TRABAJO GRUPO 98: MEMORIA

Construcción de un Compilador

Asignatura: Procesadores de Lenguajes

Luis Bravo Collado

Inés Hernández Sánchez

Andrea Blanco Gómez

ÍNDICE

1. INTRODUCCIÓN

Para elaborar esta práctica hemos implementado en Java una serie de clases. Para la primera parte, el analizador léxico, tenemos la clase Analizador Lexico, la clase TS, para la estructura de la tabla de símbolos, y la clase escribeFichero para la salida tras ejecutar el compilador (devuelve tokens), donde además hemos hecho las pruebas. Nuestro grupo tenía como opciones específicas el operador asignación con y lógico (&=), el comentario de bloque (/**/), las cadenas con comillas simples (‘’), la sentencia repetitiva (for) y un analizador sintáctico descendente recursivo.

2. DISEÑO DEL ANALIZADOR LÉXICO

El analizador léxico es la primera fase de un compilador que recibe de entrada un código fuente (secuencia de caracteres) y produce una salida compuesta de tokens y símbolos, pedidos por el analizador sintáctico (siguiente fase).

Hay cuatro fases en el diseño del analizador léxico:

2.1 Tokens

2.2 Gramática (Léxica)

2.3 Autómata Finito Determinista

2.4 Acciones Semánticas y Errores

2.1 Tokens

Los tokens son unidades lógicas que resultan de agrupar los caracteres del fichero de entrada. Son suministrados al analizador sintáctico por el analizador léxico.

En nuestro caso, hemos añadido los siguientes tokens a nuestro compilador:

< MAS, - >	‘+’	
< MENOR, < >	‘<’	
< DOSIGUAL, - >	‘==‘	
< AND, - >	‘&&’	
< ID, POS_TS >	identificador	
< PALRES, n >	palabra reservada	*Dentro de este token, cada palabra reservada tiene su propio token (!= nombre).
< CADENA, CAD >	cadena	
< ESPECIAL, - >	‘&=’	
< ENT, valor >	valor entero	
< PARAB, - >	‘(‘	
< PARCER, - >	‘)’	
< EOL, - >	end of line	
< LLAVEABR, - >	'{‘	
< LLAVECERR, - >	‘}’	
< COMA, - >	‘,’	
< PUNTOCOMA, - >	‘;’	
< IGUAL, - >	‘=‘	

* {‘int’,’bool’,’string’,’prompt’,’print’,’true’,’false’,’var’,’return’,’for’,’function’}

2.2 Gramática

Conjunto de reglas que definen el léxico: gramática regular G = <N, T, P, S>.

N = símbolos no terminales: {A, B, C, D, E, F, G, H}

T = símbolos terminales = {‘eol’ , ‘+’ , ‘;’ , ‘,’ , ‘{’ , ‘}’ , ‘&’ , ‘(’ , ‘)’ , ‘l’ , ‘d’ , ‘=’ , ‘/’ , ‘c’ , ‘*’, ‘<’}

S = axioma inicial.

P = reglas de producción:

A -> delA | eol | + | ; | , | { | } | &A |) | (| lB | dC | ’D | =E | /F | <

B -> lB | dB | -B | λ

C -> dC | λ

D -> cD | ’

E -> = | λ

F -> *G

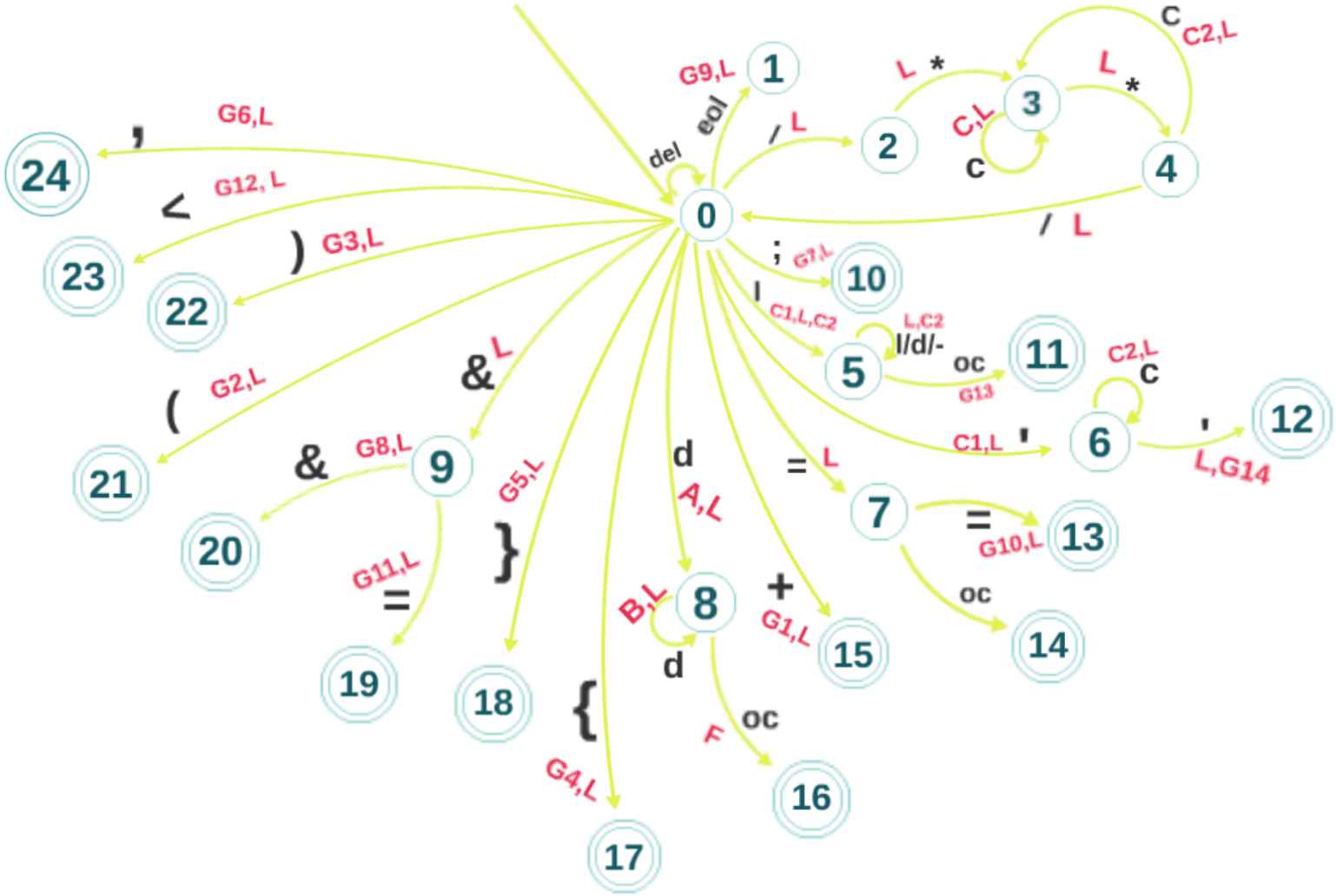
G -> cG | *H

H -> /A | cG

OBSERVACIONES:
l= letra
d= dígito
c= cualquier caracter

2.3 Autómata

Construido a partir de la gramática anterior. Señalando los símbolos que detecta la transición entre dos estados diferentes. En rojo, aparecen las acciones semánticas asociadas a cada transición.



