PRÁCTICA PROCESADORES DE LENGUAJES

Analizador Léxico, Tabla de Símbolos, Analizador Sintáctico y Analizador Semántico

Grupo 64

Jason Felipe Vázquez

ÍNDICE

1. Objetivo

Esta práctica consiste en el diseño e implementación de un Procesador, que realizará el Análisis Léxico y la Tabla de Símbolos, para un lenguaje en concreto, Javascript-PL.

2. Opciones de Grupo

Está práctica pertenece al grupo 64 donde hemos decidido añadir los tokens optativos con tal de facilitar la comprensión y la complejidad del trabajo.

Por tanto, la lista de tokens incluyendo lo específicos de nuestro grupo tendrá el siguiente aspecto:

Las opciones obligatorias para el grupo 64 son:

- Sentencias: **Sentencia repetitiva (while)**
- Técnica de Análisis Sintáctico: **Descendente**
- Operadores especiales: **Asignación con división (/=)**
- Comentario: **Comentario de bloque (/* */)**
- Cadenas: **Con comillas dobles (" ")**

Los elementos opcionales seleccionados por el grupo son*:

- Operador Aritmético: **suma (+)**
- Operador Lógico: **negación (!)**
- Operador Relacional: **distinto (! =)**

- Constante Lógica **(false)**
- Constante Lógica **(true)**
- **EOF**

3. Analizador Léxico

El principal objetivo del analizador léxico será analizar el fichero fuente, carácter a carácter y generar los tokens necesarios o errores encontrados, a través del gestor de errores, una vez realizado todo esto, transmitirá al analizador sintáctico.

- Alert --> **<alert, ->**
- Boolean --> **<boolean, ->**
- Break --> **<break, ->**
- Function --> **<function, ->**
- If --> **<if, ->**
- Input --> **<input, ->**
- Number --> **<number, ->**
- Return --> **<return, ->**
- String --> **<string, ->**
- While--> **<While, ->**
- Let --> **<let, ->**
- Const. entera --> **<Entero, PosTs>**
- Cadena (“) --> **<Cadena, Cadena (‘c*’)>**
- Identificador --> **<ID, Número>**
- Coma (,) --> **<Coma, ->**
- Punto y coma (;) --> **<PuntoComa, ->**
- Abrir paréntesis (**(**) --> **<ap, ->**
- Cerrar paréntesis (**)**) --> **<cp, ->**
- Abrir llave (**{**) --> **<al, ->**
- Cerrar llave (**{**) --> **<cl, ->**
- Asignación con división (/=) --> **<Divigual, ->**

Además, se han elegido los siguientes tokens opcionales:

- Operador aritmético:(+) --> **<suma, ->**
- Operador relacional: (!=) --> **<distinto, ->**
- Operador lógico: (!) --> **<negación, ->**

3.1. Gramática del lenguaje

Se ha diseñador la siguiente gramática que permitirá implementar los tokens citados anteriormente:

S -> del S| lA | dB | /C | !E | " F| ce | = | +

A -> lA| dA| λ

B -> dB| λ

C -> *D |=

D -> *G

G -> c1G | *H

H -> /S

E -> =| λ

F -> cF| "

Caracteres utilizados que toman valores:

l = ({A…Z}, {a…z})

d = { 0…9 }

c = { cualquier carácter- (**"**) }

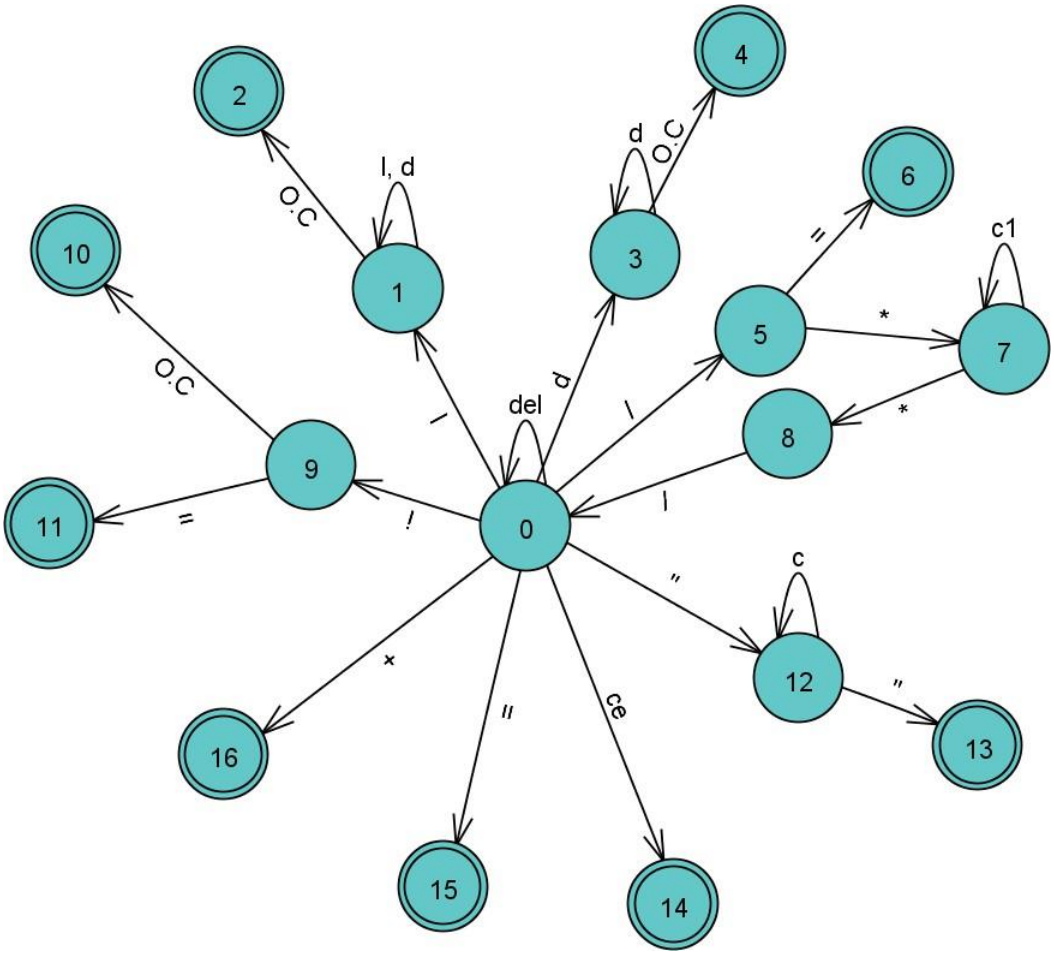
ce = { { } ( ) ; **,** }

del = { b, < tab >, < eol >, < eof > }

c1= { cualquier carácter – { * } }

O.**C** = en el autómata significará cualquier otro carácter.

