

Universidad Politécnica de Madrid

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software (DLSIS)

Escuela técnica superior de ingenieros informáticos

Práctica Procesadores de Lenguajes

JavaScript-PDL

GRUPO 68



CAMPUS
DE EXCELENCIA
INTERNACIONAL

Práctica Procesadores de Lenguajes

Autores Grupo 68:

Raúl Muñoz Dávila 20M063

Alfonso Mateos Vicente 20M041

Aaron Moyano Alcacer 20M020

Resumen:

Esta memoria define el diseño y construcción del Procesador creado por el grupo 68 en el lenguaje JavaScript-PDL. Este lenguaje contiene características específicas para cada grupo y otras globales para todos los grupos disponibles. Además, contiene la descripción del funcionamiento de los módulos del procesador de dicho lenguaje (analizador léxico, sintáctico y semántico), así como el funcionamiento y diseño empleado para la tabla de símbolos y, finalmente, un anexo con casos de prueba para determinar el funcionamiento del mismo.

Fecha:

9 de enero de 2023, Campus Montegancedo (Boadilla del Monte), Universidad Politécnica de Madrid.

Índice:

1. Objetivo.....	4
2. Introducción.....	4
3. Especificación del grupo.....	4
4. Metodología de trabajo.....	5
5. Analizador léxico.....	5
6. Analizador sintáctico.....	8
7. Analizador semántico.....	20
8. Tabla de Símbolos.....	23
9. Gestor de errores.....	25
10. Casos de prueba.....	25

1. Objetivo:

Se desea diseñar e implementar un Procesador de Lenguajes, que realice el Análisis Léxico, Sintáctico y Semántico (incluyendo la Tabla de Símbolos y el Gestor de Errores), para el lenguaje de programación *JavaScript-PL*.

2. Introducción:

Utilizando los conocimientos de programación adquiridos durante la carrera o autodidácticamente, junto con los adquiridos a lo largo de la asignatura PDL, se implementarán los diferentes módulos de un procesador de lenguajes al uso. Todo ello utilizando el lenguaje JavaScript.

Nuestro procesador trabaja de la siguiente manera. Primero, el analizador léxico comienza a procesar los caracteres del fichero fuente y genera el primer token, que se envía al analizador sintáctico, el cual comprueba que sigue con la estructura general del lenguaje, si no, se llama al gestor de errores. Finalmente, el módulo semántico comprueba que el token recibido tenga sentido en el contexto del lenguaje y lo introduce en la tabla de símbolos, en caso contrario se lanza un error.

Una vez procesado el primer token el analizador léxico continúa leyendo el fichero fuente, si encuentra alguna palabra no admitida lanza un error, si no, se repite el proceso hasta haber leído todo el fichero o se haya lanzado algún error. Entonces se libera la memoria ocupada por la tabla de simbolos.

3. Especificación del grupo:

Para la realización de la práctica se han definido unas normas generales y específicas. Atendiendo al número de grupo 68, se han asignado estas características:

- Sentencias: **Sentencia repetitiva (do-while)**
- Operadores especiales: **Asignación con resta (-=)**
- Técnicas de Análisis Sintáctico: **Descendente Recursivo**
- Comentarios: **Comentario de bloque /* */**
- Cadenas: **Con comillas simples ('')**

Además, del resto de opciones posibles se han elegido:

- Operadores aritméticos: **suma (+) y multiplicación (*)**
- Operadores relacionales: **and (&&) y or (||)**
- Operadores lógicos: **mayor que (>) y menor que (<)**

4. Metodología del trabajo:

Los miembros del grupo han colaborado y puesto en acuerdo todas las actividades para llevar a cabo la práctica de la forma más amena y efectiva posible. Tanto el diseño pedido, la implementación del código y la realización de memoria ha sido revisada y dirigida por todos los miembros del grupo.

5. Analizador léxico:

La función principal del analizador léxico es leer el fichero carácter a carácter y generar los tokens de los leídos. Además, también encontrar los posibles errores. Una vez hecho esto transmite toda la información adquirida al analizador sintáctico.

5.1. Tokens:

Se ha diseñado el siguiente número de tokens para agrupar las características del punto 3.

Nombre del token	<Identificador,	Atributo>
Suma (+)	< SUM	->
Multiplicación (*)	< MUL	->
AND (&&)	< AND	->
OR ()	< OR	->
Cadena (' ')	< CAD	cadena>
Entero	< INT	valor>
Asignación (=)	< ASIGN	->
Parent apertura ((< PAR_A	->
Parent cerrad))	< PAR_C	->
Corchete apert ({	< BRK_A	->
Corchete cerra })	< BRK_C	->
Asignación decremento (-=)	< ASIGN_SUB	->
Punto y coma ;	< SCOL	->
Coma ,	< COMM	->
Mayor	< GRT	->
Menor	< LOW	->
Id	< ID	Pos.TS>
PALABRAS RESERVADAS		
Booleano	< BOOL	->
Function	< FUNC	->
Print	< PRNT	->
return	< RET	->
input	< IN	->
let	< LET	->
do	< DO	->
while	< WHILE	->

5.2. Gramática del Lenguaje:

Para los tokens anteriores, el grupo se ha puesto de acuerdo para diseñar la siguiente gramática.

Primero de todo, definimos los caracteres utilizados los cuáles toman los valores:

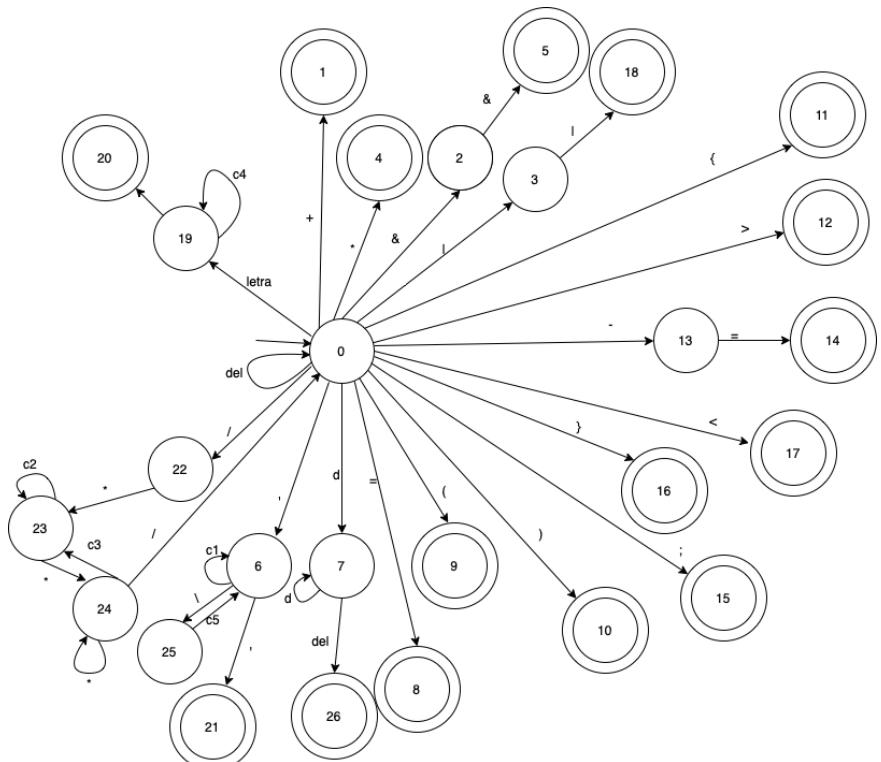
- **letra** = ($\{A \dots Z\}$, $\{a \dots z\}$)
- **d** = {0,...,9}
- **o.c.** = otro carácter
- **del** = {< tab >, < eol >, < eof >}
- **c1** : = cualquier carácter salvo “ “ “
- **c2** : = cualquier carácter salvo “ * “
- **c3** : = cualquier carácter salvo “ * / “
- **c4** : = cualquier carácter salvo “ ; “ y “del”
- **c5** : = “ \ “, “ ‘ “

Tras esto, definimos la gramática pertinente y el AFD del analizador:

- $S \rightarrow \text{del } S \mid + A \mid \& B \mid | C \mid * D \mid ' F \mid d G \mid = H \mid (I \mid) J \mid \{ K \mid > L \mid - M \mid ; N \mid \}$
- $O \mid < P \mid \text{letra } R \mid / V$
- $B \rightarrow \& E$
- $C \rightarrow | Q$
- $F \rightarrow \backslash Y \mid ' U \mid c1 F$
- $G \rightarrow \text{oc } Z \mid d G$
- $M \rightarrow = N$
- $R \rightarrow c4 R \mid \text{del } T$
- $V \rightarrow * W$
- $W \rightarrow * X \mid c2 W$
- $X \rightarrow * X \mid c3 W \mid / S$
- $Y \rightarrow ' F$

ANEXO:

$S \leftrightarrow 0$
 $A \leftrightarrow 1$
 $B \leftrightarrow 2$
 $C \leftrightarrow 3$
 $D \leftrightarrow 4$
 $E \leftrightarrow 5$
 $F \leftrightarrow 6$
 $G \leftrightarrow 7$
 $H \leftrightarrow 8$
 $I \leftrightarrow 9$
 $J \leftrightarrow 10$
 $K \leftrightarrow 11$
 $L \leftrightarrow 12$
 $M \leftrightarrow 13$
 $N \leftrightarrow 14$
 $S \leftrightarrow 15$
 $O \leftrightarrow 16$
 $P \leftrightarrow 17$
 $Q \leftrightarrow 18$
 $R \leftrightarrow 19$
 $T \leftrightarrow 20$
 $U \leftrightarrow 21$
 $V \leftrightarrow 22$
 $W \leftrightarrow 23$
 $X \leftrightarrow 24$
 $Y \leftrightarrow 25$
 $Z \leftrightarrow 26$



5.4. Acciones Semánticas:

Primero de todo, se definen las distintas acciones:

- **LEER:** Lee el siguiente carácter del fichero fuente:
 $c \equiv \text{leer}()$
- **CONCATENAR LEXEMA:** Concatena el carácter dado por parámetro al lexema dado como parámetro:
 $\text{lex} \equiv \emptyset$
 $\text{concatenar}(c) \equiv \text{lex} = \text{lex} \oplus c$
- **CONCATENAR VALOR:** Añade un dígito dado como parámetro a la derecha del valor dado como parámetro:
 $\text{valor} \equiv 0$
 $\text{valor}() \equiv \text{valor} = d$
 $\text{valorA}() \equiv \text{valor} = \text{valor} * 10 + c$
- **GENERAR TOKEN:** Genera el token que entra como parámetro de forma que lo inserta en la tabla de símbolos actual.
- **COMPROBAR RANGO VALOR:** Comprueba que el valor sea menor que 32767.
- **COMPROBAR RANGO CADENA:** Comprueba que la longitud de la cadena sea menor que 64.

Por tanto, las acciones semánticas en cada transición son:

- **0:0** $c = \text{leer}()$
- **0:1** $c = \text{leer}(); \text{Gen_token}(\text{SUM} ,)$
- **0:2** $c = \text{leer}()$
- **2:5** $c = \text{leer}(); \text{Gen_token}(\text{AND} ,)$
- **0:4** $c = \text{leer}(); \text{Gen_token}(\text{MUL} ,)$
- **0:3** $c = \text{leer}()$
- **3:18** $c = \text{leer}(); \text{Gen_token}(\text{AND} ,)$
- **0:11** $c = \text{leer}(); \text{Gen_token}(\text{BRK_A} ,)$
- **0:12** $c = \text{leer}(); \text{Gen_token}(\text{GRT} ,)$
- **0:13** $c = \text{leer}()$
- **13:14** $c = \text{leer}(); \text{Gen_token}(\text{ASIGN_SUB} ,)$
- **0:17** $c = \text{leer}(); \text{Gen_token}(\text{LOW} ,)$
- **0:16** $c = \text{leer}(); \text{Gen_token}(\text{BRK_C} ,)$
- **0:15** $c = \text{leer}(); \text{Gen_token}(\text{SCOL} ,)$
- **0:10** $c = \text{leer}(); \text{Gen_token}(\text{PAR_A} ,)$
- **0:9** $c = \text{leer}(); \text{Gen_token}(\text{PAR_C} ,)$
- **0:8** $c = \text{leer}(); \text{Gen_token}(\text{ASIGN} ,)$

- **0:7** c = leer() ; valor = concatenar_valor(d, valor)
- **7:7** d = leer() ; valor = concatenar_valor(d, valor)
- **7:26** c = leer() ; Comprobar_rango_valor(valor) ; Gen_token(INT , valor)
- **0:6** c = leer() ; concatenar(c, lex)
- **6:6** c = leer() ; concatenar(c, lex)
- **6:25** c = leer() ; concatenar(c, lex)
- **25:6** c = leer() ; concatenar(c, lex)
- **6:21** c = leer() ; concatenar(c) ; Comprobar_rango_cadena(lex) ; Gen_token(CADENA , lex)
- **0:22** c = leer()
- **22:23** c = leer()
- **23:23** c = leer()
- **23:24** c = leer()
- **24:23** c = leer()
- **24:24** c = leer()
- **24:0** c = leer()
- **0:19** c = leer() ; concatenar(c, lex)
- **19:19** c = leer() ; concatenar(c, lex)
- **19:20** c = leer() ; Gen_token(lex ,)

6. Analizador Sintáctico:

La función principal del Analizador Sintáctico es comunicarse entre los dos analizadores restantes. Solicita al Analizador Léxico un token nuevo y a su vez pide al Analizador Semántico una correcta inserción en la Tabla de Símbolos, de una forma correcta y coherente según el lenguaje definido.

6.1. Gramática:

Para que el funcionamiento del Analizador Sintáctico sea correcto y ambos Analizadores restantes se coordinen apropiadamente, se ha definido una gramática con las características que se han atribuido específicamente para nuestro grupo.

