DEPARTAMENTO DE LENGUAJES Y SISTEMAS INFORMATICOS E INGENIERIA DE SOFTWARE (DLSIS)

ETSI INFORMÁTICOS (UPM)

ADRIÁN RAPOSO POZUELO, ÁLVARO LÓPEZ MARTÍNEZ,
CARLOS DE LEGUINA LEÓN

PRÁCTICA PROCESADORES DE LENGUAJES

JAVASCRIPT-PdL

Grupo 81

PROCESADORES DE LENGUAJES

GMI SEMESTRE 5

CURSO 2022-2023

ÍNDICE

1. OBJETIVO

La Práctica consistirá en el diseño y construcción de un Analizador de una versión del lenguaje JavaScript llamado JavaScript-PdL. En esta práctica, recibiremos un fichero de texto de este lenguaje, y nuestro programa detectará si el código recibido está léxica, sintáctica y semánticamente correcto con la ayuda de una Tabla de Símbolos y un Gestor de Errores.

2. LENGUAJE DEL GRUPO

Se nos ha sido asignado las siguientes características del lenguaje JavaScript-Pdl para analizar:

- Sentencias: Sentencia Repetitiva “do - while”
- Operador Especial: Asignación con Suma (‘+=’)
- Técnica de Análisis Sintáctico: Descendente con Tablas
- Comentarios: Comentarios de Bloque (/*…*/)
- Cadenas: con Comillas Dobles (“…”)

Adicionalmente hemos seleccionado:

- Operadores Aritméticos: Suma(‘+’)
- Operadores Lógicos: AND (‘&&’)
- Operadores Relacionales: Igual (‘=’)
- Valores “Booleanos”: “true” y “false”
- Carácter Final de Fichero: “EOF”

3. INTRODUCCIÓN

La Primera Entrega consiste en la implementador del Analizador Léxico y el diseño inicial de la estructura de la Tabla de Símbolos para el lenguaje JavaScript-Pdl. Se ha decido utilizar el lenguaje de programación de Java para la implementación del Procesador de Lenguajes pedido.

Para ello, se han desarrollado siguientes 'Scripts':

- Analizador.java: encargado de ejecutar el resto de 'Scripts' y que recibe el código de JavaScript-Pdl mediante "PruebasTexto.txt" para generar los 'Tokens', la Tabla Global de Símbolos, las Tablas de Símbolos Locales, parse y posibles errores.
- Analizador_Lexico.java: 'Script' que implementa el Analizador Léxico, es decir, realiza la función del Autómata Finito Determinista (AFD) y la identificación de los 'Tokens'.
- Analizador_Sintactico_Semantico.java: 'Script' que implementa el Analizador Sintáctico, es decir, comprueba que la sintaxis del programa es la correcta, mediante el uso de la gramática LL(1) y mediante la implementación del EDT (Esquema de Traducción) que comprueba que la semántica del programa sea correcta.
- GestorTS.java: clase encargada de almacenar y gestionar los 'Tokens' de tipo Identificadores generados en el "Analizador_Lexico.java" y sus respectivos atributos semánticos.
- Token.java: se encarga de crear los 'Tokens'.
- PruebasTexto.txt: fichero de texto que recibe el código de lenguaje JavaScript-Pdl que queremos analizar.
- GestorErrores.java: se encarga de almacenar los errores encontrados en el código y producir el fichero de errores.
- Gui.java: interfaz gráfica creada con la clase "*Swing.java*" que permite un fácil manejo del trabajo.

4. FUNCIONAMIENTO DEL PROGRAMA

Una vez ejecutado el script "Gui.java", el "Analizador.java" recibe el código JavaScript-Pdl a través de PruebasTexto.txt además de crear un directorio donde se almacenarán los ficheros: "Lista de Tokens" ,"Tabla de símbolos" "Errores " y "Parse".

Una vez ejecutado el script anterior, se llama al "Analizador_Sintactico_Semantico.java" (que llama a "Analizador_Lexico.java") para comprobar que la sintaxis y la semántica del código sea la correcta y crea y almacena el fichero "Parse.txt" y con ayuda del Analizador Léxico, el fichero "TS.txt". El "Analizador_Lexico.java" mediante la ayuda del resto de clases, creará y almacenará el fichero de texto "Tokens.txt", los cuales almacenan respectivamente los 'Tokens'.

Si durante la ejecución del programa se detectan errores de código, se devolverá una lista con tales en el fichero pertinente.

5. ANALIZADOR LÉXICO

La tarea del Analizador Léxico será generar los 'Tokens' necesarios o los Errores encontrados.

5.1. TOKENS

Los 'Tokens' son una dupla formados por un Código y un Atributo. En nuestro caso, el Código será numérico y no siembre habrá atributo.

LEXEMA	'TOKEN' ASIGNADO
"boolean"	<1, >
"do"	<2, >
"function"	<3, >
"if"	<4, >
"input"	<5, >
"int"	<6, >
"let"	<7, >
"print"	<8, >
"return"	<9, >
"string"	<10, >
"while"	<11, >
"constante entera"	<12, valor>
"cadena" (")	<13, cadena ("c*") >
"identificador"	<14, posiciónTS >
"+="	<15, >
'='	<16, >
','	<17, >
';'	<18, >
'('	<19, >
')'	<20, >
'{'	<21, >
'}'	<22, >

'+'	<23, >
"&&"	<24, >
"=="	<25, >
"false"	<26, >
"true"	<27, >
"EOF"	<28, >

5.2. GRAMÁTICA

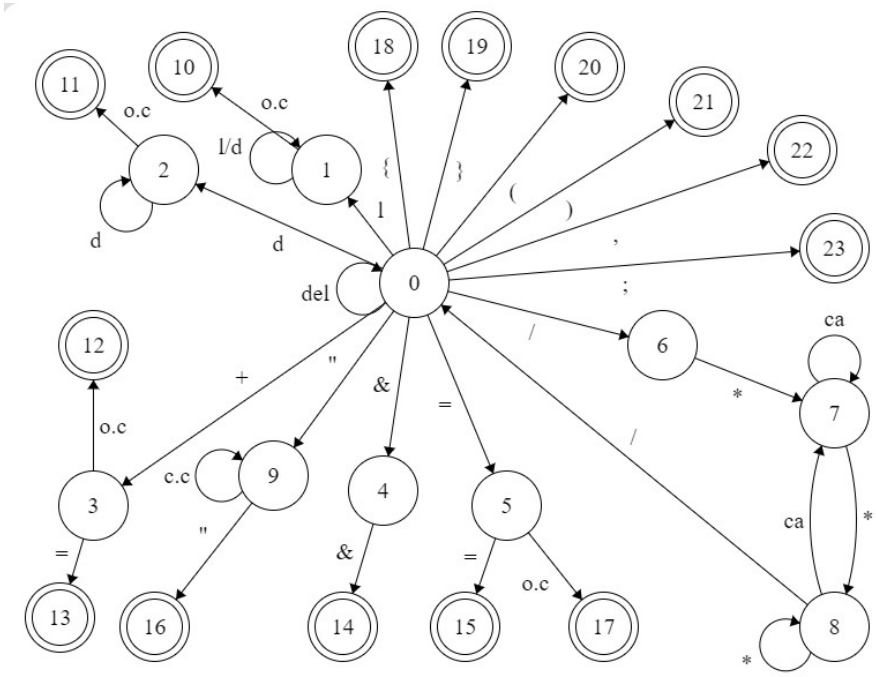
La gramática que implementa el lenguaje de nuestro analizador es:

