Memoria del proyecto de Procesadores de Lenguajes

Grupo 39:

Ignacio Amaya de la Peña (t11m021)
Alejandro Hernández Munuera (t11m033)
Miguel Zaballa Pardo (s10m034)

Índice

Introducción

El objetivo del proyecto es realizar un procesador del lenguaje JavaScript con las especificaciones correspondientes a este grupo. Además de las especificaciones generales del lenguaje nos correspondía implementar vectores, las sentencias do-while, el operador de autoincremento ++ y el realizar un análisis sintáctico ascendente LR.

De las especificaciones opcionales hemos implementado los comentarios /* */, los enteros en base decimal, el operador aritmético +, el operador de relación ==, el operador de asignación = y el operador lógico &&.

La práctica la hemos desarrollado en Python debido a sus diversas ventajas respecto a java en el tratamiento de las listas y de las tuplas. Esta decisión ha resultado ser un acierto, ya que en ciertos puntos de la práctica Java nos habría supuesto algunos problemas a la hora de implementarlo.

Analizador léxico

El primer paso para desarrollar el analizador léxico fue hacer las elecciones en las especificaciones del lenguaje para saber que tokens íbamos a generar y que condiciones iba a tener cada uno. Después generamos la gramática y el autómata correspondiente.

Palabras clave:

var
new
Array
prompt
if
do
while
function
document.write
return

Tokens:

< palabraClave , lexema >
< id , lexema >
< entero , valor >
< (, None >
<) , None >
< [, None >
<] , None >
< { , None >
< } , None >
< cadena , lexema >
< ==, None >
< = , None >

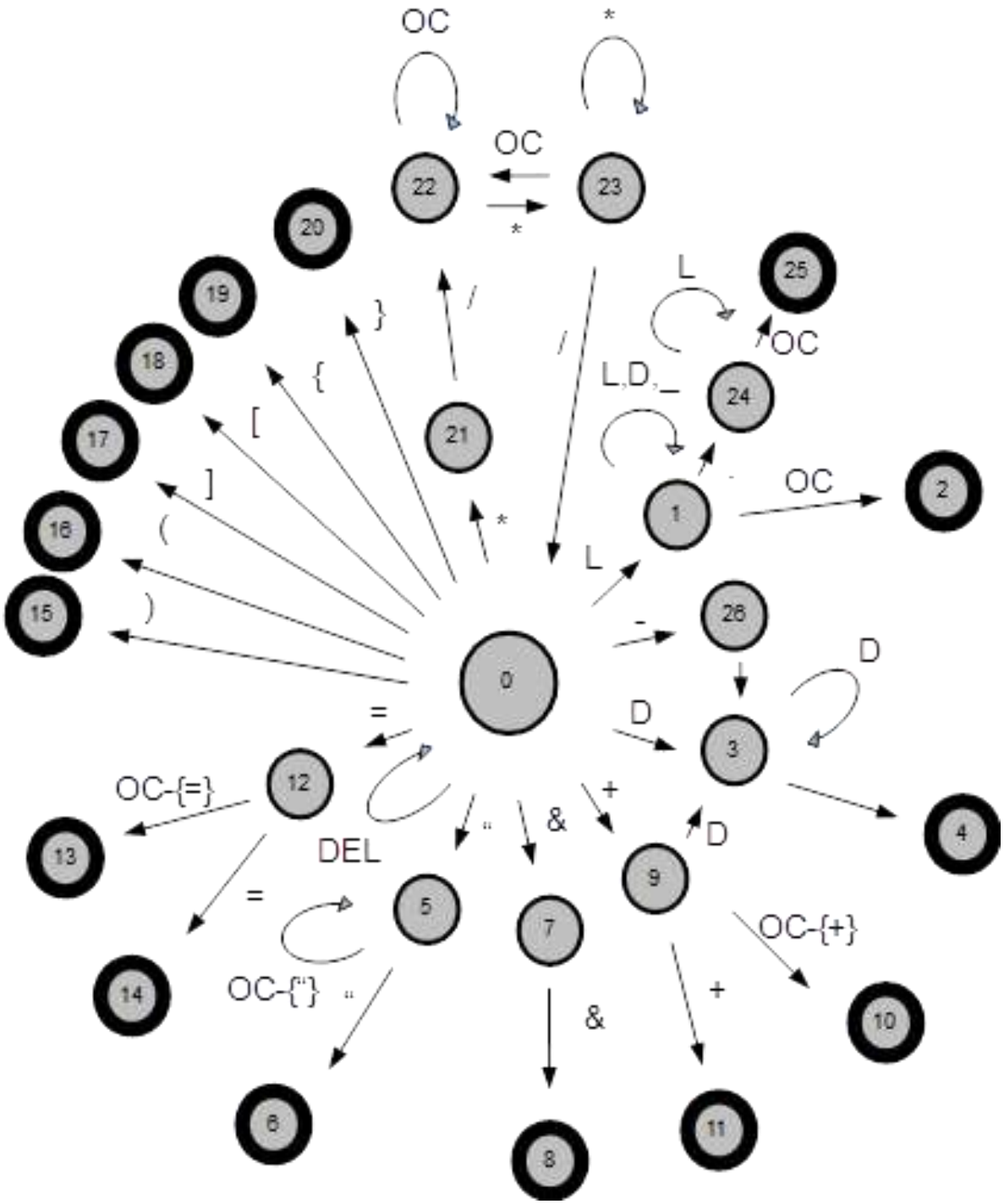
< + , None >
< ++ , None >
< && , None >

Gramática:

l : letra
d : dígito
c : cualquier carácter o dígito
q : cualquier carácter o dígito menos *

S -> lA | dB | “C | &D | +E | =F | { | } | [|] | (|) | /G | -B1
A -> lA | d A | _A | .A1 | λ
A1 -> lA1 | dA1 | _A1 | λ
B -> dB | λ
B1 -> dB
C -> cC | “
D -> &
E -> + | dB | λ
F -> = | λ
G -> *H
H -> *J | qH
J -> cH | /S

Autómata:


OC
*
OC
22
23
*
20
19
18
17
16
15
}
/
{
[
]
(
)
21
*
/
L
25
OC
L,D,_
24
1
OC
2
L
-
26
D
0
=
D
3
12
OC-{=}
13
14
DEL
=
OC-{"}
"
5
6
"
&
7
8
+
D
9
OC-{+}
+
&
10
11
4

Acciones semánticas:

(0 , 1) : leer, lexema = c
(0 , 26) : leer, negativo = True
(0 , 3) : leer, valor = d
(0 , 15) : leer, generarToken(< (, None >)
(0 , 16) : leer, generarToken(<) , None >)
(0 , 17) : leer, generarToken(< { , None >)
(0 , 18) : leer, generarToken(< } , None >)
(0 , 19) : leer, generarToken(< [, None >)
(0 , 20) : leer, generarToken(<] , None >)
(1 , 1) : leer, lexema = lexema + c
(1 , 2) : if (comprobarPalabraClave(lexema)) :
generarToken(< palabraClave , lexema)
vaciar(lexema)
else:
generaToken (< id , lexema >)
vaciar(lexema)
(1 , 24) : leer, lexema=lexema+c
(24 , 24) : leer, lexema=lexema+c
(24 , 25) : if (comprobarPalabraClave(lexema)) :
generarToken(< palabraClave , lexema)
vaciar(lexema)
else:
Error
(3 , 3) : leer, valor= valor * 10 + d
(3 , 4) : if (valor< 32767 && negativo=False):
generarToken(< entero , valor >)
vaciar(valor)
elseif(valor < 32767 && negativo=True)
generarToken(< entero , -valor >)
vaciar(valor)
else:
Error
(5 , 5) : leer, lexema=lexema+c
(5 , 6) : leer, generarToken(< cadena , lexema >) , vaciar(lexema)
(26 , 3) : leer, valor=d
(12 , 13) : leer, generarToken(< == , None >)