Axioma = P1

Terminales = { () id entero cadena ; { } if else do while string let input print return || && > < + * function , = int boolean true false -= }

NoTerminales = { P1 P O1 O1O1 O2 O2O2 O3 O3O3 O4 O4O4 O5 O5O5 O6 O6O6 E EE E1 E2 S1 S2 S3 S4 S5 S6 S Q1 Q12 Q13 Q14 Q2 Q3 Q4 T Q5 Q F FF K1 K2 C }

Producciones = {

1. **P1** -> **P**
2. **P** -> **Q P | F P | λ**
3. **O1** -> **O2 O1O1**
4. **O1O1** -> **|| O2 O1O1 | λ**
5. **O2** -> **O3 O2O2**
6. **O2O2** -> **&& O3 O2O2 | λ**
7. **O3** -> **O4 O3O3**
8. **O3O3** -> **> O4 O3O3 | λ**
9. **O4** -> **O5 O4O4**
10. **O4O4** -> **< O5 O4O4 | λ**
11. **O5** -> **O6 O5O5**
12. **O5O5** -> **+ O6 O5O5 | λ**
13. **O6** -> **E O6O6**
14. **O6O6** -> *** E O6O6 | λ**
15. **E** -> **id EE | (O1) | entero | cadena | true | false**
16. **EE** -> **(E1) | λ**
17. **E1** -> **O1 E2 | λ**
18. **E2** -> **, O1 E2 | λ**
19. **S1** -> **id S2**
20. **S2** -> **= O1 ; | (E1) ; | -= O1 ;**
21. **S3** -> **print O1 ;**
22. **S4** -> **input id ;**
23. **S5** -> **return S6 ;**
24. **S6** -> **O1 | λ**
25. **S** -> **S1 | S3 | S4 | S5**
26. **Q1** -> **if(O1) Q12**
27. **Q12** -> **S | Q13 Q14**
28. **Q14** -> **else Q13 | λ**
29. **Q13** -> **{ C }**
30. **Q2** -> **let id T ;**
31. **Q3** -> **S**
32. **Q4** -> **Q5**
33. **T** -> **string | int | boolean**
34. **Q5** -> **do Q13 while (O1) ;**
35. **Q** -> **Q1 | Q2 | Q3 | Q4**
36. **F** -> **function id FF**
37. **FF** -> **T (K1) Q13 | (K1) Q13**

38. $K1 \rightarrow T \text{ id } K2 \mid \lambda$
 39. $K2 \rightarrow , T \text{ id } K2 \mid \lambda$
 40. $C \rightarrow Q C \mid \lambda$

}

6.2. Comprobación Gramática LL(1):

Para comprobar una correcta gramática LL(1), se han seguido la propiedades que se necesitan para que sea una gramática de dicho tipo.

A su vez, el programa SDGLL1 vuelve a comprobar si está definida correctamente la gramática. En dicho caso, no saldría por pantalla ningún error definido en el *Manual de Usuario*.

Esta comprobación se basa en el conocimiento de la siguiente condición:

Una gramática es LL(1) si para cada terminal A para el que haya varios consecuentes ($A \rightarrow a \mid b \mid c$) se tiene que para cada par de la forma $A \rightarrow a \mid b$:

- $\text{FIRST}(a) \cap \text{FIRST}(b) = \emptyset$
- Suponiendo $b = \lambda$ entonces: $\text{FIRST}(a) \cap \text{FOLLOW}(A) = \emptyset$

Teniendo esto en cuenta, el programa devuelve el siguiente análisis terminado correctamente:

```

Analizando simbolo 01
Analizando producción 01 -> 02 0101
Analizando simbolo 02
Analizando producción 02 -> 03 0202
Analizando simbolo 03
Analizando producción 03 -> 04 0303
Analizando simbolo 04
Analizando producción 04 -> 05 0404
Analizando simbolo 05
Analizando producción 05 -> 06 0505
Analizando simbolo 06
Analizando producción 06 -> E 0606
Analizando simbolo E
Analizando producción E -> id EE
FIRST de E -> id EE = { id }
Analizando producción E -> ( 01 )
FIRST de E -> ( 01 ) = { ( }
Analizando producción E -> entero
FIRST de E -> entero = { entero }
Analizando producción E -> cadena
FIRST de E -> cadena = { cadena }
Analizando producción E -> true
FIRST de E -> true = { true }
Analizando producción E -> false
FIRST de E -> false = { false }
FIRST de E = { ( cadena entero false id true )
FIRST de 06 -> E 0606 = { ( cadena entero false id true )
FIRST de 06 = { ( cadena entero false id true }
FIRST de 05 -> 06 0505 = { ( cadena entero false id true }
FIRST de 05 = { ( cadena entero false id true }
FIRST de 04 -> 05 0404 = { ( cadena entero false id true }
FIRST de 04 = { ( cadena entero false id true }
FIRST de 03 -> 04 0303 = { ( cadena entero false id true }
FIRST de 03 = { ( cadena entero false id true }
FIRST de 02 -> 03 0202 = { ( cadena entero false id true }
FIRST de 02 = { ( cadena entero false id true }
FIRST de 01 -> 02 0101 = { ( cadena entero false id true }
FIRST de 01 = { ( cadena entero false id true }

FIRST de S -> S3 = { print }
Analizando producción S -> S4
Analizando simbolo S4
Analizando producción S4 -> input id ;
FIRST de S4 -> input id ; = { input }
FIRST de S4 = { input }
FIRST de S -> S4 = { input }
Analizando producción S -> S5
Analizando simbolo S5
Analizando producción S5 -> return S6 ;
FIRST de S5 -> return S6 ; = { return }
FIRST de S5 = { return }
FIRST de S -> S5 = { return }
FIRST de S = { id input print return }
FIRST de Q3 -> S = { id input print return }
FIRST de Q3 = { id input print return }
FIRST de Q -> Q3 = { id input print return }
Analizando producción Q -> Q4
Analizando simbolo Q4
Analizando producción Q4 -> Q5
Analizando simbolo Q5
Analizando producción Q5 -> do Q13 while ( 01 );
FIRST de Q5 -> do Q13 while ( 01 ) ; = { do }
FIRST de Q5 = { do }
FIRST de Q4 -> Q5 = { do }
FIRST de Q4 = { do }
FIRST de Q -> Q4 = { do }
FIRST de Q = { do id if input let print return }
FIRST de C -> Q C = { do id if input let print return }
Analizando producción C -> lambda
FIRST de C -> lambda = { lambda }
FIRST de C = { do id if input let print return lambda }
Calculando FOLLOW de C
FOLLOW de C = { }

Analizando simbolo E1
Analizando producción E1 -> 01 E2
FIRST de E1 -> 01 E2 = { ( cadena entero false id true }
Analizando producción E1 -> lambda

```

Analizando producción E2 \Rightarrow lambda
 FIRST de E2 \Rightarrow lambda = { lambda }
 FIRST de E2 = { , lambda }
 Calculando FOLLOW de E2
 Calculando FOLLOW de E1
 FOLLOW de E1 = {} }
 FOLLOW de E2 = {} }
 Calculando FOLLOW de S6
 FOLLOW de S6 = { ; }
 FOLLOW de 01 = { , ; }
 FOLLOW de 0101 = { , ; }
 Analizando símbolo 0202
 Analizando producción 0202 \Rightarrow && 03 0202
 FIRST de 0202 \Rightarrow && 03 0202 = { && }
 Analizando producción 0202 \Rightarrow lambda
 FIRST de 0202 \Rightarrow lambda = { lambda }
 FIRST de 0202 = { && lambda }
 Calculando FOLLOW de 0202
 Calculando FOLLOW de 02
 FOLLOW de 02 = { , ; } }
 FOLLOW de 0202 = { , ; } }
 Analizando símbolo 0303
 Analizando producción 0303 \Rightarrow > 04 0303
 FIRST de 0303 \Rightarrow > 04 0303 = { > }
 Analizando producción 0303 \Rightarrow lambda
 FIRST de 0303 \Rightarrow lambda = { lambda }
 FIRST de 0303 = { > lambda }
 Calculando FOLLOW de 0303
 Calculando FOLLOW de 03
 FOLLOW de 03 = { && } ; ; } }
 FOLLOW de 0303 = { && } ; ; } }
 Analizando símbolo 0404
 Analizando producción 0404 \Rightarrow < 05 0404
 FIRST de 0404 \Rightarrow < 05 0404 = { < }
 Analizando producción 0404 \Rightarrow lambda
 FIRST de 0404 \Rightarrow lambda = { lambda }
 FIRST de 0404 = { < lambda }
 Calculando FOLLOW de 0404
 Calculando FOLLOW de 04
 FOLLOW de 04 = { && } ; > ; } }
 FOLLOW de 0404 = { && } ; > ; } }
 Analizando símbolo 0505
 Analizando producción 0505 \Rightarrow + 06 0505
 FIRST de 0505 \Rightarrow + 06 0505 = { + }
 Analizando producción 0505 \Rightarrow lambda
 FIRST de 0505 \Rightarrow lambda = { lambda }
 FIRST de 0505 = { + lambda }
 Calculando FOLLOW de 0505
 Calculando FOLLOW de 05
 FOLLOW de 05 = { && } ; < > ; } }
 FOLLOW de 0505 = { && } ; < > ; } }
 Analizando símbolo 0606
 Analizando producción 0606 \Rightarrow * E 0606
 FIRST de 0606 \Rightarrow * E 0606 = { * }
 Analizando producción 0606 \Rightarrow lambda
 FIRST de 0606 \Rightarrow lambda = { lambda }
 FIRST de 0606 = { * lambda }
 Calculando FOLLOW de 0606
 Calculando FOLLOW de 06
 FOLLOW de 06 = { && } + ; < > ; } }
 FOLLOW de 0606 = { && } + ; < > ; } }
 Analizando símbolo C
 Analizando producción C \Rightarrow Q C
 Analizando símbolo Q
 Analizando producción Q \Rightarrow Q1
 Analizando símbolo Q1
 Analizando producción Q1 \Rightarrow if (01) Q12
 FIRST de Q1 \Rightarrow if (01) Q12 = { if }
 FIRST de Q1 = { if }
 FIRST de Q \Rightarrow Q1 = { if }
 Analizando producción Q \Rightarrow Q2
 Analizando símbolo Q2
 Analizando producción Q2 \Rightarrow let id T ;
 FIRST de Q2 \Rightarrow let id T ; = { let }
 FIRST de Q2 = { let }
 FIRST de Q \Rightarrow Q2 = { let }
 Analizando producción Q \Rightarrow Q3
 Analizando símbolo Q3
 Analizando producción Q3 \Rightarrow S
 Analizando símbolo S
 Analizando producción S \Rightarrow S1
 Analizando símbolo S1
 Analizando producción S1 \Rightarrow id S2
 FIRST de S1 \Rightarrow id S2 = { id }
 FIRST de S1 = { id }
 FIRST de S \Rightarrow S1 = { id }
 Analizando producción S \Rightarrow S3
 Analizando símbolo S3
 Analizando producción S3 \Rightarrow print 01 ;
 FIRST de S3 \Rightarrow print 01 ; = { print }
 FIRST de S3 = { print }

FOLLOW de EE = { && } * + ; < > ; } }
 Analizando símbolo F
 Analizando producción F \Rightarrow function id FF
 FIRST de F \Rightarrow function id FF = { function }
 FIRST de F = { function }
 Analizando símbolo FF
 Analizando producción FF \Rightarrow T (K1) Q13
 Analizando símbolo T
 Analizando producción T \Rightarrow string
 FIRST de T \Rightarrow string = { string }
 Analizando producción T \Rightarrow int
 FIRST de T \Rightarrow int = { int }
 Analizando producción T \Rightarrow boolean
 FIRST de T \Rightarrow boolean = { boolean }
 FIRST de T = { boolean int string }
 FIRST de FF \Rightarrow T (K1) Q13 = { boolean int string }
 Analizando producción FF \Rightarrow (K1) Q13
 FIRST de FF \Rightarrow (K1) Q13 = { } }
 FIRST de FF = { (boolean int string)
 Analizando símbolo K1
 Analizando producción K1 \Rightarrow T id K2
 FIRST de K1 \Rightarrow T id K2 = { boolean int string }
 Analizando producción K1 \Rightarrow lambda
 FIRST de K1 \Rightarrow lambda = { lambda }
 FIRST de K1 = { boolean int string lambda }
 Calculando FOLLOW de K1
 FOLLOW de K1 = {} }
 Analizando símbolo K2
 Analizando producción K2 \Rightarrow , T id K2
 FIRST de K2 \Rightarrow , T id K2 = { } }
 Analizando producción K2 \Rightarrow lambda
 FIRST de K2 \Rightarrow lambda = { lambda }
 FIRST de K2 = { , lambda }
 Calculando FOLLOW de K2
 FOLLOW de K2 = {} }
 Analizando símbolo P
 Analizando producción P \Rightarrow Q P
 FIRST de P \Rightarrow Q P = { do id if input let print return }
 Analizando producción P \Rightarrow F P
 FIRST de P \Rightarrow F P = { function }
 Analizando producción P \Rightarrow lambda
 FIRST de P \Rightarrow lambda = { lambda }
 FIRST de P = { do function id if input let print return lambda }
 Calculando FOLLOW de P1
 Calculando FOLLOW de P1
 FOLLOW de P1 = { \$ (final de cadena) }
 FOLLOW de P = { \$ (final de cadena) }
 Analizando símbolo P1
 Analizando producción P1 \Rightarrow P
 FIRST de P1 \Rightarrow P = { do function id if input let print return lambda }
 FIRST de P1 = { do function id if input let print return lambda }
 Analizando símbolo Q12
 Analizando producción Q12 \Rightarrow S
 FIRST de Q12 \Rightarrow S = { id input print return }
 Analizando producción Q12 \Rightarrow Q13 Q14
 Analizando símbolo Q13
 Analizando producción Q13 \Rightarrow { C }
 FIRST de Q13 \Rightarrow { C } = { } }
 FIRST de Q13 = { } }
 FIRST de Q12 \Rightarrow Q13 Q14 = { } }
 FIRST de Q12 = { id input print return } }
 Analizando símbolo Q14
 Analizando producción Q14 \Rightarrow else Q13
 FIRST de Q14 \Rightarrow else Q13 = { else }
 Analizando producción Q14 \Rightarrow lambda
 FIRST de Q14 \Rightarrow lambda = { lambda }
 FIRST de Q14 = { else lambda }
 Calculando FOLLOW de Q14
 Calculando FOLLOW de Q12
 Calculando FOLLOW de Q1
 Calculando FOLLOW de Q
 FOLLOW de Q = { do function id if input let print return } \$ (final de cadena)
 FOLLOW de Q1 = { do function id if input let print return } \$ (final de cadena)
 FOLLOW de Q12 = { do function id if input let print return } \$ (final de cadena)
 FOLLOW de Q14 = { do function id if input let print return } \$ (final de cadena)
 Analizando símbolo S2
 Analizando producción S2 \Rightarrow = 01 ;
 FIRST de S2 \Rightarrow = 01 ; = { = }
 Analizando producción S2 \Rightarrow (E1) ;
 FIRST de S2 \Rightarrow (E1) ; = { } }
 Analizando producción S2 \Rightarrow = 01 ;
 FIRST de S2 \Rightarrow = 01 ; = { -- }
 FIRST de S2 = { (--) }
 Analizando símbolo S6
 Analizando producción S6 \Rightarrow 01
 FIRST de S6 \Rightarrow 01 = { cadena entero false id true }
 Analizando producción S6 \Rightarrow lambda
 FIRST de S6 \Rightarrow lambda = { lambda }
 FIRST de S6 = { cadena entero false id true lambda }

6.3. Tabla LL(1):

La tabla proporcionada por el programa SDGLL1 dada la gramática mencionada anteriormente.

