| B1 | B1 → al C cl B2 | B1 → S | -- | -- | -- | -- | -- | -- | -- | -- | -- | B1 → S | -- | -- | B1 → S | -- | -- | -- | -- | -- | B1 → S | -- | -- | -- | -- |
| B2 | -- | B2 → lambda | -- | -- | -- | B2 → lambda | -- | B2 → else al C cl | -- | B2 → lambda | -- | B2 → lambda | B2 → lambda | -- | B2 → lambda | B2 → lambda | -- | -- | -- | -- | B2 → lambda | -- | -- | -- | B2 → lambda |
| C | -- | C → B C | -- | -- | -- | C → lambda | -- | -- | -- | -- | -- | C → B C | C → B C | -- | C → B C | C → B C | -- | -- | -- | -- | C → B C | -- | -- | -- | -- |
| E | -- | -- | E → R E1 | -- | E → R E1 | -- | -- | -- | E → R E1 | -- | -- | E → R E1 | -- | -- | -- | -- | -- | -- | E → R E1 | -- | -- | -- | -- | -- | -- |
| E1 | -- | -- | -- | E1 → lambda | -- | -- | E1 → lambda | -- | -- | -- | -- | -- | -- | E1 → ii R E1 | -- | -- | -- | -- | -- | E1 → lambda | -- | -- | -- | -- | -- |
| F | -- | -- | -- | -- | -- | -- | -- | -- | -- | F → function H id ap A cp al C cl | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| H | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | H → lambda | -- | -- | -- | -- | -- | -- | -- | -- | -- | H → T | H → T | H → T | -- |
| K | -- | -- | -- | K → c T id K | -- | -- | K → lambda | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| L | -- | -- | L → E Q | -- | L → E Q | -- | L → lambda | -- | L → E Q | -- | -- | L → E Q | -- | -- | -- | -- | -- | -- | L → E Q | -- | -- | -- | -- | -- | -- |
| P | -- | P → B P | -- | -- | -- | -- | -- | -- | -- | P → F P | -- | P → B P | P → B P | -- | P → B P | P → B P | -- | -- | -- | -- | P → B P | -- | -- | -- | P → lambda |
| Q | -- | -- | -- | Q → c E Q | -- | -- | Q → lambda | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| R | -- | -- | R → U R1 | -- | R → U R1 | -- | -- | -- | R → U R1 | -- | -- | R → U R1 | -- | -- | -- | -- | -- | -- | R → U R1 | -- | -- | -- | -- | -- | -- |
| R1 | -- | -- | -- | R1 → lambda | -- | -- | R1 → lambda | -- | -- | -- | -- | -- | -- | R1 → lambda | -- | -- | R1 → m U R1 | -- | -- | R1 → lambda | -- | -- | -- | -- | -- |
| S | -- | S → alert ap E cp pc | -- | -- | -- | -- | -- | -- | -- | -- | -- | S → id S1 | -- | -- | S → input ap id cp pc | -- | -- | -- | -- | -- | S → return X pc | -- | -- | -- | -- |
| S1 | -- | -- | S1 → ap L cp pc | -- | -- | -- | -- | -- | -- | -- | S1 → i E pc | -- | -- | -- | -- | -- | -- | S1 → mi E pc | -- | -- | -- | -- | -- | -- | -- |
| T | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | T → tipoBoolean | T → tipoCadena | T → tipoNumber | -- |
| U | -- | -- | U → V | -- | U → V | -- | -- | -- | U → V | -- | -- | U → V | -- | -- | -- | -- | -- | -- | U → n U | -- | -- | -- | -- | -- | -- |
| V | -- | -- | V → ap E cp | -- | V → cadena | -- | -- | -- | V → entero | -- | -- | V → id V1 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| V1 | -- | -- | V1 → ap L cp | V1 → lambda | -- | -- | V1 → lambda | -- | -- | -- | -- | -- | -- | V1 → lambda | -- | -- | V1 → lambda | -- | -- | V1 → lambda | -- | -- | -- | -- | -- |
| X | -- | -- | X → E | -- | X → E | -- | -- | -- | X → E | -- | -- | X → E | -- | -- | -- | -- | -- | -- | X → E | X → lambda | -- | -- | -- | -- | -- |