S→ delS | lA | dB | +C | &D |=E | /F | "G | { | } | (|) | , | ;
A→ lA | dA | λ
B→ dB | λ
C→ = | λ
D→ &
E→ = | λ
F→ *H
G→ ccG | "
H→ caH | *I
I→ caH | *I | /S

del = {"tabulaciones", "espacios", "saltos de línea"}
d = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
l = {a – z, A - Z}∪{ '_' }
cc = {"cualquier carácter"} - {' " '}
ca = {"cualquier carácter"} – {' * '}
oc = {"otro carácter"}

5.3. AUTÓMATA FINITO DETERMINISTA (AFD)

El autómata utilizado para el entendimiento del programa es el siguiente:



5.4. ACCIONES SEMÁNTICAS.

Estas son las acciones semánticas implementadas en el proyecto:

0:0 leer
0:1 lexema l; leer
1:1 lexema ⊕ l|d; leer
1:10 if(Palabra reservada = encontrar palabra reservada(lexema)) then Generar token(Cod_PalabraReservda, -)
else if(pos = buscar ts(lexema) then Generar token(Cod_Id, pos)
else InsertsToken(Id, pos) Generar token(Cod_Id, pos)
0:2 valor = d; leer
2:2 valor = valor * 10 + d; leer
2:11 if (valor < 2^(16) - 1) then Generar token(entero, valor)
else Error("número fuera de rango")
0:3 leer
3:12 Generar token (suma, -)
3:13 Generar token(Preincremento,-)
0:4 l leer
4:14 Generar token(And, -)
0:5 leer
5:17 Generar token(Asignación, -)
5:15 Generar token (Igual, -)
0:6 leer
6:7 leer
7:7 leer
7:8 leer
8:7 leer
8:8 leer
8:0 leer
0:9 lexema = ''; leer
9:9 lexema = lexema ⊕ cc, cont+1; leer
9:16 lexema = lexema ⊕''; if(cont <= 64) then Generar token(Cadena,cadena "c")
else Error (“Una cadena no puede contener más de ‘64’ carácteres”)
0:18 Generar token ({, -)
0:19 Generar token (}, -)
0:20 Generar token ((, -)
0:21 Generar token ( ) , -)
0:22 Generar token (, , -)
0:23 Generar token (; , -)

5.5. ERRORES

Los errores encontrados en el analizador solo pueden ser Errores Léxicos, es debido a que solo se ha implementado el Analizador Léxico, porque no se ha desarrollado el Analizador Sintáctico ni Semántico.

Los posibles errores son:

- Error 1: el analizador no reconoce el carácter leído. Este error se ejecutará cuando el Analizador Léxico se encuentre con un carácter no admitido por la gramática en el "estado 0" del autómata.
- Error 2: carácter no esperado, después de ' & ' se espera el carácter ' & '. Este error se ejecuta cuando el Analizador Léxico se encuentra con el carácter ' & ', precedido por cualquier otro carácter distinto al esperado.
- Error 3: carácter no esperado, después de '/' se espera el carácter ' * '. Este error se ejecuta cuando el Analizador Léxico se encuentra con el carácter ' * ', precedido por cualquier otro carácter distinto al esperado.
- Error 4: la cadena de caracteres tiene una longitud mayor que '64' caracteres. Este error se ejecuta cuando el Analizador Léxico se encuentra con una cadena de caracteres de una longitud mayor a 64.
- Error 5: la Constante Entera está fuera del rango disponible. Este error se ejecutará cuando el Analizador Léxico se encuentre con un número entero mayor o igual que 2^{16}.

6. TABLA DE SÍMBOLOS.

La Tabla de Símbolos es generada por el Analizador Léxico y el Semántico. El Analizador Léxico es el encargado de almacenar los lexemas de los analizadores encontrados en el código analizado. Con ayuda del Analizador Semántico, se podrá identificar si estos se encuentran dentro de una función, entrando así en la tabla local, o fuera de una, siendo parte de la Tabla Global.

Los lexemas se introducirán en sus respectivas tablas cuando los identificadores estén siendo declarados o especificados en los argumentos de una función. Si aparece un identificador no declarado previamente, se introducirá en la Tabla Global y se tratará, por ende, como una variable global.

El Analizador Semántico se encargará de introducir el resto de los atributos de los identificadores. Si se trata de un identificador distinto a una función, se almacenará el tipo del identificador y su desplazamiento. En el caso de que se trate una función, se almacenará el tipo (función), el número de parámetros (y sus respectivos tipos), el tipo de retorno, y una etiqueta para identificar la función.

La estructura de la Tabla de Símbolos Global estará formada por 7 columnas y n filas, siendo n el número de identificadores introducidos en la Tabla Global.

La estructura de la Tabla de Símbolos Local estará formada por 4 columnas, al carecer de anidamiento de funciones, y n filas, siendo n el número de identificadores introducidos en la Tabla Local.

Para diferenciar las posiciones de ambas tablas, se usarán números enteros positivos en la Tabla Global y número enteros negativos para la Local.

La estructura de la tabla sería la siguiente:

POSICIÓN TABLA DE SIMBOLOS GLOBAL	LEXEMA	TIPO	DESPLAZAMIENTO	Nº PARAMETROS	TIPO RETORNO	ETIQUETA
"número entero positivo"	"identificador"	"tipo"	"desplazamiento global"	"nº parámetros de una función"	"tipo retorno de una función"	"etiqueta de una función"

POSICIÓN TABLA DE SIMBOLOS LOCAL	LEXEMA	TIPO	DESPLAZAMIENTO
"número entero negativo"	"identificador"	"tipo"	"desplazamiento local"

En el nuestro analizador se han implementado mediante la clase "GestorTS.java", que mediante la Estructura de Almacenamiento de Datos "ArrayList" va almacenando los lexemas asociados a los identificadores y asignando una posición en dicho array, a su vez, almacenada en otro ArrayList que agrupa las tablas tanto Global como Locales.