2.4 Acciones Semánticas

Listamos las acciones semánticas utilizadas y necesarias en el autómata del punto 2.3. Se generan los tokens según los datos de entrada.

G_1 : GenToken(MAS, -)

G_2 : GenToken(PARAB, -)

G_3 : GenToken(PARCERR, -)

G_4 : GenToken(LLAVEAB, -)

G_5 : GenToken(LLAVECERR, -)

G_6 : GenToken(COMA, -)

G_7 : GenToken(PUNTOCOMA, -)

G_8 : GenToken(AND, -)

G_9 : GenToken(eol, -)

G_{10} : GenToken(DOSIGUAL, -)

G_{11} : GenToken(ESPECIAL, -)

G_{12} : GenToken(MENOR, -)

G_{13}: p=BuscaTS.n(pal)
GenToken (PALRES, p)

//Se busca la palabra en el array (n) para comprobar si es palabra reservada (ver código)

G_{14}:
```
if(pal = null) then
        p= BuscaTS(pal)
        GenToken(ID, p)
  else
        Error("Identificador no válido")
```

C_1: pal= ∅

C_2: concatenar(pal)

L: Leer: se leen todas las transiciones excepto en las de o.c.

E: Error: todas las transiciones no indicadas producirían una acción de error.

G_{15}: GenToken (CADENA, CAD)

A: num=d

B: num=num*10+d

F: if(num< 2^{15}) then
GenToken(ENTERO, num)
else (“Error, numero fuera de rango”)

3. TABLA DE SIMBOLOS

La tabla de símbolos es una estructura de datos donde se va a almacenar información relativa a los símbolos (identificadores) que aparecen en el programa fuente a analizar. La información relativa a los identificadores puede ser obtenida durante la fase de análisis léxico y es posible que otros módulos la necesiten después.

Su estructura: nombre (del id, lexema), y atributos (tipo, desplazamiento). Para las funciones hay tres atributos extra: numero de parámetros, tipo de parámetro y tipo de retorno.

El siguiente es un ejemplo de un tabla de símbolos después de analizar un archivo de prueba con nuestro código en java:

```
*LEXEMA : x
ATRIBUTOS:
        +tipo :bool
        +desp :1
*LEXEMA : y
ATRIBUTOS:
        +tipo :bool
        +desp :1
```

4. DISEÑO DEL ANALIZADOR SINTÁCTICO

El analizador sintáctico es el principal elemento del compilador. Recibe los tokens del analizador léxico a medida que los va tratando: trata un token, pide otro. En nuestro caso, utilizamos un analizador sintáctico descendente, que genera un análisis a izquierdas de una cadena de entrada dada.

4.1 Gramática

1. S' -> W
2. W -> SW
3. W -> FW
4. W -> eof
5. S -> varTid G';
6. S -> for(I;E;I){A}
7. S -> N
8. S -> λ
9. F -> function F' id(P){A}
10. A -> SA
11. A -> λ
12. F' -> λ
13. F' -> T
14. T -> int
15. T -> bool
16. T -> string
17. P -> λ
18. P -> TidT'
19. T' -> λ
20. T' -> ,TidT'
21. I -> idG
22. G -> &=E
23. G -> =E
24. G -> H
25. G' -> G
26. G' -> λ
27. E -> RE'
28. E -> (E)E'
29. E' -> && E
30. E' -> λ
31. E' -> ==E
32. E' -> <R
33. E' -> +E
34. H -> (C)
35. H -> λ
36. R -> idH
37. R -> numero
38. R -> cadena
39. R -> true
40. R -> false
41. C -> EC'
42. C -> λ
43. C' -> ,EC'
44. C' -> λ
45. N -> I;
46. N -> prompt(id);
47. N -> print(E);
48. N -> return K;
49. K -> E
50. K -> λ

4.2 Demostración Condición LL de la gramática

Ya que utilizamos un analizador sintáctico descendente recursivo predictivo, debemos demostrar que nuestra gramática anterior es de tipo LL(1).

Para ello hemos utilizado el software SDGLL1. Dentro del programa tenemos que definir el conjunto de símbolos terminales, el conjunto de símbolos no terminales, el axioma y la lista de producciones. Nos ha salido un análisis satisfactorio. Luego nuestra gramática es válida.

4.3 Procedimientos

```
PROCEDURE S'
  BEGIN
    IF siguiente_token ϵ {function, eof, var,
for, int, print, prompt, return, id} THEN
      W;
END
```

```
PROCEDURE W
  BEGIN
   IF siguiente_token='function' THEN
     F;
     W;
  ELSE IF siguiente_token='eof' THEN
     valida_token('eof');
  ELSE IF siguiente_token ϵ {var, for, int, print,
prompt, return, id} THEN
      S;
      W;
```

```
PROCEDURE S
  BEGIN
    IF siguiente_token='var' THEN
      valida_palres('var');
      T;
      valida_token('id');
      G';
      valida_token('puntocoma');
    ELSE IF siguiente_token='for' THEN
      valida_palres('for');
      valida_token('parab');
      I;
      valida_token('puntocoma');
      E;
      valida_token('puntocoma');
      I;
      valida_token('parcerr');
      valida_token('llaveabr');
      A;
      valida_token('llavecerr');
    ELSE IF siguiente_token ϵ {print, prompt,
return, id} THEN
      N;
      S;
  END
```

```
PROCEDURE F
  BEGIN
    IF siguiente_token='function' THEN
      valida_palres('function');
      F';
      valida_token('id');
      valida_token('parab');
      P;
      valida_token('parcerr');
      valida_token('llaveabr');
      S;
      valida_token('llavecerr');
   END
END
```

```
PROCEDURE A
 BEGIN
    IF siguiente_token ϵ {var, for, print,
prompt, return, id} THEN
       S;
       A;
    ELSE
    END
END
```

```
PROCEDURE F'
  BEGIN
    IF siguiente_token ϵ {int, bool, string} THEN
       T;
    ELSE
    END
   END
```

```
PROCEDURE T
  BEGIN
    IF siguiente_token='int' THEN
       valida_palres('int');
    ELSE IF siguiente_token='bool' THEN
       valida_palres('bool');
    ELSE IF siguiente_token='string' THEN
       valida_palres('string');
  END
END
```

```
PROCEDURE P
  BEGIN
    IF siguiente_token ϵ {int, bool, string} THEN
       T;
       valida_token('id');
       T';
    ELSE
    END
  END
```

```
PROCEDURE T'
  BEGIN
     IF siguiente_token=',' THEN
        valida_token('coma');
        T;
        valida_token('id');
        T';
     ELSE
     END
   END
END
```

