PDL
Memoria

Alfonso Ventosa	x15m027
Fernando Magdalena	x15m018
Jorge Vázquez	y16m023

ÍNDICE

Introducción

El objetivo de este proyecto es ser capaces de implementar en JAVA/Python un compilador con todas sus fases para un lenguaje muy similar a JavaScript propuesto por el profesorado; a partir de ahora referido como JS-PDL.

Analizador Léxico

En esta sección describiremos el funcionamiento de la primera fase del compilador: el analizador léxico, que se encarga de leer caracter por caracter el código de entrada y reconoce los distintos componentes significativos del lenguaje (tokens), como pueden ser nombres de variables, palabras reservadas, expresiones, operadores, etc.

Tokens

Podemos dividir el código en sus componentes individuales con significado propio mediante los siguientes tipos de tokens:

<palabra_reservada, ->
<s_puntuacion,->
<operador, ->
<cte_bool, valor>
<cte_int, valor>
<cte_cadena, cadena>
<id, lexema>

Donde palabra_reservada∈{var, int, string, bool, function, return, print, prompt, false, true, for, if}, s_puntuacion∈{;, ,, (,), {, }, $}, y operador∈{+, -, *, /, %, !, ++, --, <, >, <=, >=, |=, &&, ||, }.

Con todos los símbolos del lenguaje enumerados e identificados por un String con el identificador del token como valor, tenemos todas las herramientas para poder diseñar la gramática formal que genera este lenguaje.

Gramática Formal

Una gramática formal está definida por:

- Un axioma: S
- Un conjunto de símbolos no terminales: {A, B, C, D, E, F, G, H, K, N, M, O, P, Q}
- Un conjunto de símbolos terminales: {+, -, *, /, %, &, |, =, !, <, >, {, }, (,), ;, ’,’ , L, d, del, cr, ” }
- Un conjunto de producciones P, tal que

  P={

  S -> * | /K | % | + | - | &A | ‘|’B | =C | !D | <E | >F | { | } | (|) | ; | , | LG | dH | del S | cr S | “P

  A -> &

  B -> ‘|’ | =

  C -> = | λ

  D -> = | λ

  E -> = | λ

  F -> = | λ

  G -> c_1 G | λ

  H -> dH | λ

  K-> λ | *N | /O

  N -> c_6N | *M

  M -> / | c_7N

  O -> c_2 O | c_3

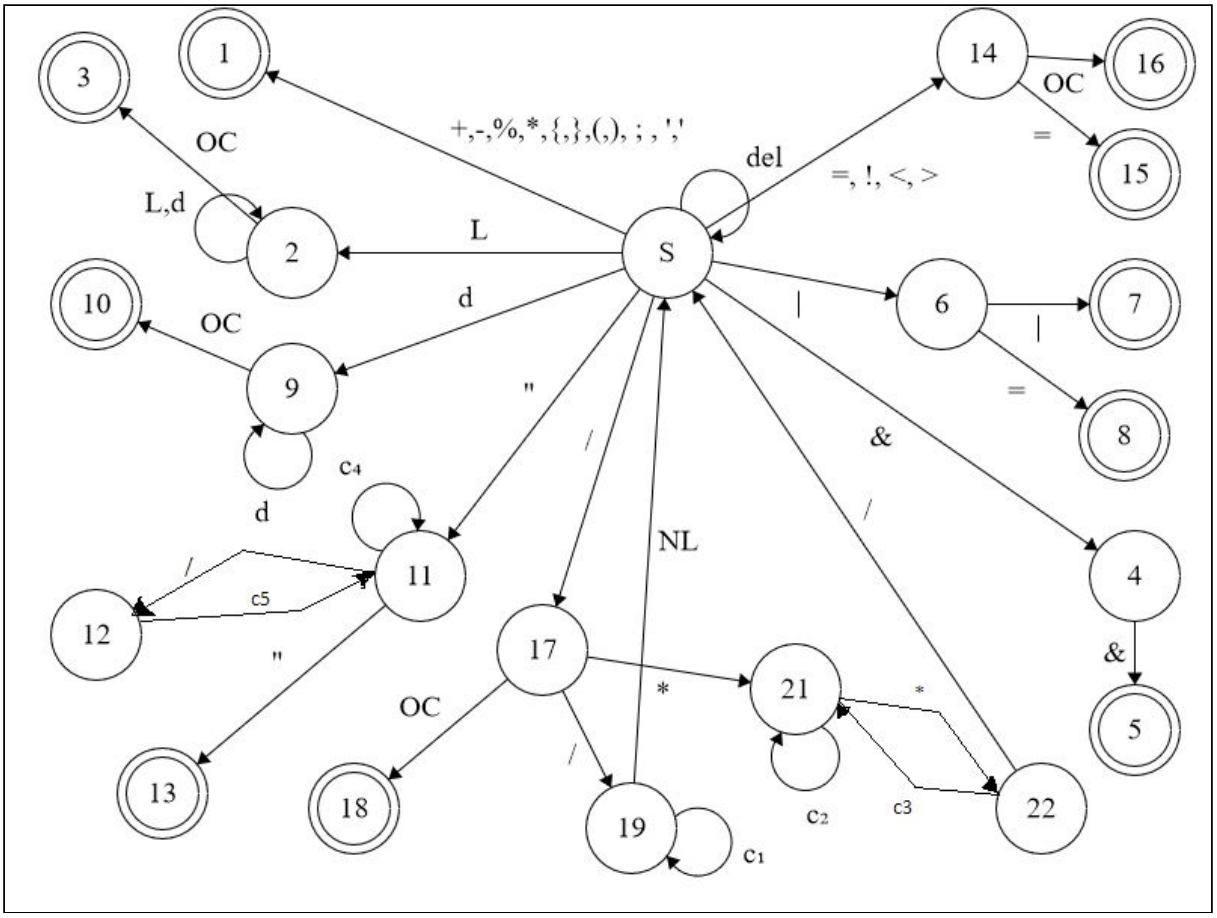
  P -> “ | c_4P | \Q

  Q -> c_5P

  }

En nuestro caso, se puede apreciar cómo se generan los operadores propios del lenguaje de manera trivial (símbolos no terminales A hasta F y terminales que resultan directamente del axioma). Las palabras reservadas se generan a partir de una letra desde el axioma, y después generando más letras gracias al símbolo no terminal *G*. En el caso de los valores enteros, se parte del axioma con un dígito, y se genera con el símbolo no terminal H. Por último, en el caso de los comentarios, partimos del axioma con una forward-slash (“/”), y vamos al símbolo no terminal K. Ahí podemos terminar (símbolo de división), añadir otra barra para un comentario en línea (e ir al símbolo no terminal O), o añadir un asterisco para comentario en bloque, yendo al símbolo N. En el caso de un comentario en línea, podemos seguir leyendo todos los caracteres que queramos excepto el salto de línea (conjunto de caracteres c_2), o podemos leer un salto de línea y terminar. En el caso de un comentario en bloque, podemos leer cualquier carácter para continuar en N, o un asterisco para ir a

Autómata Finito Determinista



Comentarios:

- Debido al software utilizado para dibujar el grafo del autómata, no hemos podido hacer dos vectores diferentes en los casos en los que dos estados están unidos en ambas direcciones (por ejemplo el 11 y el 12). Por ello, clarificamos los caracteres que corresponden a cada vector.
- Del estado 11 al 12 pasamos leyendo el carácter / .
- Del estado 12 al 11 pasamos leyendo un carácter que pertenezca al conjunto de caracteres c_5.
- Del estado 21 al 22 pasamos leyendo el carácter *.
- Del estado 22 al 21 pasamos leyendo cualquier carácter perteneciente al conjunto c_3.

Simbología:

- c_1 representa el conjunto de caracteres que contiene a todos menos el salto de línea (NL en adelante) y el final de archivo (EOF).
- c_2 representa el conjunto de caracteres que contiene a todos menos EOF y *.
- c_3 representa el conjunto de caracteres que contiene a todos menos EOF y /.

- c_4 representa el conjunto de caracteres que contiene a todos menos NL,EOF, / y “.
- c_5 representa el conjunto de caracteres que contiene a todos menos EOF y NL.

Acciones Semánticas del Analizador Léxico

El autómata implementado desarrollará las siguientes funciones semánticas, teniendo en cuenta el estado en el que se encuentra, el carácter que está leyendo y los que ha leído previamente.

En todas las transiciones de estados marcadas con O.C. (otro carácter diferente al que requieren el resto de transiciones que parten de dicho estado) no se lee un carácter, sino que se vuelve al estado S y entonces se consume el carácter.