7. ANALIZADOR SINTÁCTICO

La función del analizador sintáctico consiste en determinar si la sintaxis del programa fuente a compilar es correcta. Para ello, solicitará al analizador léxico los tokens leídos de uno en uno hasta completar el fichero, con ellos comprobará mediante una gramática específica, que se expondrá próximamente, para el método asignado que cumple la sintaxis establecida por el lenguaje "JavaScriptPdl".

El método asignado para la comprobación de la sintaxis es "El descendiente recursivo", para el cual se debe tener una gramática que no sea ni ambigua y que no sea recursiva por la izquierda, pero si factorizada por la izquierda.

7.1. GRAMÁTICA

Como se ha dicho previamente, la gramática debe cumplir las características mencionadas. Teniendo eso en cuenta, la gramática es la siguiente:

Terminales = { boolean do function if input int let print return string while cte_entera cadena("") ID += = , ; ( ) { } + && == false true EOF }

NoTerminales = { P0 P F H A K C B S S1 X L Q E E1 R R1 U U1 V V1 T W }

Axioma = P0

Producciones = {

P0 →P //// 1
P →B P //// 2
P →F P //// 3
P →EOF //// 4
F →function ID H (A) { C } //// 5
H →T //// 6
H →lambda //// 7
A →T ID K //// 8
A →lambda //// 9
K →, T ID K //// 10
K →lambda //// 11
C →B C //// 12
C →lambda //// 13
B →if (E) S ; //// 14
B →let ID T ; //// 15
B →do { C } while (E) ; //// 16
B →S ; //// 17
S →ID S1 //// 18
S →print E //// 19
S →input ID //// 20
S →return X //// 21
S1 →= E //// 22
S1 →(L) //// 23
S1 →+= E //// 24
X →E //// 25
X →lambda //// 26
L →E Q //// 27
L →lambda //// 28
Q →, E Q //// 29
Q →lambda //// 30
E → R E1 //// 31
E1 →&& R E1 //// 32
E1 →lambda //// 33
R →U R1 //// 34
R1 →== U R1 //// 35

R1 →lambda //// 36
U →V U1 //// 37
U1 →+ V U1 //// 38
U1 →lambda //// 39
V →ID V1 //// 40
V →(E) //// 41
V →cte_entera //// 42
V →cadena("") //// 43
V →false //// 44
V →true //// 45
V1 →(L) //// 46
V1 →lambda //// 47
T →int //// 48
T →boolean //// 49
T →string //// 50
}

7.2. TABLAS FIRST-FOLLOW

Realizaremos esta tabla para facilitar la futura creación de la tabla que comprobara las condiciones LL(1) y verificará que la gramática sea correcta.

NO TERMINAL	FIRST	FOLLOW
P0	{if, let, do, ID, print, input, return, function, EOF}	{$}
P	{if, let, do, ID, print, input, return, function, EOF}	{$}
F	{function}	{if, let, do, ID, print, input, return, function, EOF}
H	{int, boolean, string, λ}	{ '(' }
A	{ λ, int, boolean, string}	{ ')' }
K	{',' , λ }	{ ')' }
C	{if, let, do, ID, print, input, return, λ}	{ '}' }
B	{if, let, do, ID, print, input, return}	{if, let, do, ID, print, input, return, function, EOF, '}'}
S	{ID, print, input, return}	{';'}
S1	{'=', '(', '+='}	{';'}
X	{ID, '(', constante entera, Cadena ("), false, true, λ}	{';'}
L	{ID, '(', constante entera, Cadena ("), false, true, λ}	{')'}
Q	{',' , λ}	{')'}
E	{ID, '(', constante entera, Cadena ("), false, true}	{',' , ';', ')'}
E1	{'&&', λ}	{',' , ';', ')'}
R	{ID, '(', constante entera, Cadena ("), false, true}	{'&&', ',' , ';', ')'}
R1	{'==', λ}	{'&&', ',' , ';', ')'}
U	{ID, '(', constante entera, Cadena ("), false, true}	{'==', '&&', ',' , ';', ')'}
U1	{'+', λ}	{'==', '&&', ',' , ';', ')'}
V	{ID, '(', constante entera, Cadena ("), false, true}	{'+', '==', '&&', ',' , ';', ')'}
V1	{'(', λ}	{'+', '==', '&&', ',' , ';', ')'}
T	{'int', boolean, string}	{ID, '(', '='}

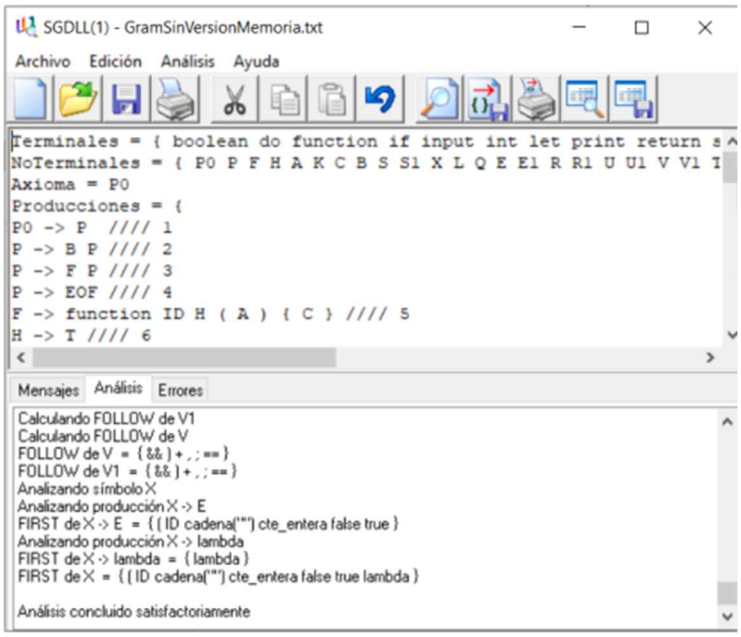
7.3. COMPROBACIÓN DE LA GRAMÁTICA

	boolean	do	function	if	input	int	let	print	return	string	while	constante entera
P0		P0 -> P	P0 -> P	P0 -> P	P0 -> P		P0 -> P	P0 -> P	P0 -> P			
P		P -> B P	P -> F P	P -> B P	P -> B P		P -> B P	P -> B P	P -> B P			
F			F -> function Id H (A) { C }									
H	H -> T					H -> T				H -> T		
A	A -> T ID K					A -> T ID K				A -> T ID K		
K												
C		C-> B C		C-> B C	C-> B C		C-> B C	C-> B C	C-> B C			
B		B -> do { C } while (E) ;		B -> if (E) S ;	B -> S ;		B -> let ID T ;	B -> S ;	B -> S ;			
S					S -> input ID			S -> print E	S -> return X			
S1												
X												X -> E
L												L -> E Q
Q												
E												E -> R E1
E1												
R												R -> U R1
R1												
U												U -> V U1
U1												
V												V -> constante entera
V1												
T	T -> boolean					T -> int				T -> string		