Para economizar espacio se ha redistribuido la tabla sintáctica que devuelve el programa y se han eliminado las celdas vacías. Teniendo esto en cuenta, la tabla queda de la siguiente manera:

NO TERMINALES	TERMINALES	ACCIÓN
O1	($O1 \rightarrow O2 O1O1$
	cadena	
	entero	
	false	
	id	
	true	
O1O1)	$O1O1 \rightarrow \lambda$
	,	
	;	
O2	($O2 \rightarrow O3 O2O2$
	cadena	
	entero	
	false	
	id	
	true	
O2O2)	$O2O2 \rightarrow \lambda$
	,	

	;	
	&&	$O2O2 \rightarrow \&\& O3 O2O2$
O3	($O3 \rightarrow O4 O3O3$
	cadena	
	entero	
	false	
	id	
	true	
O3O3)	$O3O3 \rightarrow \lambda$
	,	
	;	
	&&	
	>	$O3O3 \rightarrow > O4 O3O3$
O4	($O4 \rightarrow O5 O4O4$
	cadena	
	entero	
	false	
	id	
	true	
O4O4)	$O4O4 \rightarrow \lambda$
	,	
	;	
	&&	

	>	
	<	O4O4 → < O5 O4O4
O5	(O5 → O6 O5O5
	cadena	
	entero	
	false	
	id	
	true	
O5O5)	O5O5 → λ
	,	
	;	
	&&	
	>	
	<	
	+	O5O5 → + O6 O5O5
O6	(O6 → E O6O6
	cadena	
	entero	
	false	
	id	
	true	
O6O6)	O6O6 → λ
	,	
	;	

	&&	
	>	
	<	
	+	
	*	$O_6O_6 \rightarrow * E O_6O_6$
C	do	
	id	
	if	
	input	$C \rightarrow Q C$
	let	
	return	
	print	
	}	$C \rightarrow \lambda$
E	($E \rightarrow (O_1)$
	cadena	$E \rightarrow \text{cadena}$
	entero	$E \rightarrow \text{entero}$
	false	$E \rightarrow \text{false}$
	true	$E \rightarrow \text{true}$
	id	$E \rightarrow \text{id} EE$
E1	(
	cadena	
	entero	
	false	$E1 \rightarrow O_1 E_2$
	true	
	id	
)	$E1 \rightarrow \lambda$

EE	&&	
)	
	+	
	*	
	,	$EE \rightarrow \lambda$
	;	
	<	
	>	
	($EE \rightarrow (E1)$
F	function	$F \rightarrow \text{function id FF}$
FF	int	
	string	$FF \rightarrow T (K1) Q13$
	boolean	
	($FF \rightarrow (K1) Q13$
K1	int	
	string	$K1 \rightarrow T \text{ id } K2$
	boolean	
	($K1 \rightarrow \lambda$
K2)	$K2 \rightarrow \lambda$
	,	$K2 \rightarrow , T \text{ id } K2$
P	do	
	id	
	if	
	input	$P \rightarrow QP$
	let	

	print	
	return	
	function	$P \rightarrow FP$
	\$	$P \rightarrow \lambda$
P1	do	$P1 \rightarrow P$
	function	
	id	
	if	
	input	
	let	
	print	
	return	
	\$	
Q	if	$Q \rightarrow Q1$
	let	$Q \rightarrow Q2$
	id	
	input	$Q \rightarrow Q3$
	print	
	return	
Q1	do	$Q \rightarrow Q4$
	if	$Q1 \rightarrow \text{if (01) } Q12$
	id	
	input	$Q12 \rightarrow S$
	print	
Q12	return	
	{	$Q12 \rightarrow Q13 Q14$

Q13	{	$Q13 \rightarrow \{ C \}$
Q14	do	$Q14 \rightarrow \lambda$
	function	
	id	
	if	
	input	
	let	
	print	
	return	
	else	$Q14 \rightarrow \text{else } Q13$
Q2	let	$Q2 \rightarrow \text{let id T ;}$
Q3	id	$Q3 \rightarrow S$
	input	
	print	
	return	
Q4	do	$Q4 \rightarrow Q5$
Q5	do	$Q5 \rightarrow \text{do } Q13 \text{ while (01)}$
S	id	$S \rightarrow S1$
	print	$S \rightarrow S3$
	input	$S \rightarrow S4$
	return	$S \rightarrow S5$
S1	id	$S1 \rightarrow \text{id } S2$
S2	-=	$S2 \rightarrow \text{-- } 01 ;$
	=	
	($S2 \rightarrow (E1) ;$
S3	print	$S3 \rightarrow \text{print } 01 ;$

S4	input	$S4 \rightarrow \text{input id} ;$
S5	return	$S5 \rightarrow \text{return S6} ;$
S6	(
	cadena	
	entero	
	false	$S6 \rightarrow 01$
	id	
	true	
	;	$S6 \rightarrow \lambda$
T	boolean	$T \rightarrow \text{boolean}$
	int	$T \rightarrow \text{int}$
	string	$T \rightarrow \text{string}$

7. Analizador Semántico:

Principalmente, el analizador semántico se encarga de comprobar que el token que se ha devuelto por el analizador léxico tenga sentido en el contexto del lenguaje y, si es correcto, se introduce en una tabla de símbolos que dependerá del instante de ejecución del procesador, ya sea la tabla global o alguna local en caso de que esté procesando el interior de una función.

Algunos ejemplos de dicha comprobación son los siguientes. Comprobación de tipos, comprobación de unicidad, es decir, que las variables no estén declaradas más de una vez, comprobación de nombres, por ejemplo, que las etiquetas del do-while sean diferentes o comprobación del flujo de control, es decir, que return no aparezca fuera de una función.

7.1. Diseño de la Gramática de Atributos:

Para la gramática de atributos se ha usado tanto atributos sintéticos como atributos heredados ya que, en este caso, al tener un analizador descendente recursivo, se puede implementar ambos tipos de atributos sin dificultad añadida.

Además de los atributos clásicos como T.tipo, se necesitaban otros más específicos para cubrir algunas características del lenguaje, como, por ejemplo, el atributo return de algunas reglas semánticas, el cual sirve para comprobar el flujo de control de una función.

Con los tipos definidos se quedan los siguientes atributos para cada no terminal de la gramática:

- *Atributos generales* : zonaDecl , despl
- Oi : varTipo, tipo; $i \in \{ 1,2,3,4,5,6\}$
- $OiOj$: varTipo, tipo; $i \in \{ 1,2,3,4,5,6\}$
- E : varTipo, tipo;
- EE : varTipo, tipo, params, numParams;
- Ei : varTipo, tipo, params, numParams; $i \in \{ 1,2 \}$
- Si : varTipo, tipo, params, numParams; $i \in \{ 1,2,3,4,6 \}$
- $S5$: func, s2, varTipo, varTipo, tipo, params, numParams;
- S : tipo, return;
- Qi : tipo; $i \in \{ 2,3,4,5\}$
- $Q1$: tipo, funcLex;
- Qlj : tipo, return; $j \in \{ 2,3,4\}$
- T : tipo, despl;
- Q : tipo, return;
- F : tipo;
- FF : varLex, tipo, numParams, params, tipoRetorno, etiqRetorno;
- Ki : numParams, params;
- $C =$ tipo, return;
- **Tipos** = {
 - TIPO_OK,
 - TIPO_ERROR,
 - ENTERO,(despl = 1)
 - CADENA,(despl = 64)
 - BOOLEANO,(despl = 1)
 - NULL,
 - VOID,
 - FUNC,
 - SUB_ASIGN,
 - ASIGN}
- **Reglas semánticas** = {

CREAR_TS(NOMBRE_TS) : INDICE_TS

Crea una nueva Tabla de Símbolos con el nombre NOMBRE_TS y devuelve el índice.

DESTRUIR_TS(INDICE_TS) : VOID

Destruye la tabla de símbolos que está en INDICE_TS y no devuelve nada.

BUSCAR_INDICE_TS(NOMBRE_TS) : INDICE_TS

Busca una TS por su nombre y devuelve el índice de la tabla.

BUSCAR_TS(INDICE_TS, LEXEMA) : FILA_TS

Busca y devuelve toda la fila de la tabla situada en INDICE_TS cuyo lexema es igual a LEXEMA.

NOMBRE_TS(INDICE_TS) : CADENA

Devuelve el nombre de la TS que está en INDICE_TS.

ACTUALIZAR_TS(INDICE_TS, LEXEMA, OBJETO) : VOID

Actualiza la fila de la TS que está en INDICE_TS cuyo lexema es LEXEMA con lo que haya dentro de OBJETO y no devuelve nada.

}

7.2. Traducción dirigida por la sintaxis (TDS)

Para la TDS se ha procedido a elegir EDT, por tanto el esquema de traducción queda de la siguiente manera:

Axioma = P1

Terminales = { () id entero cadena ; { } if else do while string let input print return || && > < + * function , = int boolean true false -= }

No Terminales = { P1 P O1 O1O1 O2 O2O2 O3 O3O3 O4 O4O4 O5 O5O5 O6 O6O6 E EE E1 E2 S1 S2 S3 S4 S5 S6 S Q1 Q12 Q13 Q14 Q2 Q3 Q4 T Q5 Q F FF K1 K2 C }