(12 , 14) : generarToken(< = , None >)
(9 , 10) : generarToken(< + , None >)
(9 , 11) : leer, generarToken(< ++ , None >)
(9 , 3) : leer, valor=d
(7 , 8) : generarToken(< && , None >)
(0 , 21), (21 , 22), (22 , 22), (22 , 23) , (23 , 22) , (23, 23) , (23 , 0), (0 , 9), (0 , 5), (0 , 7), (0 , 12) : leer

Implementación:

Decidimos implementar el autómata de tal manera que fuera generando los tokens y metiéndolos en una lista. Lo primero fue crear las funciones para que cada nodo comparara el carácter leído con las distintas opciones que este ofrecía. El segundo paso fue desarrollar las funciones semánticas de cada nodo dependiendo del nodo al que se va a pasar y después crear la clase token y la clase nodo con las funciones semánticas y las funciones de comparación pertinentes. Con esto ya hecho creamos una clase lector que se encargaba de leer carácter a carácter un archivo y comenzando en el nodo 0 el lector iba pasando el carácter leído al nodo en el que se encontraba en ese momento generando error si el carácter no es el adecuado y pasando al nodo 0 si el nodo en el que nos encontramos es terminal. Debido a la manera en que implementamos el analizador léxico se nos presentaron dos problemas: hay que tener cuidado a la hora de sumar debido a que si escribes el signo + pegado al número el analizador léxico lo interpreta como un entero no como suma y también que no obligamos a que entre dos tokens distintos sea necesario que haya un separador.

Analizador sintáctico

Esta parte fue la más costosa porque para desarrollar nuestro analizador sintáctico ascendente LR era necesario tener en cuenta la tabla de acciones y goto, y toda esta información tuvimos que pasarla a mano a nuestro proyecto. El primer paso de esta parte fue realizar la gramática para luego con la herramienta Sefalas construir la tabla de acciones y goto. Este paso supuso muchos problemas, ya que al principio, con los primeros intentos de gramáticas, saltaban conflictos en la tabla de acciones al crear las tablas. Cuando conseguimos solventar el problema y pasar todos los datos de las tablas al proyecto nos dimos cuenta de la gramática tenía errores y tuvimos que rehacerla y volver a pasar todos los datos de nuevo. Con las tablas de acciones y goto y con las producciones de la gramática decidimos crear una pila. También cambiamos el formato de salida del analizador léxico para que nos diera una lista de las palabras de los tokens que había leído del archivo y añadiese $ al final para que el analizador funcionase y crease las funciones desplazar y reducir.

Gramática:

1. S -> H
2. H -> H A
3. H -> H D
4. H -> H W
5. H -> H F
6. H -> H P
7. H -> H M
8. H -> lambda
9. H1 -> H1 A
10. H1 -> H1 D1
11. H1 -> H1 W
12. H1 -> H1 F
13. H1 -> H1 P

14. H1 -> H1 M
15. H1 -> lambda
16. A -> I = E
17. A -> I = ++ I
18. I -> id
19. I -> id [N]
20. D -> var id
21. D -> var id = new Array (entero)
22. D -> id (N1)
23. D -> ++ I
24. N1 -> N
25. N1 -> lambda
26. D -> function id ( ) { H1 return }
27. D -> function id (id) { H1 return }
28. D -> function id ( ) { H1 return E }
29. D -> function id (id) { H1 return E }
30. D1 -> var id
31. D1 -> var id = new Array (entero)
32. D1 -> id (N1)
33. D1 -> ++ I
34. W -> do { H1 } while (E)
35. F -> if (E) R
36. R -> A
37. R -> D
38. R -> W
39. R -> F
40. R -> P
41. R -> M
42. P -> prompt (id)
43. M -> document.write (N)
44. M -> document.write (cadena)
45. E -> E && E1
46. E -> E1
47. E1 -> N == N
48. E1 -> N
49. N -> N + entero
50. N -> N + id
51. N -> N + id [N]
52. N -> N + id (N1)
53. N -> entero

54. N -> id
55. N -> id [N]
56. N -> id (N1)

Implementación:

Teniendo ya como datos la tabla de acciones y goto y la lista de producciones de la correspondiente gramática, lo primero fue cambiar el formato de la salida del léxico, cogiendo de todos los tokens la palabra de la izquierda excepto de las palabras clave y añadimos $ al final de esta lista.
Creamos una lista, con el elemento 0 correspondiente al estado 0, que haría de la pila necesaria para realizar el algoritmo del analizador sintáctico. Luego realizamos las funciones desplazar y reducir:
La función desplazar mete en la pila la palabra leída de la lista de palabras y luego el estado que ha leído en la tabla de acciones y elimina el primer elemento de las palabras.
La función reducir saca de la pila el doble de elementos que haya a la derecha de la producción correspondiente comprobando si uno de cada dos elementos que saca corresponde con la producción y luego mete el elemento leído de la tabla goto. Sí la reducción ha ido bien mete el número de la producción en una lista llamada parse.

Con esto ya hecho empezando con la pila solo con el estado 0 y la lista de palabras correspondiente a los tokens generados en orden vamos leyendo de la tabla de acciones según el algoritmo usando reducir cuando lea una r y desplazar cuando lea una s de la tabla de acciones, este procedimiento termina cuando lea la palabra aceptar de la tabla de acciones y la longitud de la pila sea tres.
Debido a no tener en cuenta el token coma las funciones solo pueden tener un argumento y siempre tienen que tener un return.

Analizador semántico

Debido a que en nuestro lenguaje los identificadores solo pueden ser enteros (ya que el tipo booleano también lo tratamos como entero debido a la conversión automática de tipos básicos), vectores y funciones, en esta parte hicimos una función que va leyendo los tokens y dependiendo de lo que lea realiza las siguientes acciones:

Si detecta que estamos ante una función, mete la variable en la tabla de símbolos principal si no ha sido declarada antes y el argumento de ésta la añade en la tabla de símbolos hija. Si no tiene argumentos esto se indica en la tabla de símbolos principal en la variable correspondiente a esta función y a partir de ahí trabajaremos en adelante con la tabla hija.

Si lee var mete en la tabla de símbolos la variable si no ha sido declarada antes con el tipo que corresponda dependiendo de si lo que viene después es '= new Array(entero)' o no.

Si lee return comprueba que el valor que retorna la función usa identificadores ya declarados, modifica la variable correspondiente a la función en la tabla de símbolos principal indicando que devuelve un tipo entero y elimina la tabla de símbolos hija.

Si lee un identificador entonces comprueba si ha sido declarada antes y que tipo es, dependiendo del tipo se asegura si está siendo usada de forma correcta:

Si es de tipo vector que después del identificador venga '['

Si es de tipo function que después del identificador halla '(' y comprobar si debería tener argumentos o no y si devuelve algo o no.

Pruebas

Pruebas correctas:

PruebaC1.js

CÓDIGO

```
var T=new Array(7)
T[1]=5
function a(){
var B= new Array(2)
var f
f=B[T[1]]
return f
}
function b(){
var a
a=2
var c
c=3
var d
d=8
return
}
var c
c=a()
b()
```

PARSE

Ascendente 8 21 3 53 19 53 48 46 16 2 15 31 10 30 10 18 53 55 55 48 46 16 9 54 48 46 28 3 15 30 10 18 53 48 46 16 9 30 10 18 53 48 46 16 9 30 10 18 53 48 46 16 9 26 3 20 3 18 25 56 48 46 16 2 25 22 3 1

