Práctica Procesadores de Lenguajes

Grupo 17

Andrés Ollero Morales
Víctor Alejandro Sanz Ararat
Gabriel de Oliveira Trindade

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior De Ingenieros Informáticos
Universidad Politécnica de Madrid
Enero 2023

Índice

1. Diseño final del Procesador **2**

2. Diseño del Analizador Léxico **2**
2.1. Diseño de Tokens . . . 2
2.2. Gramática Regular (Tipo III) . . . 2
2.3. Autómata Finito Determinista . . . 3
2.4. Acciones Semánticas . . . 3
2.5. Errores . . . 4

3. Diseño del Analizador Sintáctico **5**
3.1. Gramática de Contexto Libre (Tipo II) . . . 5
3.2. First y Follow . . . 6
3.3. Demostración LL1 . . . 7
3.4. Tabla de Análisis . . . 11

4. Diseño del Analizador Semántico **12**
4.1. Traducción Dirigida Por la Sintáxis . . . 12

5. Diseño de la Tabla de Símbolos Completa **17**
5.1. Descripción de su estructura final y organización . . . 17

A. Casos de Prueba **18**
A.1. Prueba Funcional 1 . . . 18
A.2. Prueba Funcional 2 . . . 21
A.3. Prueba Funcional 3 . . . 23
A.4. Prueba Funcional 4 . . . 26
A.5. Prueba Funcional 5 . . . 29
A.6. Prueba No Funcional 1 . . . 31
A.7. Prueba No Funcional 2 . . . 31
A.8. Prueba No Funcional 3 . . . 31
A.9. Prueba No Funcional 4 . . . 32
A.10. Prueba No Funcional 5 . . . 32

1. Diseño final del Procesador

La implementación fue realizada en el lenguaje de programación Java. Nuestro grupo contaba con las siguientes opciones de práctica:

- Sentencias: Sentencia de selección múltiple (switch-case)
- Operadores especiales: Asignación con resto (%=)
- Técnicas de Análisis Sintáctico: Descendente con tabla
- Comentarios: Comentario de bloque (/* */)
- Cadenas: Con comillas dobles (¨¨)

De manera opcional, decidimos también implementar el token default para las sentencias switch-case.

2. Diseño del Analizador Léxico

2.1. Diseño de Tokens

```
<palabraReservada, boolean>         <palabraReservada, input>
<palabraReservada, break>           <palabraReservada, int>
<palabraReservada, case>            <palabraReservada, let>
<palabraReservada, function>        <palabraReservada, switch>
<palabraReservada, print>           <palabraReservada, return>
<palabraReservada, string>          <palabraReservada, if>
<palabraReservada, default>

<suma, >            <puntoComa, >       <asignacion, >
<negacion, >        <coma, >            <dosPuntos, >
<abrePar, >         <cierraPar, >       <comparacion, >
<abreLlave, >       <cierraLlave, >     <asignacionResto, >
<id, nºTS>          <constEnt, nº>      <cadena, "lexema">
```

2.2. Gramática Regular (Tipo III)

```
S -> "A | dB | %C | =D | _T | cT | {  |  }  |  (  |  )  |  +  |
      :  |  ;  |  ,  |  ! | del S
T -> cT | _T | dT | O.C
A -> lA | "
B -> dB | O.C
C -> =
D -> = | O.C
E -> \ E' | O.C
E' -> *E" | O.C E
E'' -> \ | O.C E'

donde c = letras (a-z, A-Z), d = dígitos (0-9), l = cualquier cosa
y O.C = otro caracter distinto
```

2.3. Autómata Finito Determinista

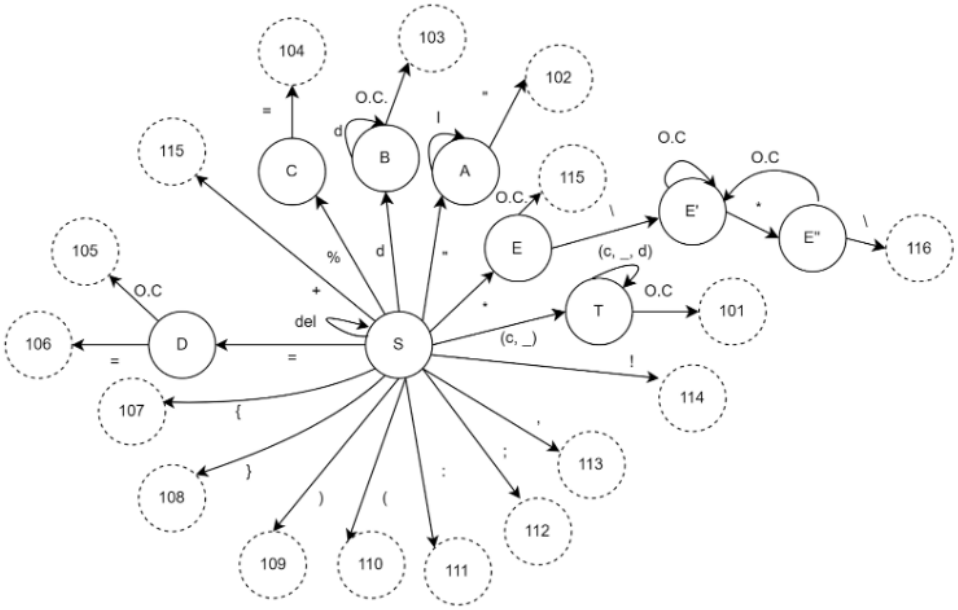


Figura 1: Implementación de la gramática con el autómata.

2.4. Acciones Semánticas

Se realiza la acción semántica LeerSigCaracter en todas las transiciones salvo en:

```
T : 101
B : 103
D : 105
S : T -> lexema = l; contador = 1;
T : T -> lexema = lexema + l; contador++;
T : 101 -> if (BuscarPalabraReservada(lexema) == 1)
                then GenerarToken(palabraReservada, “”)
            else if (BuscarTablaSimbolos(lexema) == 1)
                then GenerarToken(id, posTS)
            else (BuscarTablaSimbolos(lexema) == 0)
                then AñadirEntradaTabla(id, posTS) &&
                    GenerarToken(palabraReservada, posTS);
S : A -> contador = 0; lexema = "";
A : A -> lexema = lexema + l; contador++;
A : 102 -> if (contador <= 64)
                then GT(cadena, lexema)
    else
        then error(“longitud de la cadena excede el limite”);
S : B -> valor = valor_ascii(d);
B : B -> valor = valor + valor_ascii(d);
```

```
B : 103 -> if (valor <= 32767)
                then GenerarToken(constEnt, valor);
    else
                then error(‘‘valor del entero excede el limite’’);
C : 104 -> if (c != ‘=’)
                then error(‘‘expected ’%=’’);
           else
                GenerarToken(asignacionResto, );
D : 105 -> GenerarToken(igual, );
D : 106 -> GenerarToken(comparación, );
S : 107 -> GenerarToken(abreLlave, );
S : 108 -> GenerarToken(cierraLlave, );
S : 109 -> GenerarToken(abrePar, );
S : 110 -> GenerarToken(cierraPar, );
S : 111 -> GenerarToken(dosPuntos, );
S : 112 -> GenerarToken(puntoComa, );
S : 113 -> GenerarToken(coma, );
S : 114 -> GenerarToken(negación, );
S : 115 -> GenerarToken(suma, );

- GenerarToken (atributo, valor): Genera el token en un fichero de
                                  salida tokens.txt.
- BuscarPalabraReservada (lexema): Busca la palabra reservada en la
                                   respectiva tabla de palabras reservada.
- AñadirEntradaTabla(lexema, posTS): Añadimos el nombre de la variable a
                                     la tabla de símbolos.
```