3.2. Autómata



3.3. Acciones semánticas

0:0 Leer;

0:1 lexema= l; Leer

1:2 if (cod = buscarPalabra_clave (lexema)) then GenToken (Palabra_clave, cod);
Elseif (pos = buscarTS (lexema)) then GenToken (lexema, pos);
Else InsertToken (lexema, pos) GenToken (lexema, pos);

0:3 valor= d; Leer

3:3 valor= valor*10 + d; Leer

3:4 if (valor <= 32767) then GenToken (entero,valor);
Else error (“numero fuera de rango”);

0:5 Leer;

5:6 lexema = lexema $\oplus$ =; GenToken(Divigual,-);

5:7 leer;

7:7 leer;

7:8 leer;

0:9 lexema;

9:11 Gentoken(distinto,-);

9:10 Gentoken(negación, -);

0:12 Leer;

12:12 lexema= lexema ⊕ c; Leer;

12:13 Gentoken(Cadena, lexema);

0:14 Gentoken (ce, cod);

0:15 Gentoken(igua, -)

0:16 Gentoken(suma, -)

4. Tablas de Símbolos

Las tablas de símbolos que generaremos en nuestro programa, serán del siguiente tipo:

- Tabla de símbolos Global: tabla global de información.
- Tabla de símbolos Local: tabla de símbolos de funciones. Tabla local.

A continuación, se mostrarán el diseño de nuestra tabla de símbolos citadas anteriormente:

- Tabla de Símbolos Global, serán del siguiente tipo:

  * LEXEMA: *'lexema correspondiente'*

    + Tipo: *' tipo correspondiente'*

    + Despl: *'desplazamiento correspondiente según el "Tipo" '*

  Si se trata de una función:

  * LEXEMA: *'x'*

    + Tipo: *' funcion'*

    + TipoRetorno: *'Tipo de retorno correspondiente'*

    + NumParam: *'Número de parámetros de la función'*

    + TipoParam01: *'Tipo del primer parámetro'*

    + TipoParam02: *'Tipo del segundo parámetro'*

    + EtiqFuncion: *'Etiqueta de la función será del tipo '*

- Tabla de símbolos Local:

    * LEXEMA: *'lexema correspondiente'* (parametro de funcion)
        + Tipo: *'tipo correspondiente '*
        + Despl: *'desplazamiento correspondiente según el "Tipo" '*

Para separar la TSG de la TSL se usará el siguiente separador:

---

5. Analizador Sintáctico

El principal objetivo del analizador sintáctico será analizar el programa fuente y determinar si será correcto la sintaxis, para ello se apoya en el analizador léxico, recibiendo tokens de este, los cuales son tratados y una vez acabado el proceso vuelve a pedir otro token hasta completar el fichero.

Además, la gramática que se va a usar va a diferir en el analizador sintáctico. Como el analizador sintáctico es descendente recursivo, tiene que cumplirse la condición LL (1), es decir cumpliendo las condiciones siguientes:

- No tiene que ser una gramática ambigua.
- No presente recursividad por la izquierda.
- Factorizada por la izquierda.

Teniendo todo esto en cuenta, se ha desarrollado la siguiente gramática:

P -> B P

P -> F P

P -> eof

B -> let T id ;

B -> if (R) S

B -> while (R) { C }

B -> S

T -> number

T -> boolean

T -> string

S -> id Sa

S -> return X ;

S -> alert (R) ;

S -> input (id) ;

Sa -> = R ;

Sa -> != R ;

Sa -> (L) ;

X -> R

X -> lambda

C -> B C

C -> lambda

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

H -> T

H -> lambda

A -> T id K

A -> lambda

K -> , T id K

K -> lambda

L -> R Q

L -> lambda

Q -> , R Q

Q -> lambda

R -> U Ra

Ra -> = R

Ra -> != R

Ra -> lambda

U -> V Ua

Ua -> + U

Ua -> lambda

V -> id Va

V -> entero

V -> cadena

V -> true

V -> false

V -> (R)

V -> ! Vb

Ep -> /= T Ep

Ep -> lambda

Va -> (L)

Va -> lambda

Vb -> true

Vb -> false

Vb -> id

Vb -> (R)

Sa -> /= R ;