El resto de acciones semánticas se resumen como sigue:

G_{0-1}:= Gen_Token(TipoOperador, ÍndiceEnTabla).
G_{0-4}:= Crea variable Token, añade & a la variable.
G_{4-5}:= Concatena lo que está leyendo, el carácter &, a la variable Token; y Gen_Token(Op_Logico,0).
G_{0-6}:= Crea variable Token, añade | a la variable.
G_{6-7}:= Concatena lo que está leyendo a la variable Token. Gen_Token(Op_Logico,1).
G_{6-8}:= Concatena lo que está leyendo a la variable Token. Gen_Token(Op_Aritmetico,8).
G_{0-14}:= Crea variable Token, añade = || < || > || ! a la variable.
G_{14-15}:= Concatena lo que está leyendo a la variable Token. Genera el token correspondiente, de entre los correspondientes a los operadores ==, <=, >= o !=.
G_{14-16}:=Concatena lo que está leyendo a la variable Token. Gen_Token(TipoOperador, IndiceEnTabla)
G_{0-2}:= Crea variable Token, añade la letra leída a la variable.
G_{2-2}:= Añade lo que está leyendo a la variable token, siempre que sea una letra o un número.
G_{2-3}:= Para otro carácter, Gen_Token(lexema), comparando si es una palabra reservada, y si no lo es lo añade a la tabla de símbolos global, que contiene los nombres de variables, funciones, constantes, etc.
G_{0-9}:= Crea variable Token, añade el dígito leído a la variable.
G_{9-9}:= Token = (Token*10)+dígito_leído, siempre que el carácter leído sea un dígito.
G_{9-10}:= Cuando lee otro carácter Gen_Token(entero, valor), si está en el rango de int.

$G_{0\text{-}17}$:= Crea la variable Token, y añade el caracter leído (/) a la variable.
$G_{17\text{-}18}$:= Si el siguiente carácter no es / o *, Gen_Token(Op_Aritmetico,3).
$G_{17\text{-}19}$:= Sabemos que es comentario en línea. Variable Token carece ya de valor.
$G_{19\text{-}19}$:= Leemos carácter, siempre que sea distinto del salto de línea.
$G_{19\text{-}20}$:= Con salto de línea volvemos al estado inicial.
$G_{17\text{-}21}$:= Sabemos que es comentario en bloque. Variable Token carece ya de valor.
$G_{22\text{-}23}$:= Comentario terminado, volvemos al estado inicial.
$G_{0\text{-}11}$:= Crea variable Token, que va a ser una cadena de texto.
$G_{11\text{-}11}$:= Si caracter leido diferente de " o \, leer caracter y concatenar a variable Token.
$G_{11\text{-}12}$:= Leer caracter, pero sabemos que aunque el siguiente caracter sea ", la cadena de texto no acaba.
$G_{12\text{-}11}$:= Leer caracter y concatenar a la variable Token.
$G_{11\text{-}13}$:= Gen_Token(STRING, Token).

Tratamiento de Errores en el Analizador Léxico

En este apartado especificaremos qué errores léxicos se pueden generar y cómo los tratamos. Cabe mencionar que en todos los estados del autómata, si se proporciona un carácter que no tiene una transición válida, se generaría el error CaracterNoEsperado. En cualquier caso, todos los errores quedan recogidos en la siguiente tabla:

Error	Descripción	Estado/Transición en la que se presenta
CaracterNoEsperado	Caracter leido no tiene una transición válida en el autómata.	
CadenaIncompleta	Una cadena de texto no ha sido cerrada con el correspondiente ".	Estado 11, leyendo NL o EOF. Estado 12 leyendo EOF.
ComentarioIncompleto	Un comentario en bloque no ha sido cerrado.	Estado 21 y 22 leyendo EOF.

Analizador sintáctico

El tipo de analizador sintáctico asignado a nuestro grupo era el Descendente por Tablas, o LL(1). Para ello, teníamos que generar una gramática de tipo LL(1) que no tuviera casos conflicto.

La gramática utilizada es la siguiente:

```
NoTerminales = { J D T I E G GG EE X XX F H A AA C S R B SS L AR RR M RRR }
Terminales = { var id int string $ = ( ) ! bool + ++ - -- < > >= <= == { } function return ; , * |= / % && || print
prompt false true for char cte_int cte_cadena cte_logica }
Axioma = J
Producciones = {
J -> D J
J -> F J
J -> S J
J -> $
D -> var T id I ;
T -> int
T -> string
T -> bool
I -> = E
I -> |= E
I -> lambda
E -> G EE
G -> ( X )
G -> id GG
G -> ! B
G -> cte_int
G -> cte_cadena
G -> cte_logica
GG -> lambda
GG -> --
GG -> ++
EE -> + G EE
EE -> - G EE
EE -> lambda
EE -> * G EE
EE -> / G EE
EE -> % G EE
X -> E XX
XX -> == E
EE -> || G EE
XX -> < E
XX -> > E
XX -> >= E
XX -> <= E
F -> function H id ( A ) { C }
H -> T
H -> lambda
A -> T id AA
A -> lambda
AA -> , T id AA
AA -> lambda
C -> D C
C -> S C
C -> lambda
S -> id M ;
S -> return R ;
S -> print ( X ) ;
S -> prompt ( id ) ;
S -> for ( D ; X ; SS ) { C }
R -> X
```

```
R -> lambda
B -> ( X )
B -> cte_logica
B -> id
SS -> id M
SS -> print ( X )
XX -> lambda
M -> ++
M -> --
M -> I
GG -> L
EE -> && G EE
S -> L ;
L -> ( AR )
AR -> E RR
AR -> lambda
RR -> , RRR
RR -> lambda
M -> L
RRR -> E RR
}
```

Cabe hacer mención del uso de algunos de los símbolos no terminales de la gramática.

- El programa principal parte del axioma J; y se va componiendo recursivamente con las producciones #1, #2 y #3 en declaraciones, funciones y sentencias respectivamente. Así mismo, utilizamos la producción #4 para generar el token fin de fichero.
- Una declaración (no terminal D) se compone de una reserva de memoria para la variable (var T id), que puede tener un tipo (no terminal T) o no; y una inicialización que es opcional mediante los operadores = ó |= (no terminal I).
- Utilizamos el no terminal E para referirnos a una expresión, que puede estar formada por variables (no terminal G); con o sin postdecremento/incremento (no terminal GG), un literal o una expresión entre paréntesis (producción G -> (X)), y a continuación la posibilidad de una serie de operaciones aritmético lógicas (no terminal EE).
- Utilizamos el no terminal F para generar una función, que puede tener o no tipo de retorno, argumentos (no terminal A para el primero y AA para sucesivos argumentos), y un cuerpo de función (no terminal C).
- Un cuerpo de una función está compuesto de declaraciones o sentencias.
- Las sentencias (no literal S) pueden ser asignaciones, retornos de función, entrada y salida estandard o sentencias de control como un bucle for, en los que hay una sentencia simple de una sola línea (no terminal SS).
- Para retornar valores en las funciones utilizamos el no terminal R.
- Para referirnos a un valor booleano, utilizamos el no terminal B.
- Para una llamada a una función, utilizaremos el no terminal L después de un identificador, que a su vez generará los argumentos pasados a la función gracias a AR, RR, RRR.

Hemos utilizado la herramienta SDGLL1 disponible en la web del departamento para procesar esta gramática y obtener la tabla sintáctica para el autómata de LL1. Dicha herramienta incluye una comprobación de que la gramática es correcta para este tipo de analizador sintáctico, y por ende, nos permite comprobar que efectivamente esta gramática es propensa para este tipo de analizador. La tabla sintáctica resultante se puede ver con más detalle en el anexo.

La implementación del autómata ha sido realizada mediante la siguiente estructura de datos:

```
HashMap<String,HashMap<String,ArrayList<String>>>
```

Cada producción se corresponde con un ArrayList de String, que representa cada uno de los símbolos en el consecuente. Cada celda puede tener una sola producción para que cumpla las condiciones del LL1, por lo que con un HashMap asociamos cada columna con su producción. Por último, asociamos cada fila de la tabla al HashMap que la representa mediante el HashMap más exterior a la estructura de datos anteriormente mencionada.

Por lo tanto, para inicializar la primera fila de la tabla, se codificaría de la siguiente manera:

```
Map<String, Map<String, ArrayList<String>>> tablaTransicion = new HashMap<>();

ArrayList<String> produccionAcierrap = new ArrayList<>();
ArrayList<String> produccionAbool = new ArrayList<>();
Map<String, ArrayList<String>> filaA = new HashMap<>();
produccionAcierrap.add("38");
filaA.put(")", produccionAcierrap);
produccionAbool.add("37");
produccionAbool.add("T");
produccionAbool.add("id");
produccionAbool.add("AA");
filaA.put("bool", produccionAbool);
filaA.put("int", produccionAbool);
filaA.put("string", produccionAbool);
tablaTransicion.put("A", filaA);
```

Así mismo, también dispondremos de dos listas, con los símbolos terminales y no terminales, que posteriormente utilizaremos en el autómata.