TABLAS DE SÍMBOLOS

##
Tabla de símbolos secundaria (borrada): a
##

| B | array | 2 | None |

| f | entero | None | None |

##
Tabla de símbolos secundaria (borrada): b
##

| a | entero | None | None |

| c | entero | None | None |

| d | entero | None | None |

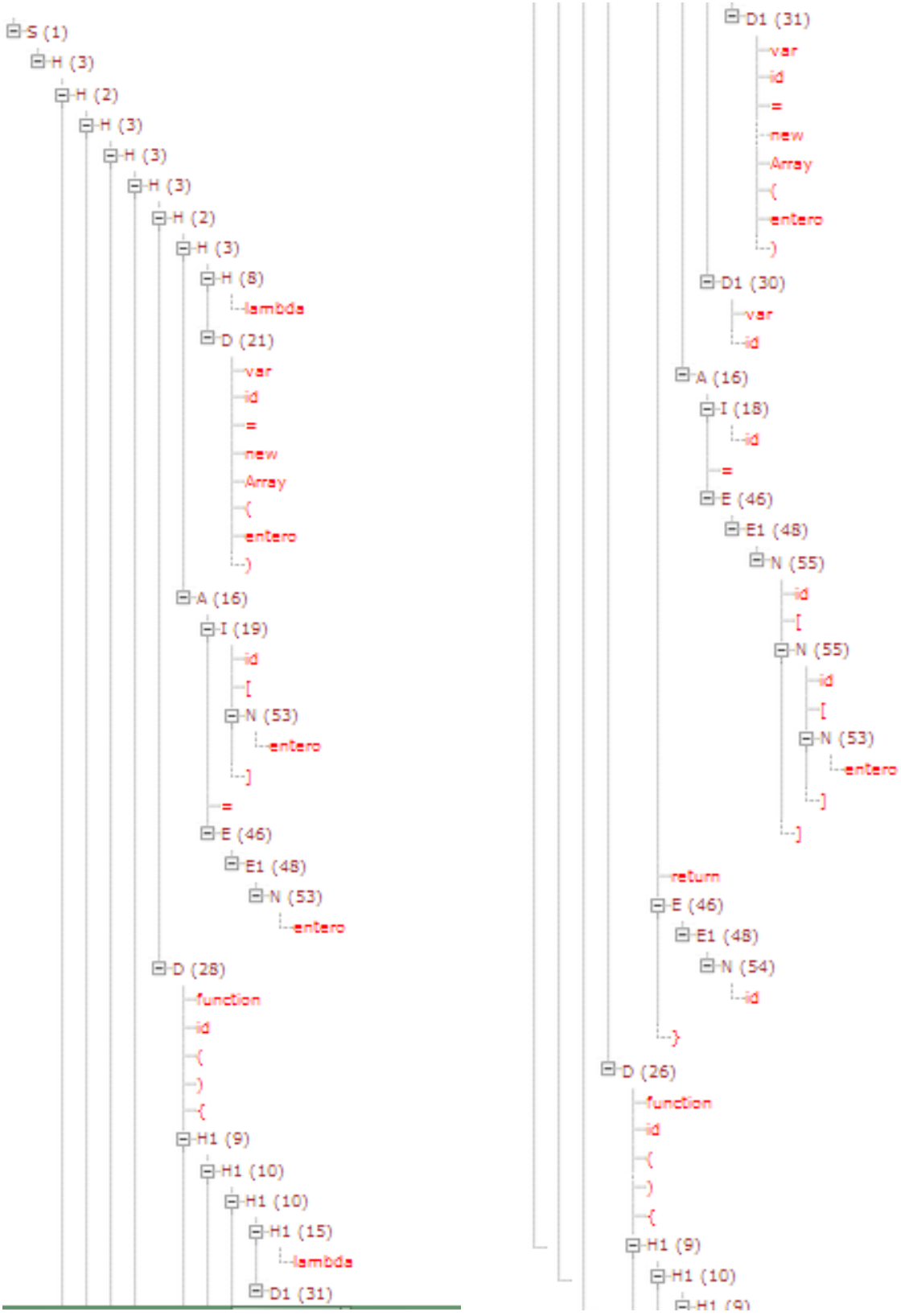
##
Tabla de símbolos principal
##

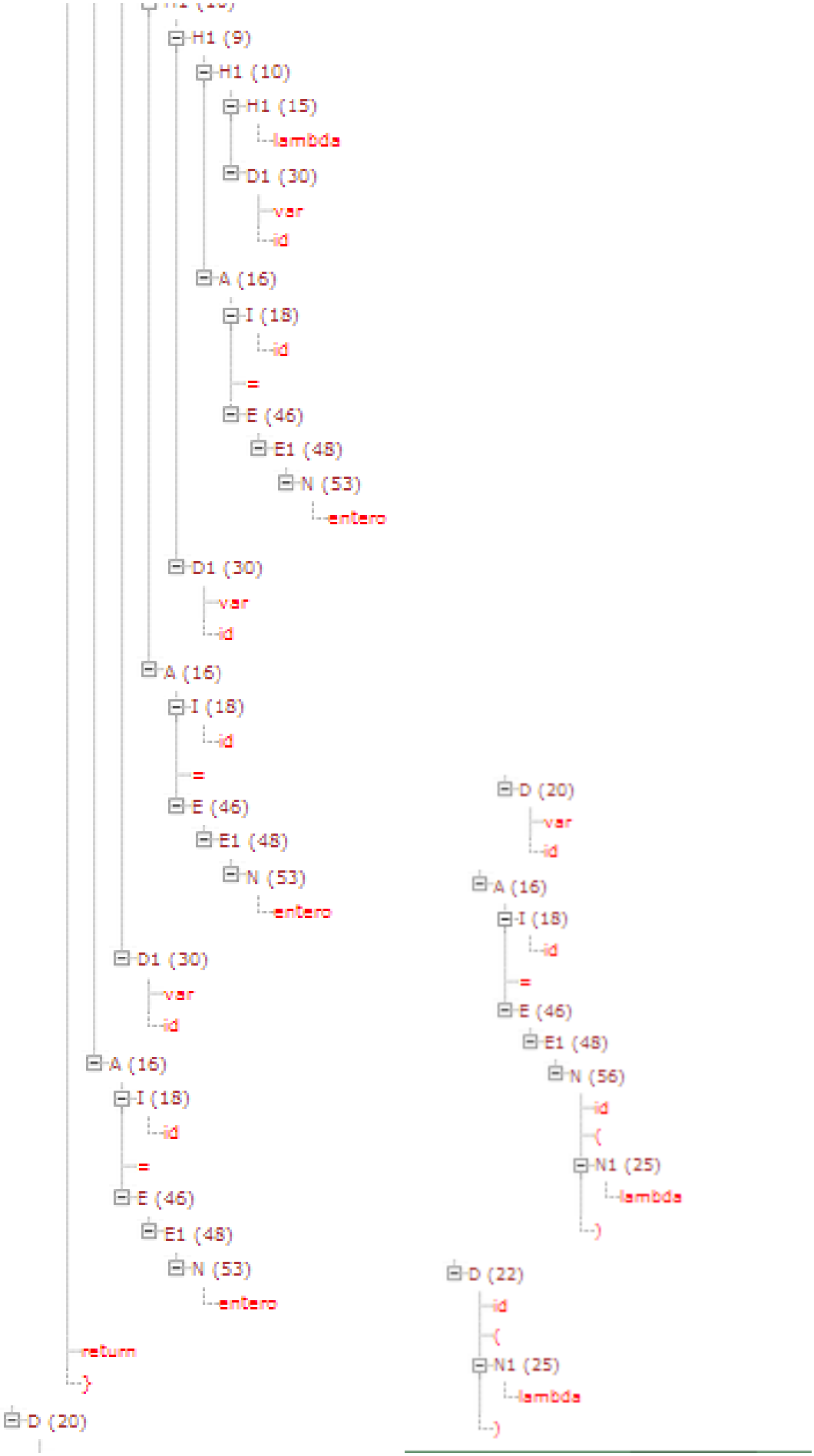
| T | array | 7 | None |

| a | function | None | entero |

| b | function | None | None |

| c | entero | None | None |


S (1)
H (3)
H (2)
H (3)
H (3)
H (3)
H (2)
H (3)
H (8)
lambda
D (21)
var
id
=
new
Array
(
entero
)
A (16)
I (19)
id
[
N (53)
entero
]
=
E (46)
E1 (48)
N (53)
entero
D (28)
function
id
(
)
{
H1 (9)
H1 (10)
H1 (10)
H1 (15)
lambda
D1 (31)
var
id
=
new
Array
(
entero
)
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (55)
id
[
N (55)
id
[
N (53)
entero
]
]
return
E (46)
E1 (48)
N (54)
id
}
D (26)
function
id
(
)
{
H1 (9)
H1 (10)
H1 (9)