.

```
PROCEDURE I
  BEGIN
    IF siguiente_token=‘id’ THEN
          valida_token(id);
          G;
  END
 END
```

```
PROCEDURE G
 BEGIN
   IF siguiente_token=‘=’THEN
         valida_token(igual);
         E;
   ELSE IF siguiente_token=‘&=’ THEN
         valida_token(especial);
         E;
   ELSE IF siguiente_token=‘(’ THEN
         H;
 END
```

```
PROCEDURE G’
 BEGIN
   IF siguiente_token ϵ {=, &=} THEN
         G;
   ELSE IF
   END
 END
```

```
PROCEDURE E
 BEGIN
   IF siguiente_token ϵ {id, numero, cadena,
true, false} THEN
         R;
         E’;
   ELSE IF siguiente_token=‘(’ THEN
         valida_token(parab);
         E;
         valida_token(parcerr);
         E’;
 END
```

```
PROCEDURE E’
 BEGIN
   IF siguiente_token=‘&&’ THEN
         valida_token(and);
         E;
   ELSE IF siguiente_token=‘==’ THEN
         valida_token(dosigual);
         E;
   ELSE IF siguiente_token=‘+’ THEN
         valida_token(mas);
         E;
   ELSE IF siguiente_token=‘<’ THEN
         valida_token(menor);
         R;
   ELSE
   END
 END
END
```

```
PROCEDURE H
 BEGIN
   IF siguiente_token=‘(’ THEN
         valida_token(parab);
         C;
         valida_token(parcerr);
   ELSE IF
   END
 END
```

```
PROCEDURE R
  BEGIN
    IF siguiente_token='id' THEN
          valida_token(id);
          H;
    ELSE IF siguiente_token='numero' THEN
          valida_token(ent);
    ELSE IF siguiente_token='cadena' THEN
          valida_token(cadena);
    ELSE IF siguiente_token='true' THEN
          valida_palres(true);
    ELSE IF siguiente_token='false' THEN
          valida_palres(false);
  END
END
```

```
PROCEDURE C'
  BEGIN
    IF siguiente_token=',' THEN
          valida_token(coma);
          E;
          C';
    ELSE IF
    END
END
```

```
PROCEDURE K
  BEGIN
    IF siguiente_token ϵ {id, numero, cadena,
true, false} THEN
           E;
    ELSE
    END
  END
END
```

```
PROCEDURE C
  BEGIN
    IF siguiente_token='id' THEN
          E;
          C';
    ELSE IF
    END
END
```

```
PROCEDURE N
  BEGIN
    IF siguiente_token='id' THEN
          I;
          valida_token(puntocoma);
    ELSE IF siguiente_token='prompt' THEN
          valida_palres(prompt);
          valida_token(parab);
          valida_token(id);
          valida_token(parcerr);
          valida_token(puntocoma);
    ELSE IF siguiente_token='print' THEN
          valida_palres(print);
          valida_token(parab);
          E;
          valida_token(parcerr);
          valida_token(puntocoma);
    ELSE IF siguiente_token='return' THEN
          valida_palres(return);
          K;
          valida_token(puntocoma);
  END
END
```

5. DISEÑO DEL ANALIZADOR SEMÁNTICO

En el Análisis Semántico se lleva a cabo la traducción dirigida por sintaxis, en la que se asocia cierta información al lenguaje. Con ella se consigue la gramática de contexto libre (GCL) y además unas reglas o acciones semánticas que permiten calcular el valor de los atributos asociados a un símbolo utilizando los valores de otros símbolos de esa misma regla sintáctica.

5.1 Traducción dirigida por sintaxis: GCL

1. S' -> W {S'-> {crearTS(despl=0)=TSG} W}
5. S -> varTid G'; {id.tipo:=buscar(id, posicion, TS, tipo)
S.tipo:=if(id.tipo!=NULL) then
tipo_ok
else
error_tipo}
6. S -> for(I;E;I){A} {S.tipo:=if((E.tipo==boolean) && (I.tipo==tipo_ok) &&
(A.tipo==tipo_ok))then
tipo_ok
else
error_tipo}
7. S -> N {S.tipo:=N.tipo}
8. S -> λ {S.tipo:=tipo_ok}
9. F -> function F' id(P){A} {F.tipo:= if(buscarTS(id)!=NULL) then
F.tipo:=F'.tipo
else
error_tipo}
10. A -> SA {A.tipo:= if(S.tipo!=NULL) then
S.tipo
else
error_tipo}
11. A -> λ {A.tipo:=ok}
12. F' -> λ {F'.tipo:=ok}
13. F' -> T {F'.tipo:=T.tipo}
14. T -> int {T.tipo:=int, T.tam:=2}
15. T -> bool {T.tipo:=boolean, T.tam:=1}
16. T -> string {T.tipo:=string, T.tam:=8}