Por último, la implementación del autómata consistiría en la función analizar, que recibe como parámetro la lista de strings representando los Tokens (símbolos terminales), y obtiene el parse de dicho código, a la vez que va generando un objeto con estructura de árbol, que posteriormente será utilizado en el analizador semántico.

El proceso de generación de este árbol consiste en, siempre que el autómata procese una producción, desarrollar en el árbol dicha producción creando los hijos correspondientes en el nodo actual. De igual manera, cuando se consume un símbolo terminal, asignamos el token correspondiente a ese terminal al nodo hijo del actual con ese nombre.

```
public boolean analizar(ArrayList<Token> listaTokens) {
                    ArrayList<Integer> arbol = new ArrayList<>();
                    ArrayList<String> codigo = convertirTokenaTerminales(listaTokens);
```

```
                pila.clear();
                pila.push("$");
                pila.push("J");
                int puntero = 0;

                Nodo nodoActual= new Nodo("ROOT",-1, null);
                nodoActual.addHijo("J",new Nodo("J",-1, nodoActual));
                Arbol asem = new Arbol(nodoActual);

                while (!pila.peek().equals("$")) {
                        String X = pila.peek();
                        String a = codigo.get(puntero);
                        if (terminales.contains(X)) {
                                if (X.equals(a)) {
                                        Nodo hijo;
                                        while(nodoActual.getHijo(X)==null) {
                                                nodoActual=nodoActual.getPadre();
                                        }
                                        hijo=nodoActual.getHijo(X);
                                        hijo.setToken(listaTokens.get(puntero));

                                        pila.pop();
                                        puntero++;
                                } else {
                                        ErrorHandler.error(2, 0, listaTokens.get(puntero).getLinea(),
"Recibido: "+a+ ". Se esperaba: "+X+".");
                                        return false;
                                }
                        } else if (noTerminales.contains(X)) {
                                pila.pop();

                                ArrayList<String> produccion = tablaTransicion.get(X).get(a);
                                if (produccion == null) {
                                        ErrorHandler.error(2, 0, listaTokens.get(puntero).getLinea(),
"Recibido: "+a+ ". Se esperaba: "+tablaTransicion.get(X).keySet().toString()+".");
                                        return false;
                                }

                                arbol.add(Integer.parseInt(produccion.get(0)) + 1);
                                for (int i = produccion.size() - 1; i > 0; i--) {
                                        pila.push(produccion.get(i));
                                }

                                Nodo hijo;
                                while(nodoActual.getHijo(X)==null) {
                                        nodoActual=nodoActual.getPadre();
                                }
                                hijo=nodoActual.getHijo(X);

                                hijo.setProdN(Integer.parseInt(produccion.get(0))+1);
                                for(int i=1;i<produccion.size();i++) {
                                        hijo.addHijo(produccion.get(i), new Nodo(produccion.get(i),
```

```
hijo));
                                        }
                                        if(produccion.size()>1) {
                                                nodoActual=hijo;
                                        }

                                } else {
                                        ErrorHandler.error(2, 1, listaTokens.get(puntero).getLinea(), "Recibido:
"+a+ ".");
                                        return false;
                                }
                        }

                        BufferedWriter fw = fileManager.getWriterParse();
                        try{
                                fw.write("Desc ");
                                for(Integer i : arbol) {
                                        fw.write(i.toString()+" ");
                                }
                                fw.flush();
                        }catch(IOException ex) {
                                ErrorHandler.error(0, 0, 0, "Error durante escritura del archivo parse.");
                        }

                        AnalizadorSemantico as = new AnalizadorSemantico(fileManager);
                        return as.analizar(asem);
                }
```

Hay que destacar la funcion convertirTokenaTerminales(ArrayList<Tokens> listaTokens), que convierte lo tokens obtenidos en el analizador léxico en los símbolos terminales de la gramática para el analizador sintáctico.

Diseño de tabla de símbolos

Hemos decidido utilizar un diseño para la tabla de símbolos sin demasiada información carente de significado, pero que pudiera ser legible de una manera más o menos sencilla por una persona.
Primero, la línea que representa el encabezado de la tabla de símbolos tiene el siguiente formato:

```
main # 0 :
```

dónde main representa el nombre del “scope” o “ámbito” de la tabla de símbolos, y 0 el identificador numérico unívoco de la tabla. Es importante resaltar que estos identificadores se van generando en orden, y por lo tanto un scope hijo de otro va a tener un identificador con mayor valor. El valor 0 pertenece al scope principal del programa, donde se declaran las variables globales.

A continuación, por cada símbolo en dicha tabla de símbolos tendremos las dos líneas siguientes:

```
* LEXEMA : 'suma'
ATRIBUTOS :
```

donde suma representaría el identificador del símbolo; seguido de una linea como la siguiente para cada atributo del símbolo:

```
+ Tipo : 'function'
```

con Tipo representando el nombre del atributo, seguido del caracter : y de, o bien un valor entero o bien una cadena entre comillas simples. Después del último atributo, y antes del siguiente símbolo, tendremos una línea de caracteres - que permite una mejor lectura del archivo.

Análisis semántico

Como hemos mencionado antes, la clase AnalizadorSemantico es la encargada de esta fase del proceso. Recibe del analizador sintáctico el árbol sintáctico, desarrollado a partir de las producciones del parse, y opera sobre esos nodos. Como apoyo, hemos creado la clase Árbol y la clase Nodo, que nos proporcionan los métodos y estructuras de datos necesarios para construir el mencionado árbol.
Cabe mencionar la clase Nodo con un poco más de detalle, pues será sobre la que en casi todo momento operamos durante el análisis semántico. En el momento de la instanciación de un nuevo objeto de tipo nodo, se inicializa un Map<String,Object> de propiedades que relaciona el nombre de una propiedad (como puede ser tipo, valor, tipoRetorno, argumentos…) con un objeto de tipo Object genérico que se especifica en el momento en el que se añade la propiedad. Este mapa de propiedades nos va a permitir guardar información entre los diferentes nodos, así como pasar informacion de padres a hijos o entre hermanos (atributos heredados) y de hijos a padres (atributos sintetizados).

Volviendo a la implementación del autómata para el analizador semántico, lo primero que haremos será obtener una lista de nodos a partir del árbol sintáctico recibido como parámetro, recorriendolo en postorden. Esto me va a permitir que dado un nodo X, cuando tenga que ejecutar X en el analizador semántico todos sus nodos hijo, así como todos los hermanos a la izquierda, van a tener ya todos los valores y propiedades resueltos, y podré operar los valores de X a partir de ellos.

A continuación, de manera similar a como hicimos en el analizador léxico, vamos enviando a la función calcularCódigo los nodos, y está, dependiendo de la producción que haya generado los nodos hijo a partir del nodo en cuestión que se está ejecutando, realizará una u otra tarea.

Por ejemplo, para el caso de la produccion {D -> var T id I ;}, podemos obtener el tipo de la variable que se va a declarar de la propiedad tipo de T (en adelante utilizaremos la notación T.tipo), el identificador de la variable de id.Token.Lexema, y el valor de inicialización (si hay alguno), de I.valor. Comprobaremos entonces si el tipo de valor inicializado en I es el mismo que el declarado en T, si existe ese identificador en la tabla de símbolos actual (contexto actual), y si no está, inicializar la variable.

Las acciones semánticas correspondientes a las producciones se pueden ver en la siguiente tabla, aunque las que no tienen acciones semánticas no están incluidas en la tabla:

Nº de producción	Producción	Acción semántica
5	D -> var T id I ;	{D.tipo=T.tipo; D.id=id.lexema; D.valor = I.valor}
6	T -> int	{T.tipo= int}
7	T -> string	{T.tipo=string}
8	T -> bool	{T.tipo=bool}
9	I->= E	{I.valor=E.valor;I.tipo=E.tipo}
10	I ->|= E	{I.valor=E.valor; I.tipo=E.tipo}
11	I-> lambda	{I.tipo="void"}
12	E -> G EE	{if G.tipo=EE.tipo; E.tipo=G.tipo}
13	G -> (X)	{G.valor=X.valor; G.tipo=X.tipo}
14	G -> id GG	{if (G.tipo=id.tipo) G.valor=id.valor}
15	G -> ! B	{if (B.tipo=bool) {G.valor=not B.valor}}
16	G -> cte_int	{G.valor=cte_int.valor}
17	G -> cte_cadena	{G.valor = cte_cadena.valor}
18	G -> cte_logica	{G.valor = cte_logica.valor}
19	GG -> lambda	{GG.tipo="void"}
20	GG -> --	{GG.Operador==menosmenos}
21	GG -> ++	{GG.operador==masmas}
22	EE -> + G EE	{EE.tipo=G.tipo; EE.operacion=+}
23	EE -> - G, EE	{EE.tipo=G.tipo; EE.operacion=-}
24	EE -> lambda	{EE.tipo="void"}