2.5. Errores

Cuando se introduce en el código un símbolo que no pertenece al lenguaje, nuestro autómata lo omite. Los errores que contempla el código son cuando (y su correspondiente mensaje por línea de comandos):

- Se crea un identificador que no empieza por letra o subrayado.
  Error en línea: n ->Error Léxico: inicio de nombre de variable inválido.
- Se introduce un número entero superior a 32767.
  Error en línea: n ->Error Léxico: el valor numerico excede el límite de 32767.
- Se intenta crear una cadena de más de 64 caracteres.
  Error en línea: n ->Error Léxico: Cadena sobrepasa los 64 caracteres.
- Se crea una cadena sin finalizar con la doble comilla final.
  Error: La string esta mal formada.
- Se introduce exclusivamente un ’%’ sin el caracter ’=’ a continuación.
  Error: Expected ’=’ after ’%’.
- Se intenta hacer comentario de bloque y no se introducen los /* */ correctamente.
  Error en línea: n ->Error Léxico: Fin de comentario no válido.

Y por supuesto, no se genera el token.

3. Diseño del Analizador Sintáctico

3.1. Gramática de Contexto Libre (Tipo II)

Para ello hemos tomado la gramática que venía dada en las diapositivas de la explicación de la práctica modificándola a los terminales que escogimos al inicio. Eliminamos la recursividad por la izquierda y la factorizamos para cumplir la condición LL(1).

```
Axioma = PP

NoTerminales = { PP P S SS E R RR U UU V VV L Q X B T A K C F H O D }

Terminales = { ! == + id ( ) constEnt cadena %= print input return , if break
                switch case int boolean string let function ; : = { } default }

Producciones = {
    E -> R RR
    RR -> == R RR
    RR -> lambda
    R -> U UU
    UU -> + U UU
    UU -> lambda
    U -> ! V
    U -> V
    V -> id VV
    V -> ( E )
    V -> constEnt
    V -> cadena
    VV -> ( L )
    VV -> lambda
    S -> id SS
    SS -> %= E ;
    SS -> = E ;
    SS -> ( L ) ;
    S -> print R ;
    S -> input id ;
    S -> return X ;
    L -> E Q
    L -> lambda
    Q -> , E Q
    Q -> lambda
    X -> E
    X -> lambda
    B -> switch ( E ) { O }
    B -> if ( E ) S
    O -> case constEnt : C D O
    O -> default : C D O
    O -> lambda
    D -> break ;
    D -> lambda
```

```
    B -> let id T ;
    T -> int
    T -> boolean
    T -> string
    B -> S
    F -> function id H ( A ) { C }
    H -> T
    H -> lambda
    A -> T id K
    A -> lambda
    K -> , T id K
    K -> lambda
    C -> B C
    C -> lambda
    P -> B P
    P -> F P
    P -> lambda
    PP -> P
}
```

3.2. First y Follow

```
First(E) = {!, id, (, constEnt, cadena}
First(R) = {!, id, (, constEnt, cadena}
First(RR) = {==, lambda}
First(U) = {!, id, (, constEnt, cadena}
First(UU) = {+, lambda}
First(V) = {id, (, constEnt, cadena}
First(VV) = {(, lambda}
First(S) = {id, print, input, return}
First(SS) = {%, =, (}
First(L) = {!, id, (, constEnt, cadena, lambda}
First(Q) = {‘,’ , lambda}
First(X) = {!, id, (, constEnt, cadena, lambda}
First(B) = {switch, let, if, id, print, input, return}
First(O) = {case, default, lambda}
First(D) = {break, lambda}
First(T) = {int, boolean, string}
First(F) = {function}
First(H) = {function, lambda}
First(A) = {function, lambda}
First(K) = {‘,’ , lambda}
First(C) = {switch, let, if, id, print, input, return, lambda}
First(P) = {function, switch, let, if, id, print, input, return, lambda}
First(PP) = First(P)

Follow(E) = {), ;, ‘,’}
Follow(R) = {==}
Follow(RR) = {), ;, ‘,’}
```

```
Follow(U) = {+}
Follow(UU) = {==}
Follow(V) = {+}
Follow(VV) = {+}
Follow(S) = {function, switch, let, if, id, print, input, return}
Follow(SS) = {function, switch, let, if, id, print, input, return}
Follow(L) = { ) }
Follow(Q) = { ) }
Follow(X) = { ; }
Follow(B) = {function, switch, let, if, id, print, input, return}
Follow(O) = { } }
Follow(D) = {case, default}
Follow(T) = {;, id, (}
Follow(F) = {function, switch, let, if, id, print, input, return}
Follow(H) = { ( }
Follow(A) = { ) }
Follow(K) = { ) }
Follow(C) = { } }
Follow(P) = {$}
Follow(PP) = {$}
```

3.3. Demostración LL1

Para las 52 reglas de producción, hallamos que para cada No Terminal que tenga más de una producción en la gramática:
1. No exista ningún terminal se deriven de los No Terminales y sea la primera aparición (la intersección de sus firsts sea vacía).
2. Si un No Terminal se puede derivar a Lambda, el terminal producido por la regla no puede estar contenido en su follow.

Los no terminales que contienen dos o más producciones son: RR, UU, U, V, VV, S, SS, L, Q, X, B, O, D, T, H, A, K, C, P.