H1 (9)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
return
}
D (20)
D (20)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (56)
id
(
N1 (25)
lambda
)
D (22)
id
(
N1 (25)
lambda
)

PruebaC2.js

CÓDIGO

```
var T=new Array(4)
T[5]=7
function a(b)
{
return 3
}
function c(s)
{
return
}
c(a(T[5] + 1))
```

PARSE

Ascendente 8 15 30 10 30 10 18 53 48 46 16 9 31 10 18 53 55 48 46 16 9 18 54 49 48 46 16 9 54 48 46 29 3 15 30 10 30 10 18 54 49 48 46 16 9 30 10 18 54 24 56 48 46 16 9 54 48 46 29 3 20 3 20 3 18 53 24 56 24 56 48 46 16 2 1

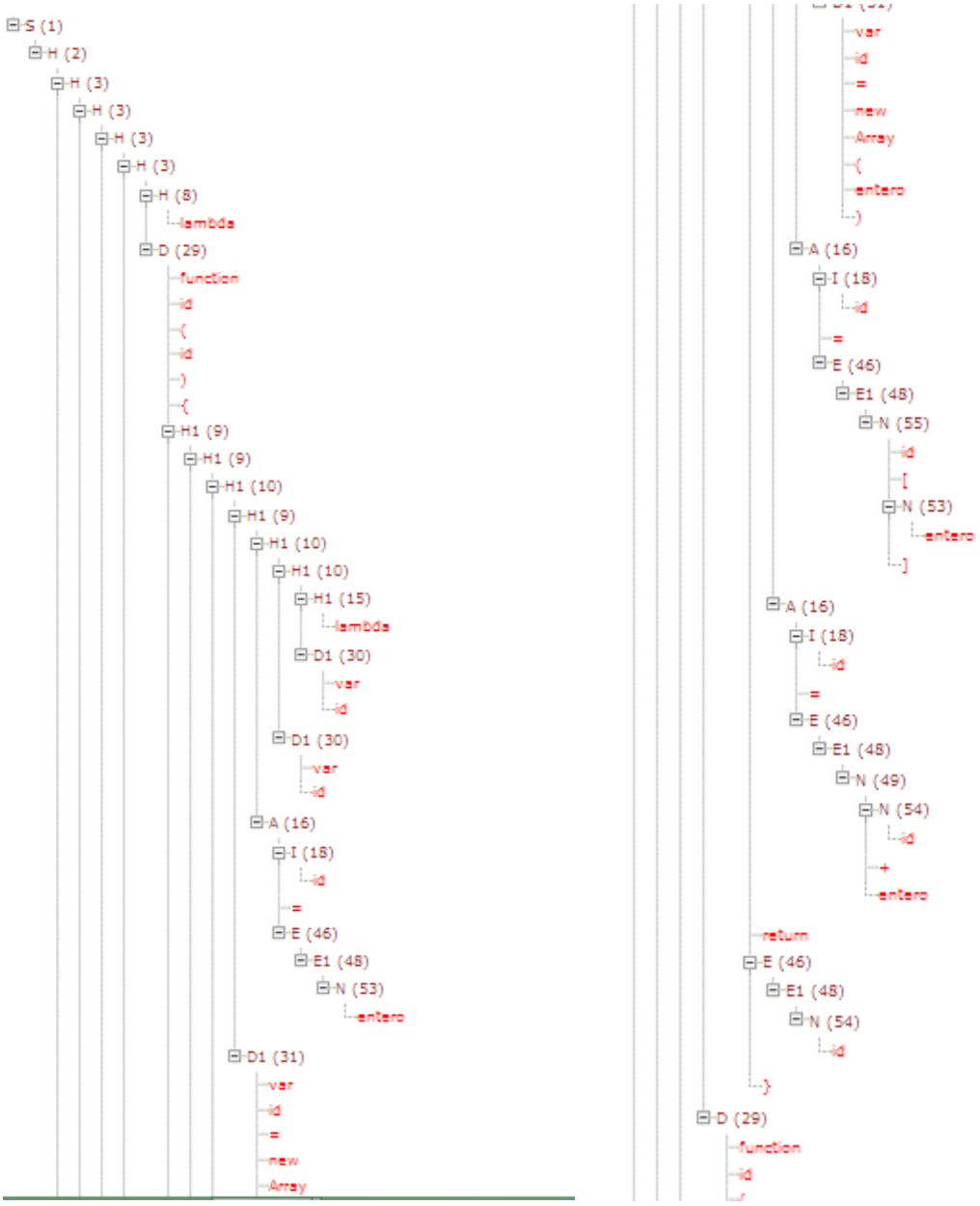
TABLAS DE SÍMBOLOS

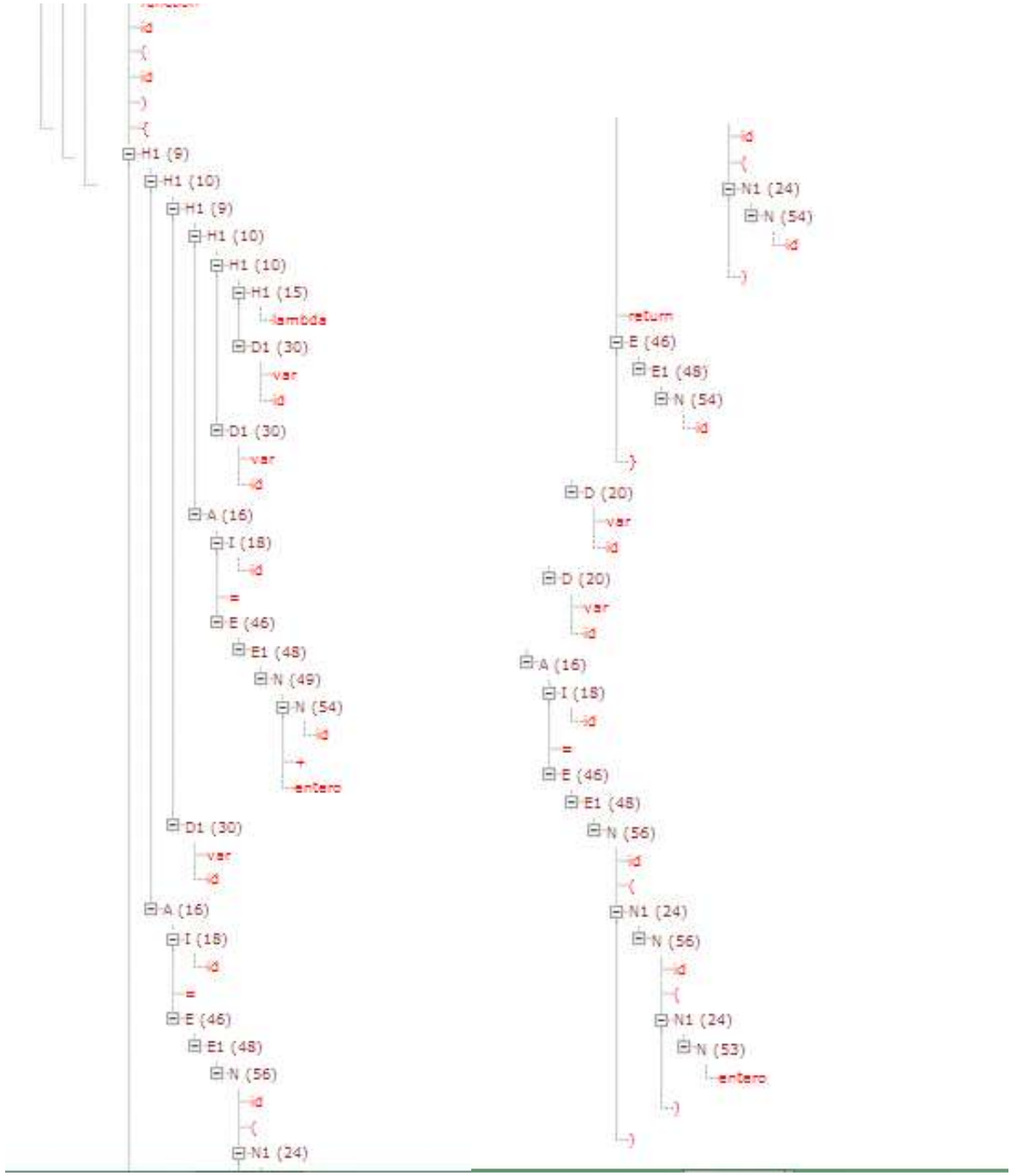
```
#######################################################
Tabla de símbolos secundaria (borrada): funcion1
#######################################################
-------------------------------------------------
| a | entero | None | None |
-------------------------------------------------
| c | entero | None | None |
-------------------------------------------------
| b | entero | None | None |
-------------------------------------------------
| A | array | 8 | None |
-------------------------------------------------


#######################################################
Tabla de símbolos secundaria (borrada): funcion2
#######################################################
-------------------------------------------------
| a | entero | None | None |
```

```
-------------------------------------------------
| b | entero | None | None |
-------------------------------------------------
| c | entero | None | None |
-------------------------------------------------
| t | entero | None | None |
-------------------------------------------------------


############################################################
Tabla de símbolos principal
############################################################
-------------------------------------------------
| funcion1 | function | entero | entero |
-------------------------------------------------
| funcion2 | function | entero | entero |
-------------------------------------------------
| b | entero | None | None |
-------------------------------------------------
| c | entero | None | None |
-------------------------------------------------------
```