25	EE ->G EE	{EE.tipo=G.tipo; EE.operacion=}
26	EE -> / G EE	{EE.tipo=G.tipo; EE.operacion=/}
27	EE -> % G EE	{EE.tipo=G.tipo; EE.operacion=%}
28	X -> E XX	{X.tipo=E.tipo ;X.Operador=XX.Operador}
29	XX -> == E	{XX.Operacion=igualigual XX.tipo=E.tipo}
30	EE -> || G EE	{EE.tipo=G.tipo EE.operacion=||}
31	XX -> < E	{XX.Operacion=menor XX.tipo=E.tipo}
32	XX -> > E	{XX.Operacion=mayor XX.tipo=E.tipo}
33	XX -> >= E	{XX.Operacion=mayorigual XX.tipo=E.tipo}
34	XX -> <= E	{XX.Operacion=menorigual XX.tipo=E.tipo}
35	F -> function H id (A) { C }	{F.tipoRetorno= H.tipoRetorno; F.id=id}
36	H -> T	{H.tipo=T.tipo}
37	H -> lambda	{H.tipo="void"}
38	A -> T id AA	{A.tipo=T.tipo A.id=id A.argumentosdef=AA.argumentosdef}
39	A -> lambda	{A.tipo="void"}
40	AA -> , T id AA	{AA.tipo=T.tipo AA.id=id AA.argumentosdef=AA.argumentodef}
41	AA -> lambda	{AA.tipo="void"}
44	C -> lambda	{C.tipo="void"}
45	S -> id M ;	{S.id= id S.valor=M.valor}
46	S -> return R ;	{S.tipoRetorno=R.tipo}
47	S -> print (X) ;	{S.tipo=X.tipo}
48	S -> prompt (id) ;	{S.id=id}
49	S ->for (D X ; SS) { C }]	{crear tsFor, tsFor.addSimbolo(D.id,D.tipo, D.valor), si(X.valor==true) {crear tsInnerFor; ejecutar(C); ejecutar(SS)}, repetir bucle
50	R -> X	{R.valor=X.valor}
51	R -> lambda	{R.tipo="void"}
52	B -> (X)	{B.valor=X.valor}

53	B -> cte_logica	{B.valor=cte_logica.valor}
54	B -> id	{B.id=id}
55	SS -> id M	{SS.valor=M.valor; SS.id=id}
56	SS -> print (X)	{print (X.valor)}
57	XX -> lambda	{XX.tipo="void"}
58	M -> ++	{MM.Operador==masmas}
59	M -> --	{MM.Operador==menosmenos}
60	M -> I	{M.tipo=I.tipo}
61	GG -> L	{GG.tipo=L.tipo}
62	EE -> && G EE	{EE.tipo=G.tipo; EE.valor=G.valor; EE.operacion=&&}
64	L -> (AR)	{L.argumentos=AR.argumentos}
65	AR -> E RR	{AR.argumentos=RR.argumentoss+[AR.argumentos.size,E.tipo,E.valor]}
66	AR -> landa	{AR.tipo="void"}
67	RR -> , RRR	{RR.argumentos=RRR.argumentos}
68	RR -> landa	{RR.tipo="void"}
69	M -> L	{M.Argumentos=L.Argumentos}
70	RRR -> E RR	{RRR.argumentos=RR.argumentos+[RRR.argumentos.size,E.tipo,E.valor]}
71	S -> if (X) SS ;	si (X.valor==true) ejecutar(SS)

En cuanto a los contextos de las variables (scopes), se tiene en cuenta lo siguiente:

- Se crea una tabla de símbolos cuando estamos declarando una función, con el objetivo de comprobar que el cuerpo de la funcion cumple las reglas semánticas igual que el resto del programa.
- Cuando llamamos a una función, crearemos una tabla de símbolos hija a la tabla actual en la que añadiremos los identificadores de los parámetros pasados y sus valores, para la posterior ejecución del cuerpo de la función.
  A la hora de retornar un valor, obtendremos la tabla de símbolos inmediatamente superior a la de la ejecución de la función, añadiremos un símbolo temporal con identificador “$$retorno$$”, que posteriormente será leído por la sentencia que llamó a la función, y una vez leído se eliminará, asignándose a la variable que corresponda, imprimiendola, etc.

- En un bucle for, crearemos una tabla de símbolos, que llamaremos “tablaFor”, hija de la tabla de símbolos actual (puede ser la del programa principal o la de la ejecución de una función, por ejemplo), que guarde la variable declarada en la declaración del for (en el caso de for(var int i=0;i<....){C} la variable i) para todas las ejecuciones del cuerpo principal C. Así mismo, cada vez que realicemos una iteración, crearemos una tabla de símbolos, que llamaremos “innerFor”, que contendrá todas las variables declaradas dentro del cuerpo del bucle for, y que no serán accesibles desde fuera.

Tratamiento de Errores en la aplicación

En esta sección describiremos todos los errores que se pueden generar en la aplicación, como se muestran al usuario y qué formato tendrán.

En primer lugar, los errores que se produzcan durante el proceso de ejecución de todas las fases se mostrarían por la salida de error estándar del sistema. Además, si se produjeran errores, se generarían los ficheros correspondientes a todas las fases anteriores a la que ha producido el error.

El mensaje de error tendrá el siguiente formato como el siguiente ejemplo:

```
Error generico. Caso de prueba elegido no esta disponible.
Informacion adicional: correcto5
Linea 0.
Codigo 0x3
```

La primera línea especifica en qué fase de la ejecución se ha producido el error (error general, léxico, sintáctico o semántico). Además, mostrará información un poco más específica de qué ha producido el error (variable no declarada, tipo no esperado…).

En la segunda línea tendremos información adicional del error, como por ejemplo qué tipo de variable se esperaba y que tipo ha recibido. Puede ser también el nombre de la variable que no ha sido declarada, el nombre del archivo que no se puede leer, u otra información que nos permita saber exactamente qué está fallando.

La tercera línea contiene el número de línea donde se ha producido el error (0 en el caso de errores no relacionados con el código como por ejemplo archivos, errores internos…). Por último, en la cuarta línea se muestra el código de error, donde el número antes del caracter x representa la fase en la que se produjo el error (0 al 3 para error general, léxico, sintáctico y semántico respectivamente), y el número inmediatamente después el tipo de error de esa fase, que se puede consultar en la siguiente tabla.

Código de error	Mensaje del error	Información adicional proporcionada
0x0	Error genérico.	Error durante la lectura o escritura del archivo.
0x1	Error genérico. No se puede leer el archivo de código.	Ruta actual de carpetas y ruta al archivo de código.
0x2	Error genérico. No se puede abrir el archivo para escritura.	Ruta actual de carpetas y ruta al archivo de código.
0x3	Error genérico. Caso de prueba elegido no está disponible.	Caso de prueba seleccionado.
1x0	Error léxico.	
1x1	Error léxico: caracter no reconocido.	Caracter recibido.
2x0	Error sintáctico: token inesperado.	Token recibido y lista de posibles tokens esperados.
2x1	Error sintáctico: token desconocido.	Token recibido.
3x0	Error semántico.	
3x1	Error semántico: el valor recibido no es del tipo declarado.	Tipo declarado y tipo recibido.
3x2	Error semántico: ya existe un identificador con ese nombre.	Nombre del identificador.
3x3	Error semántico: el valor recibido no es del tipo esperado.	Tipo recibido y tipo esperado.
3x4	Error semántico: el tipo de ambos operandos debe ser el mismo.	Tipos recibidos.
3x5	Error semántico: no existe un identificador con ese nombre.	Nombre del identificador.
3x6	Error semántico: número de argumentos no es el esperado.	Número de parámetros recibido y número de parámetros esperado.
3x7	Error semántico: el argumento recibido no es del tipo esperado.	Tipo recibido, tipo esperado y nombre del parámetro.

ANEXO: Casos de prueba para el analizador léxico

Caso 1: Asignación de variable como operación aritmética de otra variable y un entero.