	cadena(")	ID	"+="	=	,	;	(	)	{ }	"+"	&&	"=="	false	true	EOF	"$"
P0		P0 -> P													P0 -> P	
P		P -> B P													P -> EOF	
F																
H							H -> lambda									
A								A -> lambda								
K					K -> , T ID K			K -> lambda								
C		C-> B C							C -> lambda							
B		B -> S ;														
S		S -> ID S1														
S1			S1 -> += E	S1 -> = E			S1 -> (L)									
X	X -> E	X -> E				X -> lambda	X -> E						X -> E	X -> E		
L	L -> E Q	L -> E Q					L -> E Q	L -> lambda					L -> E Q	L -> E Q		
Q					, E Q			Q -> lambda								
E	E -> R E1	E -> R E1					E -> R E1						E -> R E1	E -> R E1		
E1					E1 -> lambda	E1 -> lambda		E1 -> lambda			E1 -> && R E1					
R	R -> U R1	R -> U R1					R -> U R1						R -> U R1	R -> U R1		
R1					R1 -> lambda	R1 -> lambda		R1 -> lambda			R1 -> lambda	R1 -> == U R1				
U	U -> V U1	U -> V U1					U -> V U1						U -> V U1	U -> V U1		
U1					U1 -> lambda	U1 -> lambda		U1 -> lambda		U1 -> + V U1	U1 -> lambda	U1 -> lambda				
V	V -> cadena(")	V -> ID V1					V -> (E)						V -> false	V -> true		
V1					V1 -> lambda	V1 -> lambda	V1 -> (L)	V1 -> lambda		V1 -> lambda	V1 -> lambda	V1 -> lambda				
T																

Como las celdas de la tabla LL(1) tiene o una regla o están vacías eso significa que se cumplen las condiciones para verificar que es una gramática LL(1) bien formada. Esto sería equivalente a verificar que la intersección de los Firsts de cada regla (o si se incluye lambdas en el First también el de los Follows) es el conjunto vacío.

Adicionalmente se ha utilizado el software "SDGLL1" proporcionado en la página web de la asignatura para verificar las condiciones LL1 de nuestra gramática.



7.4. IMPLEMENTACION EN EL CÓDIGO

Como hemos usado el método descendente recursivo para la implementación en el código hemos implementado lo siguiente:

- La función *generarParse()* que se usará como función principal para la ejecución del resto de funciones.
- Una función para cada no terminal que sigue una estructura recursiva empezando por el axioma. Cada función es una anidación de 'else-if' que comprueban que el token que reciben es del conjunto de los tokens esperados para que la sintaxis sea correcta . Si el token es el esperado se sigue avanzando recursivamente por el resto de las funciones hasta que se lean todos los tokens. En caso contrario se producirá un error sintáctico que especificaremos posteriormente.
- Una función auxiliar *getNextToken()* que el analizador sintáctico usa para solicitar el siguiente token leído por el analizador léxico. En caso de que la lista de tokens este vacía se devolverá null.
- Una función *escribirParse()* que se usa para crear un fichero con el formato especificado que incluye el parse generado por el analizador.

Todo esto se encuentra en la clase *Analizador_Sintactico.java*.

8. ANALIZADOR SEMÁNTICO

La función del Analizador Semántico consiste en determinar si la semántica del programa es correcta. Para ello, se implementará unas reglas semánticas en la gramática de Sintáctico que permitirán realizar las acciones semánticas pertinentes y controlar los errores.

Las posibles acciones semánticas serán:

- Comprobar que los tipos usados son los correctos mediante el uso de atributos sintetizados y heredados.
- Insertar los tipos desplazamiento y atributos de las funciones a la Tabla de Símbolos.
- Crear las tablas de símbolos o destruirlas cuando ya no sean necesarias.

Los tipos definidos por el grupo han sido:

TIPO DE VARIABLE	NOMBRE
Función	funcion
Vacío	vacio
Lógico	logico
Cadena	cadena
Entero	entero

8.1. ESQUEMA DE TRADUCCIÓN (EDT)

El Esquema de Traducción (EDT) que se encarga de recoger las acciones semánticas del Analizador Semántico es el siguiente.

Terminales = { boolean do function if input int let print return string while cte_entera cadena("") ID += = , ; ( ) { } + && == false true EOF }
NoTerminales = { P0 P F H A K C B S S1 X L Q E E1 R R1 U U1 V V1 T }
Axioma = P0
Producciones = {
P0 → {TSG=CrearTS(), desplGl=0,zonaDecl=true, TSActual=TSG}P{Destruir TS(TSG)} //// 1
P → B P{} //// 2
P → F P{} //// 3
P → EOF{} //// 4
F → function id {TSL: CrearTS(), TSAct := TSL, despL := 0, insertaEtiTSG(id.pos, nueva_et())}H (A) {insertaTipoTS(id.pos, A.tipo(Parametros) -→ H.tipo(TipoRetorno))} { C } {if(C.tipo = tipo_error) then Error("Sentencias incorrectas")
if(C.tipoRet != H.tipo and C.tipoRet != vacio) then Error("Tipo de retorno incorrecto"), DestruirTS(TSL), TSAct := TSG} //// 5
H → T {H.tipo:=T.tipo}//// 6
H → lambda{H.tipo:=vacio} //// 7
A → T id K {A:= T.tipo x K.tipo, insertarTipoTS(id.pos,T.tipo), insertaDespTS(id.pos, despL),despL := despL + T.ancho}//// 8
A → lambda {A.tipo:= tipo_ok} //// 9
K → , T id K {K:= T.tipo x K'.tipo, insertarTipoTS(id.pos,T.tipo), insertaDespTS(id.pos, despL),despL:= despL + T.ancho}//// 10
K → lambda{K.tipo:=tipo_ok} //// 11
C → B C {C.tipo:= if(B.tipo==tipo_ok)then C'.tipo, C.tipoRet:= if(B.tipoRet==C'.tipoRet) then B.tipoRet elseif(B.tipo==vacio) then C'.tipo elseif(C'.tipo==vacio) then B.tipo else tipo_error}//// 12
C → lambda {C.tipo:= tipo_ok, C.tipoRet= vacio}//// 13
B → if (E) S ;{B.tipo:= if(E.tipo==logico)then S.tipo else error_tipo} //// 14