S (1)
H (2)
H (3)
H (3)
H (3)
H (3)
H (8)
lambda
D (29)
function
id
(
id
)
{
H1 (9)
H1 (9)
H1 (10)
H1 (9)
H1 (10)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
D1 (31)
var
id
=
new
Array
(
entero
)
A (16)
I (18)
id
=
E (46)
E1 (48)
N (55)
id
[
N (53)
entero
]
A (16)
I (18)
id
=
E (46)
E1 (48)
N (49)
N (54)
id
+
entero
return
E (46)
E1 (48)
N (54)
id
}
D (29)
function
id


id
(
id
)
{
H1 (9)
H1 (10)
H1 (9)
H1 (10)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (49)
N (54)
id
+
entero
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (56)
id
(
N1 (24)
id
(
N1 (24)
N (54)
id
)
return
E (46)
E1 (48)
N (54)
id
)
D (20)
var
id
D (20)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (56)
id
(
N1 (24)
N (56)
id
(
N1 (24)
N (53)
entero
)
)

PruebaC3.js

CÓDIGO

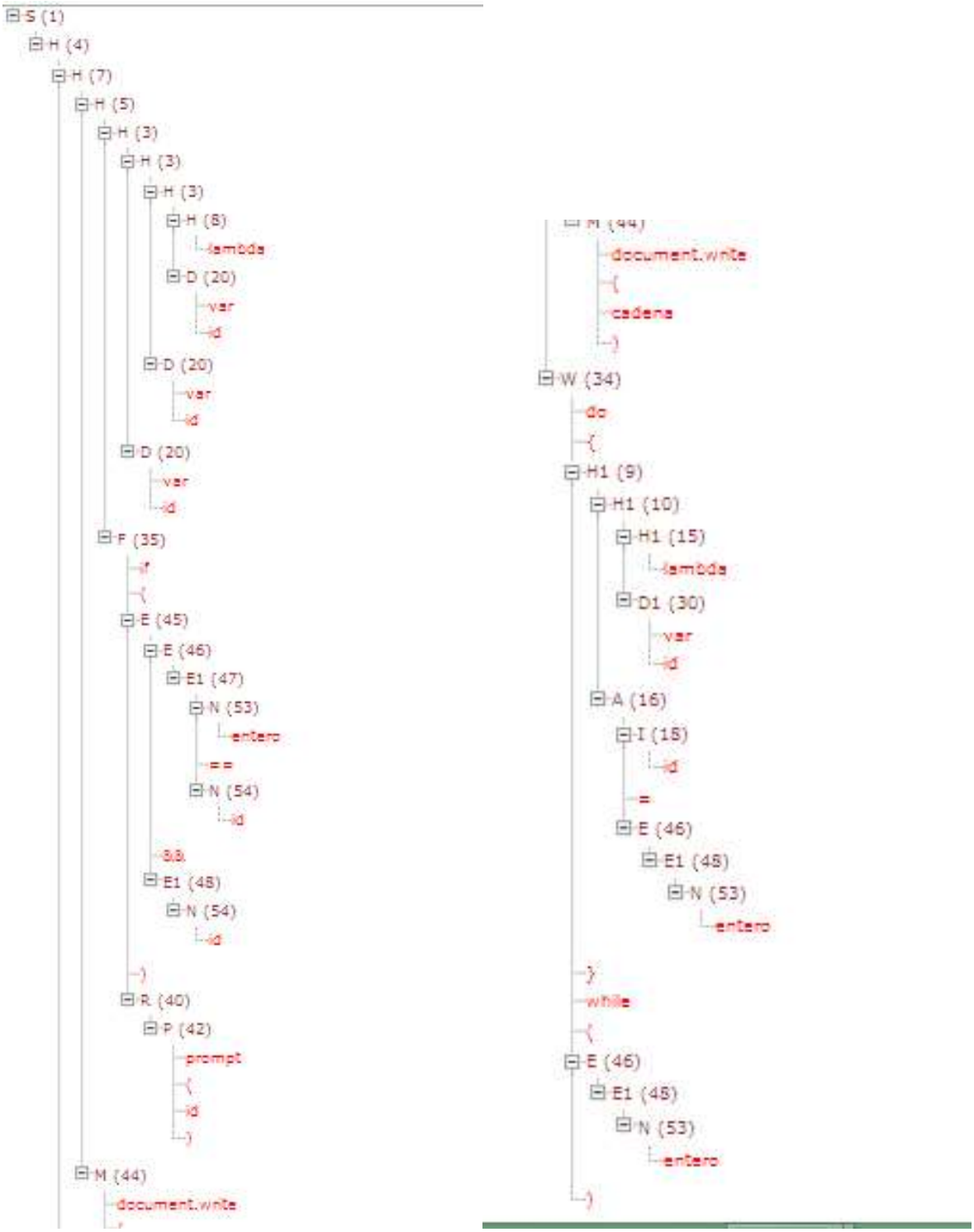
```
var b
var c
var hello
if(22==b && c)
prompt(hello)
document.write("Mi casa es verde")
do {
var perro
perro = 1
}
while (0)
```

PARSE

Ascendente 8 20 3 20 3 20 3 53 54 47 46 54 48 45 42 40 35 5 44 7 15 30 10 18 53 48 46 16 9 53 48 46 34 4 1

TABLAS DE SÍMBOLOS

```
##########################################################
Tabla de símbolos principal
##########################################################
-------------------------------------------------
| b | entero | None | None |
-------------------------------------------------
| c | entero | None | None |
-------------------------------------------------
| hello | entero | None | None |
-------------------------------------------------
| perro | entero | None | None |
-------------------------------------------------
```


S (1)
H (4)
H (7)
H (5)
H (3)
H (3)
H (3)
H (8)
lambda
D (20)
var
id
D (20)
var
id
D (20)
var
id
F (35)
if
(
E (45)
E (46)
E1 (47)
N (53)
entero
==
N (54)
id
&&
E1 (48)
N (54)
id
)
R (40)
P (42)
prompt
(
id
)
M (44)
document.write
(
cadena
)
W (34)
do
{
H1 (9)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
}
while
(
E (46)
E1 (48)
N (53)
entero
)