Entrada:
```
var var1  =var2+ 13;
```

Salida:
```
<var, >  <id, var1>  <=, >  <id, var2>  <+, >  <cte_int, 13>  <;, >
```

Caso 2: Bucle for

Entrada:
```
var int a=1;
for(var int i=0;i<10;i=i+1}(
        a=a*i;
}
```

Salida:
```
<var, >  <int, >  <id, a>  <=, >  <cte_int, 1>  <;, >  <for, >  <(, >  <var, >  <int, >  <id, i>  <=, >  <cte_int, 0>  <;, >  <id, i>  <<, >  <cte_int, 10>  <;, >  <id, i>  <=, >  <id, i>  <+, >  <cte_int, 1>  <}, >  <(, >  <id, a>  <=, >  <id, a>  <*, >  <id, i>  <;, >  <}, >
```

Caso 3: Comentarios

Entrada:
```
//Esto es un comentario simple
var a = "Hola me llamo \"JORGE\" ";
/* Esto es un comentario en
bloque */
```

Salida:
```
<var, >  <id, a>  <=, >  <cte_cadena, Hola me llamo >  <cte_cadena, JORGE>  <cte_cadena, >  <;, >
```

Caso 4: Cadena de texto incompleta

Entrada:
var string a= “Cadena incompleta

Salida:
Error léxico.
Informacion adicional: Cadena no cerrada.
Linea 1.
Codigo 1x0

Caso 5: Comentario en bloque incompleto

Entrada:
/* Comentario incompleto*

Salida:
Error léxico.
Informacion adicional: Comentario no cerrado.
Linea 1.
Codigo 1x0

Caso 6: Carácter no esperado

Entrada:
var a@=b;

Salida:
Error léxico: Caracter no esperado.
Informacion adicional: @
Linea 1.
Codigo 1x1

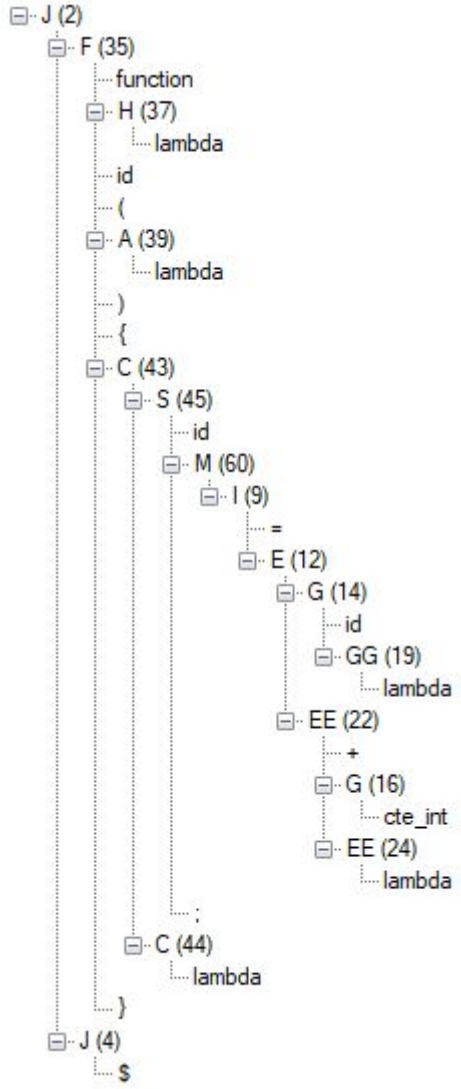
ANEXO: Casos de prueba para el analizador sintáctico

Caso 1:

Entrada
function sumaUno(){i=i+1;}

Parse:
Desc 2 35 37 39 43 45 60 9 12 14 19 22 16 24 44 4

Árbol:



Caso 2:

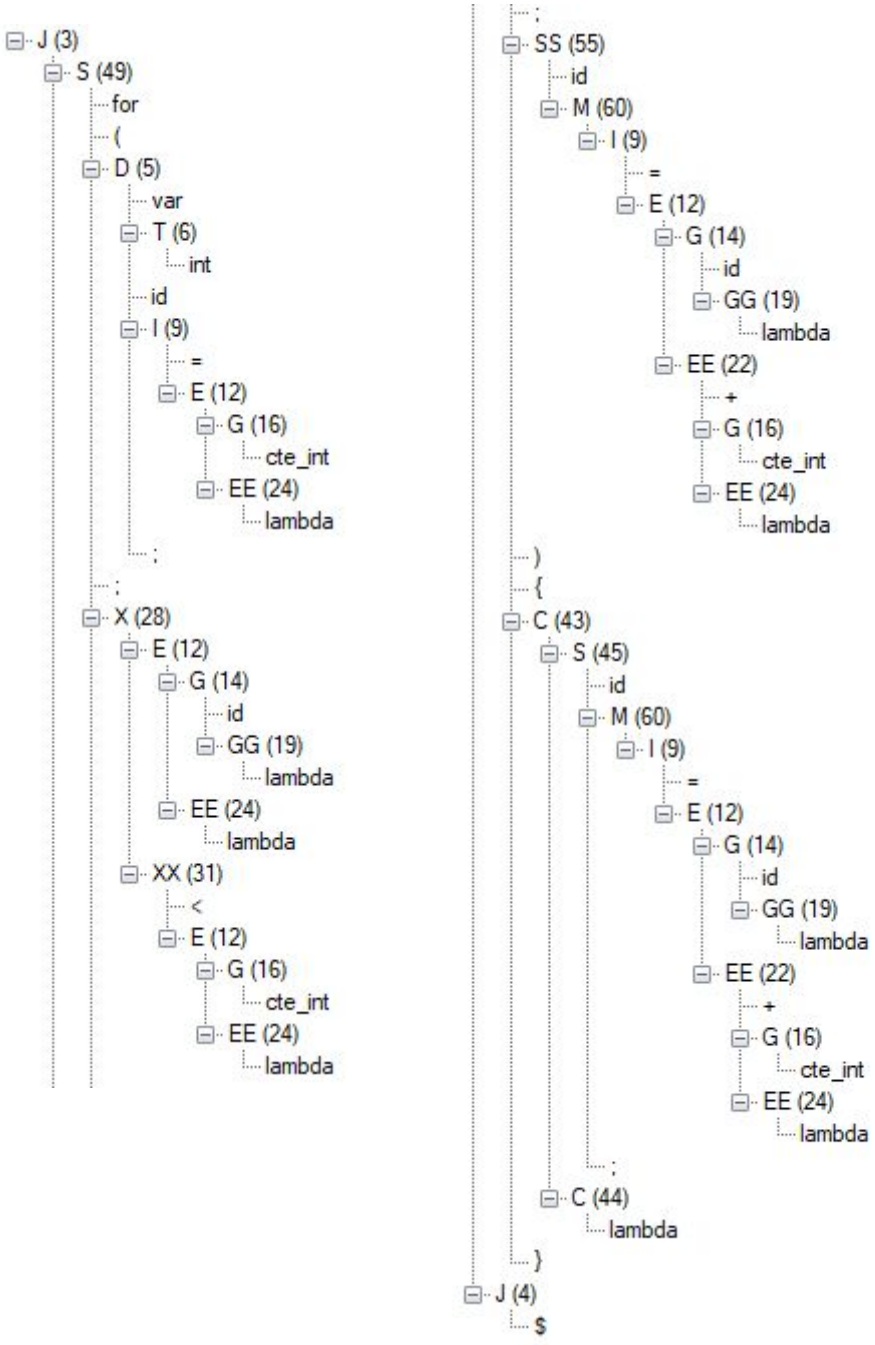
Entrada:

for(var int i=0;i<5;i=i+1) {i=i+1;}

Parse:

Desc 3 49 5 6 9 12 16 24 28 12 14 19 24 31 12 16 24 55 28 12 14 19 22 16 24 57 43 45 12 14 19 22 16 24 44 4

Árbol:



Caso 3:

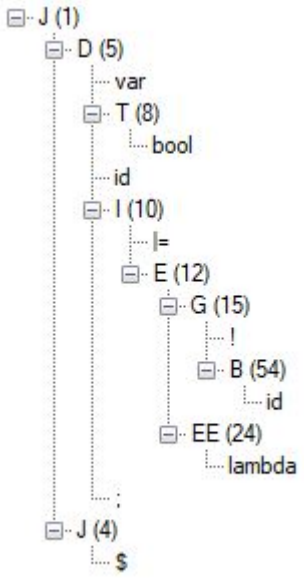
Entrada:

var bool x|=!j;

Parse:

Desc 1 5 8 10 12 15 54 24 4

Árbol:



Caso 4:

Entrada:
function suma(){i=i+1}
Error:
Error sintactico: token inesperado.
Informacion adicional: Recibido: }. Se esperaba: [&&, ||, ==, <=, %,), *, +, ,, -, /, cte_int, ;, <, >, >=].
Linea 1.
Codigo 2x0

Caso 5:

Entrada:
for(var int i=0;;i=i+1){i=i+1;}
Error:
Error sintactico: token inesperado.
Informacion adicional: Recibido: ;. Se esperaba: [!, cte_int, (, cte_cadena, cte_logica, id].
Linea 1.
Codigo 2x0

Caso 6:

Entrada:

prompt();
Error:
Error sintactico: token inesperado.
Informacion adicional: Recibido:). Se esperaba: id.
Linea 1.
Codigo 2x0