5.1 Tablas First y Follow:

FIRST de T -> number = { number }

FIRST de T -> boolean = { boolean }

FIRST de T -> string = { string }

FIRST de T = { boolean number string }

FIRST de A -> T id K = { boolean number string }

FIRST de A -> lambda = { lambda }

FIRST de A = { boolean number string lambda }

FOLLOW de A = {) }

FIRST de B -> let T id ; = { let }

FIRST de B -> if (R) S = { if }

FIRST de B -> while (R) { C } = { while }

FIRST de S -> id Sa = { id }

FIRST de S -> return X ; = { return }

FIRST de S -> alert (R) ; = { alert }

FIRST de S -> input (id) ; = { input }

FIRST de S = { alert id input return }

FIRST de B -> S = { alert id input return }

FIRST de B = { alert id if input let return while }

FIRST de C -> B C = { alert id if input let return while }

FIRST de C -> lambda = { lambda }

FIRST de C = { alert id if input let return while lambda }

FOLLOW de C = { } }

FIRST de Ep -> //= T Ep = { //= }

FIRST de Ep -> lambda = { lambda }

FIRST de Ep = { //= lambda }

Calculando FOLLOW de Ep

FOLLOW de Ep = { }

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

FIRST de F = { function }

FIRST de H -> T = { boolean number string }

FIRST de H -> lambda = { lambda }

FIRST de H = { boolean number string lambda }

FOLLOW de H = { id }

FIRST de K -> , T id K = { , }

FIRST de K -> lambda = { lambda }

FIRST de K = { , lambda }

FOLLOW de K = {) }

FIRST de V -> id Va = { id }

FIRST de V -> entero = { entero }

FIRST de V -> cadena = { cadena }

FIRST de V -> true = { true }

FIRST de V -> false = { false }

FIRST de V -> (R) = { (}

FIRST de V -> ! Vb = { ! }

FIRST de V = { ! (cadena entero false id true }

FIRST de U -> V Ua = { ! (cadena entero false id true }

FIRST de U = { ! (cadena entero false id true }

FIRST de R -> U Ra = { ! (cadena entero false id true }

FIRST de Ra -> != R = { != }

FIRST de Ra -> lambda = { lambda }

FIRST de Ra = { != = lambda }

FOLLOW de X = { ; }

FOLLOW de R = {) , ; }

FOLLOW de Ra = {) , ; }

FIRST de Sa -> = R ; = { = }

FIRST de Sa -> != R ; = { != }

FIRST de Sa -> (L) ; = { (}

FIRST de Sa -> //= R ; = { //= }

FIRST de Sa = { != (//= = }

FIRST de Ua -> + U = { + }

FIRST de Ua -> lambda = { lambda }

FIRST de Ua = { + lambda }

FOLLOW de U = { !=) , ; = }

FIRST de R = { ! (cadena entero false id true }

FIRST de L -> R Q = { ! (cadena entero false id true }

FIRST de L -> lambda = { lambda }

FIRST de L = { ! (cadena entero false id true lambda }

FOLLOW de L = {) }

FIRST de P -> B P = { alert id if input let return while }

FIRST de P -> F P = { function }

FIRST de P -> eof = { eof }

FIRST de P = { alert eof function id if input let return while }

FIRST de Q -> , R Q = { , }

FIRST de Q -> lambda = { lambda }

FIRST de Q = { , lambda }

FOLLOW de Q = {) }

FIRST de Ra -> = R = { = }

FOLLOW de Ua = { !=) , ; = }

FIRST de Va -> (L) = { (}

FIRST de Va -> lambda = { lambda }

FIRST de Va = { (lambda }

FOLLOW de V = { !=) + , ; = }

FOLLOW de Va = { !=) + , ; = }

FIRST de Vb -> true = { true }

FIRST de Vb -> false = { false }

FIRST de Vb -> id = { id }

FIRST de Vb -> (R) = { (}

FIRST de X -> R = { ! (cadena entero false id true }

FIRST de X -> lambda = { lambda }

FIRST de X = { ! (cadena entero false id true lambda

5.2 Comprobación de la gramática

Para comprobar si la gramática, es una gramática válida, se mostrará la tabla LL1, en la que se verá a simple vista si es válida o no, ya que, en caso de no ser válida, en alguna de las celdas tendría más de un movimiento(regla) para el mismo símbolo.

En nuestro caso, no ocurre eso, por tanto, se considerará una gramática válida.

A continuación, se mostrará la tabla LL1 de mi gramática. También se adjuntará un documento Excel, en caso de que no se puede visualizar correctamente (documento **Entrega\Recursos\TablaLL1.xlsx**).

6. Analizador Semántico

Este analizador semántico se encargará de computar la información una vez se tenga la estructura sintáctica del programa por tanto se considerará una fase a continuación del análisis sintáctico.

Es la que se encargará, por tanto, de detectar la validez semántica de las sentencias que han sido aceptadas por el analizador sintáctico. En caso de que haya error se lanzará un mensaje de error al Gestor de Errores.

Como mi el analizador de este grupo es descendente recursivo, se basa en una serie procedimientos donde se llama de manera recursiva para realizar su correspondiente análisis.

Cada procedimiento corresponde a un símbolo terminal de la gramática dada en el punto 3.1 de este documento, de tal manera que, al llamarse los procedimientos entre sí, el orden de estas llamadas determinará el análisis de la cadena.

En cuanto al algoritmo que se ha seguido al sido el explicado en las clases correspondientes a esta práctica por lo que no veo necesario volver a poner el mismo algoritmo.

7. Gestor de Errores

Se encargará de informar del tipo de error, línea donde se produce dicho error, en que analizador se produce (léxico, sintáctico o semántico) y una breve descripción del error.

Y serán del tipo siguiente:

“->Error en Analizador” --------” (Línea “-----”): Se esperaba '----'. Se ha recibido la expresión” ””

8. ANEXO (Pruebas)

Pruebas Correctas

Como en las especificaciones de la entrega se pide que se incluya el resultado únicamente de 1 de las 5 pruebas correctas, se mostrará, por tanto, los resultados correctos únicamente para la Prueba 1.

Por lo que se intentará que se la prueba más completa para que se observe su correcto funcionamiento.

- ### Prueba 1

Código fuente:

```
let string s;
let number uno;
let number UNO;
function number Factorial (number n)
{
        if (n != 0)          return 1;
        return n + Factorial (n + 1);
}
let number For;
let number functional;
let number While;



function string cadena (boolean log)
{
        while (!log)
        {
                print (s, "hola", 33);
                if (uno != UNO)   return s;
        }
        return s;
}
s = "El factorial ";

alert (s);
alert ("Introduce un numero.");
```

input (num);

let

boolean

booleano;

if (num != 0) *alert ("No existe el factorial de un negativo.");*

For= Factorial (num);

Tokens generados:

<let,>
<string,>
<ID,1>
<PuntoComa, >
<let,>
<number,>
<ID,2>
<PuntoComa, >
<let,>
<number,>
<ID,3>
<PuntoComa, >
<function,>
<number,>
<ID,4>
<ap, >
<number,>
<ID,5>
<cp, >
<al, >
<if,>
<ap, >
<ID,6>
<distinto,>
<Entero,0>
<cp, >
<return,>
<Entero,1>
<PuntoComa, >
<return,>
<ID,7>
<suma, >
<ID,8>
<ap, >
<ID,9>
<suma, >
<Entero,1>
<cp, >
<PuntoComa, >
<cl, >
<let,>
<number,>
<ID,10>
<PuntoComa, >
<let,>
<number,>
<ID,11>
<PuntoComa, >
<let,>
<number,>
<ID,12>
<PuntoComa, >
<function,>
<string,>
<ID,13>
<ap, >

<boolean,>
<ID,14>
<cp, >
<al, >
<while,>
<ap, >
<negacion,>
<ID,15>
<cp, >
<al, >
<ID,16>
<ap, >
<ID,17>
<Coma, >
<Cadena,"hola">
<Coma, >
<Entero,33>
<cp, >
<PuntoComa, >
<if,>
<ap, >
<ID,18>
<distinto,>
<ID,19>
<cp, >
<return,>
<ID,20>
<PuntoComa, >
<cl, >
<return,>
<ID,21>
<PuntoComa, >
<cl, >
<ID,22>
<igual,>
<Cadena,"El factorial ">
<PuntoComa, >
<alert,>
<ap, >
<ID,23>
<cp, >
<PuntoComa, >
<alert,>
<ap, >
<Cadena,"Introduce un numero.">
<cp, >
<PuntoComa, >
<input,>
<ap, >
<ID,24>
<cp, >
<PuntoComa, >
<let,>
<boolean,>
<ID,25>
<PuntoComa, >
<if,>
<ap, >
<ID,26>
<distinto,>
<Entero,0>
<cp, >
<alert,>
<ap, >
<Cadena,"No existe el factorial de un negativo.">
<cp, >
<PuntoComa, >
<ID,27>
<igual,>
<ID,28>
<ap, >