RR:
RR $\rightarrow$ == R RR
RR $\rightarrow \lambda$

First(= R RR) $\cap$ First(λ) = {==} $\cap$ {λ} $\rightarrow$ al aparecer λ:
First(== R RR) $\cap$ Follow(RR) = {==} $\cap$ {), ;, ','} = $\emptyset$

UU:
UU $\rightarrow$ + U UU
UU $\rightarrow \lambda$

First(+ U UU) $\cap$ First(λ) = {+} $\cap$ {λ} $\rightarrow$ al aparecer λ:
First(+ U UU) $\cap$ Follow(UU) = {+} $\cap$ {==} = $\emptyset$

U:

U → ! V
U → V

First(!V) ∩ First(V) = {!} ∩ {id, (, constEnt, cad} = ∅

V:

V → id VV
V → (E)
V → constEnt
V → cadena

First(id VV) ∩ First((E)) = {id} ∩ {(} = ∅
First(id VV) ∩ First(constEnt) = {id} ∩ {constEnt} = ∅
First(id VV) ∩ First(cadena) = {id} ∩ {cadena} = ∅
First((E)) ∩ First(constEnt) = {(} ∩ {constEnt} = ∅
First((E)) ∩ First(cadena) = {(} ∩ {cadena} = ∅
First(constEnt) ∩ First(cadena) = {constEnt} ∩ {cadena} = ∅

VV:

VV → (L)
VV → λ

First((L)) ∩ First(λ) = {(} ∩ {λ} → al aparecer λ:
First((L)) ∩ Follow(VV) = {(} ∩ {+} = ∅

S:

S → id SS
S → print R ;
S → input id ;
S → return X ;

First(id SS) ∩ First(print R ;) = {id} ∩ {print} = ∅
First(id SS) ∩ First(input id ;) = {id} ∩ {input} = ∅
First(id SS) ∩ First(return X) = {id} ∩ {return} = ∅
First(print R ;) ∩ First(input id ;) = {print} ∩ {input} = ∅
First(print R ;) ∩ First(return X ;) = {print} ∩ {return} = ∅
First(input id ;) ∩ First(return X ;) = {input} ∩ {return} = ∅

SS:

SS → %= E ;
SS → = E ;
SS → (L)

First(%= E) ∩ First(= E ;) = { %=} ∩ {=} = ∅
First(%= E) ∩ First((L)) = { %=} ∩ {(} = ∅
First(= E ;) ∩ First((L)) = {=} ∩ {(} = ∅

L:

L → E Q
L → λ

First(E Q) ∩ First(λ) = {!, id, (, constEnt, cadena} ∩ {λ}→ al aparecer λ:
First(E Q) ∩ Follow(L) = {!, id, (, constEnt, cadena} ∩ {)} = ∅

Q:

Q → , E Q
Q → λ

First(, E Q) ∩ First(λ) = {,} ∩ {λ}→ al aparecer λ:
First(, E Q) ∩ Follow(Q) = {,} ∩ {)} = ∅

X:

X → E
E → λ

First(E) ∩ First(λ) = {!, id, (, constEnt, cadena} ∩ {λ} → al aparecer λ:
First(E) ∩ Follow(X) = {!, id, (, constEnt, cadena} ∩ {;} = ∅

B:

B → switch (E) { O }
B → if (E) S
B → let id T ;
B → S

First(switch (E) { O }) ∩ First(if (E) S) = {switch} ∩ {if} = ∅
First(switch (E) { O }) ∩ First(let id T ;) = {switch} ∩ {let} = ∅
First(switch (E) { O }) ∩ First(S) = {switch} ∩ {id, print, input, return} = ∅
First(if (E) S) ∩ First(let id T ;) = {if} ∩ {let} = ∅
First(if (E) S) ∩ First(S) = {if} ∩ {id, print, input, return} = ∅
First(let id T ;) ∩ First(S) = {let} ∩ {id, print, input, return} = ∅

O:

O → case constEnt : C D O
O → default: C D O
O → λ

First(case constEnt: CDO) ∩ First(default: CDO) = {case} ∩ {default} = ∅
First(case constEnt : C D O) ∩ First(λ) = {case} ∩ {λ} → al aparecer λ:
First(case constEnt : C D O) ∩ Follow(O) = {case} ∩ {}} = ∅
First(default : C D O) ∩ First(λ) = {case} ∩ {λ} → al aparecer λ:
First(default : C D O) ∩ Follow(O) = {case} ∩ {}} = ∅

D:

D → break ;
D → λ

First(break ;) ∩ First(λ) = {break} ∩ {λ} → al aparecer λ:
First(break ;) ∩ Follow(D) = {break} ∩ {case, default} = $\emptyset$

T:

T → int
T → boolean
T → string

First(int) ∩ First(boolean) = {int} ∩ {boolean} = $\emptyset$
First(int) ∩ First(string) = {int} ∩ {string} = $\emptyset$
First(boolean) ∩ First(string) = {boolean} ∩ {string} = $\emptyset$

H:

H → T
H → λ

First(T) ∩ First(λ) = {int, boolean, string} ∩ {λ} → al aparecer λ:
First(T) ∩ Follow(H) = {int, boolean, string} ∩ {(} = $\emptyset$

A:

A → T id K
A → λ

First(T id K) ∩ First(λ) = {int, boolean, string} ∩ {λ} → al aparecer λ:
First(T id K) ∩ Follow(A) = {int, boolean, string } ∩ {)} = $\emptyset$

K:

K → , T id K
K → λ

First(, T id K) ∩ First(λ) = {,} ∩ {λ} → al aparecer λ:
First(, T id K) ∩$Follow(K)$ ={,} ∩ {)} = $\emptyset$

C:

C → B C
C → λ

First(B C) ∩ First(λ) = {switch, let, if, id, print, input, return} ∩ {λ} →
First(B C) ∩ Follow(C) = {switch, let, if, id, print, input, return} ∩ {}} = $\emptyset$

P:

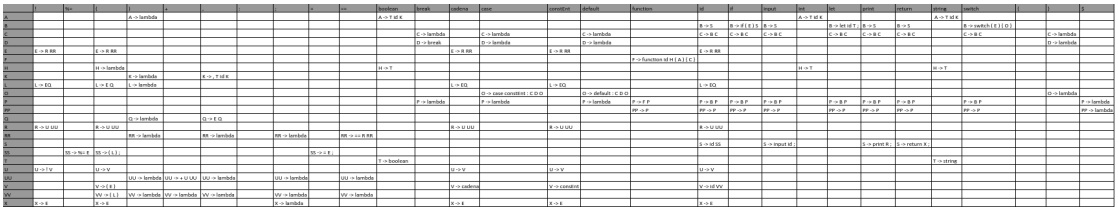
P → B P
P → F P
P → λ

First(BP)∩First(FP)={switch, let, if, id, print, input, return}∩{function} = $\emptyset$
First(B P) ∩ First(F P) = {switch, let, if, id, print, input, return} ∩ {λ} →
First(B P) ∩ Follow(P) = {switch, let, if, id, print, input, return} ∩ {$} = $\emptyset$
First(F P) ∩ First(λ) = {function} ∩ {λ} →

$$\text{First(F P)} \cap \text{Follow(P)} = \{\text{function}\} \cap \{\$\} = \emptyset$$

3.4. Tabla de Análisis

Utilizamos la herramienta Sistema Generador de Gramáticas LL1, donde introdujimos la gramática factorizada, sin recursividad por la izquierda, y nos produjo la siguiente tabla, que le indica al Analizador Sintáctico que reglas aplicar según los tokens que reciba.