Anexo: Tabla sintáctica

	A	AA	AR	B	C	D	E	EE	F	G	GG	H	I	J	L
!	-	-	AR → E RR	-	-	-	E → G EE	-	-	G → ! B	-	-	-	-	-
$	-	-	-	-	-	-	-	-	-	-	-	-	-	J → $	-
%	-	-	-	-	-	-	-	EE → % G EE	-	-	GG → lambda	-	-	-	-
&&	-	-	-	-	-	-	-	EE → && G EE	-	-	GG → lambda	-	-	-	-
(	-	-	AR → E RR	B → (X)	C → S C	-	E → G EE	-	-	G → (X)	GG → L	-	-	J → S J	L → (AR)
)	A → lambda	AA → lambda	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	I → lambda	-	-
*	-	-	-	-	-	-	-	EE → * G EE	-	-	GG → lambda	-	-	-	-
+	-	-	-	-	-	-	-	EE → + G EE	-	-	GG → lambda	-	-	-	-
++	-	-	-	-	-	-	-	-	-	-	GG → ++	-	-	-	-
,	-	AA → T id AA	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
-	-	-	-	-	-	-	-	EE → - G EE	-	-	GG → lambda	-	-	-	-
--	-	-	-	-	-	-	-	-	-	-	GG → --	-	-	-	-
//	-	-	-	-	-	-	-	EE → // G EE	-	-	GG → lambda	-	-	-	-
;	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	I → lambda	-	-
<	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
<=	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
=	-	-	-	-	-	-	-	-	-	-	-	-	I → = E	-	-
==	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
>	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
>=	-	-	-	-	-	-	-	EE → lambda	-	-	GG → lambda	-	-	-	-
bool	A → T id AA	-	-	-	-	-	-	-	-	-	-	H → T	-	-	-
char	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
cte_cadena	-	-	AR → E RR	-	-	-	E → G EE	-	-	G → cte_cadena	-	-	-	-	-
cte_int	-	-	AR → E RR	-	-	-	E → G EE	-	-	G → cte_int	-	-	-	-	-
cte_logica	-	-	AR → E RR	B → cte_logica	-	-	E → G EE	-	-	G → cte_logica	-	-	-	-	-
false	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
for	-	-	-	-	C → I S C	-	-	-	-	-	-	-	-	J → I S J	-
function	-	-	-	-	-	-	-	-	F → function H id (A) { C }	-	-	-	-	J → F J	-
id	-	-	AR → E RR	B → id	C → S C	-	E → G EE	-	-	G → id GG	-	H → lambda	-	J → S J	-
if	-	-	-	-	C → I S C	-	-	-	-	-	-	-	-	J → I S J	-
int	A → T id AA	-	-	-	-	-	-	-	-	-	-	H → T	-	-	-
print	-	-	-	-	C → S C	-	-	-	-	-	-	-	-	J → S J	-
prompt	-	-	-	-	C → S C	-	-	-	-	-	-	-	-	J → S J	-
return	-	-	-	-	C → S C	-	-	-	-	-	-	-	-	J → S J	-
string	A → T id AA	-	-	-	-	-	-	-	-	-	-	H → T	-	-	-
true	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
var	-	-	-	-	C → D C	D → var T id ;	-	-	-	-	-	-	-	J → D J	-
{	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
||	-	-	-	-	-	-	-	EE → || G EE	-	-	GG → lambda	-	-	-	-
}	-	-	-	-	C → lambda	-	-	-	-	-	-	-	-	-	-
$ (final de cadena)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

M	R → X	RR → lambda	RRR → E RR	S → for (D ; X ; SS) { C }	S → if (X) SS ;	S → id M ;	S → print (X) ;	S → prompt (id) ;	S → return R ;	T → string	T → int	SS → print (X)	SS → id M	X → E XX	XX → lambda	XX → < E	XX → <= E	XX → == E	XX → > E	XX → >= E	T → bool	M → I	M → L	M → ++	M → .	M → (	M → I
R → X			RRR → E RR			R → X									R → lambda												
RR → lambda			RRR → E RR			R → X																				
RRR → E RR						R → X																				
S → for (D ; X ; SS) { C }																										
S → if (X) SS ;																										
S → id M ;	R → X		RRR → E RR																							
S → print (X) ;																										
S → prompt (id) ;																										
S → return R ;																										
T → string																										
T → int																										
SS → print (X)																										
SS → id M																										
X → E XX			RRR → E RR			R → X																				
XX → lambda																										
XX → < E																										
XX → <= E																										
XX → == E																										
XX → > E																										
XX → >= E																										
T → bool																										
M → I																										
M → L																										
M → ++																										
M → .																										
M → (																										
M → I																										
X → E XX																										
XX → lambda																										
XX → < E																										
XX → <= E																										
XX → == E																										
XX → > E																										
XX → >= E																										
T → bool																										
M → I																										
M → L																										
M → ++																										
M → .																										
M → (																										
M → I																										

Caso de prueba: Funcionamiento completo

Tokens

```
<var, >
<int, >
<id, a>
<=, >
<cte_int, 1>
<;, >
<var, >
<int, >
<id, b>
<=, >
<cte_int, 2>
<;, >
<var, >
<int, >
<id, c>
<=, >
<cte_int, 3>
<;, >
<var, >
<int, >
<id, d>
<=, >
<cte_int, 4>
<;, >
<function, >
<int, >
<id, suma>
<(, >
<int, >
<id, x>
<,, >
<int, >
<id, y>
<), >
<{, >
<return, >
<id, x>
<+, >
<id, y>
<;, >
<}, >
<function, >
<int, >
```

<id, resta>
<(, >
<int, >
<id, x>
<,, >
<int, >
<id, y>
<), >
<{, >
<return, >
<id, x>
<-, >
<id, y>
<;, >
<}, >
<function, >
<int, >
<id, mult>
<(, >
<int, >
<id, x>
<,, >
<int, >
<id, y>
<), >
<{, >
<return, >
<id, x>
<*, >
<id, y>
<;, >
<}, >
<function, >
<int, >
<id, div>
<(, >
<int, >
<id, x>
<,, >
<int, >
<id, y>
<), >
<{, >
<return, >
<id, x>
</, >
<id, y>
<;, >
<}, >
<var, >
<int, >
<id, e>

<=, >
<id, suma>
<(, >
<id, a>
<,, >
<id, b>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es 3>
<), >
<;, >
<print, >
<(, >
<id, e>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es false>
<), >
<;, >
<print, >
<(, >
<id, e>
<<, >
<cte_int, 3>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es true>
<), >
<;, >
<print, >
<(, >
<id, e>
<<=, >
<cte_int, 3>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es true>
<), >
<;, >
<print, >
<(, >
<id, e>
<==, >
<cte_int, 3>