# ANALIZADOR SEMÁNTICO

El analizador semántico se encarga de gestionar acciones sobre las estructuras sintácticas que permitan la interpretación de estas estructuras además de la posible generación de errores. Para ello hemos diseñado un esquema de traducción que se encarga de gestionar las acciones semánticas en base a la gramática del analizador sintáctico.

# ACCIONES SEMÁNTICAS

(Las acciones semánticas relativas a errores de variables repetidas no están puestas en el esquema de traducción al haber sido ya implementadas anteriormente en el procesador léxico).

```
P1 ->     {TSG = Crea TS
               TsActual = TSG
               DespG = 0}
     P         {DestruyeTS(TSG)}

P ->  B P' {}
P ->  F P' {}
P ->  λ    {}

F ->  function H id
               {TSL = CreaTS
               TsActual = TSL
               ZonaDecl = true
               InsertaEtTS(id.pos, nuevaEt())
               ZonaDecl = false
               DespL = 0}
          ( A ) { C }
               {If (C.tipoRet!=H.tipo)
                      Error ("Tipo de retorno
                      incorrecto")
               DestruyeTS (TSL)
               TSactual:= TSG}

H -> T    {H.tipo = T.tipo}
H -> λ    {H.tipo = vacío}

A -> T id K    {ZonaDecl = true
               InsertarTipoTS(id.pos, T.tipo)
               InsertarDespTS(id.pos, Desp)
               Desp = Desp + T.ancho
               ZonaDecl = false}
A -> λ         {}

K -> , T id K' {ZonaDecl = true
               InsertarTipoTS(id.pos, T.tipo)
               InsertarDespTS(id.pos, Desp)
```

```
                Desp = Desp + T.ancho
                ZonaDecl = true}
K -> λ          {}

C -> BC'        {If(B.tipoRet==vacío and
                C'.tipoRet==vacío)
                     C.tipoRet = vacio
                Else
                     C.tipoRet = B.tipoRet}
C -> λ          {C.tipo = vacio}

B -> if ( E ) B1    {If(E.tipo!=boolean)
                         Error("Expresion del if
                         incorrecta")}
B -> let T id ;     {ZonaDecl = true
                    InstertarEtiTS(id.pos)
                    InstertarTipoTS(id.pos,D.tipo)
                    InsertarDespTS(id.pos, desp)
                    Desp = Desp + T.ancho
                    ZonaDecl = false}

B -> S              {B.tipo = S.tipo
                    B.tipoRet = S.tipoRet}

B1 -> S             {}
B1 -> { C } B2      {}

B2 -> λ             {}
B2 -> else { C }    {}

T -> tipoNumber{T.tipo = entero
                    T.ancho = 1}
T -> tipoBoolean    {T.tipo = boolean
                    T.ancho = 1}
T -> tipoCadena{T.tipo = cadena
                    T.ancho = 64}

S -> id S1      {id.tipo = BuscaTS(id.pos, tipo)
                    If(id.tipo!=S1.tipo &&
                    S1.tipo!=vacio)
                         Error("Tipo de variable
                         errónea")
                    If(S1.tipo!=vacio)