	!	!=	(	)	+	,	-	.	/	:	=	==	boolean	break	cadena	case	constint	default	function	id	if	input	int	let	print	return	string	switch	{	}	$
A				A -> lambda									A -> T id K										A -> T id K				A -> T id K			
B																				B -> S	B -> if (E) S	B -> S		B -> let id T	B -> S	B -> S		B -> switch (E) { D }			
C														C -> lambda		C -> lambda		C -> lambda		C -> B C	C -> B C	C -> B C		C -> B C	C -> B C	C -> B C		C -> B C		C -> lambda	
D														D -> break		D -> lambda		D -> lambda													D -> lambda	
E	E -> R RR		E -> R RR												E -> R RR		E -> R RR				E -> R RR										
F																			F -> function id H [A] { C }												
H			H -> lambda										H -> T										H -> T				H -> T				
K				K -> lambda		K -> , T id K																										
L	L -> EQ		L -> E Q	L -> lambda											L -> EQ					L -> EQ											
O																O -> case constint : C D O		O -> default : C D O											O -> lambda	
P														P -> lambda		P -> lambda		P -> lambda	P -> F P	P -> B P	P -> B P	P -> B P		P -> B P	P -> B P	P -> B P		P -> B P			P -> lambda
PP																			PP -> P	PP -> P	PP -> P	PP -> P		PP -> P	PP -> P	PP -> P		PP -> P			PP -> lambda
Q				Q -> lambda		Q -> E Q																										
R	R -> U UU		R -> U UU												R -> U UU					R -> U UU											
RR				RR -> lambda		RR -> lambda		RR -> lambda		RR -> == R RR																						
S																				S -> id SS		S -> input id ;			S -> print R ;	S -> return X ;					
SS		SS -> %s T	SS -> ( ) ;						SS -> = E ;																							
T													? -> boolean														F -> string				
U	U -> ! V		U -> V												U -> V					U -> V											
UU				UU -> lambda	UU -> + U UU	UU -> lambda		UU -> lambda		UU -> lambda																						
V			V -> (E)												V -> cadena		V -> constint			V -> id VV											
VV			VV -> ( )	VV -> lambda	VV -> lambda	VV -> lambda		VV -> lambda		VV -> lambda																						
X	X -> E		X -> E					X -> lambda							X -> E		X -> E			X -> E											

Figura 2: Tabla de Análisis de LL1 (Click para una hoja de cálculo con mayor resolución).

4. Diseño del Analizador Semántico

4.1. Traducción Dirigida Por la Sintáxis

Implementamos un Esquema de Traducción (EdT) sobre la Gramática de Contexto Libre del Analizador Sintáctico:

Variables:

TSG = Tabla de Símbolos Global
DespG = Desplazamiento de la TS Global
TSActual = Tabla de Símbolos Actual
TSL = Tabla de Símbolos Local
DespL = Desplazamiento de la TS Local
ZonaDecl = Zona Declarativa

Métodos:

crearTS()
destruirTS(TSi)
insertaTipoTS(posicion de X, tipo de X)
insertaEtTS(posicion de X, nueva etiqueta para la TS)
nuevaEt() //las funciones no tienen desplazamiento en la TSG, sino una etiqueta
tipoDeFunción() //devuelve el tipo de la función actual
buscaTipoTS(posicion de X)
initCases() //comprueba que cada case del switch no repita el numero

Bloque Axioma:

PP → {TSG:= crearTS(), DespG := 0, TSActual := TSG}$_{1,1}$ P {destruirTS(TSG)}$_{1,2}$

P → B P {if (P2.tipo = tipo.vacio && B.tipo = tipo.ok)
then P.tipo := tipo.ok
else if (P2.tipo = tipo.ok && B.tipo = tipo.ok)
then P.tipo:= tipo.ok
else then
P.tipo := tipo.error;
POP(2)}$_2$

P → {ZonaDecl := true}$_{3,1}$ F P {if (P2.tipo = tipo.vacio && F.tipo = tipo.ok)
then P.tipo := tipo.ok
else if (P2.tipo = tipo.ok && F.tipo = tipo.ok)
then P.tipo:= tipo.ok
else then
P.tipo := tipo.error;
POP(2)}$_{3,2}$

P → λ {P.tipo := tipo.vacío}$_4$

Bloque declaración de funciones:

F → function id {TSL := crearTS(), DespL := 0} $_{5,1}$
H (A) {insertaTipoTS(id.pos, H.tipo); insertaEtTS(id.pos, nuevaEt())}$_{5,2}$
{ C {destruirTS(TSL)}$_{5,3}$}
{if (C.tipo != tipo.ok && C.tipo != tipo.vacío)
then error("tipos de return y función no coincide")
else F.tipo := tipo.ok
POP(9)}$_{5,4}$

H → T {H.tipo := T.tipo; POP(1)}$_{6}$

H → λ {H.tipo := tipo.vacío}$_{7}$

A → T id K {A.tipo := T.tipo, insertaTipoTS(id.pos, T.tipo), insertaDespTS(id.pos, DespL), DespL := DespL + T.ancho; POP(3)}$_{8}$

A → λ {A.tipo := tipo.vacío}$_{9}$

K → , T id K {K.tipo := T.tipo, insertaTipoTS(id.pos, T.tipo), insertaDespTS(id.pos, DespL), DespL := DespL + T.ancho; POP(4)}$_{10}$

K → λ {K.tipo := tipo.vacio}$_{11}$

C → B C {if(C2.tipo = tipo.vacio)
then C.tipo = B.tipo
else C.tipo = C2.tipo
POP(2)}$_{12}$

C → λ {C.tipo := tipo.vacío}$_{13}$

Bloque sentencias compuestas y declaración de variables:

B → if (E) S {if (E.tipo == boolean && S.tipo != tipo.vacio)
then B.tipo := tipo.ok
else B.tipo := tipo.error
POP(5)}$_{14}$

B → switch {initCases()}$_{15,1}$
(E) {if (E.tipo = tipo.ok)
then B := tipo.ok
else B.tipo := tipo.error)}$_{15,2}$
{ O } {if (E.tipo := constEnt && tipo.O := tipo.ok)
then B.tipo := tipo.ok
else B.tipo := tipo.error
POP(7)}$_{15,3}$