17. P -> λ {P.tipo:=ok}

18. P -> TidT' {id.tipo:=buscar(id, posicion, TS, tipo)
P.tipo:=if(id.tipo!=NULL) then
tipo_ok
else
error_tipo}

19. T' -> λ {T'.tipo:=ok}

20. T' -> ,TidT' {id.tipo:=buscar(id, posicion, TS, tipo)
T'.tipo:=if(id.tipo!=NULL) then
tipo_ok
else
error_tipo}

21. I -> idG {id.tipo:=buscar(id, posicion, TS, tipo)
I.tipo:=if(id.tipo!=NULL) then
tipo_ok
else
error_tipo}

22. G -> &=E {G.tipo:= E.tipo, G.tam:=E.tam}

23. G -> =E {G.tipo:= E.tipo, G.tam:=E.tam}

24. G -> H {G.tipo:=H.tipo}

25. G' -> G {G'.tipo:=G.tipo}

26. G' -> λ {G'.tipo:=ok}

27. E -> RE' {E.tipo:=if(R.tipo:=E'.tipo) then
tipo_ok
else if(E'.tipo:=NULL)
tipo_ok
else
error_tipo}

28. E -> (E) {E.tipo:=ok}

29. E' -> && E {E'.tipo:=if(E.tipo!=NULL) then
boolean
else
error_tipo}

30. E' -> λ {E'.tipo:=ok}

31. E' -> ==E {E'.tipo:=E.tipo}

32. E' -> <R {E'.tipo:=if(R.tipo!=NULL) then
boolean
else
error_tipo}

33. E' -> +E {E'.tipo:=E.tipo}

34. H -> (C) {H.tipo:=C.tipo}

35. H -> λ {H.tipo:=ok}

36. R -> idH {id.tipo:=buscar(id, posicion, TS, tipo)
R.tipo:=if(id.tipo!=NULL) then
tipo_ok
else
error_tipo}

37. R -> numero {R.tipo:=int}

38. R -> cadena {R.tipo:=char}

39. R -> true {R.tipo:=boolean}

40. R -> false {R.tipo:=boolean}

41. C -> EC' {if(C'.tipo==NULL) then C.tipo:=E.tipo}

42. C -> λ {C.tipo:=ok}

43. C' -> ,EC' {C'.tipo:=E.tipo}

44. C' -> λ {C'.tipo:=ok}

45. N -> I; {N.tipo:=I.tipo}

46. N -> prompt(id); {if(buscar(id)==NULL) then
añadirTS(id)
N.tipo:=id.tipo}

47. N -> print(E); {N.tipo:= if(E.tipo!=NULL) then
E.tipo
else
error_tipo}

48. N -> return K; N.tipo:= if(K.tipo!=NULL) then
K.tipo
else
error_tipo}

49. K -> E {K.tipo:=E.tipo}

50. K -> λ {K.tipo:=ok}

6. PRUEBAS DE CÓDIGO FINALES

Al ejecutar la clase EscribeFichero de nuestro proyecto, tras haber insertado la sintaxis de la prueba en la que estamos, el programa devuelve 3 partes:

1. Análisis Léxico
2. Análisis Sintáctico
3. Tabla de Símbolos

Para las pruebas incorrectas el programa devuelve los mensajes de error detallando lo incorrecto de cada prueba.

6.1 Pruebas correctas

Prueba 1

```
var int x;
var int y;
x=39;
y=x+2;
```

Tabla de Símbolos:

```
*LEXEMA : x
ATRIBUTOS:
        +tipo :int
        +desp :2
*LEXEMA : y
ATRIBUTOS:
        +tipo :int
        +desp :2
```

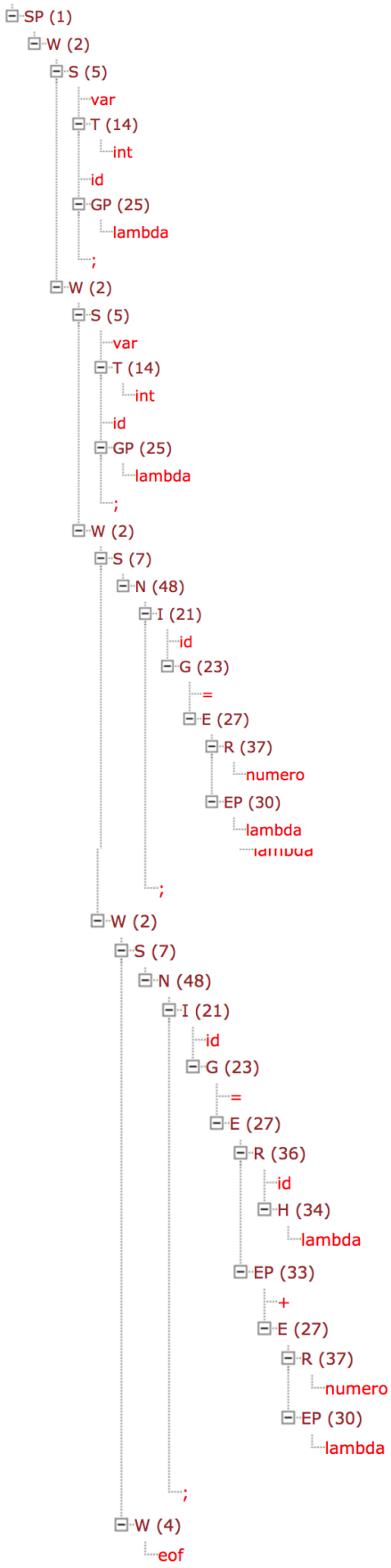
Análisis Léxico (Tokens):

```
<var,->
<int,->
<ID,1>
<PUNTOCOMA,->
<var,->
<int,->
<ID,2>
<PUNTOCOMA,->
<ID,1>
<IGUAL,->
<ENT,39>
<PUNTOCOMA,->
<ID,2>
<IGUAL,->
<ID,1>
<MAS,->
<ENT,2>
<PUNTOCOMA,->
<eof,->
```

Análisis Sintáctico (Parse):

```
Des 1 2 5 14 25 2 5 14 25 2 7 48 21 23 27 37 30 2 7 48 21 23 27 36 34 33 27 37 30 4
```

Árbol:


SP (1)
W (2)
S (5)
var
T (14)
int
id
GP (25)
lambda
;
W (2)
S (5)
var
T (14)
int
id
GP (25)
lambda
;
W (2)
S (7)
N (48)
I (21)
id
G (23)
=
E (27)
R (37)
numero
EP (30)
lambda
;
W (2)
S (7)
N (48)
I (21)
id
G (23)
=
E (27)
R (36)
id
H (34)
lambda
EP (33)
+
E (27)
R (37)
numero
EP (30)
lambda
;
W (4)
eof

Prueba 2

```
function int suma(int x, int y){
var int z=5;
x=y+z;

return x+y; }
var int t=suma(2,2);
```

Tabla de Símbolos:

```
*LEXEMA : suma
ATRIBUTOS:
      +tipo :function
      +numeroparametros :2
      +tipoparametro :int
      +tipoparametro :int
      +tiporetorno :int
*LEXEMA : t
ATRIBUTOS:
      +tipo :int
      +desp :2
Tabla de la funci□n    suma     #  1
*LEXEMA : x
      +tipo :int
      +desp :2
*LEXEMA : y
      +tipo :int
      +desp :2
*LEXEMA : z
      +tipo :int
      +desp :2
```

