Entrega final

Procesadores de lenguajes

Descripción del diseño final del procesador

Grupo 102

Iñigo Aranguren Redondo
Pablo Beltrán de Casso
Ignacio de las Alas-Pumariño

15 de enero de 2020

1. Introducción

El objetivo de la práctica es el desarrollo e implementación de un procesador del lenguaje Javascript-PL, el procesador recibirá un archivo de texto para leer y deberá generar los archivos: Tokens, Tabla de símbolos, Parse y Errores. Nuestro grupo deberá implementar los comentarios de linea (//), las comillas simples (‘ ‘), las sentencias repetitivas (while) y como operador especial la asignación con resta. En cuanto al diseño del analizador sintáctico se deberá emplear la técnica Descendente con Tablas.

Para la realización de esta práctica hemos decidido llevar esto a cabo utilizando el lenguaje de programación Kotlin, con el que no teníamos ninguna experiencia previa. El motivo de esta decisión es que Kotlin es 100% intercompatible con Java, ya que compila a bytecode para la JVM, y con Java estamos muy familiarizados tras haberlo estudiado durante los últimos años. Kotlin es ahora mismo el lenguaje oficial de desarrollo de aplicaciones para Android, por lo que su demanda es creciente. Además, es un lenguaje moderno, con una sintaxis cómoda y más compacta que Java, y es un lenguaje híbrido que soporta tanto programación orientada a objetos como programación funcional, lo cual puede resultar útil en algunas situaciones. En general, hemos querido aprovechar esta oportunidad para aprender un lenguaje nuevo con la tranquilidad de que si la tarea se complicaba mucho teníamos Java como una red de seguridad, pudiendo continuar el desarrollo en él sin perder el trabajo ya hecho.

2. Diseño del Analizador Léxico

El Analizador léxico se encargará de leer un archivo de texto fuente e identificar los tokens contenidos en él para luego más tarde usarlos en el analizador sintáctico. Los comentarios, en nuestro caso los comentarios de linea (//) se omiten.

A. Tokens

Identificación de tokens :

1. < number, *valor* >

El analizador al encontrar un número lo guarda en un token cuyo código es ***“number“*** y cuyo valor es el número encontrado. Por las características del lenguaje de programación el máximo entero válido será el 32767.

2. < cadena, *Lexema* >

El analizador al encontrar una cadena que comienza por un carácter **“ ‘ “** comienza a leer hasta que encuentra el otro carácter **“ ‘ “** , una vez finalizado lo guarda en un token cuyo código es **“cadena”** y donde *lexema* es la cadena leída incluyendo los caracteres **“ ‘ “**. Por las características del lenguaje de programación una cadena no puede contener más de 64 caracteres.

3. < ariOp, >

El analizador al encontrar un operador aritmético lo guarda en un token cuyo código es el **"*ariOp*"** que es el operador aritmético analizado. En esta primera entrega hemos implementado el operador aritmético **"+", "*", "%", "/", "-"** . De tal forma que al encontrar un operador aritmético **"+"** el token queda de la siguiente manera : **< +, >.**

4. < relationOp, >

El analizador al encontrar un operador de relación lo guarda en un token cuyo código es el **"*relationOp*"** que es el operador de relación analizado. En esta primera entrega hemos implementado el operador de relación **"<"**. De tal forma que al encontrar un operador de relación **"<"** el token queda de la siguiente manera : **< <, >.**

5. < logiOp, >

El analizador al encontrar un operador lógico lo guarda en un token cuyo código es el **"*logicOp*"** que es el operador lógico analizado. En esta primera entrega hemos implementado el operador lógico **"!"**. De tal forma que al encontrar un operador lógico **"!"** el token queda de la siguiente manera : **< !, >.**

6. < assignment, >

El analizador al encontrar un operador de asignación lo guarda en un token cuyo código es **"*assignment*"** que es el operador de asignación analizado. En esta primera entrega hemos implementado el operador de asignación **"="** y **"-="**. De tal forma que al encontrar un operador de asignación **"-="** el token queda de la siguiente manera : **< -=, >.**

7. < id, *Pos_TS* >

El analizador cuando encuentra una palabra bien formada y no es palabra reservada la guarda en un token cuyo código es **"*id*"** y donde **"*Pos_TS*"** es la posición que ocupa este identificador en la tabla de símbolos.