O → case constEnt {if(comprobarCases(constEnt) = true)
then añadirCases(constEnt)
else error(üno o multiples casos con el mismo valor")}$_{16,1}$
: C D O {if (C.tipo = tipo.ok || C.tipo = tipo.vacio) && (O2.tipo = tipo.ok ||
O2.tipo = tipo.vacio))
then O.tipo := tipo.ok
else then O.tipo := tipo.error
POP(6)}$_{16,2}$

O → default : C D O {if (C.tipo = tipo.ok && (O2.tipo = tipo.ok ||
O2.tipo = tipo.vacio))
then O.tipo := tipo.ok
else then O.tipo := tipo.error
POP(5)}$_{17}$

O → λ {O.tipo := tipo.vacío}$_{18}$

D → break ; {D.tipo := tipo.ok; POP(2)}$_{19}$

D → λ {D.tipo := tipo.vacio}$_{20}$

B → {ZonaDecl = true}$_{21,1}$ let id T {ZonaDecl = false}$_{21,2}$
; {B.tipo := tipo.ok
if (TSL = null)
insertaTipoTSL(id.pos, T.tipo), insertaDespTSL(id.pos, DespL),
DespL := DespL + T.ancho
else
insertaTipoTSG(id.pos, T.tipo), insertaDespTSG(id.pos, DespL),
DespG := DespG + T.ancho
POP(4)}$_{21,3}$

T → int {T.tipo = constEnt; POP(1)}$_{22}$

T → boolean {T.tipo = boolean; POP(1)}$_{23}$

T → string {T.tipo = cadena; POP(1)}$_{24}$

B → S {if (S.tipo != tipo.error && S.tipo != tipo.vacio))
then B.tipo := tipo.ok
else B.tipo := tipo.error
POP(1)}$_{25}$

Sentencias simples:

S → id SS {if (SS.tipo = tipo.ok || (buscaTipoTSL(id.pos) != null &&
SS.tipo = buscaTipoTSL(id.pos)))
then S.tipo:=tipo.ok
else if (SS.tipo = tipo.ok || (buscaTipoTSG(id.pos) != null &&
SS.tipo = buscaTipoTSG(id.pos)))

then S.tipo:=tipo.ok
else S.tipo := tipo.error
POP(2)}$_{26}$

SS → %= E ; {if (E.tipo = constEnt)
then SS.tipo := tipo.ok
else SS.tipo := tipo.error
POP(3)}$_{27}$

SS → = E ; {if (E.tipo != tipo.error && E.tipo != tipo.vacio)
then SS.tipo := E.tipo
else SS.tipo := tipo.error
POP(3)}$_{28}$

SS → (L) ; {if (L.tipo != tipo.error)
then SS.tipo:= tipo.ok
else SS.tipo := tipo.error
POP(4)}$_{29}$

S → print R ; {if (R.tipo = boolean || R.tipo = tipo.error)
then S:=tipo.error
else S:=tipo.ok
POP(3)}$_{30}$

S → input id ; {if (buscaTipoTSL(id.pos) = null || buscaTipoTSL(id.pos) = tipo.boolean)
then S.tipo := tipo.error
else if (buscaTipoTSG(id.pos) = null ||
buscaTipoTSG(id.pos) = tipo.boolean)
then S.tipo := tipo.error
else S.tipo := tipo.ok
POP(3)}$_{31}$

S → return X ; {if (tipoDeFuncion() = X.tipo)
then S.tipo := tipo.ok
else if (tipoDeFuncion() = null)
then error(“return fuera de una función”)
else S.tipo := tipo.error;
error("tipo de return no coincide con el tipo de la función");
POP(3)}$_{32}$

L → E Q {if (E.tipo != tipo.error && E.tipo != tipo.vacío && Q.tipo != tipo.error)
then L.tipo := tipo.ok
else L.tipo := tipo.error
POP(2)}$_{33}$

L → λ {L.tipo := tipo.vacio}$_{34}$

Q → , E Q {if (E.tipo != tipo.error && E.tipo != tipo.vacío && Q2.tipo != tipo.error)
then Q.tipo := tipo.ok
else Q.tipo := tipo.error
error("Error semantico: Expresion mal formada o vacia");
POP(3)}$_{35}$

Q → λ {Q.tipo := tipo.vacio}$_{36}$

X → E {X.tipo = E.tipo; POP(1)}$_{37}$

X → λ {X.tipo := tipo.vacio}$_{38}$

Expresiones:

E → R RR {if (RR.tipo != tipo.vacío && R.tipo = tipo.constEnt)
then E.tipo:=boolean
else if (RR.tipo != tipo.error && R.tipo != tipo.error)
then E.tipo := R.tipo
else E.tipo = tipo.error
POP(2)}$_{40}$

RR → == R RR {if (R.tipo = tipo.ConstEnt && RR2.tipo != tipo.error)
then RR.tipo:=ok
else RR.tipo := tipo.error
error("error semántico: expresión mal formada");
POP(3)}$_{41}$

RR → λ {RR.tipo:=tipo.vacío}$_{42}$

R → U UU {if (UU.tipo = tipo.vacío)
then R.tipo:=U.tipo
else if (U.tipo = tipo.constEnt && UU.tipo = tipo.ok)
then R.tipo := tipo.constEnt
else R.tipo := tipo.vacio
POP(2)}$_{43}$

UU → + U UU {if (U.tipo = constEnt && UU2.tipo != tipo.error)
then UU.tipo := tipo.ok
else then UU.tipo := tipo.error
error("error semantico: operando solo admite valores enteros");
POP(3)}$_{44}$

UU → λ {UU.tipo:=tipo.vacío}$_{45}$

U → ! V {if (V.tipo = tipo.boolean)
then U.tipo := tipo.boolean
else V.tipo := tipo.error
error("error semantico: operando solo admite tipo booelan");

POP(2)}$_{45}$

U → V {U.tipo := V.tipo; POP(1)}$_{46}$

V → id VV {if ((buscaTipoTSL(id) != null) && VV.tipo != tipo.error)
then V.tipo := (buscaTipoTSL(id))
if ((buscaTipoTSG(id) != null) && VV.tipo != tipo.error)
then V.tipo := (buscaTipoTSG(id))
else V.tipo := tipo.error
error("error semantico: id no declarado previamente");
POP(2)}$_{47}$

V → (E) {V.tipo := E.tipo; POP(3)}$_{48}$

V → constEnt {V.tipo := constEnt; POP(1)}$_{49}$

V → cadena {V.tipo := cadena; POP(1)}$_{50}$

VV → (L) {if (L.tipo != tipo.error)
then VV.tipo := tipo.ok
else VV.tipo := tipo.error;
POP(3)}$_{51}$

VV → λ {VV.tipo := tipo.vacio}$_{52}$

5. Diseño de la Tabla de Símbolos Completa

5.1. Descripción de su estructura final y organización

La tabla de símbolos está compuesta por dos tipos de tabla: las tablas de función, que tienen un número asignado que se muestra despues de # (siendo este numero mayor que cero). El cero estará asignado al otro tipo de tabla, tabla global.