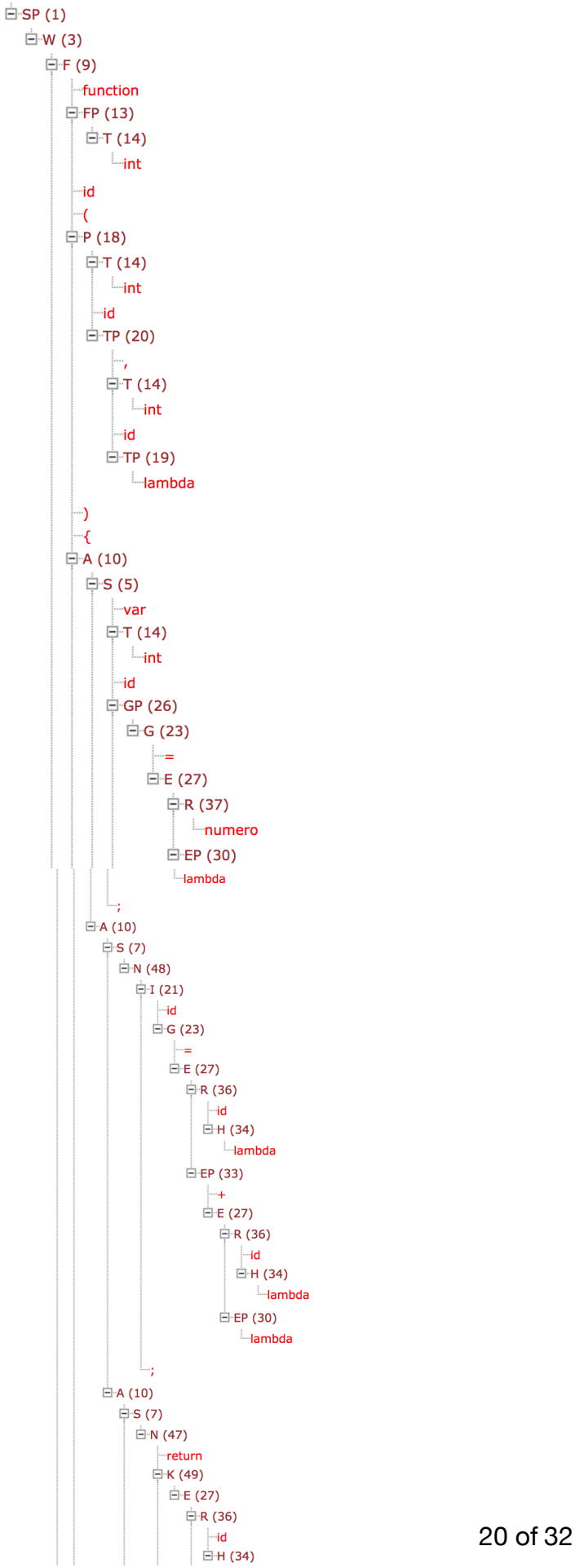
Análisis Léxico (Tokens):

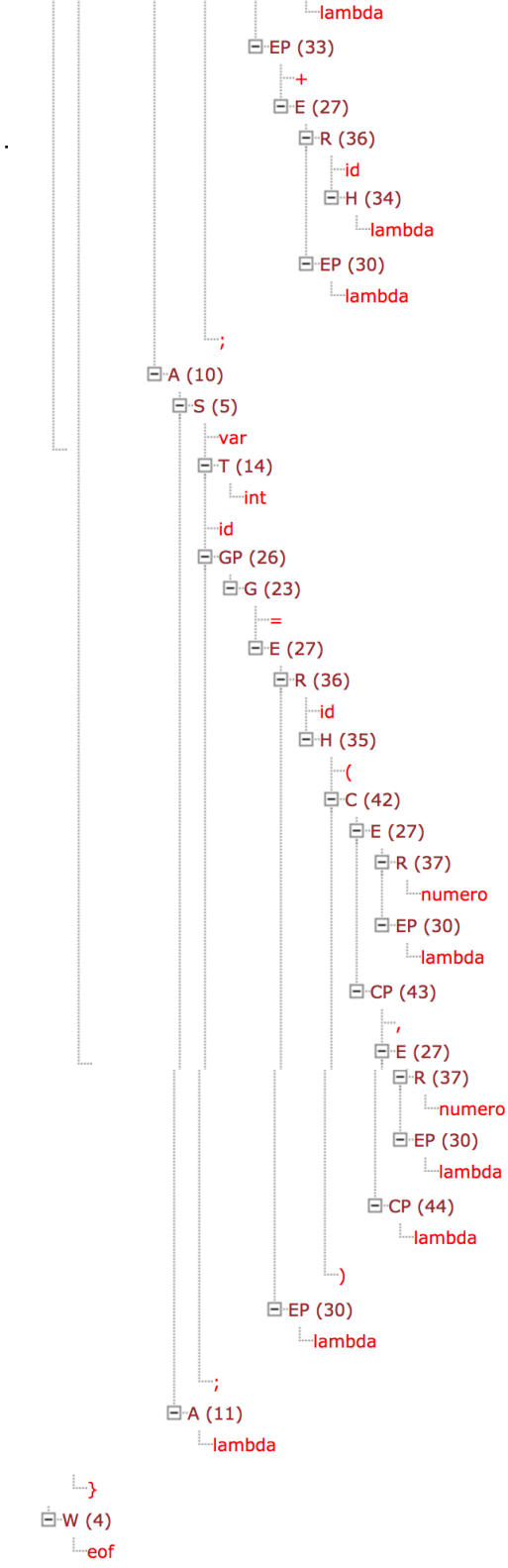
```
<function,->
<int,->
<ID,1>
<PARAB,->
<int,->
<ID,2>
<COMA,->
<int,->
<ID,3>
<PARCERR,->
<LLAVEAB,->
<var,->
<int,->
<ID,4>
<IGUAL,->
<ENT,5>
<PUNTOCOMA,->
<ID,2>
<IGUAL,->
<ID,3>
<MAS,->
<ID,4>
<PUNTOCOMA,->
<return,->
<ID,2>
<MAS,->
<ID,3>
<PUNTOCOMA,->
<LLAVECERR,->
<var,->
<int,->
<ID,5>
<IGUAL,->
<ID,1>
<PARAB,->
<ENT,2>
<COMA,->
<ENT,2>
<PARCERR,->
<PUNTOCOMA,->
<eof,->
```

Análisis Sintáctico (Parse):

```
Des 1 3 9 13 14 18 14 20 14 19 10 5 14 26 23 27 37 30 10 7 48 21 23 27 36 34 33
27 36 34 30 10 7 47 49 27 36 34 33 27 36 34 30 11 2 5 14 26 23 27 36 35 42 27 37
30 43 27 37 30 44 30 4
```

Árbol:


SP (1)
W (3)
F (9)
function
FP (13)
T (14)
int
id
(
P (18)
T (14)
int
id
TP (20)
,
T (14)
int
id
TP (19)
lambda
)
{
A (10)
S (5)
var
T (14)
int
id
GP (26)
G (23)
=
E (27)
R (37)
numero
EP (30)
lambda
;
A (10)
S (7)
N (48)
I (21)
id
G (23)
=
E (27)
R (36)
id
H (34)
lambda
EP (33)
+
E (27)
R (36)
id
H (34)
lambda
EP (30)
lambda
;
A (10)
S (7)
N (47)
return
K (49)
E (27)
R (36)
id
H (34)


lambda
EP (33)
+
E (27)
R (36)
id
H (34)
lambda
EP (30)
lambda
;
A (10)
S (5)
var
T (14)
int
id
GP (26)
G (23)
=
E (27)
R (36)
id
H (35)
(
C (42)
E (27)
R (37)
numero
EP (30)
lambda
CP (43)
,
E (27)
R (37)
numero
EP (30)
lambda
CP (44)
lambda
)
EP (30)
lambda
;
A (11)
lambda
}
W (4)
eof