<ID,29>

<cp, >

<PuntoComa, >

<EOF, >

Tabla de Símbolos:

TABLA GLOBAL #1 :

* LEXEMA: 's'
    + Tipo: 'string'
    + Despl: '0'

* LEXEMA: 'uno'
    + Tipo: 'Entero'
    + Despl: '64'

* LEXEMA: 'UNO'
    + Tipo: 'Entero'
    + Despl: '65'

* LEXEMA: 'Factorial'
    + Tipo: 'Funcion'
    + TipoRetorno: 'Entero'
    + NumParam: '1'
    + TipoParam01:'Entero'
    + EtiqFuncion: 'EtFactorial01'

* LEXEMA: 'For'
    + Tipo: 'Entero'
    + Despl: '66'

* LEXEMA: 'functional'
    + Tipo: 'Entero'
    + Despl: '67'

* LEXEMA: 'While'
    + Tipo: 'Entero'
    + Despl: '68'

* LEXEMA: 'print'
    + Tipo: 'Entero'
    + Despl: '69'

* LEXEMA: 'cadena'
    + Tipo: 'Funcion'
    + TipoRetorno: 'string'
    + NumParam: '1'
    + TipoParam01:'Bool'
    + EtiqFuncion: 'Etcadena02'

* LEXEMA: 'num'
    + Tipo: 'Entero'
    + Despl: '70'

* LEXEMA: 'booleano'
    + Tipo: 'Bool'
    + Despl: '71'

TABLA DE LA Funcion Factorial #2 :

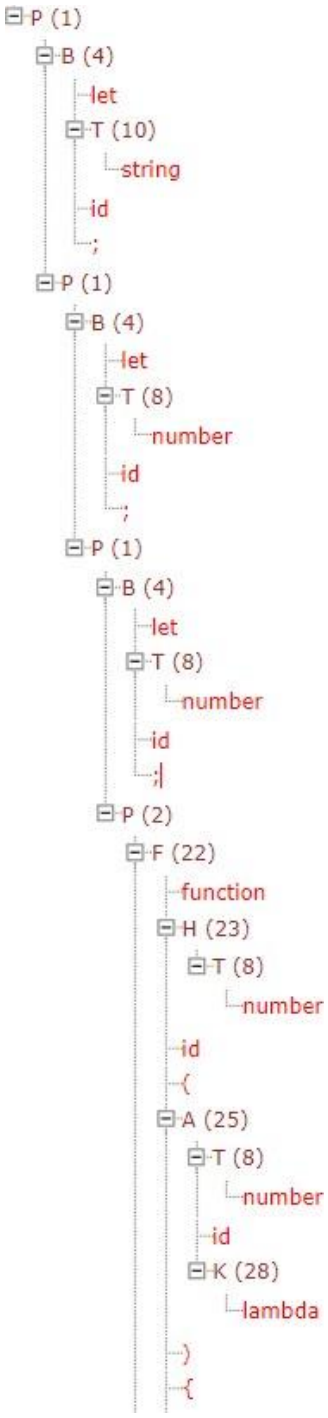
* LEXEMA: 'n' (parametro de funcion)
    + Tipo: 'Entero'
    + Despl: '0'

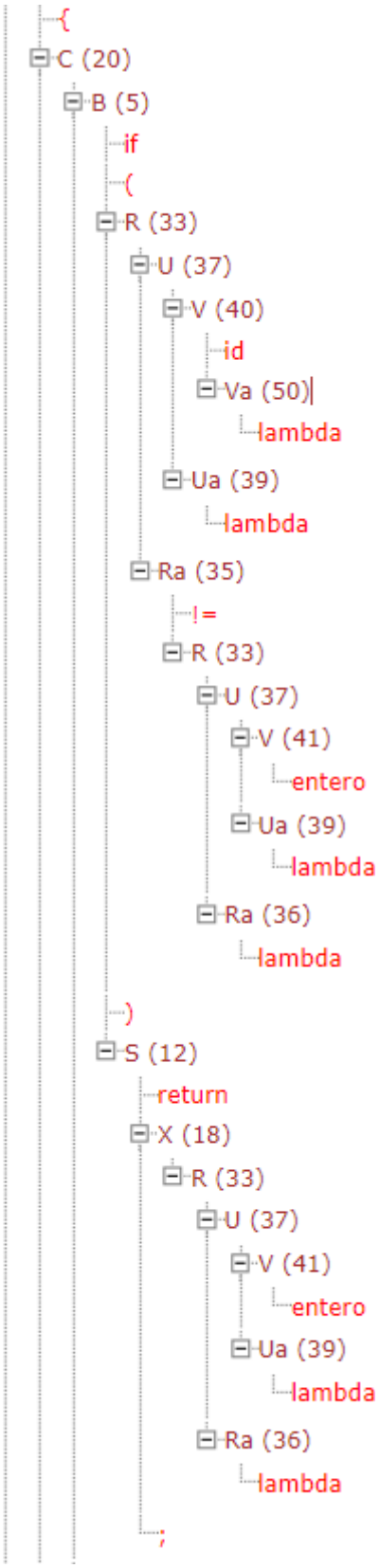
TABLA DE LA Funcion cadena #3 :