Las tablas de las función estarán conformadas por un solo tipo de entrada: las entradas de variable (ya que el lenguaje que no admite anidar funciones). Estas entradas tendrán como parámetros: el lexema que les identifica, su tipo y el desplazamiento relativo a la función.

La tabla global estará conformada por dos tipos de entradas: la entrada de variable (descrita anteriormente) y la entrada de función, que tiene como parámetros: el lexema, que será único para cada función, el tipo (que será siempre función), el número de parámetros recibidos por la función, una dupla de atributos para cada uno de los parámetros (indicando su tipo y el modo de paso), el tipo de retorno de la función y la etiqueta asignada a la función.

A. Casos de Prueba

A.1. Prueba Funcional 1

Código fuente:

```
function funcion1 string (int arg1, boolean arg2, boolean arg3, boolean arg4)
{
  if (arg4 == arg3)
      return arg2;
  return arg1;
}
```

Fichero de Tokens:

```
<palabraReservada, function>
<id, 1>
<palabraReservada, boolean>
<abrePar, >
<palabraReservada, boolean>
<id, 2>
<coma, >
<palabraReservada, boolean>
<id, 3>
<coma, >
<palabraReservada, boolean>
<id, 4>
<coma, >
<palabraReservada, boolean>
<id, 5>
<cierraPar, >
<abreLlave, >
<palabraReservada, if>
<abrePar, >
<id, 5>
<comparacion, >
<id, 4>
<cierraPar, >
<palabraReservada, return>
<id, 3>
<puntoComa, >
<palabraReservada, return>
<id, 2>
<puntoComa, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#1:
* LEXEMA : 'arg1'
  ATRIBUTOS :
    +tipo : 'booleanR'
    +despl : 1
* LEXEMA : 'arg2'
  ATRIBUTOS :
    +tipo : 'booleanR'
    +despl : 2
* LEXEMA : 'arg3'
  ATRIBUTOS :
    +tipo : 'booleanR'
    +despl : 3
* LEXEMA : 'arg4'
```

```
  ATRIBUTOS :
    +tipo : 'booleanR'
    +despl : 4
#0:
* LEXEMA : 'funcion1'
  ATRIBUTOS :
    +tipo : 'funcion'
     +numParam : 4
      +TipoParam1 : 'booleanR'
       +ModoParam1 : 1
      +TipoParam2 : 'booleanR'
       +ModoParam2 : 1
      +TipoParam3 : 'booleanR'
       +ModoParam3 : 1
      +TipoParam4 : 'booleanR'
       +ModoParam4 : 1
    +TipoRetorno : 'booleanR'
    +EtiqFuncion : 'Etfuncion101'
```

Fichero de Parse y Árbol Sintáctico:

```
Descendente 52 50 40 41 37 43 37 45 37 45 37 45 37 46 47 29 1 4 8 9 14 6 2 4 8 9
    14 6 3 21 26 1 4 8 9 14 6 3 47 39 21 26 1 4 8 9 14 6 3 48 51
```


PP (52)
P (50)
F (40)
function
id
H (41)
T (37)
boolean
(
A (43)
T (37)
boolean
id
K (45)
,
T (37)
boolean
id
K (45)
,
T (37)
boolean
id
K (45)
,
T (37)
boolean
id
K (46)
lambda
)
{
C (47)
B (29)
if
(
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (2)
—
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
)
S (21)
return
X (26)
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
;
C (47)
B (39)
S (21)
return
X (26)
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
;
C (48)
lambda
}
P (51)
lambda

A.2. Prueba Funcional 2

Código fuente:

```
let global int;
global = 4;
function funcion2 int (int arg1)
{
  return (arg1 + global);
}
```

Fichero de Tokens:

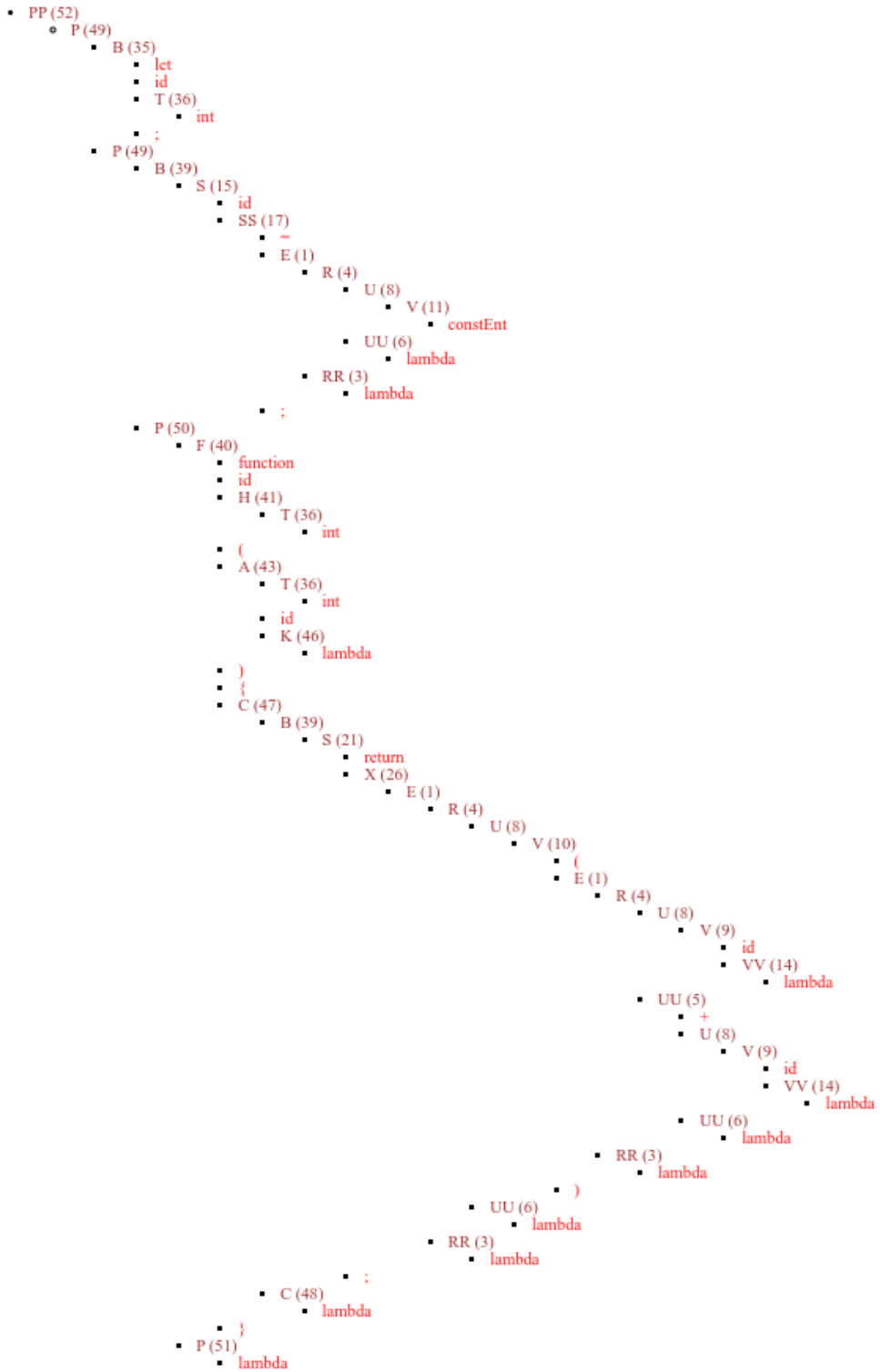
```
<palabraReservada, let>
<id, 1>
<palabraReservada, int>
<puntoComa, >
<id, 1>
<igual, >
<constEnt, 4>
<puntoComa, >
<palabraReservada, function>
<id, 2>
<palabraReservada, int>
<abrePar, >
<palabraReservada, int>
<id, 3>
<cierraPar, >
<abreLlave, >
<palabraReservada, return>
<abrePar, >
<id, 3>
<suma, >
<id, 1>
<cierraPar, >
<puntoComa, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#1:
* LEXEMA : 'arg1'
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 1
#0:
* LEXEMA : 'global'
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 1
* LEXEMA : 'funcion2'
  ATRIBUTOS :
    +tipo : 'funcion'
     +numParam : 1
      +TipoParam1 : 'constEnt'
       +ModoParam1 : 1
    +TipoRetorno : 'constEnt'
    +EtiqFuncion : 'Etfuncion201'
```