Prueba 3

```
var bool x=true;
var bool y=false;
y&=x;
```

Tabla de Símbolos:

```
*LEXEMA : x
ATRIBUTOS:
      +tipo :bool
      +desp :1
*LEXEMA : y
ATRIBUTOS:
      +tipo :bool
      +desp :1
```

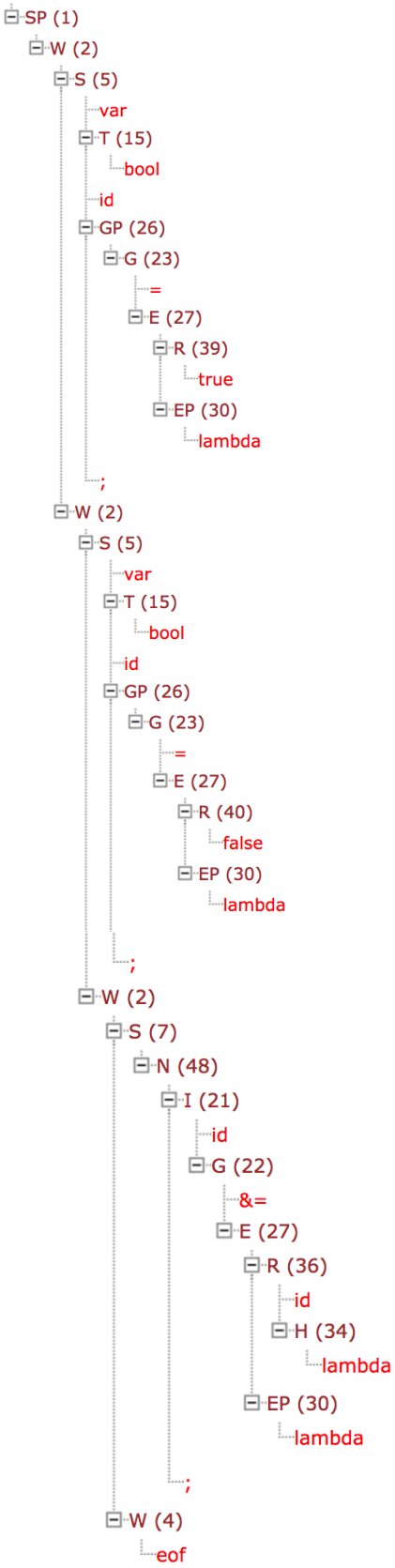
Análisis Léxico (Tokens):

```
<var,->
<bool,->
<ID,1>
<IGUAL,->
<true,->
<PUNTOCOMA,->
<var,->
<bool,->
<ID,2>
<IGUAL,->
<false,->
<PUNTOCOMA,->
<ID,2>
<ESPECIAL,->
<ID,1>
<PUNTOCOMA,->
<eof,->
```

Análisis Sintáctico (Parse):

```
Des 1 2 5 15 26 23 27 39 30 2 5 15 26 23 27 40 30 2 7 48 21 22 27 36 34 30 4
```

Árbol:


SP (1)
W (2)
S (5)
var
T (15)
bool
id
GP (26)
G (23)
=
E (27)
R (39)
true
EP (30)
lambda
;
W (2)
S (5)
var
T (15)
bool
id
GP (26)
G (23)
=
E (27)
R (40)
false
EP (30)
lambda
;
W (2)
S (7)
N (48)
I (21)
id
G (22)
&=
E (27)
R (36)
id
H (34)
lambda
EP (30)
lambda
;
W (4)
eof

Prueba 4

```
function bool booleano (bool x){
x=false;
return x;}
var int y;
var int z=10;
var bool xx=true;
for(y=10;y<z;z=z+1){
booleano(xx);}
;
```

Tabla de Símbolos:

```
*LEXEMA : booleano
ATRIBUTOS:
      +tipo :function
      +numeroparametros :1
      +tipoparametro :bool
      +tiporetorno :bool
*LEXEMA : y
ATRIBUTOS:
      +tipo :int
      +desp :2
*LEXEMA : z
ATRIBUTOS:
      +tipo :int
      +desp :2
*LEXEMA : xx
ATRIBUTOS:
      +tipo :bool
      +desp :1
Tabla de la funci□n   booleano    #  1
*LEXEMA : x
      +tipo :bool
      +desp :1
```

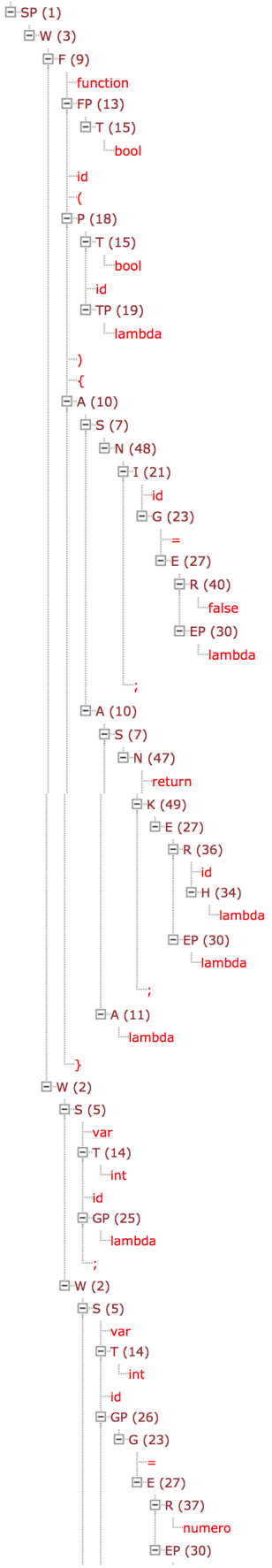
Análisis Léxico (Tokens):