8. < keyword, >

El analizador cuando encuentra una palabra bien formada y es palabra reservada la guarda en un token cuyo código es **"*keyword*"** que es la palabra reservada encontrada. Por ejemplo, al encontrar la palabra reservada **"If"**, el token quedará de la siguiente manera : **< If, >.** Las Keyword que detectamos son **: Boolean, string, false, function, if, int, return, true, false, var, while, print, input.**

9. < (, > Token paréntesis abierto (**"("**), propio del lenguaje.

10. <), > Token paréntesis cerrado (**")"**), propio del lenguaje.

11. < {, > Token llave abierta (**“{“**), propio del lenguaje.

12. < }, > Token llave cerrada (**“}”**), propio del lenguaje.

13. < ;, > Token punto y coma (**“;”**), propio del lenguaje.

14. < ,, > Token coma (**“,”**), propio del lenguaje.

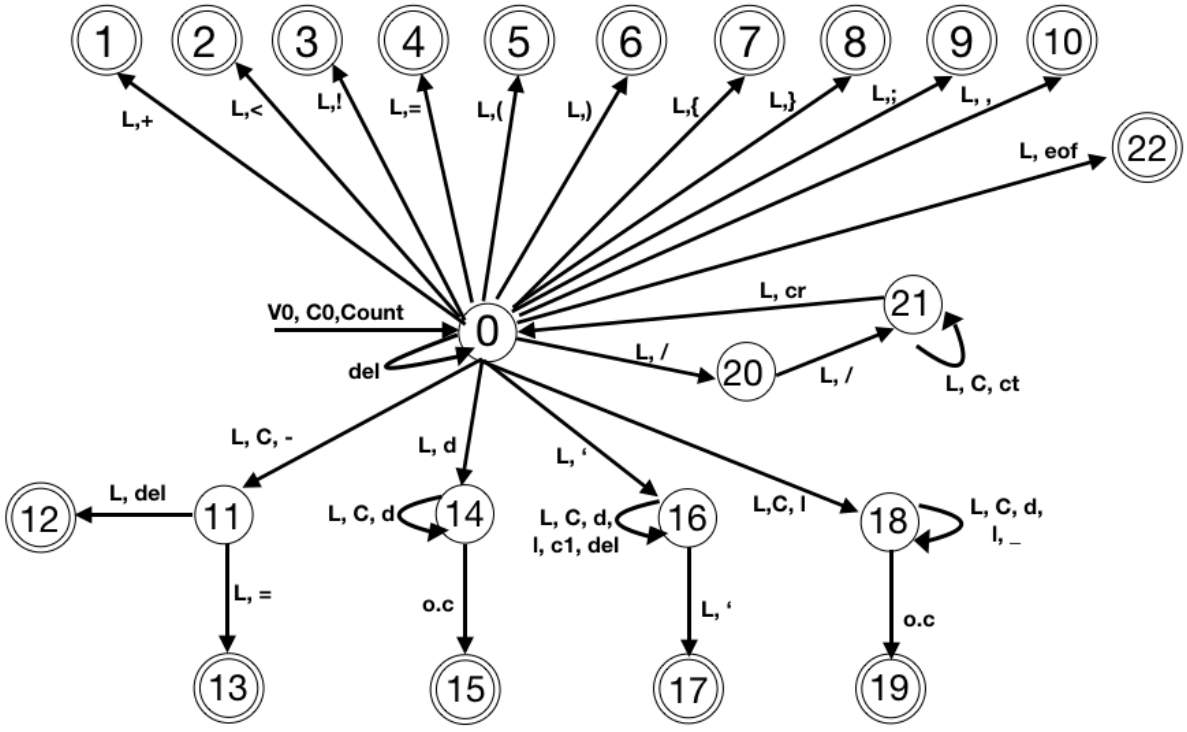
15. < eof, > Token end of file (**eof**), token para conocer fin del archivo

B. Gramática

S => + | < | ! | = | (|) | { | } | ; | , | -A | dB | ‘C | ID | /E | delS | eof
A => = | λ
B => dB | λ
C => dC | IC | delC | c1C | ‘
D => dD | ID | _D | λ
E => /F
F => ctF | crS

d: [0,9], l: [a,z] U [A,Z], del: <blanco>, c1: cualquier carácter excepto ‘
cr: carriage return (Salto de linea) eof: end of file ct: cualquier caracter