Fichero de Parse y Árbol sintáctico:

```
Descendente 52 49 35 36 49 39 15 17 1 4 8 11 6 3 50 40 41 36 43 36 46 47 39 21 26
    1 4 8 10 1 4 8 9 14 5 8 9 14 6 3 6 3 48 51
```


PP (52)
P (49)
B (35)
let
id
T (36)
int
;
P (49)
B (39)
S (15)
id
SS (17)
=
E (1)
R (4)
U (8)
V (11)
constEnt
UU (6)
lambda
RR (3)
lambda
;
P (50)
F (40)
function
id
H (41)
T (36)
int
(
A (43)
T (36)
int
id
K (46)
lambda
)
{
C (47)
B (39)
S (21)
return
X (26)
E (1)
R (4)
U (8)
V (10)
(
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (5)
+
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
)
UU (6)
lambda
RR (3)
lambda
;
C (48)
lambda
}
P (51)
lambda

A.3. Prueba Funcional 3

Código fuente:

```
let global1 int;
let global2 int;
if (global1 == global2)
    global1 = 0;
switch(global1)
{
    case 0:
        print "global 1 igual a global 2";
        break;
    default:
        print "global 1 es diferente a global 2";
        break;
}
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<palabraReservada, int>
<puntoComa, >
<palabraReservada, let>
<id, 2>
<palabraReservada, int>
<puntoComa, >
<palabraReservada, if>
<abrePar, >
<id, 1>
<comparacion, >
<id, 2>
<cierraPar, >
<id, 1>
<igual, >
<constEnt, 0>
<puntoComa, >
<palabraReservada, switch>
<abrePar, >
<id, 1>
<cierraPar, >
<abreLlave, >
<palabraReservada, case>
<constEnt, 0>
<dosPuntos, >
<palabraReservada, print>
<cadena, "global 1 igual a global 2">
<puntoComa, >
<palabraReservada, break>
<puntoComa, >
<palabraReservada, default>
<dosPuntos, >
<palabraReservada, print>
<cadena, "global 1 es diferente a global 2">
<puntoComa, >
<palabraReservada, break>
<puntoComa, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#0:
* LEXEMA : 'global1'
```

```
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 1
* LEXEMA : 'global2'
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 2
```

Fichero de Parse y Árbol sintáctico:

```
Descendente 52 49 35 36 49 35 36 49 29 1 4 8 9 14 6 2 4 8 9 14 6 3 15 17 1 4 8 11
    6 3 49 28 1 4 8 9 14 6 3 30 47 39 19 4 8 12 6 48 33 31 47 39 19 4 8 12 6 48 33
     32 51
```


PP (52)
P (49)
B (35)
let
id
T (36)
int
;
P (49)
B (35)
let
id
T (36)
int
;
P (49)
B (29)
if
(
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (2)
==
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
)
S (15)
id
SS (17)
=
E (1)
R (4)
U (8)
V (11)
constEnt
UU (6)
lambda
RR (3)
lambda
;
P (49)
B (28)
switch
(
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
)
{
O (30)
case
constEnt
:
C (47)
B (39)
S (19)
print
R (4)
U (8)
V (12)
cadena
UU (6)
lambda
;
C (48)
lambda
D (33)
break
;
O (31)
default
:
C (47)
B (39)
S (19)
print
R (4)
U (8)
V (12)
cadena
UU (6)
lambda
;
C (48)
lambda
D (33)
break
;
O (32)
lambda
}
P (51)
lambda

A.4. Prueba Funcional 4

Código fuente:

```
let int1 int;
function funcion4_1 int ()
{
    int1 = int1 + 1;
    return int1;
}
function funcion4_2 ()
{
    print funcion4_1();
}
funcion4_2();
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<palabraReservada, int>
<puntoComa, >
<palabraReservada, function>
<id, 2>
<palabraReservada, int>
<abrePar, >
<cierraPar, >
<abreLlave, >
<id, 1>
<igual, >
<id, 1>
<suma, >
<constEnt, 1>
<puntoComa, >
<palabraReservada, return>
<id, 1>
<puntoComa, >
<cierraLlave, >
<palabraReservada, function>
<id, 3>
<abrePar, >
<cierraPar, >
<abreLlave, >
<palabraReservada, print>
<id, 2>
<abrePar, >
<cierraPar, >
<puntoComa, >
<cierraLlave, >
<id, 3>
<abrePar, >
<cierraPar, >
<puntoComa, >
```

Tabla de Símbolos:

```
#1:
#2:
#0:
* LEXEMA : 'int1'
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 1
* LEXEMA : 'funcion4_1'
```

```
  ATRIBUTOS :
    +tipo : 'funcion'
     +numParam : 0
    +TipoRetorno : 'constEnt'
    +EtiqFuncion : 'Etfuncion4_101'
* LEXEMA : 'funcion4_2'
  ATRIBUTOS :
    +tipo : 'funcion'
     +numParam : 0
    +TipoRetorno : 'vacio'
    +EtiqFuncion : 'Etfuncion4_201'
```