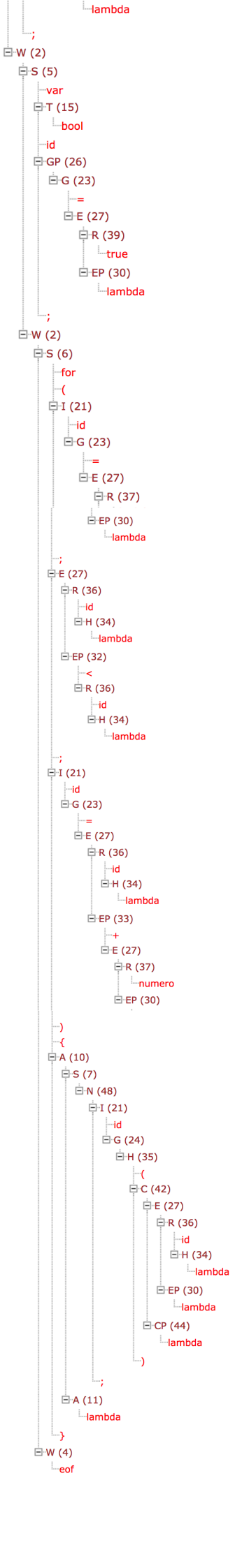
```
<function,->
<bool,->
<ID,1>
<PARAB,->
<bool,->
<ID,2>
<PARCERR,->
<LLAVEAB,->
<ID,2>
<IGUAL,->
<false,->
<PUNTOCOMA,->
<return,->
<ID,2>
<PUNTOCOMA,->
<LLAVECERR,->
<var,->
<int,->
<ID,3>
<PUNTOCOMA,->
<var,->
<int,->
<ID,4>
<IGUAL,->
<ENT,10>
<PUNTOCOMA,->
<var,->
<bool,->
<ID,5>
<IGUAL,->
<true,->
<PUNTOCOMA,->
<for,->
<PARAB,->
<ID,3>
<IGUAL,->
<ENT,10>
<PUNTOCOMA,->
<ID,3>
<MENOR,->
<ID,4>
<PUNTOCOMA,->
<ID,4>
<IGUAL,->
<ID,4>
<MAS,->
<ENT,1>
<PARCERR,->
<LLAVEAB,->
<ID,1>
<PARAB,->
<ID,5>
<PARCERR,->
<PUNTOCOMA,->
<LLAVECERR,->
<eof,->
```

Análisis Sintáctico (Parse):

```
Des 1 3 9 13 15 18 15 19 10 7 48 21 23 27 40 30 10 7 47 49 27 36 34 30 11
2 5 14 25 2 5 14 26 23 27 37 30 2 5 15 26 23 27 39 30 2 6 21 23 27 37 30 27 36 34 32
36 34 21 23 27 36 34 33 27 37 30 10 7 48 21 24 35 42 27 36 34 30 44 11 4
```

Árbol:


SP (1)
W (3)
F (9)
function
FP (13)
T (15)
bool
id
(
P (18)
T (15)
bool
id
TP (19)
lambda
)
{
A (10)
S (7)
N (48)
I (21)
id
G (23)
=
E (27)
R (40)
false
EP (30)
lambda
;
A (10)
S (7)
N (47)
return
K (49)
E (27)
R (36)
id
H (34)
lambda
EP (30)
lambda
;
A (11)
lambda
}
W (2)
S (5)
var
T (14)
int
id
GP (25)
lambda
;
W (2)
S (5)
var
T (14)
int
id
GP (26)
G (23)
=
E (27)
R (37)
numero
EP (30)


lambda
;
W (2)
S (5)
var
T (15)
bool
id
GP (26)
G (23)
=
E (27)
R (39)
true
EP (30)
lambda
;
W (2)
S (6)
for
(
I (21)
id
G (23)
=
E (27)
R (37)
EP (30)
lambda
;
E (27)
R (36)
id
H (34)
lambda
EP (32)
<
R (36)
id
H (34)
lambda
;
I (21)
id
G (23)
=
E (27)
R (36)
id
H (34)
lambda
EP (33)
+
E (27)
R (37)
numero
EP (30)
)
{
A (10)
S (7)
N (48)
I (21)
id
G (24)
H (35)
(
C (42)
E (27)
R (36)
id
H (34)
lambda
EP (30)
lambda
CP (44)
lambda
)
;
A (11)
lambda
}
W (4)
eof

Prueba 5

```
var int x;
x=3+2;
var bool b;
b=(x+3<5)&&(7<8);
var int j;
for(j=0;true;j=j+1){
print(16);}
```

Tabla de Símbolos:

```
*LEXEMA : x
ATRIBUTOS:
      +tipo :int
      +desp :2
*LEXEMA : b
ATRIBUTOS:
      +tipo :bool
      +desp :1
*LEXEMA : j
ATRIBUTOS:
      +tipo :int
      +desp :2
```

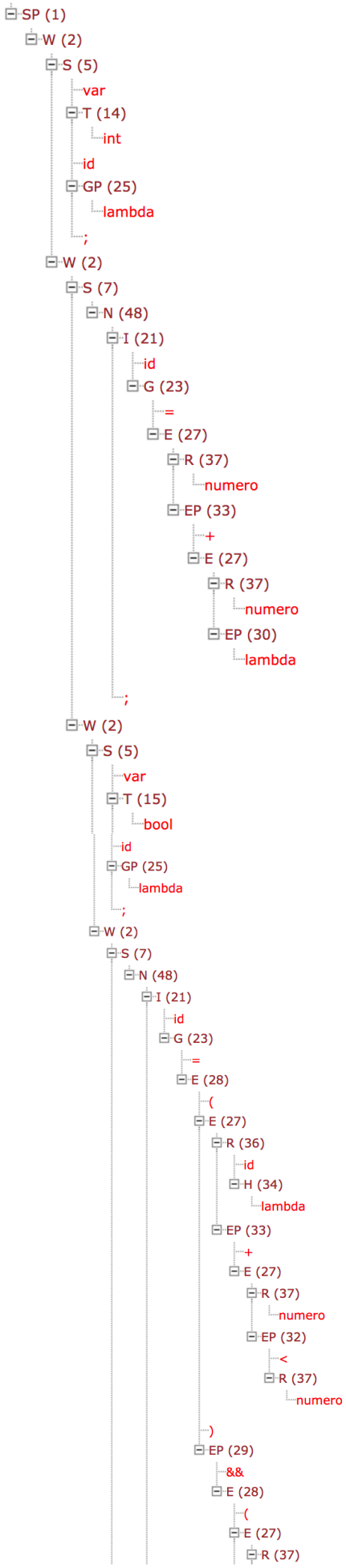
Análisis Léxico (Tokens):