Producciones = {

```

P-> Q P {}
P-> F P {}
P-> lambda {}
01 -> 02 {
    O1O1.varTipo = O2.varTipo
}
0101 {
    O1.varTipo = if( O1O1.varTipo == NULL ) then O2.varTipo ; else O1O1.varTipo
    O1.tipo = if( O2.tipo == tipo_ok && O1O1.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0101 -> || 02 0101_1 {
    O1O1.varTipo = if( O2.varTipo == O1O1.varTipo == BOOLEANO ) then BOOLEANO ; else tipo_error
    O1O1.tipo = if( O2.varTipo == O1O1.varTipo == BOOLEANO ) then tipo_ok ; else tipo_error
    O1O1_1.varTipo = O1O1.varTipo
}
0101 -> lambda {
    O1O1.tipo = tipo_ok
    O1O1.varTipo = NULL
}
02 -> 03 {
    O2O2.varTipo = O3.varTipo
}
0202 {
    O2.varTipo = if( O2O2.varTipo == NULL ) then O3.varTipo ; else O2O2.varTipo
    O2.tipo = if( O3.tipo == tipo_ok && O2O2.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0202 -> && 03 0202_1 {
    O202.varTipo = if( O3.varTipo == 0202.varTipo == BOOLEANO ) then BOOLEANO ; else tipo_error
    O202.tipo = if( O3.varTipo == 0202.varTipo == BOOLEANO ) then tipo_ok ; else tipo_error
    O202_1.varTipo = 0202.varTipo
}
0202 -> lambda {
    O202.tipo = tipo_ok
    O202.varTipo = NULL
}
03 -> 04 {
    O3O3.varTipo = O4.varTipo
}
0303 {
    O3.varTipo = if( O3O3.varTipo == NULL ) then O4.varTipo ; else O3O3.varTipo
    O3.tipo = if( O4.tipo == tipo_ok && O3O3.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0303 -> 04 0303_1 {
    O303.varTipo = if( O4.varTipo == 0303.varTipo == ENTERO ) then BOOLEANO ; else tipo_error
    O303.tipo = if( O4.varTipo == 0303.varTipo == ENTERO ) then tipo_ok ; else tipo_error
    O303_1.varTipo = 0303.varTipo
}
0303 -> lambda {
    O303.tipo = tipo_ok
    O303.varTipo = NULL
}
04 -> 05 {
    O4O4.varTipo = O5.varTipo
}
0404 {
    O4.varTipo = if( O4O4.varTipo == NULL ) then O5.varTipo ; else O4O4.varTipo
    O4.tipo = if( O5.tipo == tipo_ok && O4O4.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0404 -> <5 0404_1 {
    O404.varTipo = if( 05.varTipo == 0404.varTipo == ENTERO ) then BOOLEANO ; else tipo_error
    O404.tipo = if( 05.varTipo == 0404.varTipo == ENTERO ) then tipo_ok ; else tipo_error
    O404_1.varTipo = 0404.varTipo
}
0404 -> lambda {
    O404.tipo = tipo_ok
    O404.varTipo = NULL
}
05 -> 06 {
    O5O5.varTipo = O6.varTipo
}
0505 {
    O5.varTipo = if( O5O5.varTipo == NULL ) then O6.varTipo ; else O5O5.varTipo
    O5.tipo = if( O6.tipo == tipo_ok && O5O5.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0505 -> +6 0505_1 {
    O505.varTipo = if( O6.varTipo == 0505.varTipo == ENTERO ) then ENTERO ; else tipo_error
    O505.tipo = if( O6.varTipo == 0505.varTipo == ENTERO ) then tipo_ok ; else tipo_error
    O505_1.varTipo = 0505.varTipo
}
0505 -> lambda {
    O505.tipo = tipo_ok
    O505.varTipo = NULL
}
06 -> E {
    O6O6.varTipo = E.varTipo
    O6.varTipo = E.varTipo
}
0606 {
    O6.varTipo = if( O6O6.varTipo == NULL ) then E.varTipo ; else O6O6.varTipo
    O6.tipo = if( E.tipo == tipo_ok && O6O6.tipo == tipo_ok ) then tipo_ok ; else tipo_error
}
0606 -> * E 0606_1 {
    O606.varTipo = if( E.varTipo == 0606.varTipo == ENTERO ) then ENTERO ; else tipo_error
    O606.tipo = if( E.varTipo == 0606.varTipo == ENTERO ) then tipo_ok ; else tipo_error
    O606_1.varTipo = 0606.varTipo
}
0606 -> lambda {
    O606.tipo = tipo_ok
    O606.varTipo = NULL
}
E -> id {
    EE.varTipo = BUSCAR_TS( TS_ACTUAL, id.lex ).tipo
    if( !BUSCAR_TS( TS_ACTUAL, id.lex ).tipo == FUNC ) then {
        EE.numParams = BUSCAR_TS( TS_ACTUAL, id.lex ).numParams
        EE.params = BUSCAR_TS( TS_ACTUAL, id.lex ).params
    } else {
        EE.numParams, EE.params = NULL
    }
}
EE {
    E.varTipo = if( BUSCAR_TS( TS_ACTUAL, id.lex ).tipo == FUNC ) then BUSCAR_TS( TS_ACTUAL, id.lex ).tipoRetorno ; else BUSCAR_TS( TS_ACTUAL, id.lex ).tipo
    E.tipo = EE.tipo
}
E -> ( 01 ) {
    E.varTipo = O1.varTipo
    E.tipo = O1.tipo
}
E -> entero {
    E.tipo = tipo_ok
    E.varTipo = ENTERO
}
E -> cadena {
    E.tipo = tipo_ok
    E.varTipo = CADENA
}
S4 -> input id : {
    S4.tipo = if( BUSCAR_TS( TS_ACTUAL, id.lex ).tipo == ENTERO || BUSCAR_TS( TS_ACTUAL, id.lex ).tipo == CADENA ) then tipo_ok ; else tipo_error
}
S5 -> return S6 : {
    S5.func = NOMBRE_TS( TS_ACTUAL )
    S5.tipo = if( S5.func != "GLOBAL" ) then
                if( BUSCAR_TS( BUSCAR_INDICE_TS( "GLOBAL" ), S5.func ).tipoRetorno == S6.tipo ) then
                    tipo_ok ; else tipo_error
                else tipo_error
}
S6 -> 01 {
    S6.tipo = O1.tipo
    S6.varTipo = O1.varTipo
}
S -> S1 {
    S.tipo = S1.tipo
    S.return = FALSE
}
S -> S3 {
    S.tipo = S3.tipo
    S.return = FALSE
}
S -> S4 {
    S.tipo = S4.tipo
    S.return = FALSE
}
S -> S5 {
    S.tipo = S5.tipo
    S.return = TRUE
}
Q1 -> if( 01 ) Q12 {
    Q1.tipo = if( O1.tipo == bool ) then Q12.tipo ; else tipo_error
    Q12.funcLex = NULL
}
Q12 -> S {
    Q12.tipo = S.tipo
    Q12.return = S.return
}
Q12 -> Q13 Q14 {
    Q12.tipo = if( Q12.tipo == tipo_ok && Q14.tipo == tipo_ok ) then tipo_ok ; else tipo_error
    Q12.return = Q13.return && Q14.return
}
Q14 -> else Q13 {
    Q14.tipo = Q13.tipo
    Q14.return = Q13.return
}
Q14 -> lambda {
    Q14.tipo = tipo_ok
    Q14.return = FALSE
}
Q13 -> { C } {
    Q13.tipo = C.tipo
    Q13.return = C.return
}
Q2 -> let {
    zonaDecl = true
}
id T : {
    zonaDecl = false
    ACTUALIZAR_TS( TS_ACTUAL, id.lex, {
        tipo = T.tipo
    })
}
Q3 -> S {
    Q3.tipo = S.tipo
}
Q4 -> Q5 {
    Q4.tipo = Q5.tipo
}
T -> string {
    T.tipo = CADENA
}
T -> int {
    T.tipo = ENTERO
}
T -> boolean {
    T.tipo = BOOLEANO
}
Q5 -> do Q13 while( 01 ) : {
    Q5.tipo = if( Q13.tipo == tipo_ok && O1.tipo == tipo_ok ) then tipo_ok ; else tipo_error
    Q5.return = Q13.return
}
Q -> Q1 {
    Q.tipo = Q1.tipo
    Q.return = Q1.return
}
Q -> Q2 {
    Q.tipo = Q2.tipo
}
Q -> Q3 {
    Q.tipo = Q3.tipo
}
Q -> Q4 {
    Q.tipo = Q4.tipo
}
F -> function {
    zonaDecl = true
}
id {
    F.tipo = FUNC
    FF.varLex = id.lex
    TS_ACTUAL = CREAR_TS( FF.varLex )
}
FF {
    F.tipo = if( FF.tipo == tipo_ok ) then FUNC ; else tipo_error
}
FF -> T( K1 ) {
    ACTUALIZAR_TS( 0, FF.varLex, {
        tipo = FUNC
        numParams = K1.numParams
        params = K1.params
        tipoRetorno = T.tipo
        etiqRetorno = FF.varLex
    })
    zonaDecl = false
}
Q13 {
    FF.tipo = if( Q13.return ) then tipo_ok ; else tipo_error
    DESTRUIR_TS( TS_ACTUAL )
    TS_ACTUAL = BUSCAR_INDICE_TS( "GLOBAL" )
}

```

```

E -> true {
    E.tipo = tipo_ok
    E.varTipo = BOOLEANO
}
E -> false {
    E.tipo = tipo_ok
    E.varTipo = BOOLEANO
}
EE -> ( E1 ) {
    EE.tipo = if( E1.numParams == EE.numParams && E1.params == EE.params ) then tipo_ok ; else E1.tipo
}
EE -> lambda {
    EE.tipo = tipo_ok
}
E1 -> 01 E2 {
    E1.tipo = if( O1.tipo == tipo_ok && E2.tipo == tipo_ok ) then tipo_ok ; else tipo_error
    E1.numParams = 1 + E2.numParams
    E1.params = [O1.tipo, ...E2.params]
}
E1 -> lambda {
    E1.tipo = tipo_ok
    E1.numParams = 0
    E1.params = []
}
E2 -> , 01 E2_1 {
    E2.tipo = if( O1.tipo == tipo_ok && E2_1.tipo == tipo_ok ) then tipo_ok ; else tipo_error
    E2.numParams = 1 + E2_1.numParams
    E2.params = [O1.tipo, ...E2_1.params]
}
S1 -> id {
    id = BUSCAR_TS( TS_ACTUAL, id.lex )
    if( id.tipo == FUNC ) then {
        S2.numParams = id.numParams
        S2.params = id.params
    } else {
        S2.numParams = tipo_ok
        S2.params = tipo_ok
    }
}
S2 {
    S1.tipo = if( S2.tipo == FUNC ) then
        if( id.tipo == FUNC ) then tipo_ok ; else tipo_error
    else if( S2.tipo == ASIGN ) then
        if( id.tipo == S2.varTipo ) then tipo_ok ; else tipo_error
    else if( S2.tipo == SUB_ASIGN ) then
        if( id.tipo == ENTERO && S2.varTipo == ENTERO ) then tipo_ok ; else tipo_error
    else tipo_error
}
S2 -> = 01 ;
S2.tipo = ASIGN
S2.varTipo = O1.tipo
}
S2 -> ( E1 ) ; {
    S2.tipo = if( S2.numParams == E1.numParams AND S2.params == E1.params ) then FUNC ; else tipo_error
    S2.varTipo = tipo_ok
}
S3 -> point 01 ;
S3.tipo = if( O1.tipo == ENTERO || O1.tipo == CADENA ) then tipo_ok ; else tipo_error
}

FF -> ( K1 ) {
    ACTUALIZAR_TS(0, FF.varLex, {
        tipo = FUNC
        numParams = K1.numParams
        params = K1.params
        tipoRetorno = VOID
        etipoRetorno = FF.varLex
    })
    zonaDecl = false
}
Q13 {
    FF.tipo = tipo_ok
    DESTRUIR_TS( TS_ACTUAL )
    TS_ACTUAL = BUSCAR_INDICE_TS( "GLOBAL" )
}
K1 -> T id K2 {
    ACTUALIZAR_TS( TS_ACTUAL, id.lex, {
        tipo = T.tipo
    })
    K1.numParams = 1 + K2.numParams
    K1.params = [T.tipo, ...K2.params]
}
K1 -> lambda {
    K1.numParams = 0
    K1.params = []
}
K2 -> , T id K2_1 {
    ACTUALIZAR_TS( TS_ACTUAL, id.lex, {
        tipo = T.tipo
    })
    K2.numParams = 1 + K2_1.numParams
    K2.params = [T.tipo, ...K2_1.params]
}
K2 -> lambda {
    K2.numParams = 0
    K2.params = []
}
C -> Q C_1 {
    C.tipo = if( Q.tipo == tipo_ok && C_1.tipo == tipo_ok ) then tipo_ok ; else tipo_error
    C.return = Q.return || C_1.return
}
C -> lambda {
    C.tipo = tipo_ok
    C.return = FALSE
}
E2 -> lambda {
    E2.tipo = tipo_ok
    E2.numParams = 0
    E2.params = []
}
S2 -> = 01 ;
S2.tipo = SUB_ASIGN
S2.varTipo = O1.tipo
}
P1 -> {
    zonaDecl = true
    despl = 0
    TS_ACTUAL = BUSCAR_INDICE_TS( "GLOBAL" )
}
P {
    zonaDecl = false
    DESTRUIR_TS( TS_ACTUAL )
}
S6 -> lambda {
    S6.tipo = VOID
}
}

```

8. Tabla de Símbolos (TS)

La tabla de símbolos permite almacenar y consultar la información relevante del programa fuente. Su funcionamiento es bastante intuitivo. Las variables globales y declaración de funciones se almacenan en la tabla global de forma que de cada identificador podemos encontrar todos sus atributos buscándolo por la dirección en la tabla de símbolos que nos da el propio token. Por otro lado, por cada función se crea una tabla local en la que se añadirán los identificadores del cuerpo de la función así como los parámetros de la susodicha. Así podemos analizar el comportamiento del programa tanto del programa general como de cada una de las funciones independientemente.

En cuanto a los atributos de la tabla, se han usado los predefinidos por el enunciado de la práctica que son: Tipo, desplazamiento, número de argumentos, lista de los tipos de los argumentos, tipo que tiene que retornar y una etiqueta en caso de ser una función.

Todos los modelos del procesador pueden o necesitan acceder al contenido de la tabla para su correcto funcionamiento. Por ejemplo, el Analizador Semántico accede a la tabla para comprobar que una variable no se declara dos veces o que se ha llamado a una función con un número de argumentos incorrecto.

8.1. Diseño de la Tabla de Símbolos

El diseño de la Tabla de Símbolos está creado de la siguiente manera. Se han tenido en cuenta las especificaciones demandadas en la práctica en relación a la tabla de símbolos y se ha elegido un formato compatible con la librería TS-20O6 de Tabla de Símbolos. Esta librería está disponible en la página web de la asignatura. La estructura se verá mejor más adelante en los distintos casos de prueba.

En particular, las tablas están definidas así.

TABLA GLOBAL #0 :

- *LEXEMA : 'lexema id1'*
+*nombre atributo1 : valor atr1*
...
+*nombre atributo i : valor atr i*
...

- *LEXEMA : 'lexema id j'*
...
+*nombre atributo k : valor atr k*

TABLA 'lexema id funcion r' #i :

- *LEXEMA : 'lexema id1'*
+*nombre atributo1 : valor atr1*
...
+*nombre atributo s : valor atr s*
...
- *LEXEMA : 'lexema id u'*
...
+*nombre atributo v : valor atr v*

Con $i, j, k, r, s, u, v \in \mathbb{N}$

9. Gestor de Errores

El Gestor de Errores se ha implementado por y para el usuario, se ha creado un Gestor enriquecido visualmente, y que aporta bastante información entendible por el usuario para que este sepa por qué, dónde produce el error, ya sea este léxico, sintáctico o semántico. Además en cada error se ha incluido un mensaje que funciona como “tip” comunicando al programador lo que el procesador esperaba encontrar y lo que ha encontrado en cambio.

El GDE implementado está almacenado en la carpeta ‘*errors*’ del código adjunto. Dicha carpeta contiene 4 archivos, uno para cada módulo (léxico, sintáctico y semántico) y otro que funciona como coordinador entre los tres módulos.

Por ejemplo, si el error lo ha detectado el Analizador Semántico, el GDE indicará al usuario que el error es léxico, en qué línea del archivo fuente se ha producido el error y una breve recomendación sobre cómo arreglarlo, o indicación de por qué se ha cometido dicho error, un ejemplo de un mensaje de error sería:

[ERROR SEMÁNTICO] Caracter inesperado '}' en la linea 27:0.

Parece que hay un error debido a que existe el riesgo de que la función no devuelva nada a pesar de no ser de tipo void.

10. Pruebas de Ejecución

A continuación se mostrarán 10 ejemplos de código, 5 con y otros 5 sin errores, para corroborar el funcionamiento del procesador implementado.

10.1 Pruebas sin errores:

En cada ejecución del código creado se añadirá el listado de Tokens, el árbol de análisis sintáctico generado por el programa VASy y el volcado de la Tabla de Símbolos.