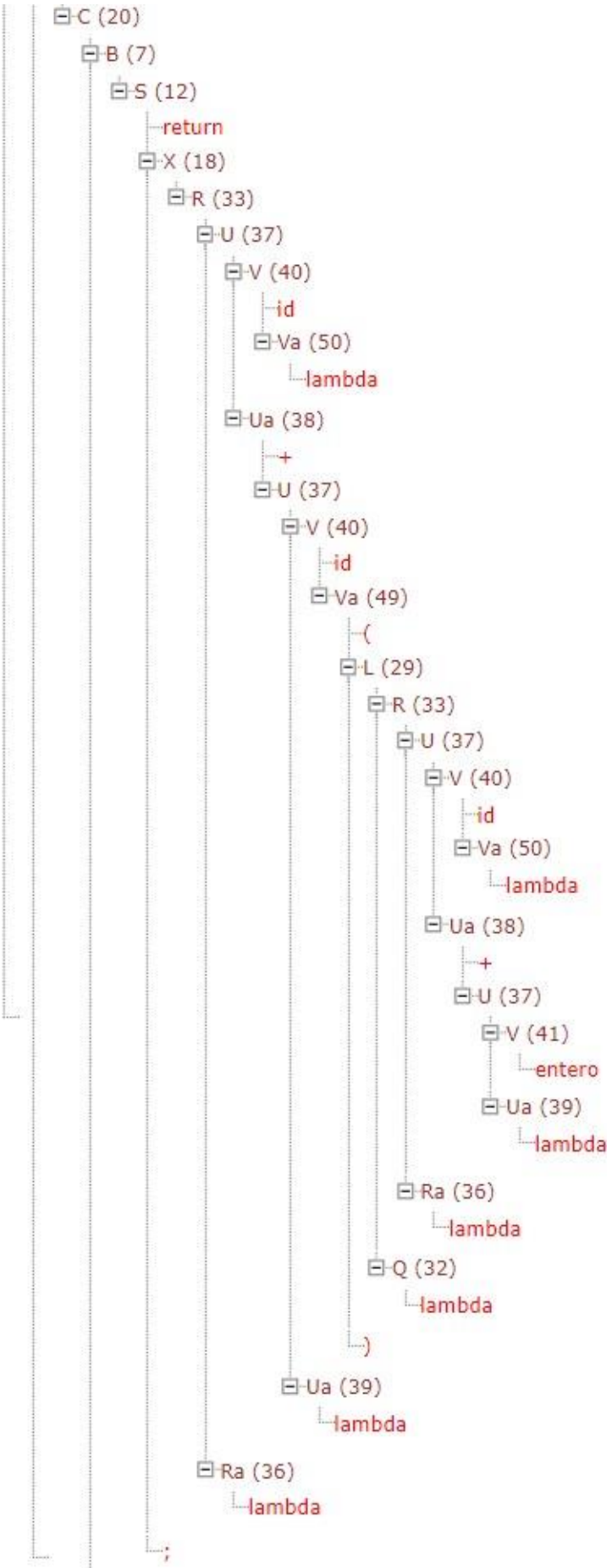
* LEXEMA: 'log' (parametro de funcion)
    + Tipo: 'Bool'
    + Despl: '0'

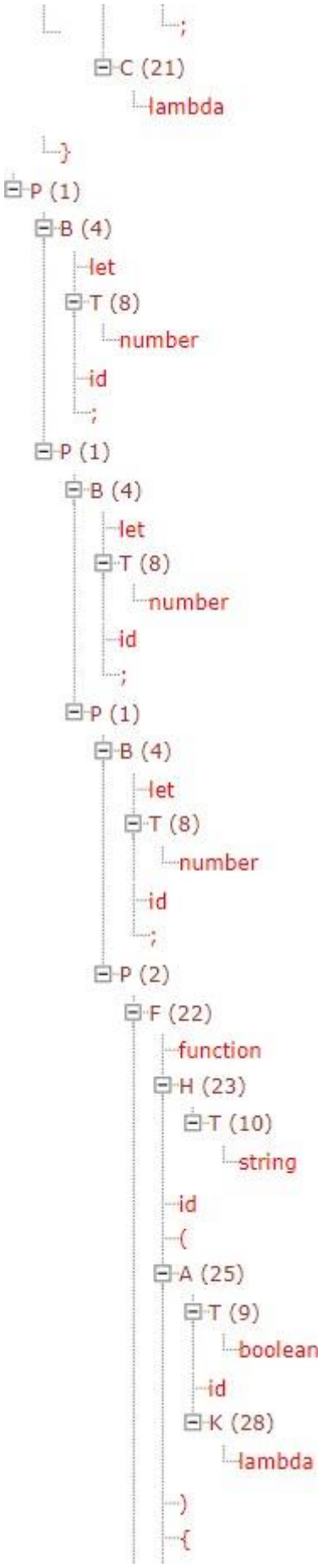
Parse obtenido: Descendente 1 4 10 1 4 8 1 4 8 2 22 23 8 25 8 28 20 5 33 37 40 50 39 35 33 37 41 39 36 12 18 33 37 41 39 36 20 7 12 18 33 37 40 50 38 37 40 49 29 33 37 40 50 38 37 41 39 36 32 39 36 21 1 4 8 1 4 8 1 4 8 2 22 23 10 25 9 28 20 6 33 37 46 53 39 36 20 7 11 17 29 33 37 40 50 39 36 31 33 37 42 39 36 31 33 37 41 39 36 32 20 5 33 37 40 50 39 35 33 37 40 50 39 36 12 18 33 37 40 50 39 36 21 20 7 12 18 33 37 40 50 39 36 21 1 7 11 15 33 37 42 39 36 1 7 13 33 37 40 50 39 36 1 7 13 33 37 42 39 36 1 7 14 1 4 9 1 5 33 37 40 50 39 35 33 37 41 39 36 13 33 37 42 39 36 1 7 11 15 33 37 40 49 29 33 37 40 50 39 36 32 39 36 3

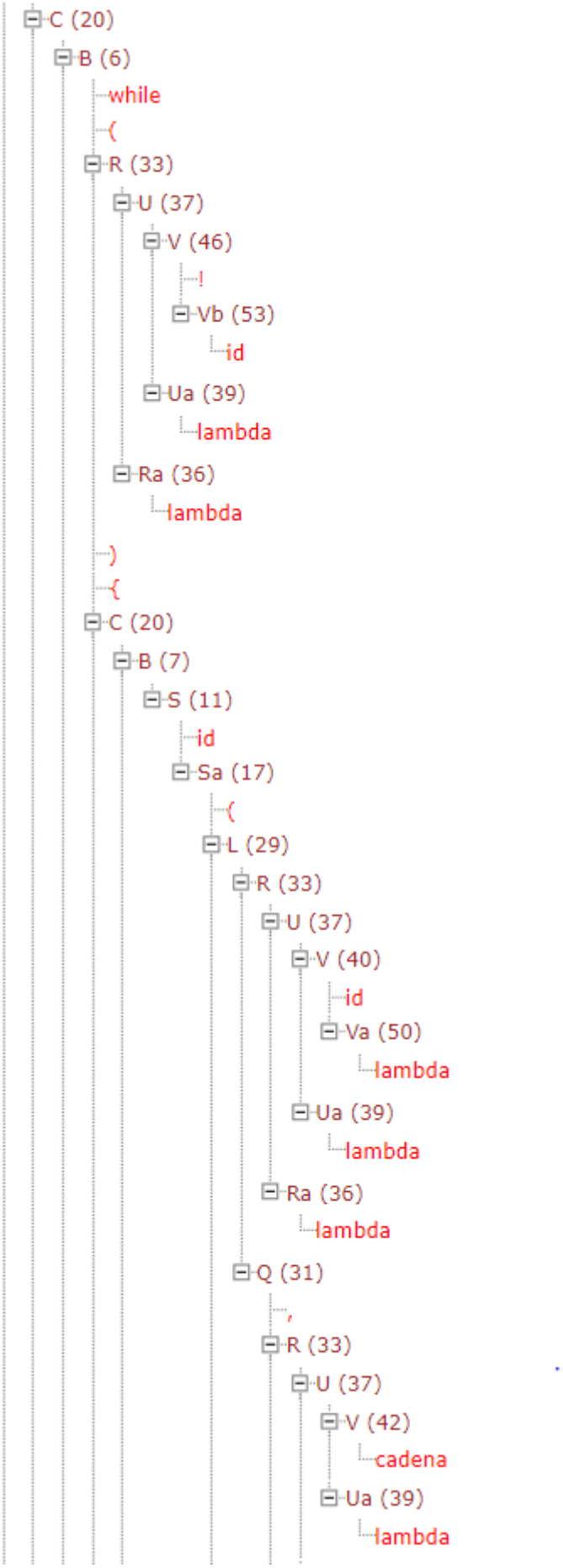
Árbol obtenido:

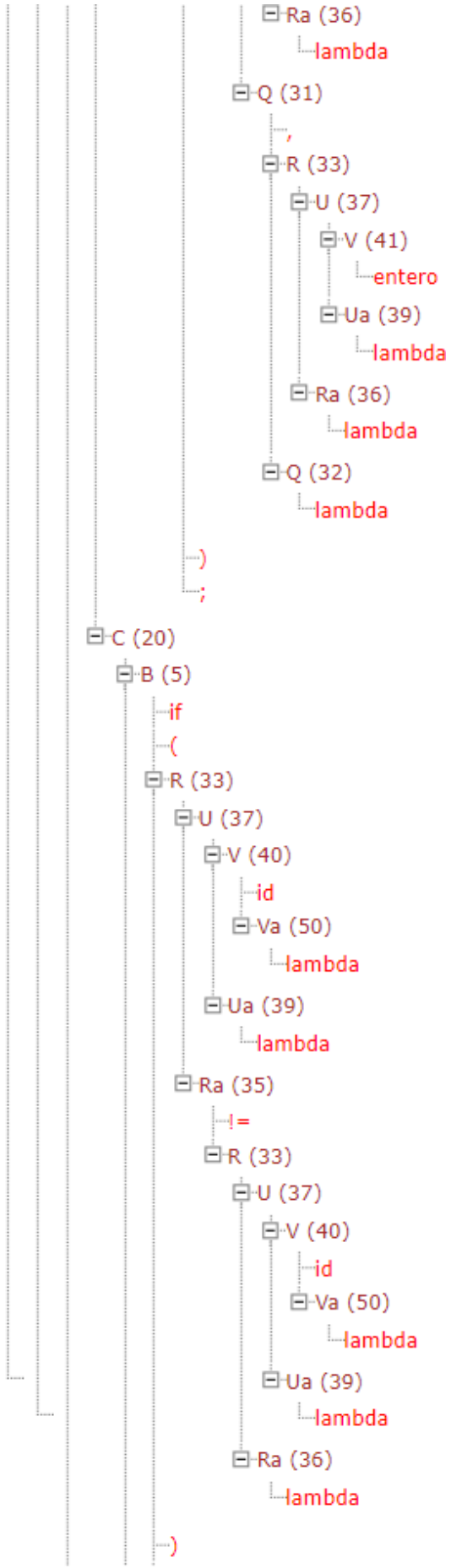


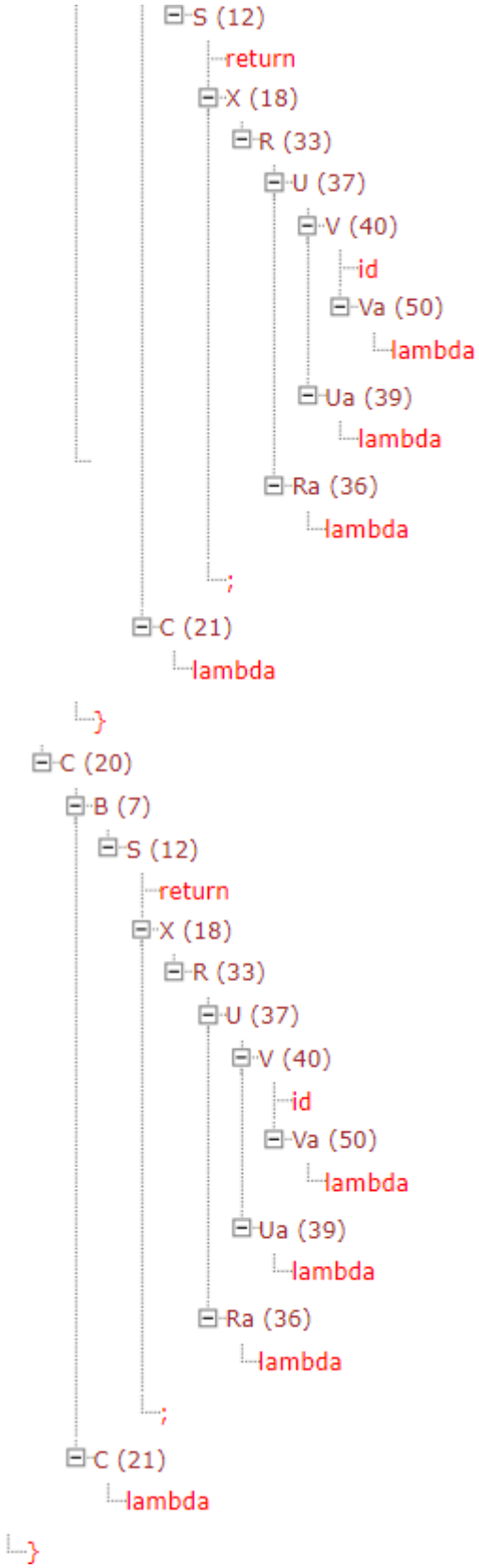

{
C (20)
B (5)
if
(
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (35)
!=
R (33)
U (37)
V (41)
entero
Ua (39)
lambda
Ra (36)
lambda
)
S (12)
return
X (18)
R (33)
U (37)
V (41)
entero
Ua (39)
lambda
Ra (36)
lambda
;