PruebaC4.js

CÓDIGO

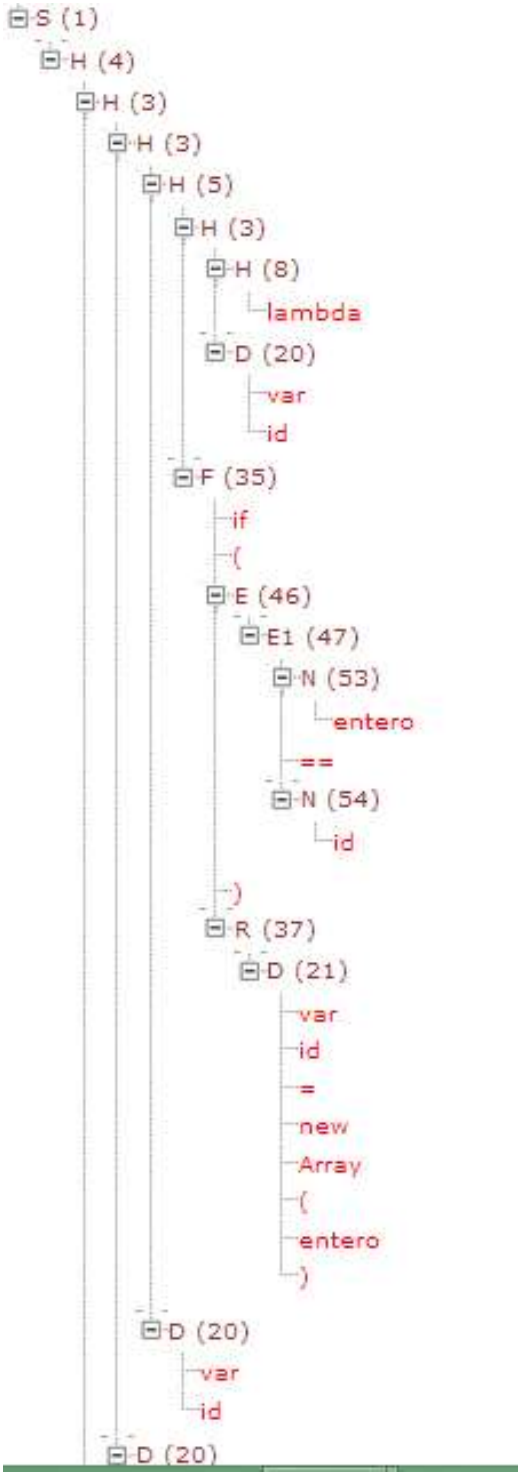
```
var b
if(4==b) var a = new Array (5)
var casa
var dedo
do {
prompt(casa)
document.write("caminata del desierto")
prompt(dedo)
document.write(casa + dedo)
casa = a[dedo]
dedo = a[3]
if (1==1) a[0]=++casa
}
while (4==dedo && 0)
```

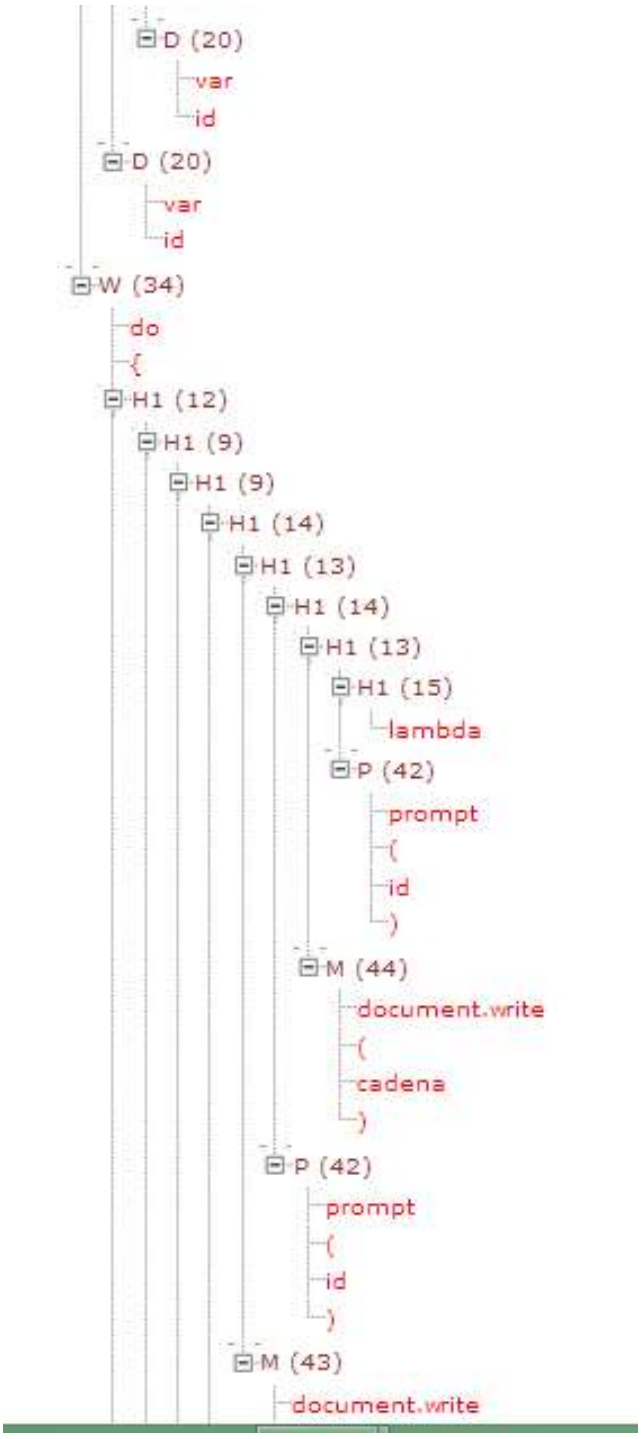
PARSE

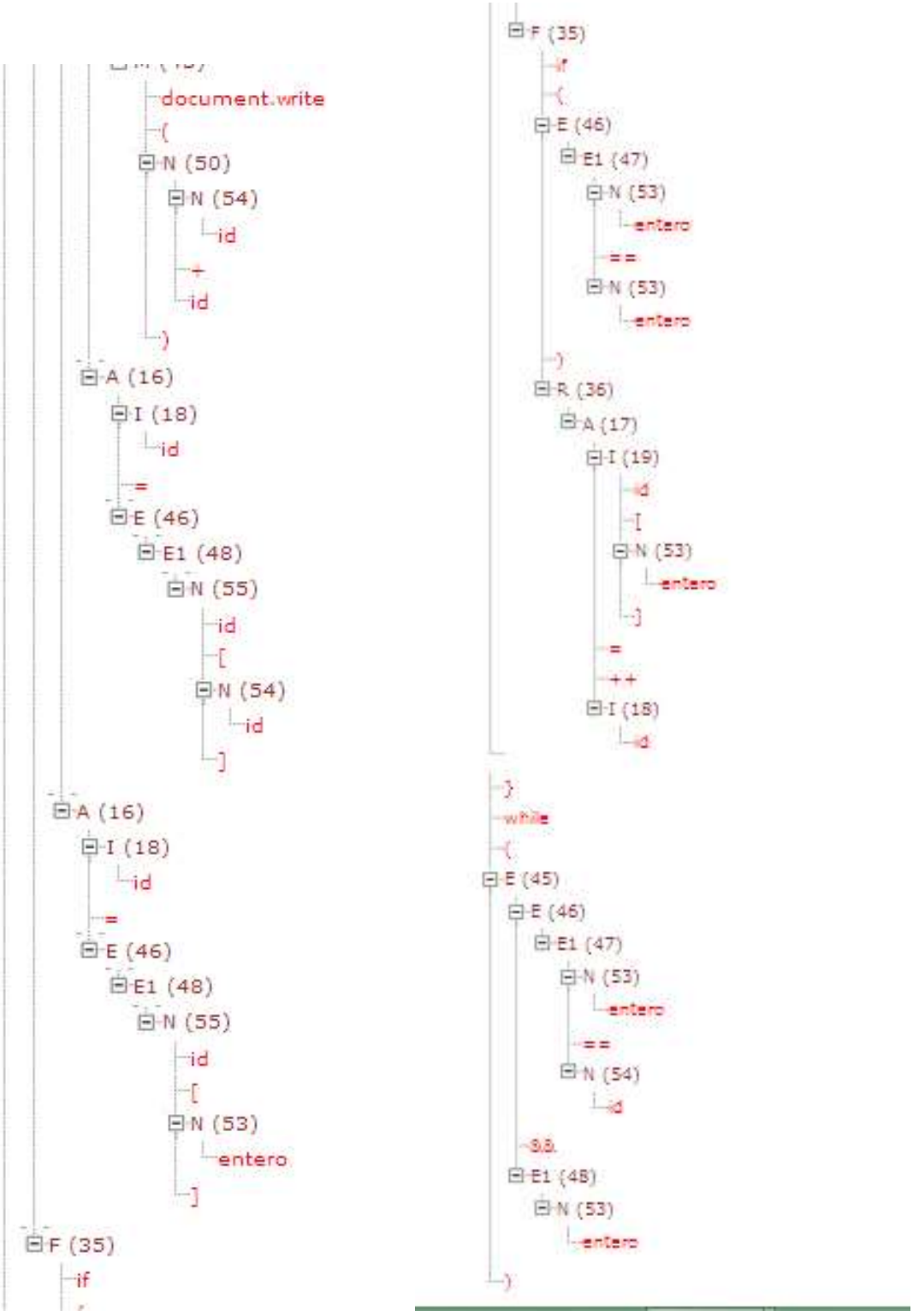
Ascendente 8 20 3 53 54 47 46 21 37 35 5 20 3 20 3 15 42 13 44 14 42 13 54 50 43 14 18 54 55 48 46 16 9 18 53 55 48 46 16 9 53 53 47 46 53 19 18 17 36 35 12 53 54 47 46 53 48 45 34 4 1