Fichero de Parse y Árbol sintáctico:

```
Descendente 52 49 35 36 50 40 41 36 44 47 39 15 17 1 4 8 9 14 5 8 11 6 3 47 39 21
    26 1 4 8 9 14 6 3 48 50 40 42 44 47 39 19 4 8 9 13 23 6 48 49 39 15 18 23 51
```


PP (52)
P (49)
B (35)
let
id
T (36)
int
;
P (50)
F (40)
function
id
H (41)
T (36)
int
(
A (44)
lambda
)
{
C (47)
B (39)
S (15)
id
SS (17)
=
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (5)
+
U (8)
V (11)
constEnt
UU (6)
lambda
RR (3)
lambda
;
C (47)
B (39)
S (21)
return
X (26)
E (1)
R (4)
U (8)
V (9)
id
VV (14)
lambda
UU (6)
lambda
RR (3)
lambda
;
C (48)
lambda
}
P (50)
F (40)
function
id
H (42)
lambda
(
A (44)
lambda
)
{
C (47)
B (39)
S (19)
print
R (4)
U (8)
V (9)
id
VV (13)
(
L (23)
lambda
)
UU (6)
lambda
;
C (48)
lambda
}
P (49)
B (39)
S (15)
id
SS (18)
(
L (23)
lambda
)
;
P (51)
lambda

A.5. Prueba Funcional 5

Código fuente:

```
function funcion5()
{
    a = 4;
}
print a;
```

Fichero de Tokens:

```
<palabraReservada, function>
<id, 1>
<abrePar, >
<cierraPar, >
<abreLlave, >
<id, 2>
<igual, >
<constEnt, 4>
<puntoComa, >
<cierraLlave, >
<palabraReservada, print>
<id, 2>
<puntoComa, >
```

Tabla de Símbolos:

```
#1:
#0:
* LEXEMA : 'funcion5'
  ATRIBUTOS :
    +tipo : 'funcion'
     +numParam : 0
    +TipoRetorno : 'vacio'
    +EtiqFuncion : 'Etfuncion501'
* LEXEMA : 'a'
  ATRIBUTOS :
    +tipo : 'constEnt'
    +despl : 1
```

Fichero de Parse y Árbol sintáctico:

```
Descendente 52 50 40 42 44 47 39 15 17 1 4 8 11 6 3 48 49 39 19 4 8 9 14 6 51
```

- PP (52)
  - P (50)
    - F (40)
      - function
      - id
      - H (42)
        - lambda
      - (
      - A (44)
        - lambda
      -)
      - {
      - C (47)
        - B (39)
          - S (15)
            - id
            - SS (17)
              - =
              - E (1)
                - R (4)
                  - U (8)
                    - V (11)
                      - constEnt
                  - UU (6)
                    - lambda
                - RR (3)
                  - lambda
              - ;
        - C (48)
          - lambda
      - }
    - P (49)
      - B (39)
        - S (19)
          - print
          - R (4)
            - U (8)
              - V (9)
                - id
                - VV (14)
                  - lambda
            - UU (6)
              - lambda
          - ;
      - P (51)
        - lambda

A.6. Prueba No Funcional 1

Código fuente:

```
let a int;
let b int;
function suma int(int a, int b)
{
    let s string;
    return (a + s + b);
}
b = "cadena de prueba"
```

Aqui podemos ver dos fallos: el primero es que se intenta realizar una suma sobre valores que no son enteros, en este caso un valor de tipo cadena, lo cual provoca que el return devuelva un error de tipos incompatibles con la función. Y el siguiente, que se intenta asignar a una variable entera, un valor de tipo cadena.

Error en linea: 6 ->Error semantico: el operando + solo admite valores enteros
Error semantico, tipo de funcion no coincide con el valor de return
Error en linea: 8 ->Error semantico: el tipo de variable no coincide con el tipo de valor que se intenta asignar

A.7. Prueba No Funcional 2

Código fuente:

```
let log boolean;
let b int;
let c string;

if (! log)
    print log;

case 22:
```

En este otro ejemplo, podemos observar que existe un error semantico, al intentar llamar a la función print con un valor booleano, que según la especificación del lenguaje no debería ser posible, además, de tener un error sintáctico por la generación de un case, sin tener un switch previamente.

Error en linea: 8 ->Error semantico: Expresion mal formada o variable boolean no admitida
Error Sintáctico: No existe regla para M[P, <palabraReservada, case>].

A.8. Prueba No Funcional 3

Código fuente:

```
switch("hola")
{
    case 1: case 2: case 4: case 1:
        print 2;
    case "hola":
        input a;
}
```

En este caso de prueba vemos que al poner una palabra reservada como nombre de una función envía un error, ya que el token que se esperaba era un id, sin embargo, al recibir el token de palabra reservada, no coincide con el de la cima de la pila y acaba la ejecución.

Error en linea: 2 ->Error semantico: Expresion incorrecta para la evaluación del switch,

debe ser de tipo entero

Error en linea: 3 ->Error semantico: existen uno o multiples cases con la misma constante entera

Error Sintáctico: El terminal de la cima de la pila “constEnt“ no coincide con el token <cadena, “hola“>enviado por el AnLex.

A.9. Prueba No Funcional 4

Código fuente:

```
let a int;
let b string;
if (a == b)
    a = 2;
let c boolean;
input c;
====
```

Aqui vemos varios errores semánticos, entre ellos intentar realizar una comparación entre un valor entero y uno de tipo cadena. Otro error por intentar realizar un input sobre una varibale booleana. Y por último un error sintáctico por intentar mandar un token = justo despues de un ;.

Error en linea: 3 ->Error semantico: La comparacion solo admite valores enteros

Error en linea: 4 ->Error semantico: La expresion no esta bien formada

Error en linea: 7 ->Error semantico: variable booleana no admitida

Error Sintáctico: No existe regla para M[P, <comparacion, >].

A.10. Prueba No Funcional 5

Código fuente:

```
let a int;
let a string;

if ("hola")
    print 2.2;
```

Aqui encontramos errores léxicos, sintácticos y semánticos. El primero léxico es intentar declarar dos veces una variable con el mismo lexema. El semántico viene dado por intentar evaluar una cadena en un if, cuando este solo admite una expresión booleana. Y el sintáctico viene dado por intentar escribir un número real, lo cual no es admitido por nuestro lenguaje.

Error en linea: 2 ->Error léxico: Variable ya declarada

Error en linea: 4 ->Error semantico: La expresion no esta bien formada

Error Sintáctico: No existe regla para M[UU, <constEnt, 2>].