C. Autómata



D. Acciones semánticas

L	Leo el siguiente caracter
C	Concat(lexema, l/d) Voy formando el lexema al concatenar lo que voy leyendo
C0	Iniciamos la variable de lectura lexema a “”
VO	Iniciamos la variable de lectura valor a 0
Count	Iniciamos el valor del contador a 0
0:1	Gen_token (+,)
0:2	Gen_token (<,)
0:3	Gen_token (!,)
0:4	Gen_token (=,)
0:5	Gen_token ((,)
0:6	Gen_token ( ),)
0:7	Gen_token ({,)
0:8	Gen_token (},)
0:9	Gen_token (;,)
0:10	Gen_token (,,)
0:22	Gen_token (eof,)
0:11	C0 = C (C0, -)
11:12	Gen_token (-,)
11:13	Gen_token (-=,)
0:14	V0 = C (V0, d)
14:14	V0 = C (V0, d)
14:15	If V0 <= 32767 then Gen_token (number, valor) else Error
0:16	C0 = C (C0, ‘) Count = 1
16:16	C0 = C (C0, d/l/del/c1) Count ++

16:17	**If(Count < 64) then** **Gen_token (string, lexema)** **else Error**
0:18	**C0 = C (C0, l)**
18:18	**C0 = C (C0, d/l/_)**

18:19	**If (buscarKeyword(C0) == true) then**
	Gen_token (Keyword,)
	else
	Gen_token (id, Pos_TS)
	
	BuscarKeyword devuelve true si C0 es una Keyword
	Pos_TS es un número unitario que sirve para identificar que posición ocupa id en la TS, se actualiza al último Pos_TS de la TS a Pos_TS++

E. Errores

En esta fase del análisis sólo detectamos tres tipos de errores destacados :

- Identificadores mal formados: identificadores que empiezan por números seguidos por letras, ya que no hay ningún caso en JavaScript en el que un número vaya seguido inmediatamente por una letra, por lo que suponemos que es un identificador que empieza por un carácter ilegal.

- Cadenas de longitud mayor a 64 caracteres : si la cadena analizada, en nuestro caso las cadenas **'….'** tienen una longitud mayor de 64 caracteres el analizador notificará del error debido al exceso de longitud de la cadena.

- Los números se tienen que poder representar con una palabra, por lo que cualquier número mayor de 32767 generará un error por número mayor de lo permitido.

Cualquier transición que no esté en el autómata generará un error. En caso de haberse detectado errores, el analizador generará un documento al final de su ejecución en el que se detallará el tipo de error detectado y la línea en la que se ha detectado dicho error.

3. Diseño del analizador sintáctico

El analizador sintáctico implementado sigue la técnica descendente con tablas. Este analizador sintáctico recibe los tokens del analizador léxico para comprobar que la estructura es la correcta, una vez comprobado que la estructura es correcta se pasa dicha estructura para que el analizador semántico la analice.

A. Gramática

La gramática desarrollada debe cumplir que sea del tipo 2, es decir que para ello debe cumplir que no es ambigua, no debe ser recursiva por la izquierda y tiene que estar factorizada. Este analizador debe cumplir con la condición LL(1). Para ello hemos comprobado que en cada noTerm la conjunción de los FIRST da el conjunto vacío. En caso de que apareciese un lamda (en la gramática del noTerm), también con los follow del noTerm.

P -> B P | F P | Eof
B -> var T S2 ; | if (E) B1 | while (E) { C } | S
B1 -> S | { C }
T -> int | string | boolean
S -> id S1 | return X ; | print (L) ; | input (id) ;
S1 -> = E ; | -= E ; | (L) ;
S2 -> id S3
S3 -> = E S4 | , S2 | LAMBDA
S4 -> , S2 | LAMBDA
X -> E | LAMBDA
C -> B C | LAMBDA
F -> function H id (A) { C }
H -> T | LAMBDA
A -> T id K | LAMBDA
K -> , T id K | LAMBDA
L -> E Q | LAMBDA
Q -> , E Q | LAMBDA
E -> ! E | U R
R -> < U | LAMBDA
U -> V U2
U1 -> + V | - V | * V | / V | % V
U2 -> U1 U2 | LAMBDA
V -> id V1 | number | (E) | cadena | false | true
V1 -> (L) | LAMBDA