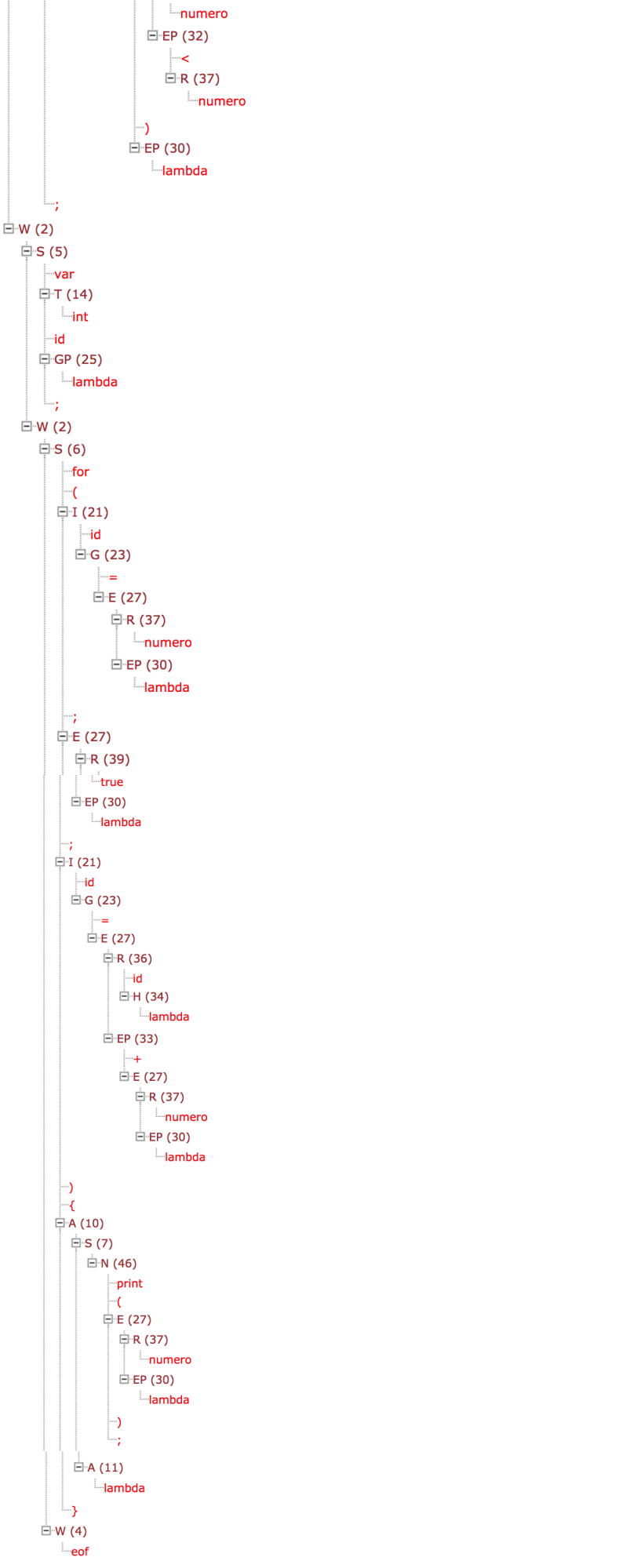
```
<var,->
<int,->
<ID,1>
<PUNTOCOMA,->
<ID,1>
<IGUAL,->
<ENT,3>
<MAS,->
<ENT,2>
<PUNTOCOMA,->
<var,->
<bool,->
<ID,2>
<PUNTOCOMA,->
<ID,2>
<IGUAL,->
<PARAB,->
<ID,1>
<MAS,->
<ENT,3>
<MENOR,->
<ENT,5>
<PARCERR,->
<AND,->
<PARAB,->
<ENT,7>
<MENOR,->
<ENT,8>
<PARCERR,->
<PUNTOCOMA,->
<var,->
<int,->
<ID,3>
<PUNTOCOMA,->
<for,->
<PARAB,->
<ID,3>
<IGUAL,->
<ENT,0>
<PUNTOCOMA,->
<true,->
<PUNTOCOMA,->
<ID,3>
<IGUAL,->
<ID,3>
<MAS,->
<ENT,1>
<PARCERR,->
<LLAVEAB,->
<print,->
<PARAB,->
<ENT,16>
<PARCERR,->
<PUNTOCOMA,->
<LLAVECERR,->
<eof,->
```

Análisis Sintáctico (Parse):

```
Des 1 2 5 14 25 2 7 48 21 23 27 37 33 27 37 30 2 5 15 25 2 7 48 21 23 28 27 36 34 33 27 37 32 37 29
28 27 37 32 37 30 2 5 14 25 2 6 21 23 27 37 30 27 39 30 21 23 27 36 34 33 27 37 30 10 7 46 27 37 30 11 4
```

Árbol:


SP (1)
W (2)
S (5)
var
T (14)
int
id
GP (25)
lambda
;
W (2)
S (7)
N (48)
I (21)
id
G (23)
=
E (27)
R (37)
numero
EP (33)
+
E (27)
R (37)
numero
EP (30)
lambda
;
W (2)
S (5)
var
T (15)
bool
id
GP (25)
lambda
;
W (2)
S (7)
N (48)
I (21)
id
G (23)
=
E (28)
(
E (27)
R (36)
id
H (34)
lambda
EP (33)
+
E (27)
R (37)
numero
EP (32)
<
R (37)
numero
)
EP (29)
&&
E (28)
(
E (27)
R (37)


numero
EP (32)
<
R (37)
numero
)
EP (30)
lambda
;
W (2)
S (5)
var
T (14)
int
id
GP (25)
lambda
;
W (2)
S (6)
for
(
I (21)
id
G (23)
=
E (27)
R (37)
numero
EP (30)
lambda
;
E (27)
R (39)
true
EP (30)
lambda
;
I (21)
id
G (23)
=
E (27)
R (36)
id
H (34)
lambda
EP (33)
+
E (27)
R (37)
numero
EP (30)
lambda
)
{
A (10)
S (7)
N (46)
print
(
E (27)
R (37)
numero
EP (30)
lambda
)
;
A (11)
lambda
}
W (4)
eof

6.2 Pruebas incorrectas

Prueba 6

```
var int t=0;
var int y=8888888;
var int x;
x=t*y;
```

-Mensajes de error:

```
Error Lexico : la representacion del numero excede los 2 bytes en la linea 2
Error Lexico : caracter invalido en la linea 4
```

Prueba 7

```
var int x;
x='hola';
```

-Mensajes de error:

```
Error Semantico en linea 2:se asigna string cuando deberia haberse asignado int
```

Prueba 8

```
var bool x;
var int u=8;
x=u+3<20&&2<3;
```

-Mensaje de error:

```
Error Sintactico en la linea  3:se recibio el token  <AND,-> pero se esperaba <PUNTOCOMA,->
```

Prueba 9

```
function int suma(int a, int b) {
var int z=a+b;
}
```

-Mensajes de error:

```
Error semantico en la linea: 2:la funcion deberia devolver int pero devuelve
```

Prueba 10

```
var bool t=true;
prompt(t);}
```

-Mensaje de error:

```
Error semantico en la linea: 2 la variable del prompt es de tipo logico
```