                         S.tipo = S1.tipo
```

```
                    param = BuscaTS(id.pos, param.tipo)
                    If(S1.tipoParam)!=vacio &&
                    S1.tipo!=param}
                         Error("Parámetros de diferente
                         tipo")
S -> alert ( E ) ;  {if(E.tipo==boolean)
                         S.tipo = Error("Input
                         incorrecto")}
S -> input ( id ) ; {if(E.tipo==boolean)
                         S.tipo = Error("Input
                         incorrecto")}
S -> return X ;{S.tipoRet = X.tipo}

S1 -> = E ;         {S1.tipo = E.tipo}
S1 -> += E ;        {S1.tipo = E.tipo}
S1 -> ( L ) ;       {S1.tipo = vacio
                    S1.tipoParam L.tipoParam}

L -> E Q            {L.tipo = E.tipo x Q.tipo}
L -> λ              {}

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

X -> E              {X.tipo = E.tipo}
X -> λ              {X.tipo = vacio}

E -> R E1           {If(E1.tipo==vacio)
                         E.tipo = R.tipo
                    Else
                         E.tipo = boolean
                    If(E1.tipo!=vacio &&
                    R.tipo!=entero)
                         Error(Operador no valido)
                    If(R.tipo!=E1.tipo &&
                    E1.tipo!=vacio)
                         Error("Operadores de diferente
                         tipo")}

E1 -> == R E1'      {If(E1'.tipo==vacio)
                         E1.tipo = R.tipo
                    Else If(R.tipo!=entero ||
                    E1'.tipo!=entero)
                         Error("Tipos no validos")}
```

```
    E1 -> λ             {E1.tipo = vacío}

    R -> U R1           {If(R1.tipo==vacio)
                             R.tipo = U.tipo
                        Else If(R1.tipo!=entero ||
                        U.tipo!=entero)
                             Error(Operador no valido)}

    R1 -> + U R1'       {If((U.tipo!=entero ||
                        R1'.tipo!=entero) && R1'!=vacio)
                             Error("Operador no valido")
                        If(U.tipo!=R1'.tipo &&
                        R1'.tipo!=vacío)
                             Error("Operadores de diferente
                             tipo")
                        R1.tipo = U.tipo}
    R1 -> λ             {R1.tipo = vacío}

    U -> ! U'           {If(U'.tipo!=boolean)
                             Error("Operador no valido")
                        U.tipo = U'.tipo}
    U -> V              {U.tipo = V.tipo}

    V ->  id V1         {Id.tipo = BuscaTipoTS(id.pos)
                        V.tipo = id.tipo}
    V -> ( E )     {V.tipo = E.tipo}
    V -> entero         {V.tipo = entero
                        V.ancho = 1}
    V -> cadena         {V.tipo = cadena
                        V.ancho = 64}

    V1 -> ( L )         {}
    V1 -> λ             {}
}
```