TABLAS DE SÍMBOLOS

```
##########################################################
Tabla de símbolos principal
##########################################################
-----------------------------------------------------
| b | entero | None | None |
-----------------------------------------------------
| a | array | 5 | None |
-----------------------------------------------------
| casa | entero | None | None |
-----------------------------------------------------
| dedo | entero | None | None |
-----------------------------------------------------
```


S (1)
H (4)
H (3)
H (3)
H (5)
H (3)
H (8)
lambda
D (20)
var
id
F (35)
if
(
E (46)
E1 (47)
N (53)
entero
==
N (54)
id
)
R (37)
D (21)
var
id
=
new
Array
(
entero
)
D (20)
var
id
D (20)


D (20)
var
id
D (20)
var
id
W (34)
do
{
H1 (12)
H1 (9)
H1 (9)
H1 (14)
H1 (13)
H1 (14)
H1 (13)
H1 (15)
lambda
P (42)
prompt
(
id
)
M (44)
document.write
(
cadena
)
P (42)
prompt
(
id
)
M (43)
document.write


document.write
(
N (50)
N (54)
id
+
id
)
A (16)
I (18)
id
=
E (46)
E1 (48)
N (55)
id
[
N (54)
id
]
A (16)
I (18)
id
=
E (46)
E1 (48)
N (55)
id
[
N (53)
entero
]
F (35)
if
F (35)
if
(
E (46)
E1 (47)
N (53)
entero
==
N (53)
entero
)
R (36)
A (17)
I (19)
id
[
N (53)
entero
]
=
++
I (18)
id
}
while
(
E (45)
E (46)
E1 (47)
N (53)
entero
==
N (54)
id
&&
E1 (48)
N (53)
entero
)

PruebaC5.js

CÓDIGO

```
function a (b){
var a
a=2 + 2
return a
}
do{
var b
document.write(b)
b=-1
var c
prompt(c)
}
while(3 + c ==6 && c)

function void (){ return }
void()
void()
void()
b=a(c)
```

PARSE

Ascendente 8 15 30 10 18 53 49 48 46 16 9 54 48 46 29 3 15 30 10 54 43 14 18 53 48 46 16 9 30 10 42 13 53 50 53 47 46 54 48 45 34 4 15 26 3 25 22 3 25 22 3 25 22 3 18 54 24 56 48 46 16 2 1

TABLAS DE SÍMBOLOS

```
##########################################################
Tabla de símbolos secundaria (borrada): a
##########################################################
-----------------------------------------------------
| b | entero | None | None |
-----------------------------------------------------
| a | entero | None | None |
-----------------------------------------------------
```