10.1.1 Prueba 1:

Se pretende analizar el siguiente programa:

/ Se necesita comprobar para el correcto funcionamiento del procesador
las siguientes cosas a mencionar
/

/ Declaraciones, funciones, tipos enteros, logicos y cadenas,
variables y su declaracion, sentencias, expresiones
comentarios,
operadores de todo tipo*
*/

```

let x int; /* Sean la dos primeras variables */
let y int;

/* Se procede a crear una funcion que devuelve la suma de la variable y otro numero */
function sumaPrueba1 int (int n)
{
    if(x < 0)      return x;
    return (x+10);
    /* Return básico */

}
/* Funcion con las 2 variables */

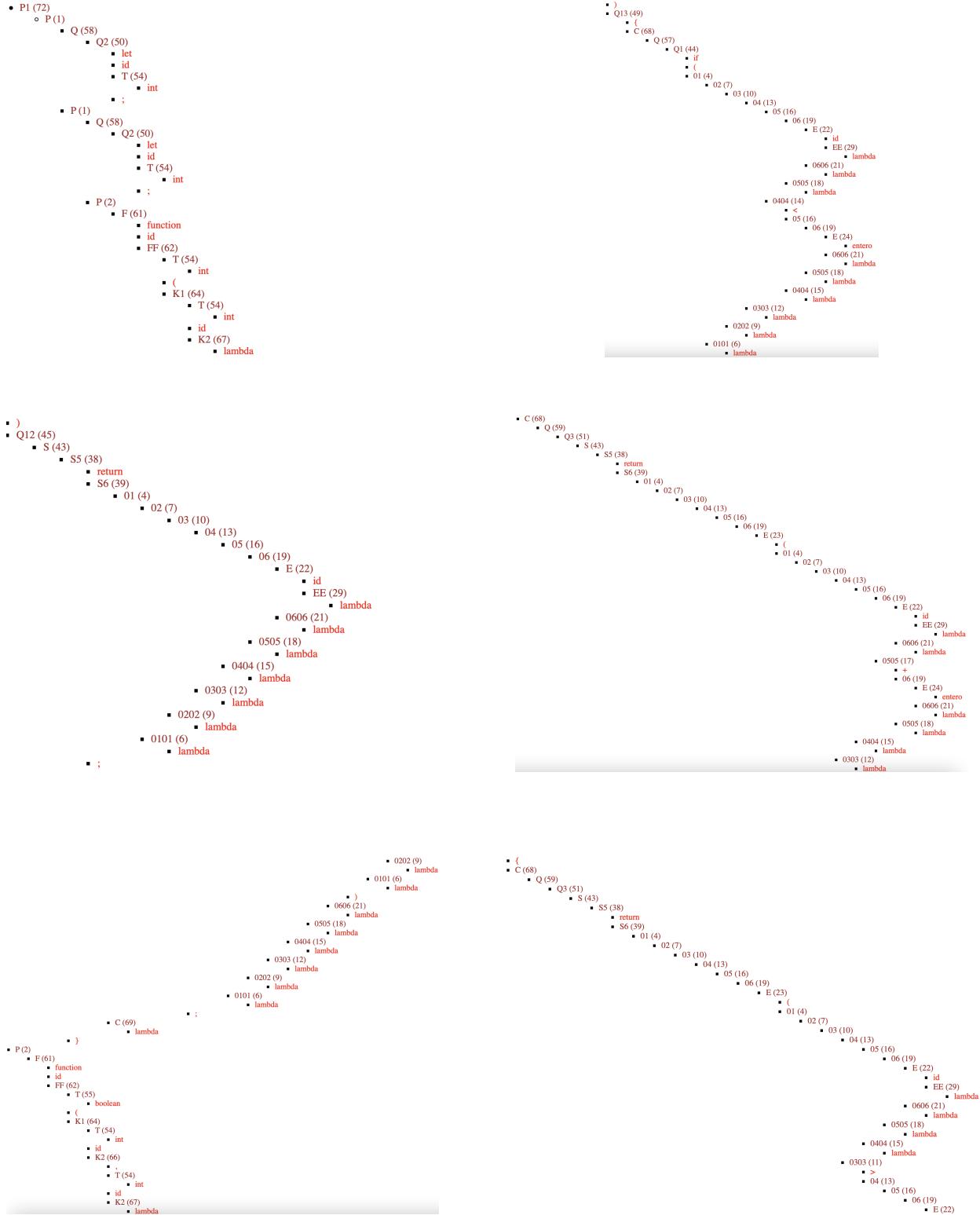
function sumaTriple boolean (int x, int y) {
    return (x > y);
}
sumaPrueba1(x);

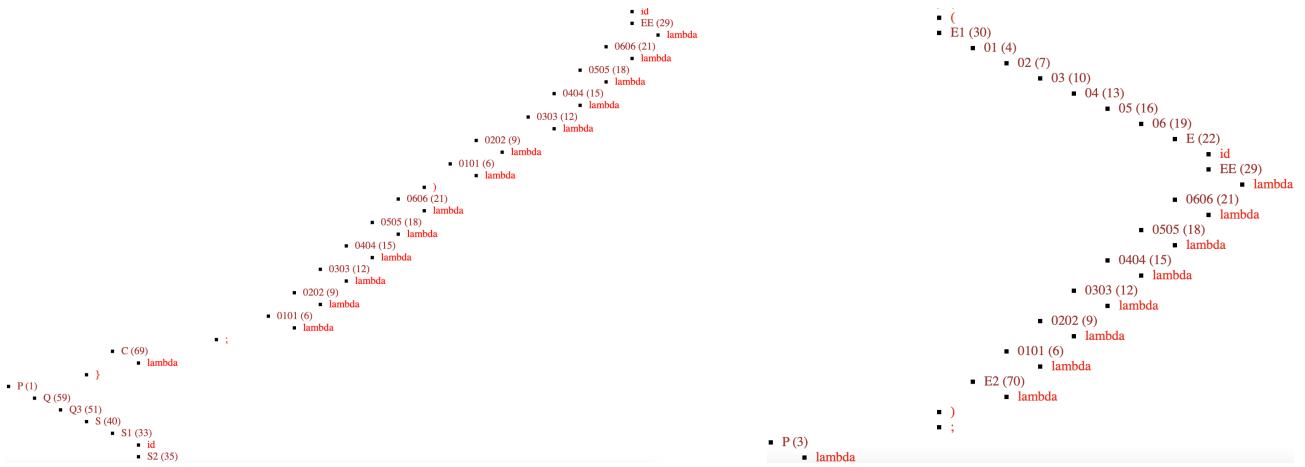
```

El volcado de los Tokens es:

<LET,>	<INTEGER, 10>
<ID,0>	<PARC,>
<INT,>	<SCOL,>
<SCOL,>	<BRKC,>
<LET,>	<FUNC,>
<ID,1>	<ID,3>
<INT,>	<BOOL,>
<SCOL,>	<PARA,>
<FUNC,>	<INT,>
<ID,2>	<ID,0>
<INT,>	<COMMA,>
<PARA,>	<INT,>
<INT,>	<ID,1>
<ID,0>	<PARC,>
<PARC,>	<BRKA,>
<BRKA,>	<RET,>
<IF,>	<PARA,>
<PARA,>	<ID,0>
<ID,0>	<GRT,>
<LOW,>	<ID,1>
<INTEGER,0>	<PARC,>
<PARC,>	<SCOL,>
<RET,>	<BRKC,>
<ID,0>	<ID,2>
<SCOL,>	<PARA,>
<RET,>	<ID,0>
<PARA,>	<PARC,>
<ID,0>	<SCOL,>
<SUM,>	

El volcado del Árbol Sintáctico es:





El volcado de las Tablas de Símbolos es:

```

TABLA GLOBAL #0 :
* LEXEMA : 'x'
+ tipo : 'ENTERO'
+ despl : 0
* LEXEMA : 'y'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'sumaPrueba1'
+ tipo : 'FUNC'
+ numParam : 1
+ TipoParam1 : 'ENTERO'
+ TipoRetorno : 'ENTERO'
+ EtiqFuncion : '_Et_sumaPrueba1'
* LEXEMA : 'sumaTriple'
+ tipo : 'FUNC'
+ numParam : 2
+ TipoParam1 : 'ENTERO'
+ TipoParam2 : 'ENTERO'
+ TipoRetorno : 'BOOLEANO'
+ EtiqFuncion : '_Et_sumaTriple'
TABLA sumaPrueba1 #1 :
* LEXEMA : 'n'
+ tipo : 'ENTERO'
+ despl : 0
TABLA sumaTriple #2 :
* LEXEMA : 'x'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'y'
+ tipo : 'ENTERO'
+ despl : 0

```

10.1.2 Prueba 2:

Se pretende analizar el siguiente programa:

```

function imprimir int (boolean l1, string cad, int x, int y) {
    l1 = (x > y);
    if(l1) {
        print 'Es cierta la implicacion';
    }
    return x;
}

```

```
function mensaje string () {  
    return 'Mensaje de prueba';
```

}

```

/* variables */

let z int;
let w int;

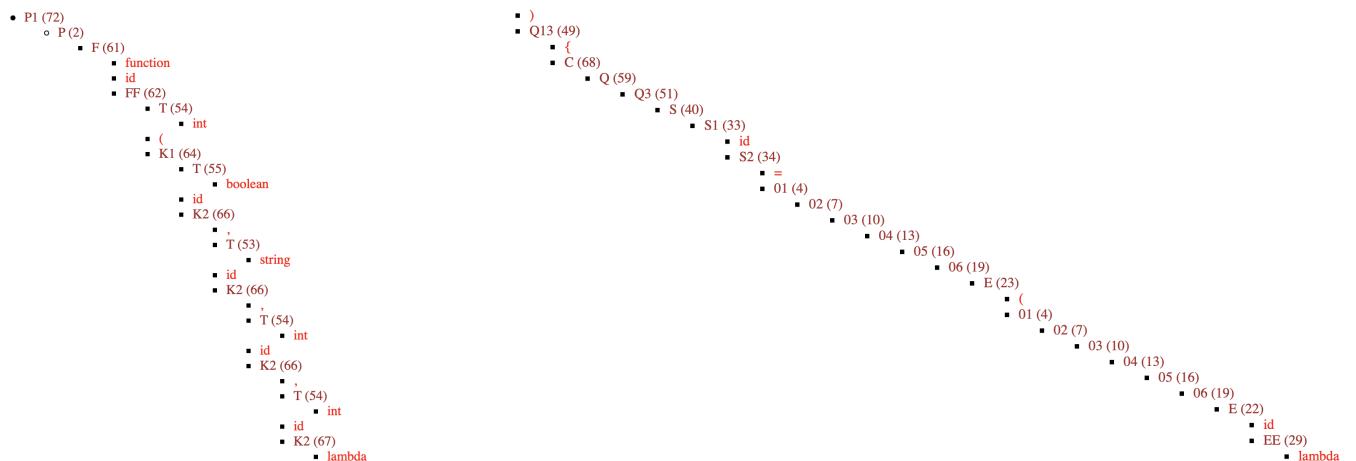
function operacion int (int z, int w)
{
    z -= w;
    return z;
}

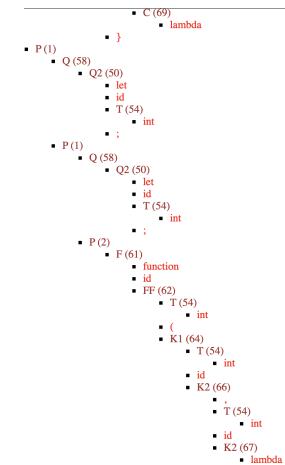
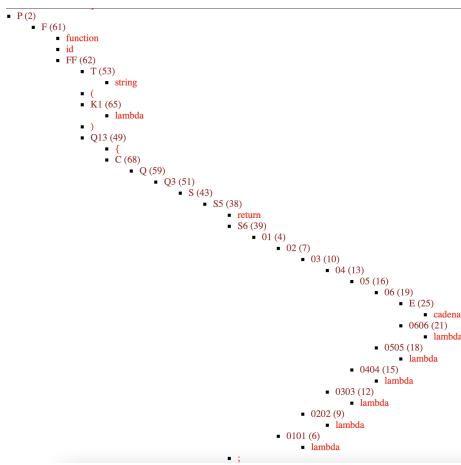
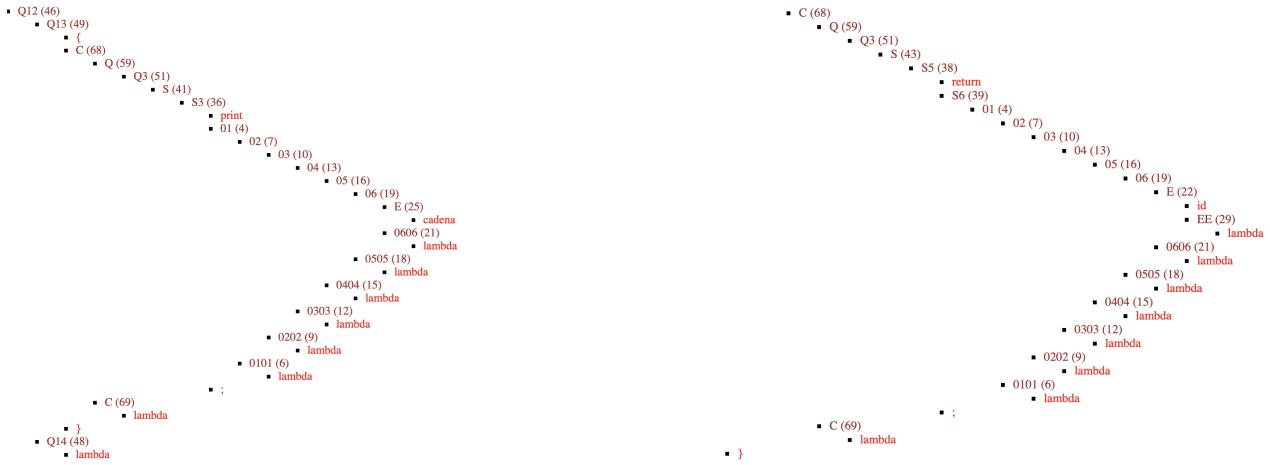
```

El volcado de los Tokens es:

```
<FUNC,>
<ID, 0>
<INT,>
<PARA,>
<BOOL,>
<ID, 0>
<COMMA,>
<STRING,>
<ID, 1>
<COMMA,>
<INT,>
<ID, 2>
<COMMA,>
<INT,>
<ID, 3>
<PARC,>
<BRKA,>
<ID, 0>
<ASIGN,>
<PARA,>
<ID, 2>
<GRT,>
<ID, 3>
<PARC,>
<SCOL,>
<LF,>
<PARA,>
<ID, 0>
<PARC,>
<BRKA,>
<PRNT,>
<CAD,"Es cierta la implicacion">
<SCOL,>
<BRKC,>
<RET,>
<ID, 2>
<SCOL,>
<BRKC,>
<ID, 1>
<SCOL,>
<RET,>
<ID, 0>
<SCOL,>
<BRKA,>
<ID, 0>
<ASIGNSUB,>
<ID, 1>
<SCOL,>
<RET,>
<ID, 0>
<SCOL,>
<BRKC,>
```

El volcado del Árbol Sintáctico es:





El volcado de las Tablas de Símbolos:

```
TABLA GLOBAL #0 :
* LEXEMA : 'imprimir'
+ tipo : 'FUNC'
+ numParam : 4
+ TipoParam1 : 'BOOLEANO'
+ TipoParam2 : 'CADENA'
+ TipoParam3 : 'ENTERO'
+ TipoParam4 : 'ENTERO'
+ TipoRetorno : 'ENTERO'
+ EtiqFuncion : 'Et_imprimir'
* LEXEMA : 'mensaje'
+ tipo : 'FUNC'
+ numParam : 0
+ TipoRetorno : 'CADENA'
+ EtiqFuncion : 'Et_mensaje'
* LEXEMA : 'z'
+ tipo : 'ENTERO'
+ despl : 0
* LEXEMA : 'w'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'operacion'
+ tipo : 'FUNC'
+ numParam : 2
|+ TipoParam1 : 'ENTERO'
|+ TipoParam2 : 'ENTERO'
+ TipoRetorno : 'ENTERO'
+ EtiqFuncion : 'Et_operacion'
TABLA imprimir #1 :
* LEXEMA : 'l1'
+ tipo : 'BOOLEANO'
+ despl : 132
* LEXEMA : 'cad'
+ tipo : 'CADENA'
+ despl : 4
* LEXEMA : 'x'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'y'
+ tipo : 'ENTERO'
+ despl : 0
TABLA mensaje #2 :
TABLA operacion #3 :
* LEXEMA : 'z'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'w'
+ tipo : 'ENTERO'
+ despl : 0
```

10.1.3 Prueba 3

Se pretende analizar el siguiente programa:

```
/* Bloque de codigo 2 */
let x int;
let y int;
if(x > y) {
    print 'Bloque';
}

let q int;

do {
    q -= 1;
} while(true);
```

```

/* Combinacion de todo */

function envuelveTodo int(int x,int y,int q) {
    do {
        print x; /* Se imprime x */
        y -= 1;
    }while(y > q);

    if(x > y && x < q)
    {
        print 'x es mayor que y pero no que q';
        if(x > 0) {
            print 'igualmente x es mayor que 0';
            if(y < 0){
                print 'no tiene sentido';
                if(q > 0){
                    print 'efectivamente';
                }
            }
        }
    }

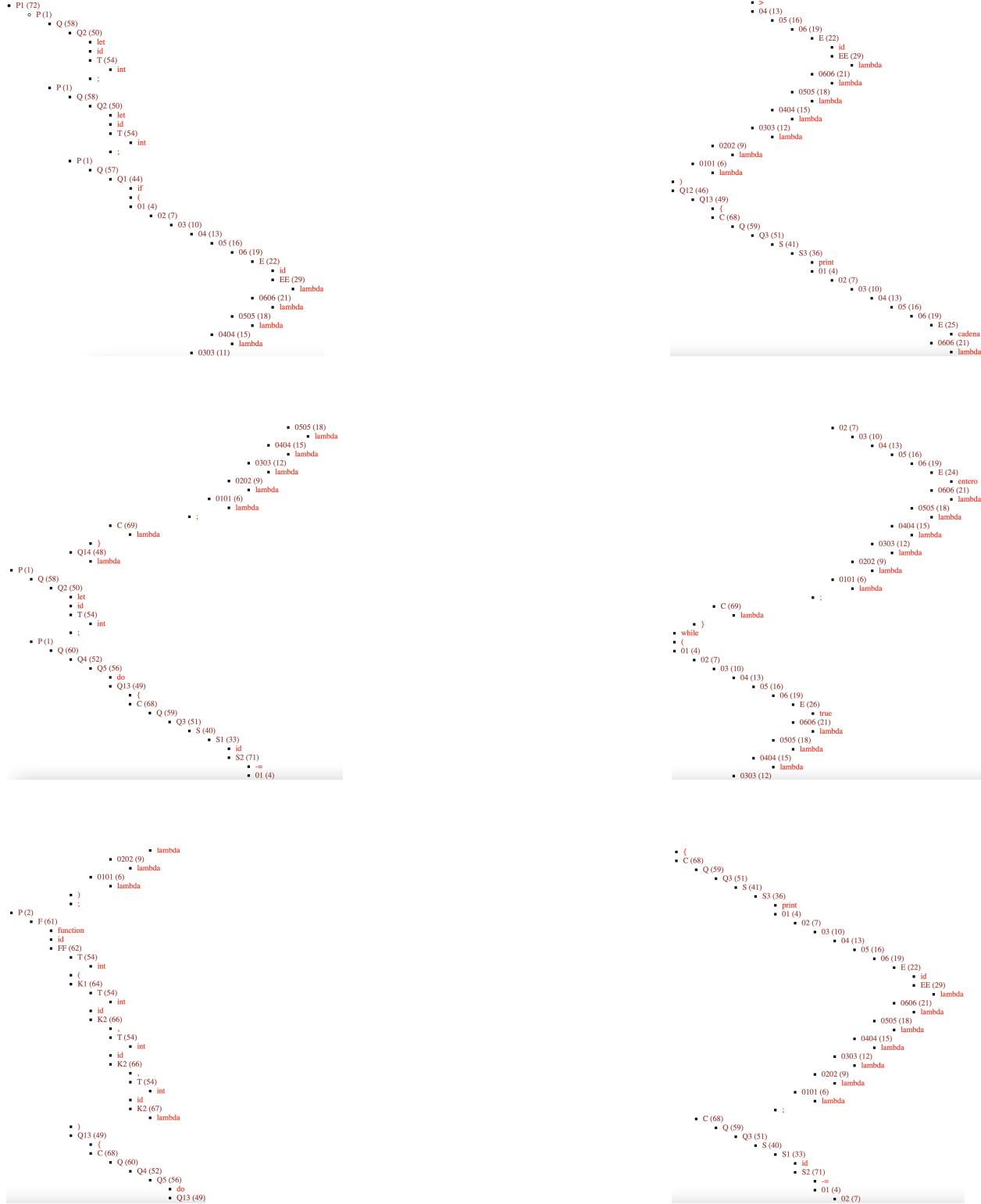
    if(q > y && y > x) print 'Ambos enteros son mayores que x';
    return x;
}

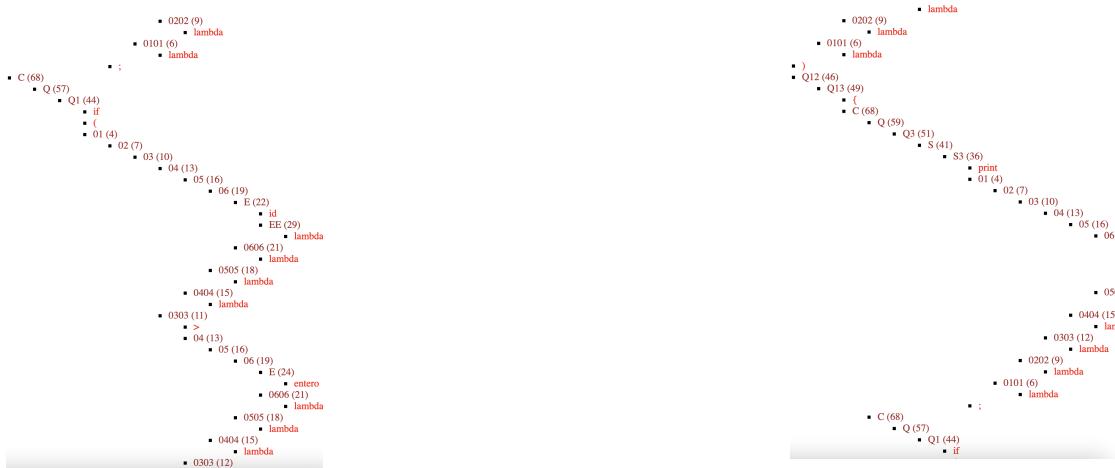
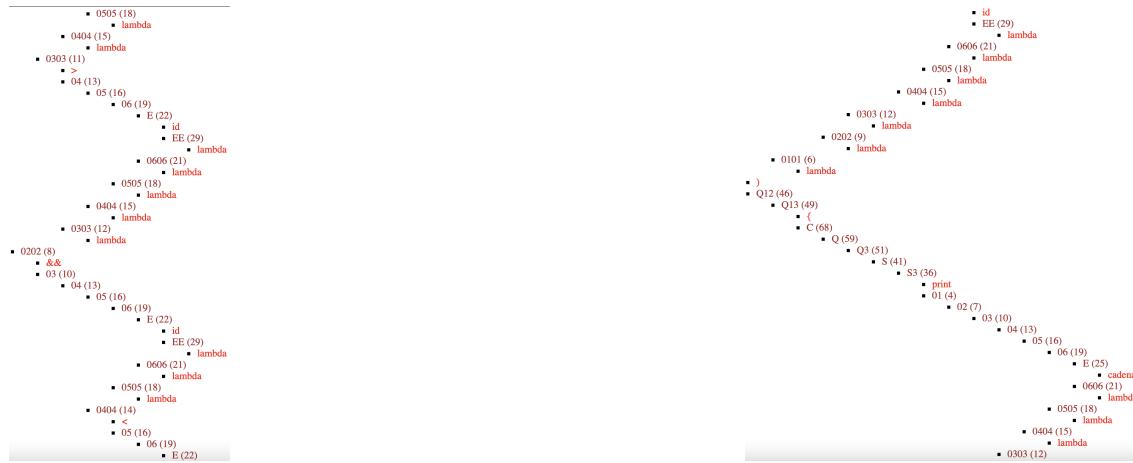
```

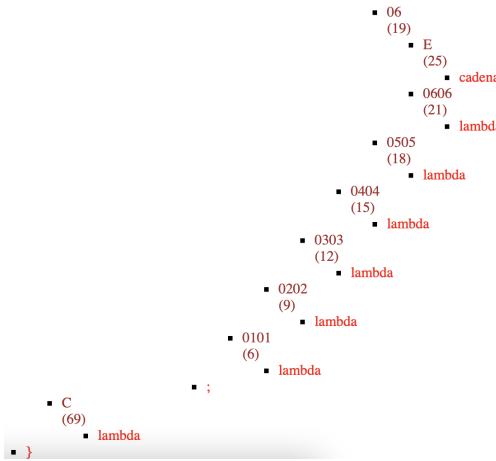
El volcado de los Tokens:

<LET,> <ID,0> <INT,> <SCOL,> <LET,> <ID,1> <INT,> <SCOL,> <IF,> <PARA,> <ID,0> <GRT,> <ID,1> <PARC,> <BRKA,> <PRNT,> <CAD,"Bloque"> <SCOL,> <BRKC,> <LET,> <ID,2> <INT,> <SCOL,> <DO,> <BRKA,> <ID,2> <ASIGNSUB,> <INTEGER,1> <SCOL,> <BRKC,> <WHILE,> <PARA,> <ID,1> <GRT,> <ID,2> <PARC,>	<PARC,> <SCOL,> <FUNC,> <ID,3> <INT,> <PARA,> <INT,> <ID,0> <COMMA,> <INT,> <ID,1> <COMMA,> <INT,> <ID,2> <PARC,> <BRKA,> <PRNT,> <ID,2> <PARC,> <BRKA,> <DO,> <BRKA,> <PRNT,> <ID,0> <GRT,> <INTEGER,0> <SCOL,> <PARA,> <ID,0> <GRT,> <ID,1> <PARC,> <BRKA,> <PRNT,> <ID,1> <SCOL,> <BRKC,> <WHILE,> <PARA,> <ID,1> <GRT,> <ID,2> <PARC,>	<PRNT,> <CAD,"no tiene sentido"> <SCOL,> <IF,> <PARA,> <ID,2> <GRT,> <INTEGER,0> <PARC,> <BRKA,> <PRNT,> <CAD,"efectivamente"> <SCOL,> <BRKC,> <BRKC,> <BRKC,> <IF,> <PARA,> <ID,2> <GRT,> <ID,1> <AND,> <ID,1> <GRT,> <ID,0> <PARC,> <PRNT,> <CAD,"Ambos enteros son mayores que x"> <SCOL,> <RET,> <ID,0> <SCOL,> <BRKC,>
---	--	---

El volcado del Árbol Sintáctico es:







El volcado de las Tablas de Símbolos:

```
TABLEA GLOBAL #0 :
* LEXEMA : 'x'
+ tipo : 'ENTERO'
+ despl : 0
* LEXEMA : 'y'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'q'
+ tipo : 'ENTERO'
+ despl : 4
* LEXEMA : 'envuelveTodo'
+ tipo : 'FUNC'
+ numParam : 3
+ TipoParam1 : 'ENTERO'
+ TipoParam2 : 'ENTERO'
+ TipoParam3 : 'ENTERO'
+ TipoRetorno : 'ENTERO'
+ EtiqFuncion : 'Et_envuelveTodo'
TABLEA envuelveTodo #1 :
* LEXEMA : 'x'
+ tipo : 'ENTERO'
+ despl : 4
* LEXEMA : 'y'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'q'
+ tipo : 'ENTERO'
+ despl : 0
```