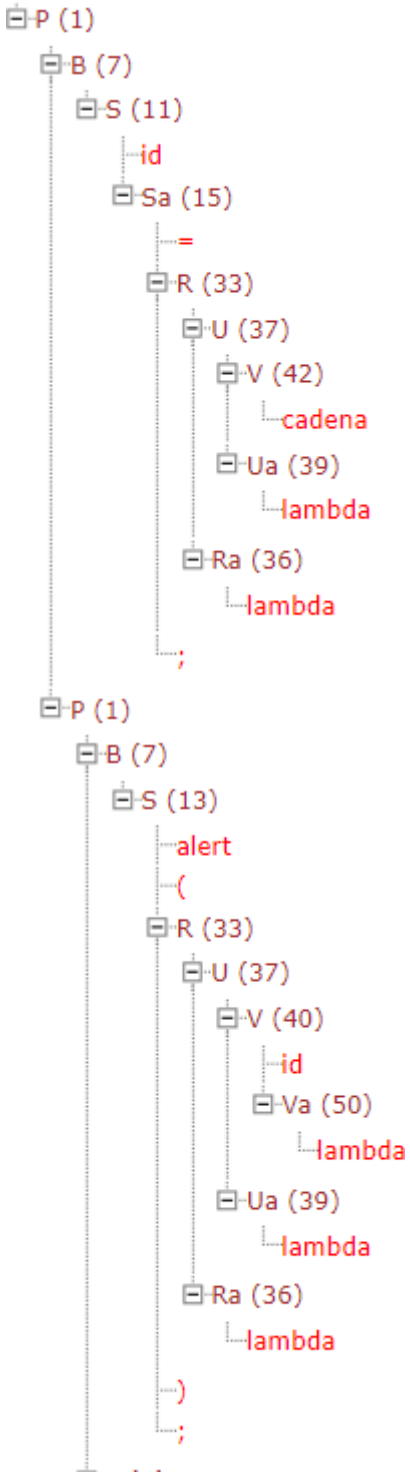

C (20)
B (7)
S (12)
return
X (18)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (38)
+
U (37)
V (40)
id
Va (49)
(
L (29)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (38)
+
U (37)
V (41)
entero
Ua (39)
lambda
Ra (36)
lambda
Q (32)
lambda
)
Ua (39)
lambda
Ra (36)
lambda
;

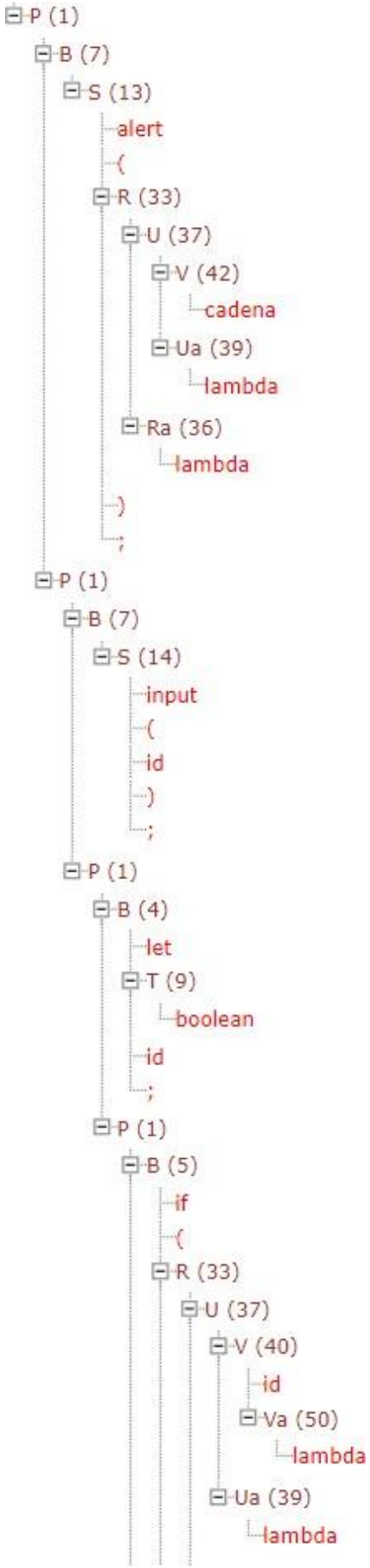

;
C (21)
lambda
}
P (1)
B (4)
let
T (8)
number
id
;
P (1)
B (4)
let
T (8)
number
id
;
P (1)
B (4)
let
T (8)
number
id
;
P (2)
F (22)
function
H (23)
T (10)
string
id
(
A (25)
T (9)
boolean
id
K (28)
lambda
)
{

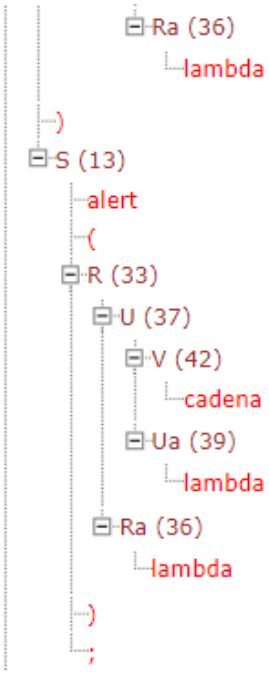

C (20)
B (6)
while
(
R (33)
U (37)
V (46)
!
Vb (53)
id
Ua (39)
lambda
Ra (36)
lambda
)
{
C (20)
B (7)
S (11)
id
Sa (17)
(
L (29)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
Q (31)
,
R (33)
U (37)
V (42)
cadena
Ua (39)
lambda

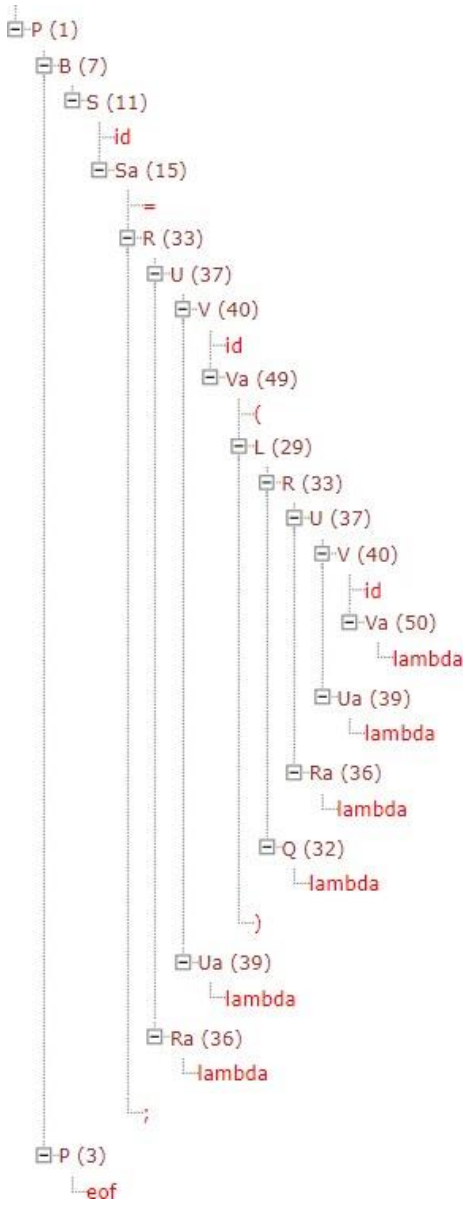

Ra (36)
lambda
Q (31)
,
R (33)
U (37)
V (41)
entero
Ua (39)
lambda
Ra (36)
lambda
Q (32)
lambda
)
;
C (20)
B (5)
if
(
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (35)
!=
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
)