# CASOS DE PRUEBA

## CORRECTOS

## 1)

```
let number a;
let number b;
let number int;
alert ('Introduce el primer operando');
input (a);
alert ('Introduce el segundo operando');
input (b);
function number operacion (number num1, number num2)
{
     return num1 + num2+77;
}

int = 0;
alert (operacion (a, b));
```

## 2)

```
let number a;

function number suma(number b, number c){
     if(a==b){
          return 0;
     } else {
          return a+b;
     }
}

a = 4;

b = suma(a,a);
```

# 3)

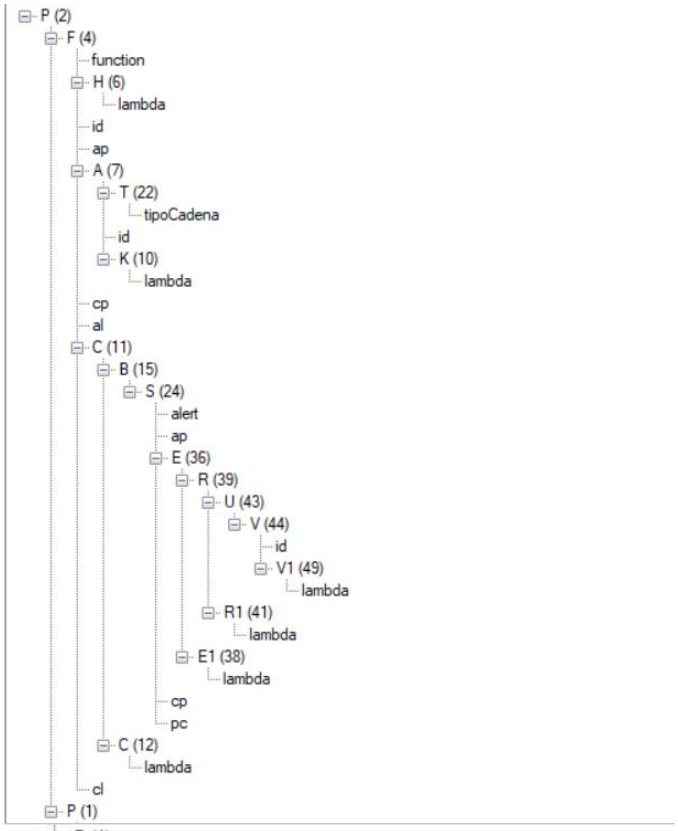
```
function print(string str){
     alert(str);
}

let string a;

input(a);

print(a);

print('Helloworld');
```

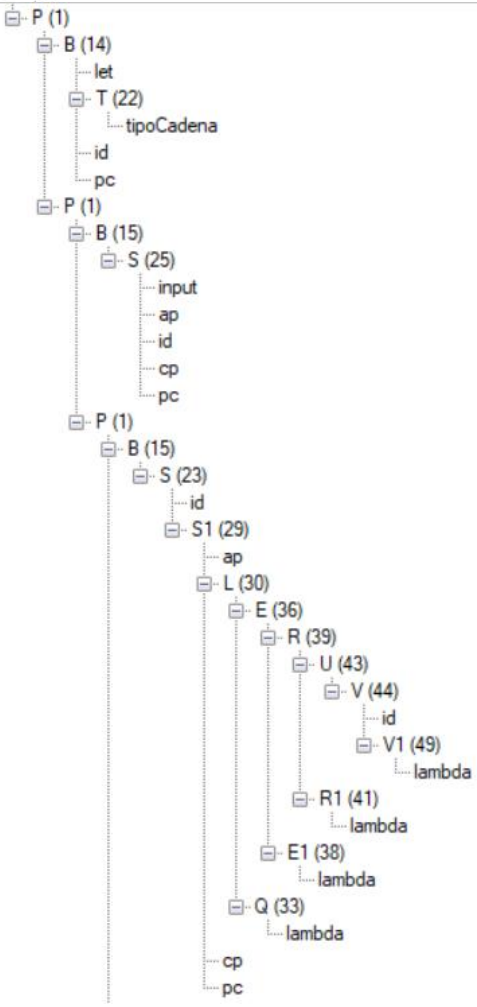
**Parse**

```
2 4 6 7 22 10 11 15 24 36 39 43 44 49 41 38 12 1 14 22 1 15 25
1 15 23 29 30 36 39 43 44 49 41 38 33 1 15 23 29 30 36 39 43
47 41 38 33 3
```

**Tokens**

```
<Function,> <Id,0> <AbrirPar,> <TipoCadena,> <Id,1>
<CerrarPar,> <AbrirLlave,> <Alert,> <AbrirPar,> <Id,1>
<CerrarPar,> <PuntoComa,> <CerrarLlave,> <Let,> <TipoCadena,>
<Id,2> <PuntoComa,> <Input,> <AbrirPar,> <Id,2> <CerrarPar,>
<PuntoComa,> <Id,0> <AbrirPar,> <Id,2> <CerrarPar,>
<PuntoComa,> <Id,0> <AbrirPar,> <Cadena,"Helloworld">
<CerrarPar,> <PuntoComa,>
```