B. Calculo de First y Follow

	FIRST	FOLLOW
P	Eof, var, if, while, id, return, print, input, function	$
B	var, if, while, id, return, print, input	var, if, while, id, return, print, input, Eof, function, }
B1	{, id, return, print, input	var, if, while, id, return, print, input, Eof, function, }
T	int, string, boolean	id
S	id, return, print, input	var, if, while, id, return, print, input, Eof, function, }
S1	= ,-=, (	var, if, while, id, return, print, input, Eof, function, }
S2	id	;
S3	=, ,, λ	;
S4	,, λ	;
X	λ, !, id, number, (, cadena, false, true	;
C	λ, var, if, while, id, return, print, input	}
F	function	Eof, var, if, while, id, return, print, input, function
H	λ, int, string, boolean	id
A	λ, int, string, boolean	)
K	,, λ	)
L	λ, !, id, number, (, cadena, false, true	)
Q	,, λ	)
E	!, id, number, (, cadena, false, true	), ,, ;
R	<, λ	), ,, ;
U	id, number, (, cadena, false, true	<,), ,, ;
U1	+, -, *, /, %	+, -, *, /, %, <,), ,, ;
U2	λ, +, -, *, /, %	<,), ,, ;
V	id, number, (, cadena, false, true	+, -, *, /, %, <,), ,, ;
V1	(, λ	+, -, *, /, %, <,), ,, ;

C. Producciones

1	P → B P	31	H → T
2	P → F P	32	H → λ
3	P → Eof	33	A → T id K
4	B → var T S2 ;	34	A → λ
5	B → if (E) B1	35	K → , T id K
6	B → while (E) { C }	36	K → λ
7	B → S	37	L → E Q
8	B1 → S	38	L → λ
9	B1 → { C }	39	Q → , E Q
10	T → int	40	Q → λ
11	T → string	41	E → ! E
12	T → boolean	42	E → U R
13	S → id S1	43	R → < U
14	S → return X ;	44	R → λ
15	S → print (L) ;	45	U → V U2
16	S → input (id) ;	46	U1 → + V
17	S1 → = E ;	47	U1 → - V
18	S1 → -= E ;	48	U1 → * V
19	S1 → (L) ;	49	U1 → / V
20	S2 → id S3	50	U1 → % V
21	S3 → = E S4	51	U2 → U1 U2
22	S3 → , S2	52	U2 → λ
23	S3 → λ	53	V → id V1
24	S4 → , S2	54	V → number
25	S4 → λ	55	V → (E)
26	X → E	56	V → cadena
27	X → λ	57	V → false
28	C → B C	58	V → true
29	C → λ	59	V1 → (L)
30	F → function H id (A) { C }	60	V1 → λ

D. Procedimiento

La implementación del código se encuentra en la clase SyntaxAnalyzer, en esta clase nos encontramos con la siguiente implementación del analizador sintáctico.

Nuestro analizador sintáctico parte de una cadena de tokens, que finaliza con el símbolo eof, que recibe del analizador léxico.

Al inicio de la ejecución el estado del programa es el siguiente:

Current_token apunta al primer token de la cadena
En la pila se encuentra P (El axioma de nuestra gramática)

```
While (cadena de tokens > 0)
        If (pila.peek es un terminal) then {
                If pila.peek == current_oken
                        Then pila.pop
                        current_token = sig_token
                Else
                        Error
        }
        Else
                switch(pila.peek) {
                  States.P -> stateP(currentToken)
                  States.B -> stateB(currentToken)
                  States.B1 -> stateB1(currentToken)
                  States.T -> stateT(currentToken)
                  States.S -> stateS(currentToken)
                  States.S1 -> stateS1(currentToken)

                States.S2 -> stateS2(currentToken)
                States.S3 -> stateS3(currentToken)
                States.S4 -> stateS4(currentToken)
                States.X -> stateX(currentToken)
                States.C -> stateC(currentToken)
                States.F -> stateF(currentToken)
                States.H -> stateH(currentToken)
                States.A -> stateA(currentToken)
                States.K -> stateK(currentToken)
                States.L -> stateL(currentToken)
                States.Q -> stateQ(currentToken)
                States.E -> stateE(currentToken)
                States.R -> stateR(currentToken)
                States.U -> stateU(currentToken)
                States.U1 -> stateU1(currentToken)
                States.U2 -> stateU2(currentToken)
                States.V -> stateV(currentToken)
                States.V1 -> stateV1(currentToken)
}
```

4. Diseño del analizador semántico

El analizador semántico se encarga de comprobar que las estructuras obtenidas por el analizador sintáctico están bien formadas. Realiza comprobaciones sobre la tabla de símbolos y estudia el significado de los elementos.