##
Tabla de símbolos secundaria (borrada): c
##

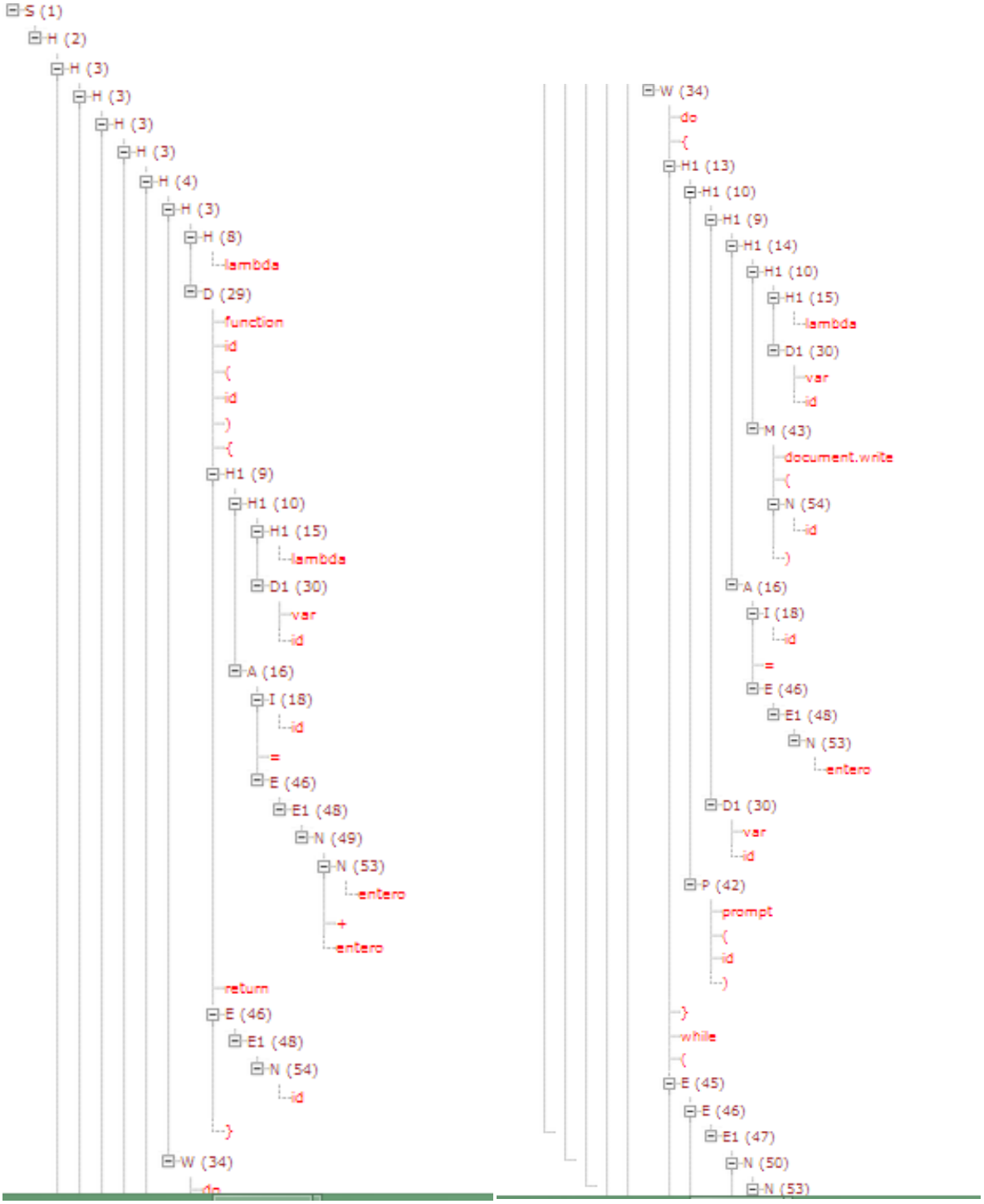
##
Tabla de símbolos principal
##

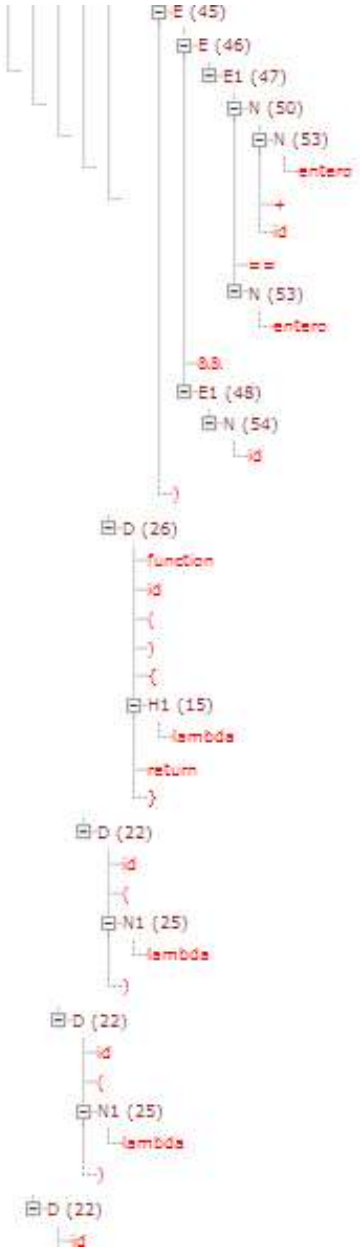
| a | function | entero | entero |

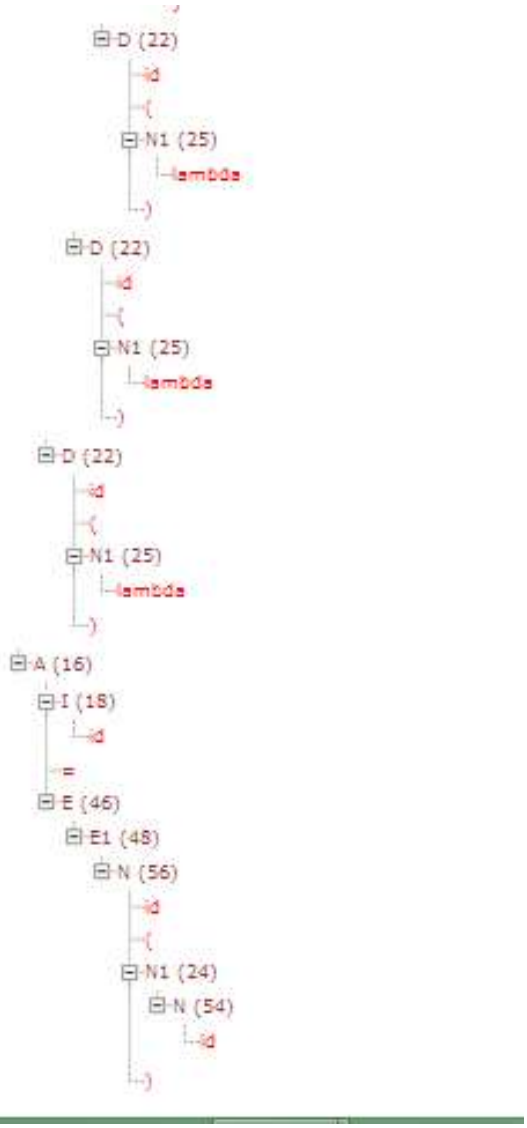
| b | entero | None | None |

| c | entero | None | None |

| void | function | None | None |


S (1)
H (2)
H (3)
H (3)
H (3)
H (3)
H (4)
H (3)
H (8)
lambda
D (29)
function
id
(
id
)
{
H1 (9)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
A (16)
I (18)
id
=
E (46)
E1 (48)
N (49)
N (53)
entero
+
entero
return
E (46)
E1 (48)
N (54)
id
}
W (34)
do
{
H1 (13)
H1 (10)
H1 (9)
H1 (14)
H1 (10)
H1 (15)
lambda
D1 (30)
var
id
M (43)
document.write
(
N (54)
id
)
A (16)
I (18)
id
=
E (46)
E1 (48)
N (53)
entero
D1 (30)
var
id
P (42)
prompt
(
id
)
}
while
(
E (45)
E (46)
E1 (47)
N (50)
N (53)


E (45)
E (46)
E1 (47)
N (50)
N (53)
entero
+
id
==
N (53)
entero
&&
E1 (48)
N (54)
id
)
D (26)
function
id
(
)
{
H1 (15)
lambda
return
}
D (22)
id
(
N1 (25)
lambda
)
D (22)
id
(
N1 (25)
lambda
)
D (22)
id


D (22)
id
(
N1 (25)
lambda
)
D (22)
id
(
N1 (25)
lambda
)
D (22)
id
(
N1 (25)
lambda
)
A (16)
I (18)
id
=
E (46)
E1 (48)
N (56)
id
(
N1 (24)
N (54)
id
)

Pruebas erróneas:

Las pruebas que tienen errores semánticos también devuelven el parse del sintáctico y la tabla de símbolos hasta el momento en que surgió el error.

PruebaE1.js

CÓDIGO

```
var T=new Array(4)
var b
T[5]=7
function a(b)
{
return
}
function c(s)
{
return a(1)

}

b= 6 + a(1 + 2)
```

ERRORES

ERROR EN EL SEMANTICO: La función a no devuelve nada

PARSE

Ascendente 8 21 3 20 3 53 19 53 48 46 16 2 15 27 3 15 53 24 56 48 46 29 3 18 53 53 49 24 52 48 46 16 2 1

TABLAS DE SÍMBOLOS

##
Tabla de símbolos secundaria (borrada): b
##

| b | entero | None | None |

##
Tabla de símbolos secundaria (borrada): c
##

| s | entero | None | None |

Tablas de símbolos cuando se ha producido el error:

##
Tabla de símbolos principal
##

| T | array | 4 | None |

| b | entero | None | None |

| a | function | entero | None |

| c | function | entero | entero |

PruebaE2.js

CÓDIGO

```
function funcion1( a ){
var c
var b
b=8
var A=new Array(8)
c=A(5)
c=c + 2
return c
}
function funcion2(a){
var b
var c
c=b + 1
var t
t=funcion1(c)
return t
}
funcion1(funcion2(10))
```

ERRORES

ERROR EN EL SEMÁNTICO:La función: A no ha sido declarada

PARSE

Ascendente 8 15 30 10 30 10 18 53 48 46 16 9 31 10 18 53 24 56 48 46 16 9 18 54 49 48 46 16 9 54 48 46 29 3 15 30 10 30 10 18 54 49 48 46 16 9 30 10 18 54 24 56 48 46 16 9 54 48 46 29 3 53 24 56 24 22 3 1

TABLAS DE SÍMBOLOS

Tablas de símbolos cuando se ha producido el error:

##
Tabla de símbolos secundaria (no borrada): funcion1
##

| a | entero | None | None |

| c | entero | None | None |

| b | entero | None | None |

| A | array | 8 | None |

##
Tabla de símbolos principal
##

| funcion1 | function | entero | None |
---

pruebaE3.js

CÓDIGO

```
var valida
var b
b=12
function prueba (a) {
var valida
b=10
document.write("hola")
}
```

ERRORES

ERROR EN EL SINTACTICO: La gramática no reconoce el texto porque en la tabla de acción la fila 348 y la columna 5 están vacíos

(esto se debe a que la función prueba no tiene return)

PruebaE4.js

CÓDIGO

```
var valida.casa
function prueba (a) {
var valida
return true
}
```

ERRORES

ERROR EN EL LÉXICO:El carácter . no está permitido en identificadores.

PruebaE5.js

CÓDIGO

```
var num
num = 4
var b = new Array(num)
b[2]=4
do{
a=++b
}
while (a == -32)
```

ERRORES

ERROR EN EL SINTACTICO: La gramática no reconoce el texto porque en la tabla de acción la fila 152 y la columna 1 están vacíos
(esto se debe a que al declarar un array su dimensión no pude ser una variable, sino una constante entera)