P (2)
F (4)
function
H (6)
lambda
id
ap
A (7)
T (22)
tipoCadena
id
K (10)
lambda
cp
al
C (11)
B (15)
S (24)
alert
ap
E (36)
R (39)
U (43)
V (44)
id
V1 (49)
lambda
R1 (41)
lambda
E1 (38)
lambda
cp
pc
C (12)
lambda
cl
P (1)


P (1)
B (14)
let
T (22)
tipoCadena
id
pc
P (1)
B (15)
S (25)
input
ap
id
cp
pc
P (1)
B (15)
S (23)
id
S1 (29)
ap
L (30)
E (36)
R (39)
U (43)
V (44)
id
V1 (49)
lambda
R1 (41)
lambda
E1 (38)
lambda
Q (33)
lambda
cp
pc

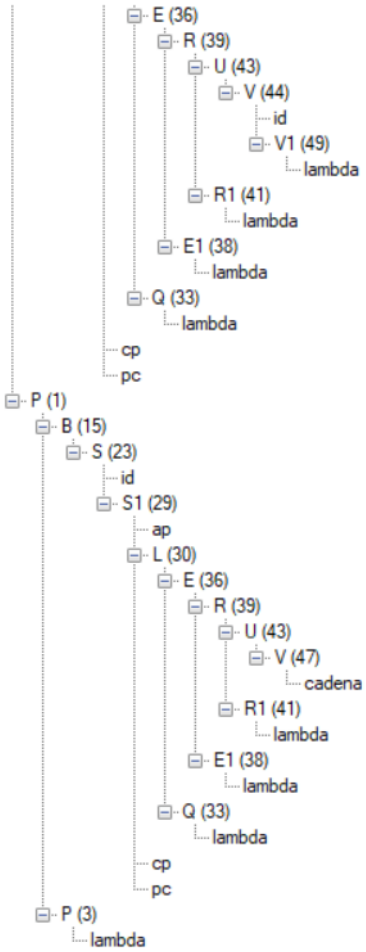



**Tabla de símbolos**

```
#0:
* 'print'
 + tipo : 'funcion'
 + numParam : '1'
 + TipoParam1 : 'cadena'
 + ModoParam1 : '1'
 + TipoRetorno : 'vacio'
 + EtiqFuncion : 'etiq_print'

* 'a'
 + tipo : 'cadena'
 + despl : '0'

#1:
* 'str'
 + tipo : 'cadena'
 + despl : '0'
```

4)

```
let boolean a;

//asignamos un valor a la variable a
a =2==2;

let boolean b;

b =0==1;

if(a){
     alert('Incorrecto');
} else{
     if(!b)
     alert('Correcto');
}
```

**5)**

```
let number a;

let number b;

a=3;

b=1;

function number asignasuma(number c, number d){

e+=(c+d);

return e;
}

b=asignasuma(a,b);
```

# ERRÓNEOS

## 1)

```
//declaramos a

let number a;

a=0;

/declaramos b

let number b;

b=0;

c=a+b;
```

```
java.lang.Exception: Error lexico [linea 6]: Comentario erroneo
```

## 2)

```
let string a;

input(a);

let string b;;

b='Pick one of this pokemon';

alert(a);

alert(b);
```

```
java.lang.Exception: Error sintactico [linea 6]: PuntoComa no
esperado
```

## 3)

```
let number a;

a= 3;

let number b;

b=1;

function print(string str){

alert(str);

}

print(a+b);
```

```
java.lang.Exception: Error semantico [linea 14]: Parametros de
llamada a la funcion incorrectos
```

## 4)

```
let number int;

int =2;

function fun(){

     alert('Esto no es divertido');

}

let string int;

int='Esto va a explotar';
```

```
java.lang.Exception: Error semantico [linea 12]: Variable
*int* repetida
```

## 5)

```
let boolean true;

true = 1==2;

let boolean false;

false = 1==1;

let number explosion;

explosion = true + false;
```

```
java.lang.Exception: Error semantico [linea 10]: Operador
invalido para la operacion suma
```