La implementación del analizador semántico se encuentra en la clase syntaxSemanticAnalyzer, en esta clase hemos implementado una serie de variables y funciones para la comprobación de tipos necesarias para el correcto funcionamiento del analizador, además de variables de localización para saber en que caso nos encontramos como **inFunctionDeclaration** para saber que nos encontramos en el caso de una función o **inVariableDeclaration** para saberlo en caso de variables.

Funciones auxiliares :

searchInTS() = busca el elemento en la tabla de símbolos, devuelve null si el elemento no ha sido ya añadido a la tabla de símbolos.

addInTS() = una vez encontrado el elemento actualizamos en la tabla de símbolos los atributos que deseemos.

getType() = devuelve el tipo del elemento.

auxType = variable local para guardar el tipo de un elemento

returnType = variable local para guardar el tipo del return

Traducción dirigida por sintaxis

P -> B P => P.tipo = correctType if B.tipo = correctType && P.tipo = correctType

P -> F P => P.tipo = correctType if B.tipo = correctType && P.tipo = correctType

P -> Eof => P.tipo = correctType

B -> var T S2 ; => inVariableDeclaration = true, IF(searchInTS(id) != null)
THEN Error (Semantic Error, is defined);
ELSE AddInTS(id, T.tipo); B.tipo = CorrectType; inVariableDeclaration = false

B → if (E) B1 => B.tipo = correctType if E.tipo = logico && B1.tipo = correctType else errorType

B → while (E) { C } => B.tipo = correctType if E.tipo = logico && C.tipo = correctType else errorType

B → S => B.tipo = S.tipo; If S.tipo = correctType then B.type = correctType else errorType

B1 → S => B1.tipo = S.tipo; If S.tipo = correctType then B1.type = correctType else errorType

B1 → { C } => B1.tipo = C.tipo; If C.tipo = correctType then B1.tipo = correctType else errorType

T → int => T.tipo = entero

T → string => T.tipo = cadena

T → boolean => T.tipo = logico

S → id S1 => if getType(id) = auxType then S.tipo = correctType else errorType

S → return X ; => if X.tipo = returnType then S.tipo = correctType else errorType

S → print (L) ; => if L.tipo != emptyType then S.tipo = correctType else errorType

S → input (id) ; => if searchInTS(id) == null then errorType
Elseif getType(id) is (entero || cadena) then correctType else errorType

S1 → = E ; => S1.tipo = correctType if E.tipo = correctType; S1.tipo = E.tipo

S1 → -= E ; => S1.tipo = correctType if E.tipo = correctType; S1.tipo = E.tipo

S1 → (L) ; => S1.tipo = L.tipo

S2 → id S3 => S2.tipo = correctType if S3.tipo =correctType =>
if searchInTS(id) != null then
auxType
addInTS(id. auxType)

S3 → = E S4 => S3.tipo = correctType if E.tipo = auxType && S4.tipo = correctType else errorType

S3 → , S2 => S3.tipo = S2.tipo

S3 → λ => S3.tipo = correctType

S4 → , S2 => S4.tipo = S2.tipo

S4 → λ => S4.tipo = correctType

X → E => X.tipo = E.tipo

X → λ => X.tipo = correctType

C → B C => C.tipo = correctType if B.tipo = C.tipo = correctType else errorType

C → λ => C.tipo = correctType

F → function H id (A) { C } => inFunctionDeclaration = true,

if (getType(id)) != null) then error (“is defined”)
else addInTS(id, funcion)
if (A.tipo != emptyType)
then addInTS (A.tipo) & parameterCount++
else parameterCount = 0
if (H.tipo == emptyType) then addInTS(id, -)
Else addInTS(id, H.tipo) & returnType = H.tipo

inFunctionDeclaration = false

F.tipo = C.tipo if (C.tipo = correctType) else errorType

H → T => H.tipo = T.tipo

H → λ => H.tipo = correctType

A → T id K => addInTS(id,T.tipo) {
A.tipo = K.tipo if (K.tipo = emptyType) else T.tipo x K.tipo }