10.1.4 Prueba 4

Se pretende analizar el siguiente programa:

```
function envuelveTodo2 string (int x, string cad1, string cad2) {
    do {
        print cad1;
        x -= 1;
    }while(x > 0);
    return cad1;
}

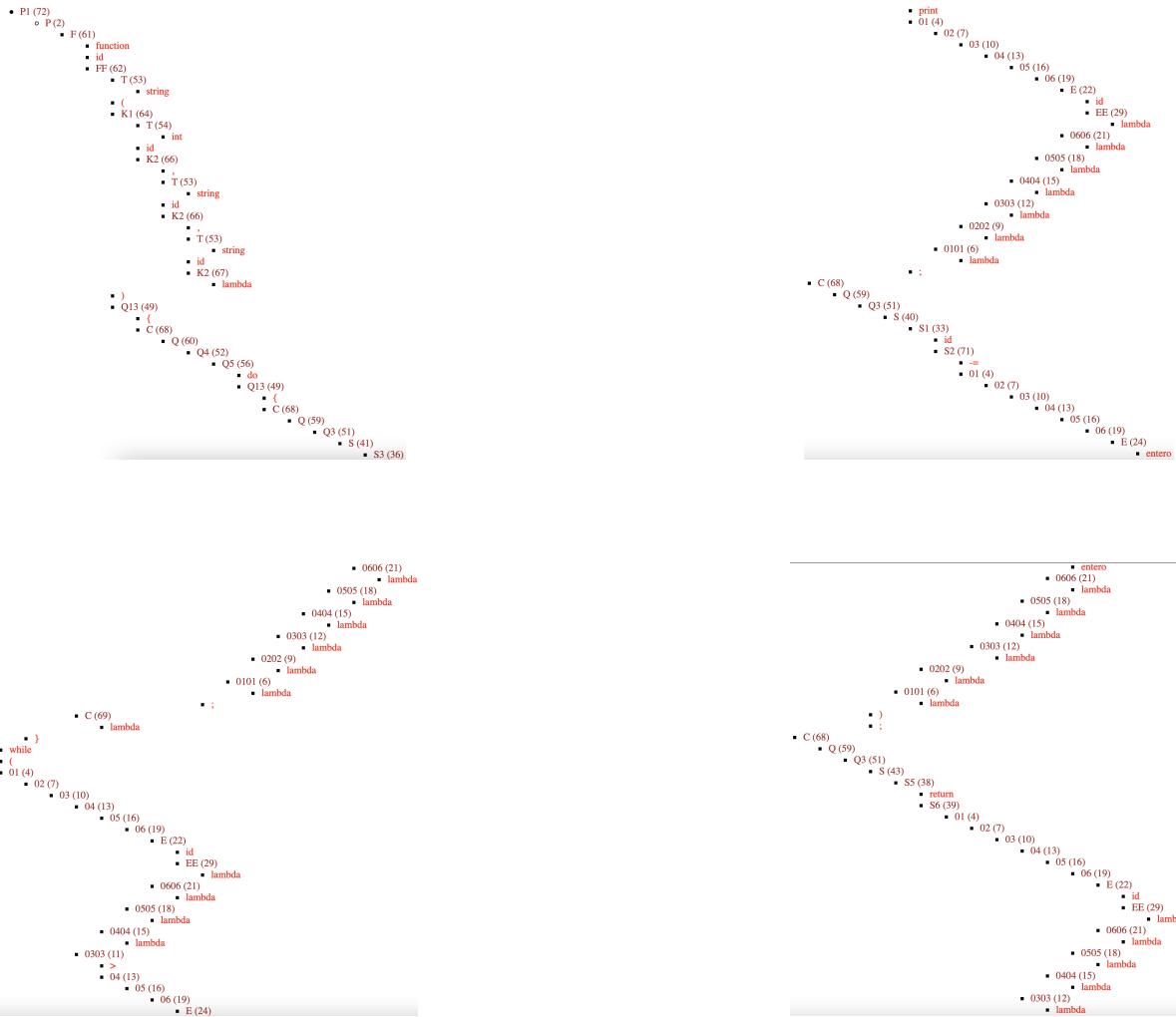
function envuelveTodoRecur int (int x, int y, string cad, boolean l1, boolean l2) {
    let z int;
    do{
        if(x > y && x > 0 && x < 5) {
            print cad;
            print 'x es la mejor variable';
            print 'y no merece la pena';
            x -= z;
            y -= l;
        }
    }while(y > 100);

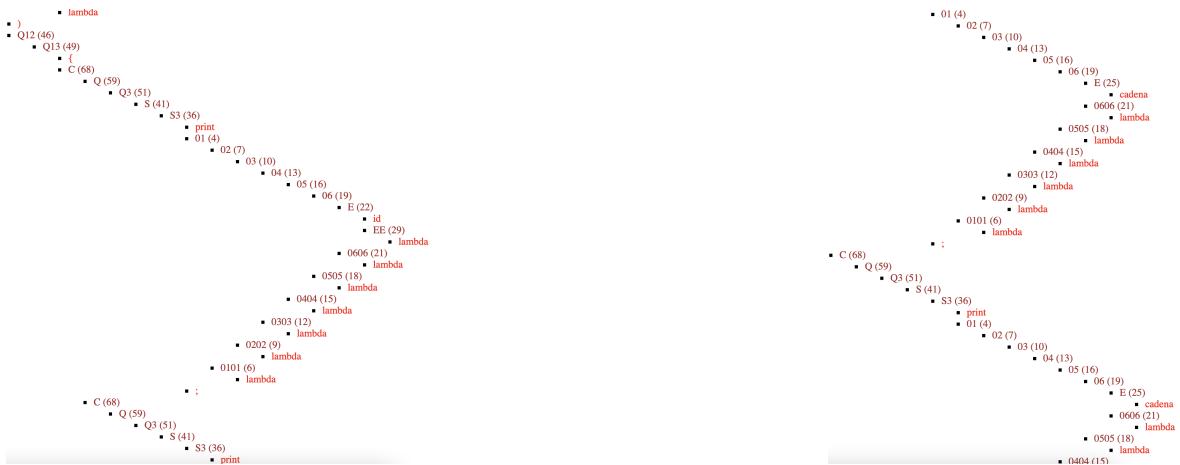
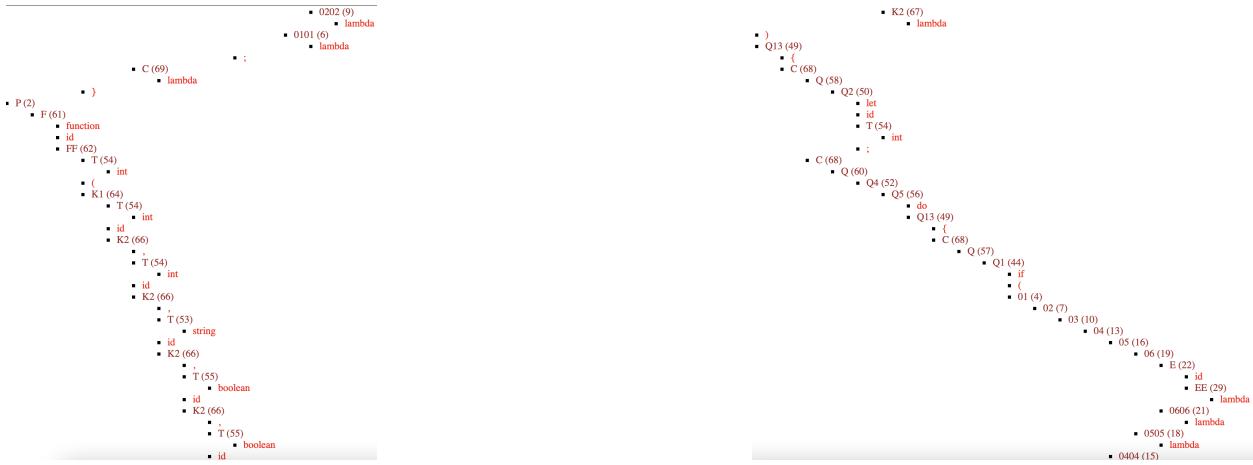
    if(l1) print y;
    if(l2) print x;
    return y;
}
```

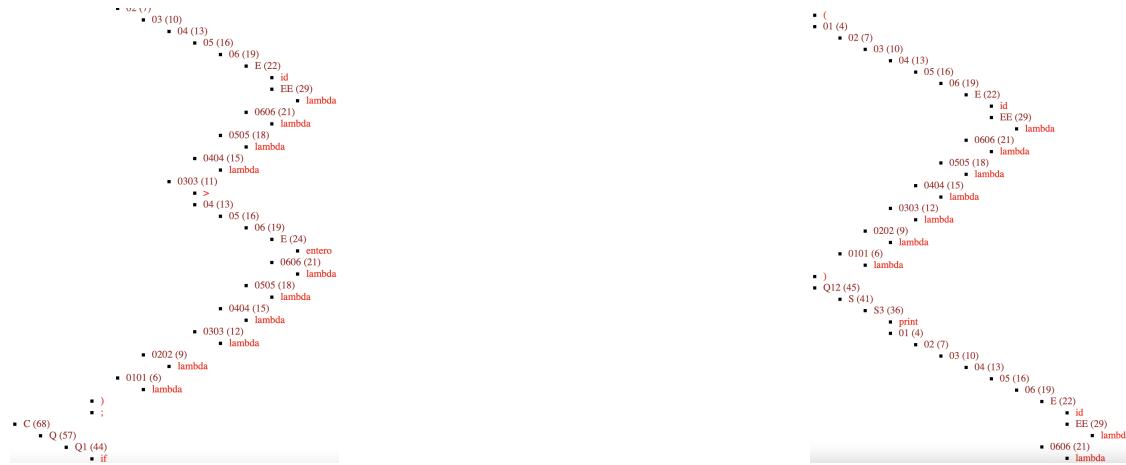
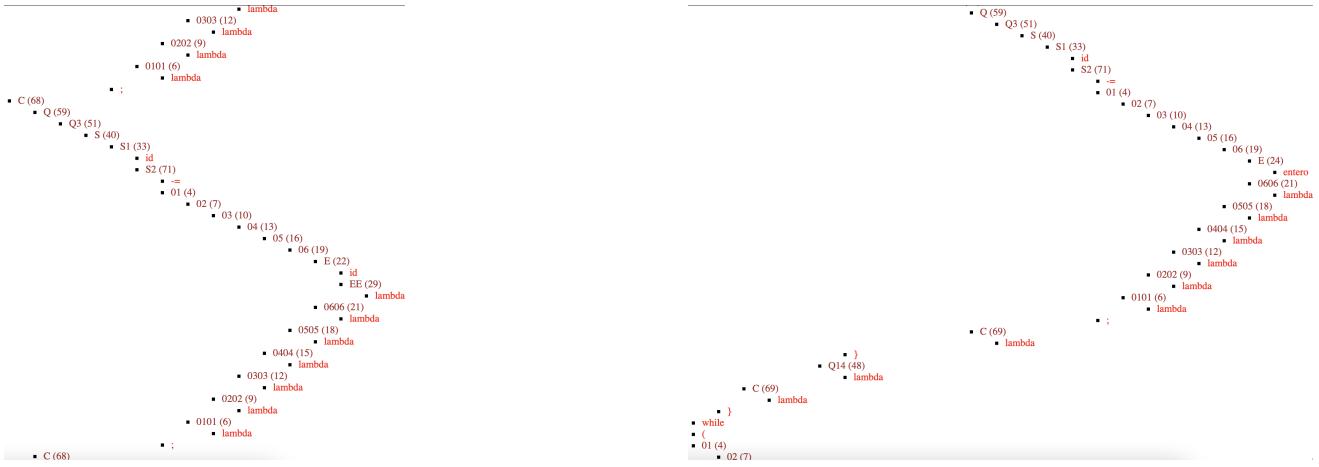
El volcado de Tokens:

<FUNC,>	<SCOL,>	<SCOL,>
<ID,0>	<RET,>	<ID,1>
<STRING,>	<ID,1>	<ASIGNSUB,>
<PARA,>	<SCOL,>	<INTEGER,1>
<INT,>	<BRKC,>	<SCOL,>
<ID,0>	<FUNC,>	<BRKC,>
<COMMA,>	<ID,1>	<WHILE,>
<STRING,>	<INT,>	<PARA,>
<ID,1>	<PARA,>	<ID,1>
<COMMA,>	<INT,>	<GRT,>
<STRING,>	<ID,0>	<INTEGER,100>
<ID,2>	<COMMA,>	<PARC,>
<PARC,>	<INT,>	<SCOL,>
<BRKA,>	<ID,1>	<IF,>
<D0,>	<COMMA,>	<PARA,>
<BRKA,>	<STRING,>	<ID,3>
<PRNT,>	<ID,2>	<PARC,>
<ID,1>	<COMMA,>	<PRNT,>
<SCOL,>	<BOOL,>	<ID,1>
<ID,0>	<ID,3>	<SCOL,>
<ASIGNSUB,>	<COMMA,>	<IF,>
<INTEGER,1>	<BOOL,>	<PARA,>
<SCOL,>	<ID,4>	<ID,4>
<BRKC,>	<PARC,>	<PARC,>
<WHILE,>	<BRKA,>	<PRNT,>
<PARA,>	<ID,5>	<ID,0>
<ID,0>	<LET,>	<SCOL,>
<GRT,>	<ID,5>	<RET,>
<INTEGER,0>	<INT,>	<ID,1>
<PARC,>	<SCOL,>	<SCOL,>

El volcado del Árbol Sintáctico es:







El volcado de las Tablas de Símbolos:

<pre>TABLA GLOBAL #0 : * LEXEMA : 'envuelveTodo2' + tipo : 'FUNC' + numParam : 3 + TipoParam1 : 'ENTERO' + TipoParam2 : 'CADENA' + TipoParam3 : 'CADENA' + TipoRetorno : 'CADENA' + EtiqFuncion : 'Et_envuelveTodo2' * LEXEMA : 'envuelveTodoRecur' + tipo : 'FUNC' + numParam : 5 + TipoParam1 : 'ENTERO' + TipoParam2 : 'ENTERO' + TipoParam3 : 'CADENA' + TipoParam4 : 'BOOLEANO' + TipoParam5 : 'BOOLEANO' + TipoRetorno : 'ENTERO' + EtiqFuncion : 'Et_envuelveTodoRecur' TABLA envuelveTodo2 #1 : * LEXEMA : 'x' + tipo : 'ENTERO' + despl : 256 * LEXEMA : 'cad1' + tipo : 'CADENA' + despl : 128 * LEXEMA : 'cad2' + tipo : 'CADENA' + despl : 0</pre>	<pre>TABLA envuelveTodoRecur #2 : * LEXEMA : 'x' + tipo : 'ENTERO' + despl : 134 * LEXEMA : 'y' + tipo : 'ENTERO' + despl : 132 * LEXEMA : 'cad' + tipo : 'CADENA' + despl : 4 * LEXEMA : 'l1' + tipo : 'BOOLEANO' + despl : 2 * LEXEMA : 'l2' + tipo : 'BOOLEANO' + despl : 0 * LEXEMA : 'z' + tipo : 'ENTERO' + despl : 136</pre>
--	---