B → let id T ;{insertaTipoTS(id.pos, T.tipo) if(TSAct = TSG) { insertaDespTS(id.pos, despG), despG := despG + T.ancho} else { insertaDespTS(id.pos, despL), despL := despL + T.ancho}B.tipo:= tipo_ok//// 15
B → do { C } while (E) ; {B.tipo:= if(E.tipo==logico)then C.tipo else error_tipo }//// 16
B → S ; {B.tipo:=S.tipo} //// 17
S → id S1 {S.tipo := if(S1.tipo == buscaTipoTS(id.pos)) then tipo_ok else tipo_error; S.tipoRet:= vacio}//// 18
S → print E {S.tipo := if(E.tipo = {cadena,entero}) then tipo_ok else tipo_error; S.tipoRet:= vacio}//// 19
S → input id {S.tipo:= if(buscaTipoTS(id.pos) != tipo_error) then tipo_ok else tipo_error; S.tipoRet:=vacio}//// 20
S → return X {S.tipo:= if(X.tipo != error_tipo) then tipo_ok else tipo_error;S.tipoRet:= X.tipo}//// 21
S1 → = E {S1.tipo:= if(E.tipo != tipo_error) E.tipo else tipo_error} //// 22
S1 → (L){S1.tipo:= if(L.tipo != tipo_error) L.tipo else tipo_error} //// 23
S1 → += E {S1.tipo:= if(E.tipo = int) then S1.tipo else tipo_error//// 24
X → E {X.tipo:= E.tipo} //// 25
X → lambda{X.tipo:=Vacio} //// 26
L → E Q {L.tipo:= E.tipo x Q.tipo}//// 27
L → lambda{L.tipo:= tipo_ok} //// 28
Q → , E Q {Q.tipo:= E.tipo x Q'.tipo} //// 29
Q → lambda {Q.tipo:= tipo_ok}//// 30
E → R E1 {E.tipo:= if(R.tipo==E1.tipo)then R.tipo else tipo_error} //// 31
E1 → && R E1 {R1:= if((R.tipo==E1'.tipo)={logico})then logico else tipo_error}//// 32
E1 → lambda{E1.tipo=tipo_ok} //// 33
R → U R1 {R.tipo:= if(U.tipo==entero AND R1.tipo == logico) then logico else if(R1.tipo==tipo_ok) then U.tipo else tipo_error} //// 34
R1 → == U R1 {R1:= if((U.tipo==entero AND R1'.tipo== logico))then logico else tipo_error} //// 35
R1 → lambda {R1.tipo:= tipo_ok} //// 36
U → V U1 {U.tipo:=if(V.tipo==U1.tipo) then V.tipo else tipo_error } //// 37
U1 → + V U1 {U1:= if((V.tipo==U1'.tipo)={entero})then V.tipo else tipo_error} //// 38
U1 → lambda {U1.tipo:=tipo_ok} //// 39
V → id V1 {V.tipo:= {V.tipo := if (buscaTipoTS(id.pos)=V1.tipo)(En el caso de las funciones tipo de lo sparametros) then buscaTipoTS(id.pos)(En el caso de las funciones tipoReturn) else tipo_error}//// 40
V → (E){V.tipo := if (E.tipo!= tipo_error) then E.tipo else tipo_error}//// 41
V → cte_entera {V.tipo:=entero} //// 42
V → cadena {V.tipo:=cadena}//// 43
V → false {V.tipo:=logico}//// 44
V → true {V.tipo:=logico}//// 45
V1 → (L){V1.tipo:=L.tipo} //// 46
V1 → lambda{V1.tipo:=tipo_ok} //// 47
T → int {T.tipo:=entero, T.ancho:=1} //// 48
T → boolean {T.tipo:=logico, T.ancho:=1} //// 49
T → string {T.tipo:=cadena, T.ancho:=64}//// 50

9. ANEXO

9.1. PRUEBAS CORRECTAS

```
/*Ejemplo Correcto 1*/

let b boolean;

b=true;

if(b)

	varglobal=5;

print varglobal;
```

- Tokens:

```
<7,>
<14,1>
<1,>
<18,>
<14,1>
<16,>
<27,>
<18,>
<4,>
<19,>
<14,1>
<20,>
<14,2>
<16,>
<12,5>
<18,>
<8,>
<14,2>
<18,>
<28,>
```

- Parse:

```
Descendente 1 2 15 49 2 17 18 22 31 34 37 45 39 36 33 2 14 31 34 37 40 47 39 36 33 18 22 31
34 37 42 39 36 33 2 17 19 31 34 37 40 47 39 36 33 4
```

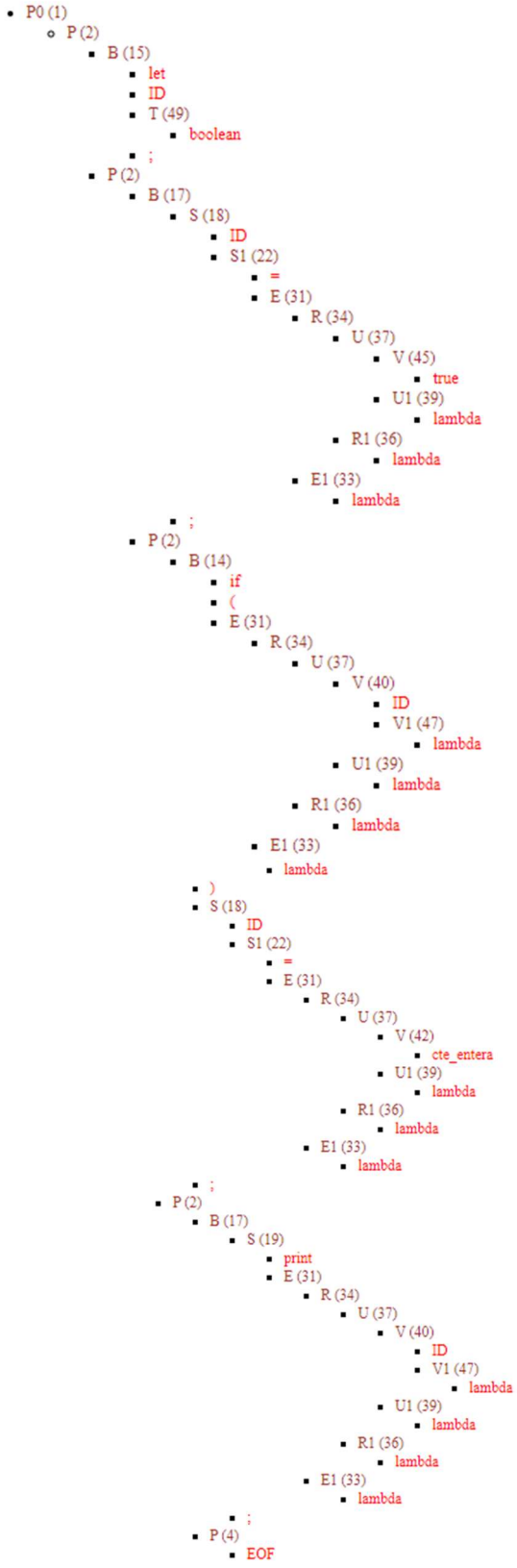
- Tabla de Símbolos:

```
CONTENIDOS DE LA TABLA GLOBAL # 1 :
* LEXEMA : 'b'
  Atributos :
  + tipo : 'logico'
  + despl : 0
  --------- ----------
* LEXEMA : 'varglobal'
  Atributos :
  + tipo : 'entero'
  + despl : 1
  --------- ----------
```