S (12)
return
X (18)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
;
C (21)
lambda
}
C (20)
B (7)
S (12)
return
X (18)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
;
C (21)
lambda
}


P (1)
B (7)
S (11)
id
Sa (15)
=
R (33)
U (37)
V (42)
cadena
Ua (39)
lambda
Ra (36)
lambda
;
P (1)
B (7)
S (13)
alert
(
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
)
;


P (1)
B (7)
S (13)
alert
(
R (33)
U (37)
V (42)
cadena
Ua (39)
lambda
Ra (36)
lambda
)
;
P (1)
B (7)
S (14)
input
(
id
)
;
P (1)
B (4)
let
T (9)
boolean
id
;
P (1)
B (5)
if
(
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda


Ra (36)
lambda
)
S (13)
alert
(
R (33)
U (37)
V (42)
cadena
Ua (39)
lambda
Ra (36)
lambda
)
;


P (1)
B (7)
S (11)
id
Sa (15)
=
R (33)
U (37)
V (40)
id
Va (49)
(
L (29)
R (33)
U (37)
V (40)
id
Va (50)
lambda
Ua (39)
lambda
Ra (36)
lambda
Q (32)
lambda
)
Ua (39)
lambda
Ra (36)
lambda
;
P (3)
eof

• Prueba 2

Código fuente:

```
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)
{
        let number res;
        res = num1+num2;
        return res;
}
int = 0;
alert (operacion (a, b));
```

• Prueba 3

Código fuente:

```
let boolean booleano;
function boolean bisiesto (number a)
{       let number bis;
        alert ("Es bisiesto?");
        input(bis);
        return (a + 4 != 0);

}
function number dias (number m, number a)
{
        let number dd;
        alert ("di cuantos dias tiene el mes ");
        alert (m);
        input(dd);
```

```
        if (bisiesto(a)) dd = dd + 1;
        return dd;
}
function boolean esFechaCorrecta (number d, number m, number a)
{
        return d != dias (m, a);
}
function demo ()
{

        if (esFechaCorrecta(20, 10, 2020)) alert ("ok");
}
let number abc;
demo();
```

- ## Prueba 4

Código fuente:

```
let number n1;
let boolean l1;
let string cad;
let number n2;
let boolean l2;
input (n1);
l1 = l2;
if (! l2) cad = "hello";
n2 /= n1 + 378;

alert(    33
                +
                n1
                +
                n2);
function boolean ff(boolean ss)
{
```

```
        l2 = ! l1;
        if (l2) l1 = ff (ss);
        varglobal = 88;
        return (ss);
}
if (ff(l1)) alert (varglobal);
```

- Prueba 5

Código fuente:

```
let number x;
let number xx;
let string ss;
let boolean boolean_1;
let boolean boolean_2;
let number y;
/*;*/
function number f1(number f1, boolean b1)
{
                alert(ss);
                x /= xx+f1;
                boolean_1 = ! boolean_2;
                return (01234);
}


function boolean f2(number f2, boolean b1)
{
                input (y);
                alert ((4+5+77+(088+f2)));
                return (b1);
}

x =
  x + 6
    + z
```

```
    + 1
    + (2
    + y
    + 6)
        ;
alert (f2 (f1 (x, boolean_1), boolean_2));
```

Pruebas Incorrectas

• Prueba 1

Código fuente:

```
let number a;
let number b;
let number int;
a = 99999;
b = 99992;
c = 123;
```

Gestor de Errores:

->Error(Linea 4)Analizador lexico: El numero 99999 sobrepasa el valor permitido

->Error(Linea 5)Analizador lexico: El numero 99992 sobrepasa el valor permitido

• Prueba 2

Código fuente:

```
let number c;
let string a;
c = 1;
for (c; c!=10; c++){
        a= “fin”;
}
```

Gestor de Errores

->Error(Linea 1)Analizador sintactico: Se esperaba ')' y se ha recibido la expresion PuntoComa

• Prueba 3

Código fuente:

```
let number numero;
numero = 89999;
<;
let number numero;
function number f(number a, number b){/*funcion sin tipo*/
        a = 33 /*falta ;*/
return a
}
```

Gestor de Errores:

->Error(Linea 2)Analizador lexico: El numero 89999 sobrepasa el valor permitido

->Error(Linea 3)Analizador lexico: No se reconoce el caracter (<)

->Error(Linea 1)Analizador sintactico: No se permite la expresiónPuntoComa

• Prueba 4

Código fuente:

```
let number a;
let number b;
a = 2
b= 3;
let number n1;
let boolean l1;
let string cad;
let number n2;
let boolean l2;
input (n1);
l1 = l2;
if (! l2) cad = "hello";
```

```
n2 = n1 + 378;


alert(     33
                    +
                    n1
                    +
                    n2);
function boolean ff(boolean ss)
{
          l2 =  l1;
          if (l2) l1 = ff (ss);
          varglobal = 0888;
          return (ss);
}
if (ff(l2)) alert (varglobal);
```

Gestor de Errores:

->Error(Linea 3)Analizador sintactico: Se esperaba ';' y se ha recibido la expresion ID

• Prueba 5

Código fuente:

```
let string s;
let number uno;
let number UNO;
function number Factorial (number n)
{
          if (n != 0)          return 1;
          return n + Factorial (n + 1);
}
let number For;
let number functional;
let number While;
```

```
function string cadena (boolean log)
{
        while (!log)
        {
                print (s, "hola", 33);
                if (uno != UNO)   return s;
        }

}
```

Gestor de Errores:

->Error(Linea 1)Analizador semantico(Línea 1): No devuelve nada la Funcion cadena

Comentario Final

Como se ha podido observar hay varios fallos en este compilador, algunos de ellos son los siguientes:

- Hay problemas cuando aparece el símbolo de negación(!) por lo que produce algún problema, aunque no siempre. Lo más seguro es que esto se solucione retocando la gramática, ya que esta no es del todo adecuada.

- En el gestor de Errores, las líneas fallan en el analizador sintáctico.

Estos problemas no se han solucionado, no porque no se ha querido sino, por falta de tiempo y porque prefería no retocar mucho código antes de la defensa de la práctica ya que podía ocasionar algún fallo aún peor.

También se han solucionado los errores producidos en las anteriores entregas en lo máximo posible.