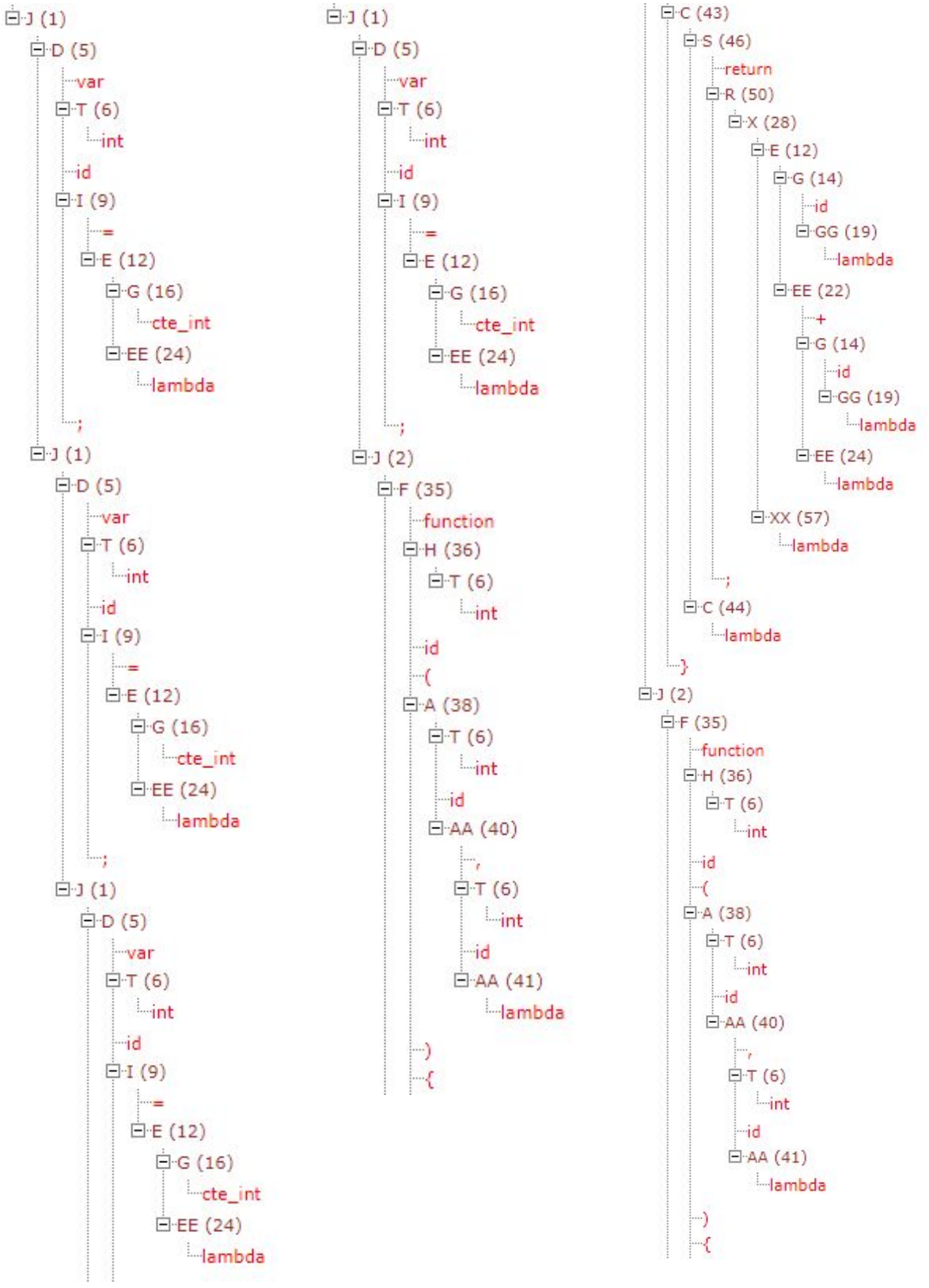
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es true>
<), >
<;, >
<print, >
<(, >
<id, e>
<>=, >
<cte_int, 3>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es false>
<), >
<;, >
<print, >
<(, >
<id, e>
<>, >
<cte_int, 3>
<), >
<;, >
<var, >
<int, >
<id, f>
<=, >
<id, resta>
<(, >
<id, d>
<,, >
<id, c>
<), >
<;, >
<print, >
<(, >
<cte_cadena, Resultado esperado es 3>
<), >
<;, >
<print, >
<(, >
<id, div>
<(, >
<id, e>
<,, >
<id, f>
<), >
<), >
<;, >

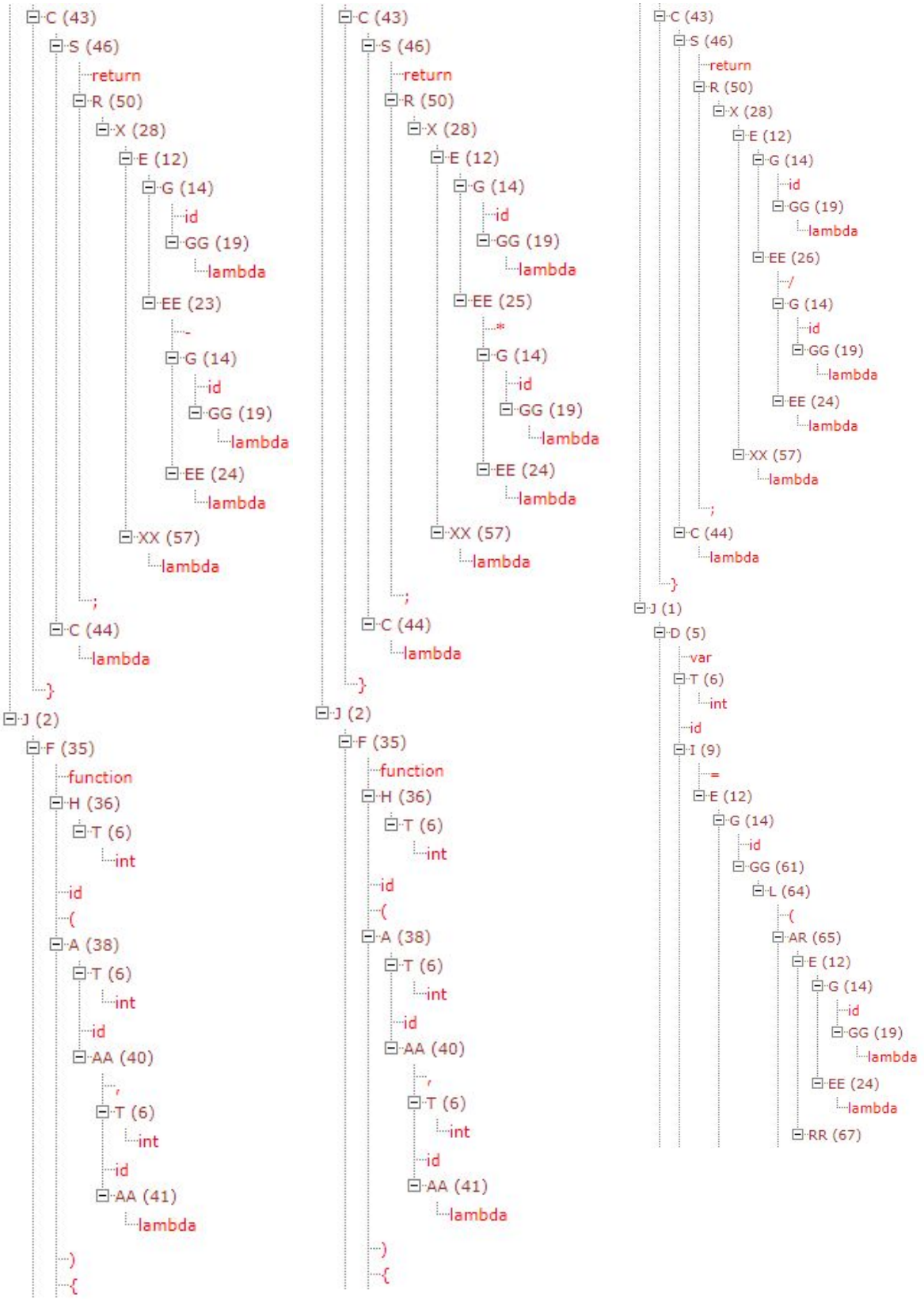
<print, >
<(, >
<cte_cadena, Resultado esperado es 8>
<), >
<;, >
<print, >
<(, >
<id, mult>
<(, >
<id, suma>
<(, >
<id, a>
<,, >
<id, c>
<), >
<,, >
<id, resta>
<(, >
<id, d>
<,, >
<id, b>
<), >
<), >
<), >
<;, >

Parse

Desc 1 5 6 9 12 16 24 1 5 6 9 12 16 24 1 5 6 9 12 16 24 1 5 6 9 12 16 24 2 35 36 6 38 6 40 6 41 43 46 50 28 12 14 19 22 14 19 24 57 44 2 35 36 6 38 6 40 6 41 43 46 50 28 12 14 19 23 14 19 24 57 44 2 35 36 6 38 6 40 6 41 43 46 50 28 12 14 19 25 14 19 24 57 44 2 35 36 6 38 6 40 6 41 43 46 50 28 12 14 19 26 14 19 24 57 44 1 5 6 9 12 14 61 64 65 12 14 19 24 67 70 12 14 19 24 68 24 3 47 28 12 17 24 57 3 47 28 12 14 19 24 57 3 47 28 12 17 24 57 3 47 28 12 14 19 24 31 12 16 24 3 47 28 12 17 24 57 3 47 28 12 14 19 24 34 12 16 24 3 47 28 12 17 24 57 3 47 28 12 14 19 24 29 12 16 24 3 47 28 12 17 24 57 3 47 28 12 14 19 24 33 12 16 24 3 47 28 12 17 24 57 3 47 28 12 14 19 24 32 12 16 24 1 5 6 9 12 14 61 64 65 12 14 19 24 67 70 12 14 19 24 68 24 3 47 28 12 17 24 57 3 47 28 12 14 61 64 65 12 14 19 24 67 70 12 14 19 24 68 24 57 3 47 28 12 17 24 57 3 47 28 12 14 61 64 65 12 14 61 64 65 12 14 19 24 67 70 12 14 19 24 68 24 67 70 12 14 61 64 65 12 14 19 24 67 70 12 14 19 24 68 24 68 24 57 4