- Errores:

```
No se ha detectado ningún error.
```

- Árbol:


P0 (1)
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (45)
true
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (2)
B (14)
if
(
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (42)
cte_entera
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (2)
B (17)
S (19)
print
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (4)
EOF

/*Ejemplo Correcto 2*/

```
let b boolean;
let b2 boolean;
do{
   varglobal+=1;
}while(b && b);
```

- Tokens:

```
<7,>
<14,1>
<1,>
<18,>
<7,>
<14,2>
<1,>
<18,>
<2,>
<21,>
<14,3>
<15,>
<12,1>
<18,>
<22,>
<11,>
<19,>
<14,1>
<24,>
<14,1>
<20,>
<18,>
<28,>
```

- Parse:
- Descendente 1 2 15 49 2 15 49 2 16 12 17 18 24 31 34 37 42 39 36 33 13 31 34 37 40 47 39 36 32 34 37 40 47 39 36 33 4
- Tabla de Símbolos:

```
CONTENIDOS DE LA TABLA GLOBAL # 1 :

* LEXEMA : 'b'
  Atributos :
  + tipo : 'logico'
  + despl : 0
  --------- ----------
* LEXEMA : 'b2'
  Atributos :
  + tipo : 'logico'
  + despl : 1
  --------- ----------
* LEXEMA : 'varglobal'
  Atributos :
  + tipo : 'entero'
  + despl : 2
  --------- ----------
```

- Errores:

```
No se ha detectado ningún error.
```

- Árbol:


P0 (1)
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (16)
do
{
C (12)
B (17)
S (18)
ID
S1 (24)
+=
E (31)
R (34)
U (37)
V (42)
cte_entera
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
C (13)
lambda
}
while
(
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (32)
&&
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
)
;
P (4)
EOF

```
/*Ejemplo Correcto 3*/
let b boolean;
let n int;
function doble int (int n){
  n=n+n;
}
n=doble(n);
```

- Tokens:

```
<7,>
<14,1>
<1,>
<18,>
<7,>
<14,2>
<6,>
<18,>
<3,>
<14,3>
<6,>
<19,>
<6,>
<14,-1>
<20,>
<21,>
<14,-1>
<16,>
<14,-1>
<23,>
<14,-1>
<18,>
<22,>
<14,2>
<16,>
<14,3>
<19,>
<14,2>
<20,>
<18,>
<28,>
```

- Parse:

Descendente 1 2 15 49 2 15 48 3 5 6 48 8 48 11 12 17 18 22 31 34 37 40 47 38 40 47 39 36 33 13 2 17 18 22 31 34 37 40 46 27 31 34 37 40 47 39 36 33 30 39 36 33 4

- Tabla de Símbolos

CONTENIDOS DE LA TABLA GLOBAL # 1 :

* LEXEMA : 'b'
  Atributos :
  + tipo : 'logico'
  + despl : 0
  --------- ----------
* LEXEMA : 'n'
  Atributos :
  + tipo : 'entero'
  + despl : 1
  --------- ----------
* LEXEMA : 'doble'
  Atributos :
  + tipo : 'funcion'
  + numParam : 1
  + tipoParam1 : 'entero'
  + tipoRetorno : 'entero'
  + etiqFuncion : 'funcion1'
  --------- ----------

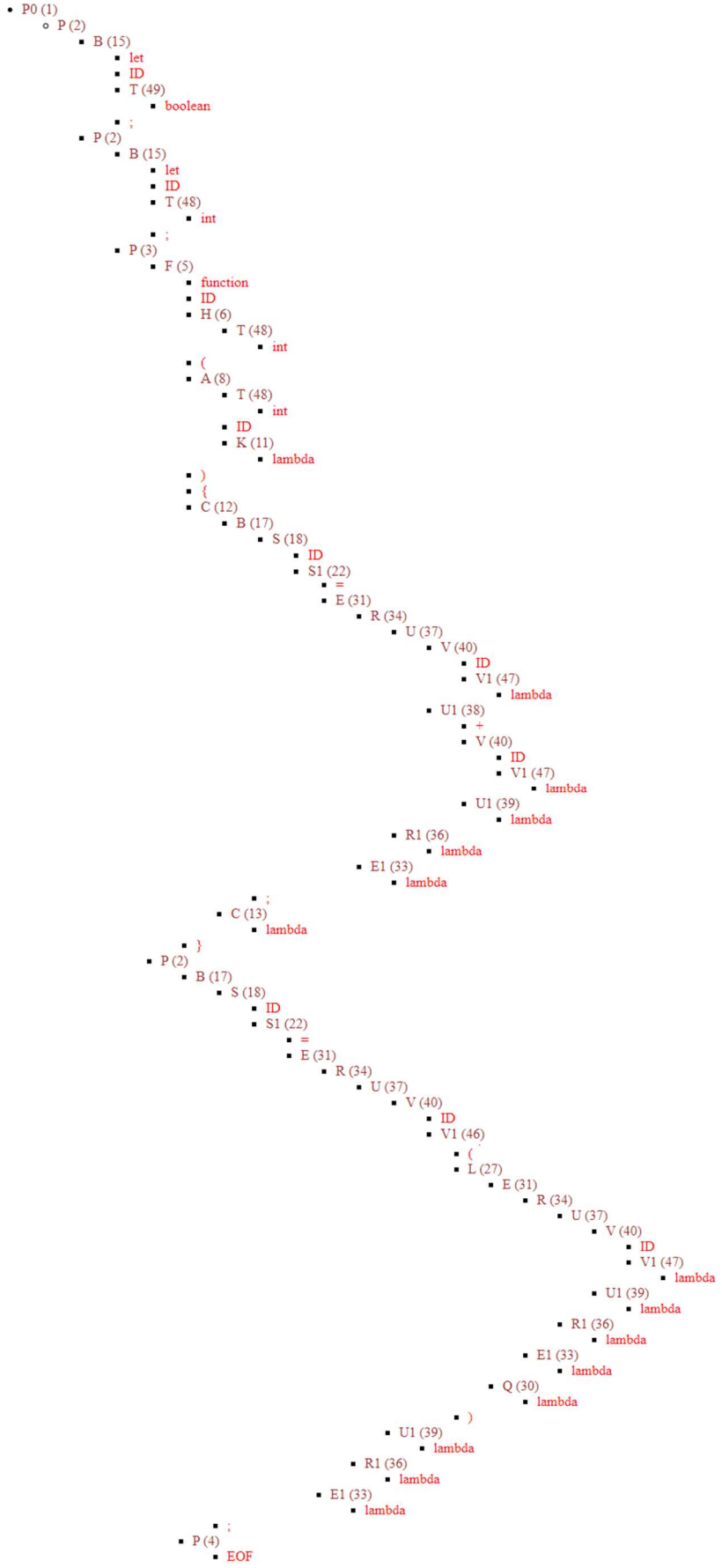
CONTENIDOS DE LA TABLA LOCAL # 2 :

* LEXEMA : 'n'
  Atributos :
  + tipo : 'entero'
  + despl : 0
  --------- ----------

- Errores:

No se ha detectado ningún error.