10.1.5 Prueba 5

Se pretende analizar el siguiente programa:

```
let prueba int;
let PRUEBA int; /* mayusculas y minusculas */

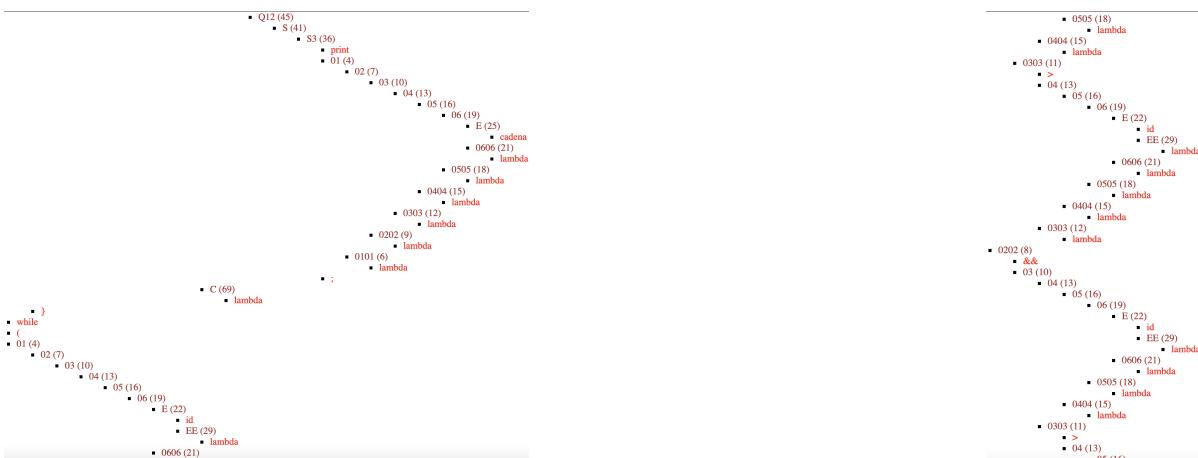
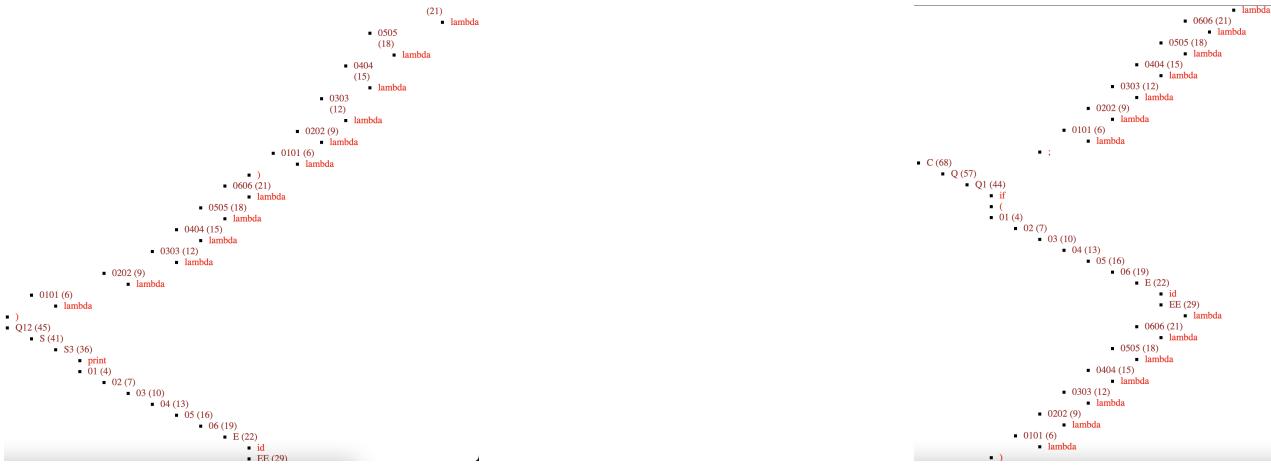
function finalTest string(int a, int b, int c, boolean l2, boolean l3, string cad) {
    let valor int;
    do {
        print a; print 'no sabemos que numero es
        m
        a
        y
        o
        r';
        a -= c;
        c -= b;
        b -= a;
        if(a > c && b < c && c > (a*2) /* comentario aqui */) print valor;
        if(l2) print 'Siempre la verdad';
    }while(b > c && c > a || c < a);
    return cad;
}
```

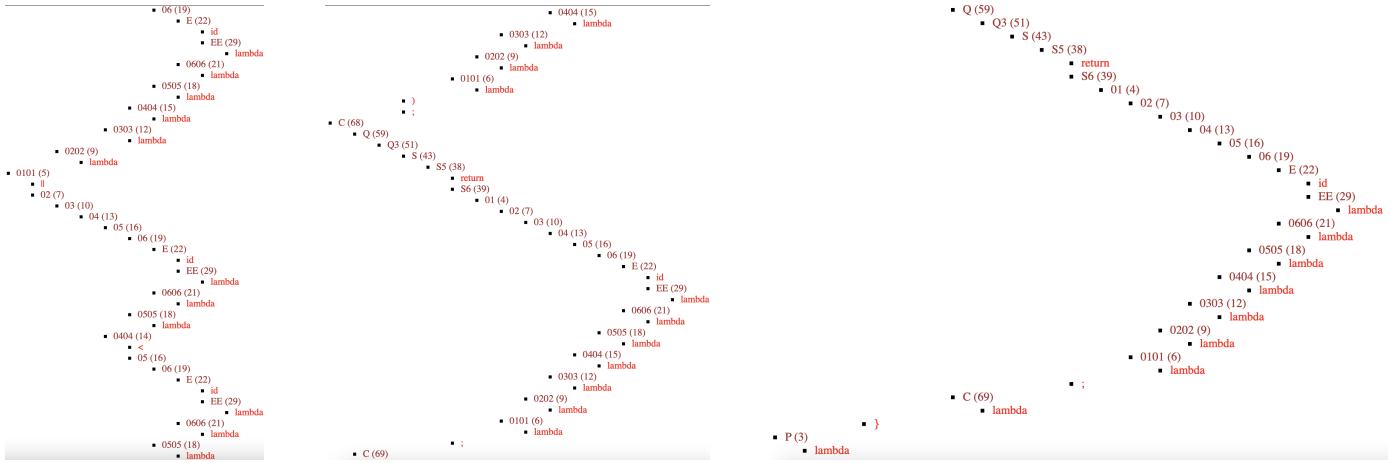
El volcado de Tokens:

<pre><LET,> <ID,0> <INT,> <SCOL,> <LET,> <ID,1> <INT,> <SCOL,> <FUNC,> <ID,2> <STRING,> <PARA,> <INT,> <ID,0> <COMMA,> <INT,> <ID,1> <COMMA,> <INT,> <ID,2> <BOOL,> <ID,3> <COMMA,> <BOOL,> <ID,4> <COMMA,></pre>	<pre><STRING,> <ID,5> <PARC,> <BRKA,> <LET,> <ID,6> <INT,> <SCOL,> <DO,> <BRKA,> <PRNT,> <ID,0> <SCOL,> <PRNT,> <CAD,"no sabemos que numero es m a y o r"> <SCOL,> <ID,0> <ASIGNSUB,> <ID,2> <SCOL,> <ID,0> <ASIGNSUB,> <ID,2> <SCOL,> <ID,2> <ASIGNSUB,></pre>	<pre><ID,1> <SCOL,> <ID,1> <ASIGNSUB,> <ID,0> <SCOL,> <IF,> <PARA,> <ID,0> <GRT,> <ID,2> <AND,> <ID,1> <LOW,> <ID,2> <AND,> <ID,2> <GRT,> <PARA,> <ID,0> <MUL,> <INTEGER,2> <PARC,> <PARC,> <PRNT,> <ID,6> <SCOL,></pre>	<pre><IF,> <PARA,> <ID,3> <PARC,> <PRNT,> <CAD,"Siempre la verdad"> <SCOL,> <BRKC,> <WHILE,> <PARA,> <ID,1> <GRT,> <ID,2> <AND,> <ID,1> <LOW,> <ID,2> <AND,> <ID,2> <GRT,> <ID,0> <OR,> <ID,2> <LOW,> <ID,0> <PARC,> <SCOL,> <RET,> <ID,5> <SCOL,> <BRKC,></pre>
---	---	--	--

El volcado del Árbol Sintáctico es:







El volcado de las Tablas de Símbolos:

```
TABLA GLOBAL #0 :
* LEXEMA : 'prueba'
+ tipo : 'ENTERO'
+ despl : 0
* LEXEMA : 'PRUEBA'
+ tipo : 'ENTERO'
+ despl : 2
* LEXEMA : 'finalTest'
+ tipo : 'FUNC'
+ numParam : 6
+ TipoParam1 : 'ENTERO'
+ TipoParam2 : 'ENTERO'
+ TipoParam3 : 'ENTERO'
+ TipoParam4 : 'BOOLEANO'
+ TipoParam5 : 'BOOLEANO'
+ TipoParam6 : 'CADENA'
+ TipoRetorno : 'CADENA'
+ EtiqFuncion : 'Et_finalTest'
```

```
TABLA finalTest #1 :
* LEXEMA : 'a'
+ tipo : 'ENTERO'
+ despl : 136
* LEXEMA : 'b'
+ tipo : 'ENTERO'
+ despl : 134
* LEXEMA : 'c'
+ tipo : 'ENTERO'
+ despl : 132
* LEXEMA : 'l2'
+ tipo : 'BOOLEANO'
+ despl : 130
* LEXEMA : 'l3'
+ tipo : 'BOOLEANO'
+ despl : 128
* LEXEMA : 'cad'
+ tipo : 'CADENA'
+ despl : 0
* LEXEMA : 'valor'
+ tipo : 'ENTERO'
+ despl : 138
```

10.2 Pruebas con errores

En cada ejecución, la combinación de los tres módulos junto con el gestor de errores interrumpe la ejecución del código en cuanto encuentra un error de cualquiera de los 3 analizadores.

10.2.1 Prueba 1

Se pretende analizar el siguiente programa:

```
/* Se necesita comprobar para el correcto funcionamiento del procesador
   las siguientes cosas a mencionar
   */
/* Declaraciones, funciones, tipos enteros, logicos y cadenas,
   variables y su declaracion, sentencias, expresiones ,
   comentarios,
   operadores de todo tipo
*/
let x int; /* Sean la dos primeras variables */
let y int;

/* Se procede a crear una funcion que devuelve la suma de la variable y otro numero */
function sumaPrueba1 int (int n)
{
    if(x < 0)      return x;
    return (x+10);
    /* Return básico */
}
/* Funcion con las 2 variables */

function sumaTriple boolean (int x, int y) {
    return (x > y);
}
sumaPrueba1(x);
```

Si intentamos procesar este fragmento de código nos salta el siguiente error:

[ERROR SEMÁNTICO] Caracter inesperado ';' en la linea 27:0.

Parece que se ha intentado llamar a una función que necesita 1 argumento y en cambio se le ha llamado con 0 argumentos.

Era lo esperado, puesto que en la línea 27 no hemos pasado como argumento ninguna variable y la función “sumaPrueba” tiene 1 argumento.

10.2.2 Prueba 2

Se pretende analizar el siguiente programa:

```
function imprimir int (boolean l1, string cad, int x, int y) {
    l1 = (x > y);
    if(l1 || ) {
        print 'Es cierta la implicacion';
    }
    return x;
}

function mensaje string () {
    return 'Mensaje de prueba';
}

/* variables */

let z int;
let w int;

function operacion int (int z, int w)
{
    z -= w;
    return z;
}
```

Si intentamos procesar este fragmento de código nos salta el siguiente error:

[ERROR SINTÁCTICO] Carácter inesperado ')' en la linea 2:9.

Es posible que falte algún elemento de los siguientes: Paréntesis de apertura '(', comillas simples ''', número entero, identificador, constante lógica falso 'false', constante lógica verdadero 'true'.

Era lo esperado puesto que el operador lógico “OR” siempre necesita dos operandos mientras que en este caso podemos apreciar que esto no se verifica en el operador OR dentro del “if” de la línea 2.

10.2.3 Prueba 3

Se pretende analizar el siguiente programa:

```
/* Bloque de codigo 2 */
let x int;
let y int;
if(x > y) {
    print 'Bloque';
}

let q int;

do {
    q -= 1;
} while(true);

/* Combinacion de todo */

function envuelveTodo int(int x,int y,int q) {
    do {
        print x; /* Se imprime x */
        y - 1;
    }while(y > q);

    if(x > y && x < q)
    {
        print 'x es mayor que y pero no que q';
        if(x > 0) {
            print 'igualmente x es mayor que 0';
            if(y < 0){
                print 'no tiene sentido';
                if(q > 0){
                    print 'efectivamente';
                }
            }
        }
    }

    if(q > y && y > x) print 'Ambos enteros son mayores que x';
    return x;
}
```

Si intentamos procesar este fragmento de código nos salta el siguiente error:

[ERROR LÉXICO] Caracter inesperado '-' en la linea 18:5.

¿Es posible que quisiese poner `-=`?

Era lo esperado puesto que el Analizador Léxico está esperando que el siguiente carácter sea “=”, por tanto encuentra el error y detiene su ejecución.

10.2.4 Prueba 4

Se pretende analizar el siguiente programa:

```
function envuelveTodo2 string (int x, string cad1, string cad2) {
    do {
        print cad1;
        x -= 1;
    }while(x > 0);
    return cad1;
}

function envuelveTodoRecur int (int x, int y, string cad, boolean l1, boolean l2) {
    let z int;
    do{
        if(x > y && x > 0 && x < 5) {
            print cad;
            print 'x es la mejor variable';
            print 'y no merece la pena';
            x -= z;
            y -= 1;
        }
    }while(y > 100);

    if(l1) print y;
    if(l2) print x;
    return y; (Borrado)
}
```

Si intentamos procesar este fragmento de código nos salta el siguiente error:

[ERROR SEMÁNTICO] Carácter inesperado '}' en la linea 27:0.

Parece que hay un error debido a que existe el riesgo de que la función no devuelva nada a pesar de no ser de tipo void.

Era lo esperado puesto que la función “envuelveTodoRecur” debería devolver un int mientras que no ha sido así.

10.2.5 Prueba 5

Se pretende analizar el siguiente programa:

```
function finalTest string(int a, int b, int c, boolean l2, boolean l3, string cad) {
    let valor int;
    do {
        print a; print 'no sabemos que numero es
        m
        a
        y
        o
        r';
        a -= c;
        c -= b;
        b -= a;
        if(a > c && b < c && c > (a*2) // comentario aqui // ) print valor;
        if(l2) print 'Siempre la verdad';
    }while(b > c && c > a || c < a);
    return cad;
}
```

Si intentamos procesar este fragmento de código nos salta el siguiente error:

[ERROR LÉXICO] Carácter inesperado '/' en la linea 12:34.

¿Es posible que tenga mal escritos los signos de comentarios: /* , */?

Era lo esperado puesto que los comentarios que se nos han encargado son del estilo “/* ... */”, el Analizador Léxico no es capaz de leer el siguiente carácter, haciendo que se interrumpa el programa.