Árbol sintáctico


J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (16)
cte_int
EE (24)
lambda
;
J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (16)
cte_int
EE (24)
lambda
;
J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (16)
cte_int
EE (24)
lambda
;
J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (16)
cte_int
EE (24)
lambda
;
J (2)
F (35)
function
H (36)
T (6)
int
id
(
A (38)
T (6)
int
id
AA (40)
,
T (6)
int
id
AA (41)
lambda
)
{
C (43)
S (46)
return
R (50)
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (22)
+
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (57)
lambda
;
C (44)
lambda
}
J (2)
F (35)
function
H (36)
T (6)
int
id
(
A (38)
T (6)
int
id
AA (40)
,
T (6)
int
id
AA (41)
lambda
)
{


C (43)
S (46)
return
R (50)
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (23)
-
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (57)
lambda
;
C (44)
lambda
}
J (2)
F (35)
function
H (36)
T (6)
int
id
(
A (38)
T (6)
int
id
AA (40)
,
T (6)
int
id
AA (41)
lambda
)
{
C (43)
S (46)
return
R (50)
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (25)
*
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (57)
lambda
;
C (44)
lambda
}
J (2)
F (35)
function
H (36)
T (6)
int
id
(
A (38)
T (6)
int
id
AA (40)
,
T (6)
int
id
AA (41)
lambda
)
{
C (43)
S (46)
return
R (50)
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (26)
/
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (57)
lambda
;
C (44)
lambda
}
J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (14)
id
GG (61)
L (64)
(
AR (65)
E (12)
G (14)
id
GG (19)
lambda
EE (24)
lambda
RR (67)

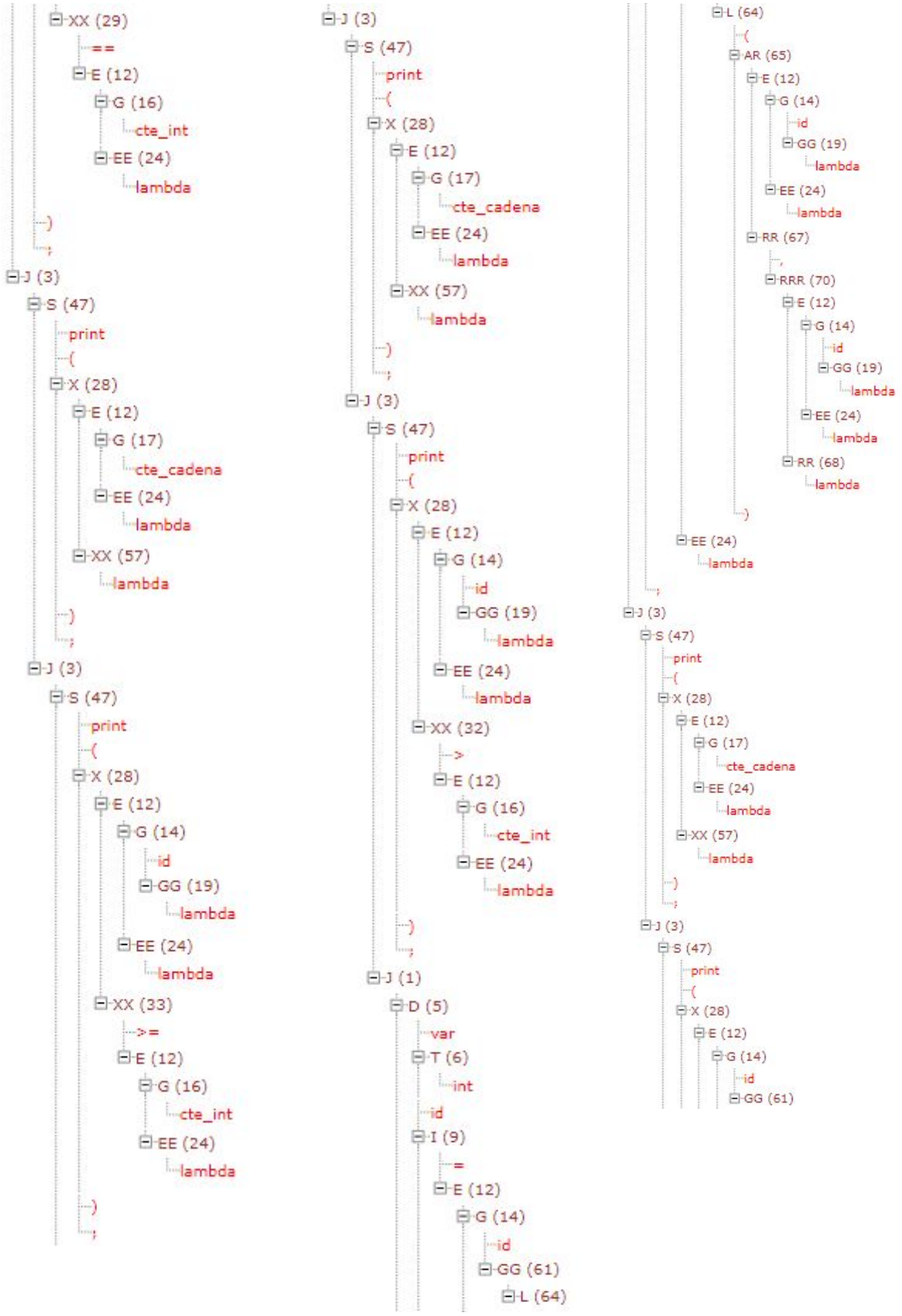

XX (29)
==
E (12)
G (16)
cte_int
EE (24)
lambda
)
;
J (3)
S (47)
print
(
X (28)
E (12)
G (17)
cte_cadena
EE (24)
lambda
XX (57)
lambda
)
;
J (3)
S (47)
print
(
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (33)
>=
E (12)
G (16)
cte_int
EE (24)
lambda
)
;
J (3)
S (47)
print
(
X (28)
E (12)
G (17)
cte_cadena
EE (24)
lambda
XX (57)
lambda
)
;
J (3)
S (47)
print
(
X (28)
E (12)
G (14)
id
GG (19)
lambda
EE (24)
lambda
XX (32)
>
E (12)
G (16)
cte_int
EE (24)
lambda
)
;
J (1)
D (5)
var
T (6)
int
id
I (9)
=
E (12)
G (14)
id
GG (61)
L (64)
(
AR (65)
E (12)
G (14)
id
GG (19)
lambda
EE (24)
lambda
RR (67)
,
RRR (70)
E (12)
G (14)
id
GG (19)
lambda
EE (24)
lambda
RR (68)
lambda
)
EE (24)
lambda
;
J (3)
S (47)
print
(
X (28)
E (12)
G (17)
cte_cadena
EE (24)
lambda
XX (57)
lambda
)
;
J (3)
S (47)
print
(
X (28)
E (12)
G (14)
id
GG (61)

Tabla de Símbolos

main # 0 :

```
* LEXEMA : 'suma'
ATRIBUTOS :
+ Tipo : 'function'
+ Despl : 8
+ EtiqFuncion : suma
+ IdParam2 : y
+ TipoParam2 : int
+ TipoParam1 : int
+ IdParam1 : x
+ TipoRetorno : int
+ NumParam : 2
--------------------
* LEXEMA : 'div'
ATRIBUTOS :
+ Tipo : 'function'
+ Despl : 8
+ EtiqFuncion : div
+ IdParam2 : y
+ TipoParam2 : int
+ TipoParam1 : int
+ IdParam1 : x
+ TipoRetorno : int
+ NumParam : 2
--------------------
* LEXEMA : 'a'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 0
--------------------
* LEXEMA : 'b'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 2
--------------------
* LEXEMA : 'c'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 4
--------------------
* LEXEMA : 'mult'
ATRIBUTOS :
+ Tipo : 'function'
+ Despl : 8
+ EtiqFuncion : mult
+ IdParam2 : y
+ TipoParam2 : int
+ TipoParam1 : int
+ IdParam1 : x
+ TipoRetorno : int
+ NumParam : 2
--------------------
* LEXEMA : 'd'
```

```
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 6
------------------
* LEXEMA : 'e'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 10
------------------
* LEXEMA : 'f'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 14
------------------
* LEXEMA : 'resta'
ATRIBUTOS :
+ Tipo : 'function'
+ Despl : 8
+ EtiqFuncion : resta
+ IdParam2 : y
+ TipoParam2 : int
+ TipoParam1 : int
+ IdParam1 : x
+ TipoRetorno : int
+ NumParam : 2
------------------



FUNC suma # 1 :
* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 2
------------------
* LEXEMA : 'y'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 0
------------------



FUNC resta # 2 :
* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 2
------------------
* LEXEMA : 'y'
ATRIBUTOS :
+ Tipo : 'int'
```

```
+ Despl : 0
-------------------



FUNC mult # 3 :
* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 2
-------------------
* LEXEMA : 'y'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 0
-------------------



FUNC div # 4 :
* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 2
-------------------
* LEXEMA : 'y'
ATRIBUTOS :
+ Tipo : 'int'
+ Despl : 0
-------------------
```