- Árbol:


P0 (1)
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (15)
let
ID
T (48)
int
;
P (3)
F (5)
function
ID
H (6)
T (48)
int
(
A (8)
T (48)
int
ID
K (11)
lambda
)
{
C (12)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (38)
+
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
C (13)
lambda
}
P (2)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (40)
ID
V1 (46)
(
L (27)
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
Q (30)
lambda
)
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (4)
EOF

```
/*Ejemplo Correcto 4*/

let b1 boolean;
let b2 boolean;
let n1 int;
let n2 int;
if((n1+3)==n2 && b1)
        b2=false;
```

- Tokens:

```
<7,>
<14,1>
<1,>
<18,>
<7,>
<14,2>
<1,>
<18,>
<7,>
<14,3>
<6,>
<18,>
<7,>
<14,4>
<6,>
<18,>
<4,>
<19,>
<19,>
<14,3>
<23,>
<12,3>
<20,>
<25,>
<14,4>
<24,>
<14,1>
<20,>
<14,2>
<16,>
<26,>
<18,>
<28,>
```

- Parse:

Descendente 1 2 15 49 2 15 49 2 15 48 2 15 48 2 14 31 34 37 41 31 34 37 40 47 38 42 39 36 33 39 35 37 40 47 39 36 32 34 37 40 47 39 36 33 18 22 31 34 37 44 39 36 33 4

- Errores:

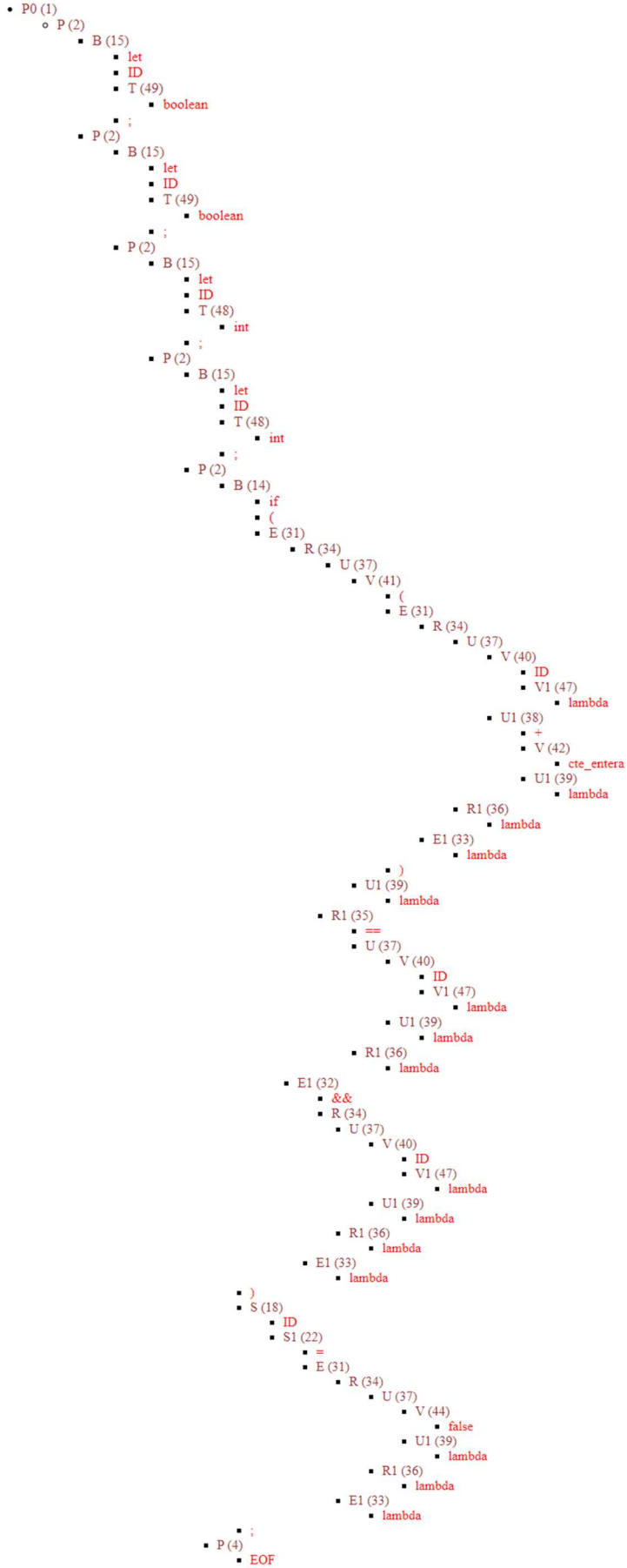
No se ha detectado ningún error.

- Tabla de Símbolos:

CONTENIDOS DE LA TABLA GLOBAL # 1 :

* LEXEMA : 'b1'
  Atributos :
  + tipo : 'logico'
  + despl : 0
  --------- ----------
* LEXEMA : 'b2'
  Atributos :
  + tipo : 'logico'
  + despl : 1
  --------- ----------
* LEXEMA : 'n1'
  Atributos :
  + tipo : 'entero'
  + despl : 2
  --------- ----------
* LEXEMA : 'n2'
  Atributos :
  + tipo : 'entero'
  + despl : 3
  --------- ----------

- Árbol:


P0 (1)
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (15)
let
ID
T (49)
boolean
;
P (2)
B (15)
let
ID
T (48)
int
;
P (2)
B (15)
let
ID
T (48)
int
;
P (2)
B (14)
if
(
E (31)
R (34)
U (37)
V (41)
(
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (38)
+
V (42)
cte_entera
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
)
U1 (39)
lambda
R1 (35)
==
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (32)
&&
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (44)
false
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (4)
EOF

```
/*Ejemplo Correcto 5*/
let n1 int ;
let n2 int;
n1=2;
n2=1;
function _Funcion (int a, int b, int c){
        a = b +c ;
         return;
}
_Funcion(varglobal1, n1, n2);
```

- Tokens:

```
<7,>
<14,1>
<6,>
<18,>
<7,>
<14,2>
<6,>
<18,>
<14,1>
<16,>
<12,2>
<18,>
<14,2>
<16,>
<12,1>
<18,>
<3,>
<14,3>
<19,>
<6,>
<14,-1>
<17,>
<6,>
<14,-2>
<17,>
<6,>
<14,-3>
<20,>
<21,>
<14,-1>
<16,>
<14,-2>
<23,>
<14,-3>
<18,>
<9,>
<18,>
<22,>
<14,3>
<19,>
<14,4>
<17,>
<14,1>
<17,>
```

<14,2>
<20,>
<18,>
<28,>

- Parse:

Descendente 1 2 15 48 2 15 48 2 17 18 22 31 34 37 42 39 36 33 2 17 18 22 31 34 37 42 39 36 33 3 5 7 8 48 10 48 10 48 11 12 17 18 22 31 34 37 40 47 38 40 47 39 36 33 12 17 21 26 13 2 17 18 23 27 31 34 37 40 47 39 36 33 29 31 34 37 40 47 39 36 33 29 31 34 37 40 47 39 36 33 30 4

- Errores:

No se ha detectado ningún error.

- Tabla de Símbolos:

CONTENIDOS DE LA TABLA GLOBAL # 1 :

* LEXEMA : 'n1'
  Atributos :
  + tipo : 'entero'
  + despl : 0
  --------- ----------
* LEXEMA : 'n2'
  Atributos :
  + tipo : 'entero'
  + despl : 1
  --------- ----------
* LEXEMA : '_Funcion'
  Atributos :
  + tipo : 'funcion'
  + numParam : 3
  + tipoParam1 : 'entero'
  + tipoParam2 : 'entero'
  + tipoParam3 : 'entero'
  + tipoRetorno : 'vacio'
  + etiqFuncion : 'funcion1'
  --------- ----------
* LEXEMA : 'varglobal1'
  Atributos :
  + tipo : 'entero'
  + despl : 2
  --------- ----------

CONTENIDOS DE LA TABLA LOCAL # 2 :

* LEXEMA : 'a'
  Atributos :
  + tipo : 'entero'
  + despl : 0
  --------- ----------
* LEXEMA : 'b'
  Atributos :
  + tipo : 'entero'
  + despl : 1
  --------- ----------
* LEXEMA : 'c'
  Atributos :
  + tipo : 'entero'
  + despl : 2
  --------- ----------

- Árbol:


P0 (1)
P (2)
B (15)
let
ID
T (48)
int
;
P (2)
B (15)
let
ID
T (48)
int
;
P (2)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (42)
cte_entera
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (2)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (42)
cte_entera
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
P (3)
F (5)
function
ID
H (7)
lambda
(
A (8)
T (48)
int
ID
K (10)
,
T (48)
int
ID
K (10)
,
T (48)
int
ID
K (11)
lambda
)
{
C (12)
B (17)
S (18)
ID
S1 (22)
=
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (38)
+
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
;
C (12)
B (17)
S (21)
return
X (26)
lambda


;
C (13)
lambda
}
P (2)
B (17)
S (18)
ID
S1 (23)
(
L (27)
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
Q (29)
,
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
Q (29)
,
E (31)
R (34)
U (37)
V (40)
ID
V1 (47)
lambda
U1 (39)
lambda
R1 (36)
lambda
E1 (33)
lambda
Q (30)
lambda
)
;
P (4)
EOF

9.2. PRUEBAS INCORRECTAS

```
/*Ejemplo Incorrecto 1*/
let n1 int;
n1=0;
function _F_u int (boolean b){
    if(n1)
        b=false;
    return b;
}
_F_u(n1)
```

- Errores:

```
Error  14 (semántico) en linea 5: El tipo de la expresión dentro del if debe ser de tipo lógico.
Error  26 (semántico) en linea 7: Tipo de retorno incorrecto en la funcion.
Error  25 (semántico) en linea 9: Argumentos incorrectos en la funcion _F_u
Error  11 (sintáctico) en linea 10: Se esperaba un ; pero se ha recibido un : EOF
```

```
/*Ejemplo Incorrecto 2*/
let n1 int;
let b1 boolean;
n1=5+b1;
do{
        n1= n1 --;
}while(5 && b1);
```

- Errores:

Error 24 (semántico) en linea 4: Para realizar una suma aritmetica los tipos de las expresiones tienen que ser iguales y de tipo entero.
Error 1 (léxico) en linea 6: El analizador no reconoce el caracter leido.
Error 1 (léxico) en linea 6: El analizador no reconoce el caracter leido.
Error 22 (semántico) en linea 7: Para realizar un AND lógico los tipos de las expresiones tienen que ser iguales y de tipo lógico.
Error 16 (semántico) en linea 7: La expresión dentro del while debe ser de tipo lógico.

```
/*Ejemplo Incorrecto 3*/
let n1 int;
let b1 boolean;
print(b1);
input n1;
if(5==b1)
   n1+=true;
do{
  /*Comentario Correcto*/
  n1 = "Hola";
};
```

- Errores:

Error 19 (semántico) en linea 4: Solo se puede de hacer print de expresiones de tipo cadena o entero.
Error 21 (semántico) en linea 7: Para hacer una asignación con suma el tipo de la expresión a asignar tiene que ser de tipo entera.
Error 17 (semántico) en linea 10: Los tipos para hacer un asignación no coinciden.
Error 12 (sintáctico) en linea 11: Se esperaba un while pero se ha recibido un : ;

```
/*Ejemplo Incorrecto 4*/
let b1 boolean;
if(b1)
  input b1;
function suma int (int n1,int n2){
  return n1+ n2;

varglobal = suma(varglobal,varglobal,varglobal);
/ /fin
```

- Errores:

```
Error  20 (semántico) en linea 4: Solo se puede de hacer input de variables de tipo cadena o entero.
Error  25 (semántico) en linea 8: Argumentos incorrectos en la funcion suma
Error  3 (léxico) en linea 9: Caracter no esperado, despues de '/' se espera el caracter '*'.
Error  3 (léxico) en linea 9: Caracter no esperado, despues de '/' se espera el caracter '*'.
Error  11 (sintáctico) en linea 10: Se esperaba un ; pero se ha recibido un : EOF
Error  6 (sintáctico) en linea 10: Se esperaba un } pero se ha recibido un : EOF
```

```
/*Ejemplo Incorrecto 5*/
let cad string;
cad="Cadena de texto de mas de 64 caracteres que debria dar error al crear el token" ;
cad="Hola";
if(true)
  print (cad);
let n1 int
```

- Errores:

```
Error  4 (léxico) en linea 3: La cadena de caracteres tiene una longitud mayor de 64 caracteres.
Error  11 (sintáctico) en linea 8: Se esperaba un ; pero se ha recibido un : EOF
```