A → λ => A.tipo = correctType

K → , T id K => addInTS(id,T.tipo) {
K.tipo = K.tipo if (K.tipo = emptyType) else T.tipo x K.tipo }

K → λ => K.tipo = correctType

L → E Q => if Q.tipo = emptyType then L.tipo = E.tipo else L.tipo = E.tipo x Q.tipo

L → λ => L.tipo = correctType

Q → , E Q => if Q.tipo = emptyType then Q.tipo = E.tipo else Q.tipo = E.tipo x Q.tipo

Q → λ => Q.tipo = correctType

E → ! E => E.tipo = correctType if E.tipo = logico else errorType

E → U R

R → < U => R.tipo = correctType if U.tipo = entero else errorType

R → λ => R.tipo = correctType

U → V U2 => U.tipo = if U2.tipo = entero Then
if V.tipo = entero then U.tipo = entero else errorType

U1 → + V => U1.tipo = entero if V.tipo = entero else errorType

U1 → - V => U1.tipo = entero if V.tipo = entero else errorType

U1 → * V => U1.tipo = entero if V.tipo = entero else errorType

U1 → / V => U1.tipo = entero if V.tipo = entero else errorType

U1 → % V => U1.tipo = entero if V.tipo = entero else errorType

U2 → U1 U2 => U2.tipo = entero if U1.tipo = entero else errorType

U2 → λ => U2.tipo = correctType

V → id V1 => V.tipo = if (V1.tipo = correctType) then{
If (searchInTS(id) == null) then errorType Else getType(id)}
Elseif (getType(id) = V1.tipo) then searchInTS(id)

Else errorType

V → number => V.tipo = entero

V → (E) => V.tipo = E.tipo

V → cadena => V.tipo = cadena

V → false => V.tipo = logico

V → true => V.tipo = logico

V1 → (L) => V1.tipo = L.tipo

V1 → λ => V1.tipo = correctType

5. Diseño de la tabla de símbolos

El analizador va guardando en una tabla todas las referencias a identificadores de variables y funciones que encuentra en el código, y al final de la ejecución vuelca la tabla en un archivo de acuerdo al siguiente formato:

• Una cabecera de tabla con el nombre de la misma (en este caso 'TABLA DE IDENTIFICADORES') seguido del carácter '#' y el número identificador de la tabla (1 en este caso).

• Una línea en blanco
• Una entrada por lexema que tendrá el siguiente formato:

‣ Una línea que comienza por "* LEXEMA :" seguido del identificador entre comillas simples, recordamos que los identificadores pueden ser identificadores de variables o identificadores de función.

‣ Una línea "ATRIBUTOS :" formada por los siguientes campos :

+ tipo : el tipo que sea el identificador { **funcion, entero, logico, cadena**}

+ tipoRetorno : (**solo para los identificadores funciones**) el tipo de retorno de la función :{ **entero, logico, cadena, void**}

+ tipoParámetros : (**solo para los identificadores funciones**) los tipos que tienen los parámetros de la función { **entero, lógico, cadena** }
+ numParámetros : (**solo para los identificadores funciones**) el numero de parámetros que tiene la función.
+ id : el numero único que sirve para identificar a cada identificador, el id es único, no hay repetidos.

‣ Una línea formada por 25 caracteres '-'. Que sirve de separación entre los diferentes identificadores que forman la tabla de símbolos.

Los identificadores (identificadores de funciones y variables) se introducen en la tabla de símbolos una vez se comprueba que ese identificador no ha sido introducido anteriormente en la tabla de símbolos. Cada Identificador tiene un id único que no se repite, este id sirve para identificar la posición que ocupa en la tabla de símbolos. El id se va incrementando en una unidad cada vez que se introduce un nuevo símbolo en la tabla.

6. Anexo

(*Prueba1*) - Primer caso de prueba sin errores:

```
function int test1(int i, boolean ok){
    var int num;
    input(num);
    while(i < 4){
        num -= i;
        if (num < 8) ok = true;
    }
    var string hola = 'Hola caracola';
    print(hola);
    return num;
}
```

TOKENS GENERADOS:

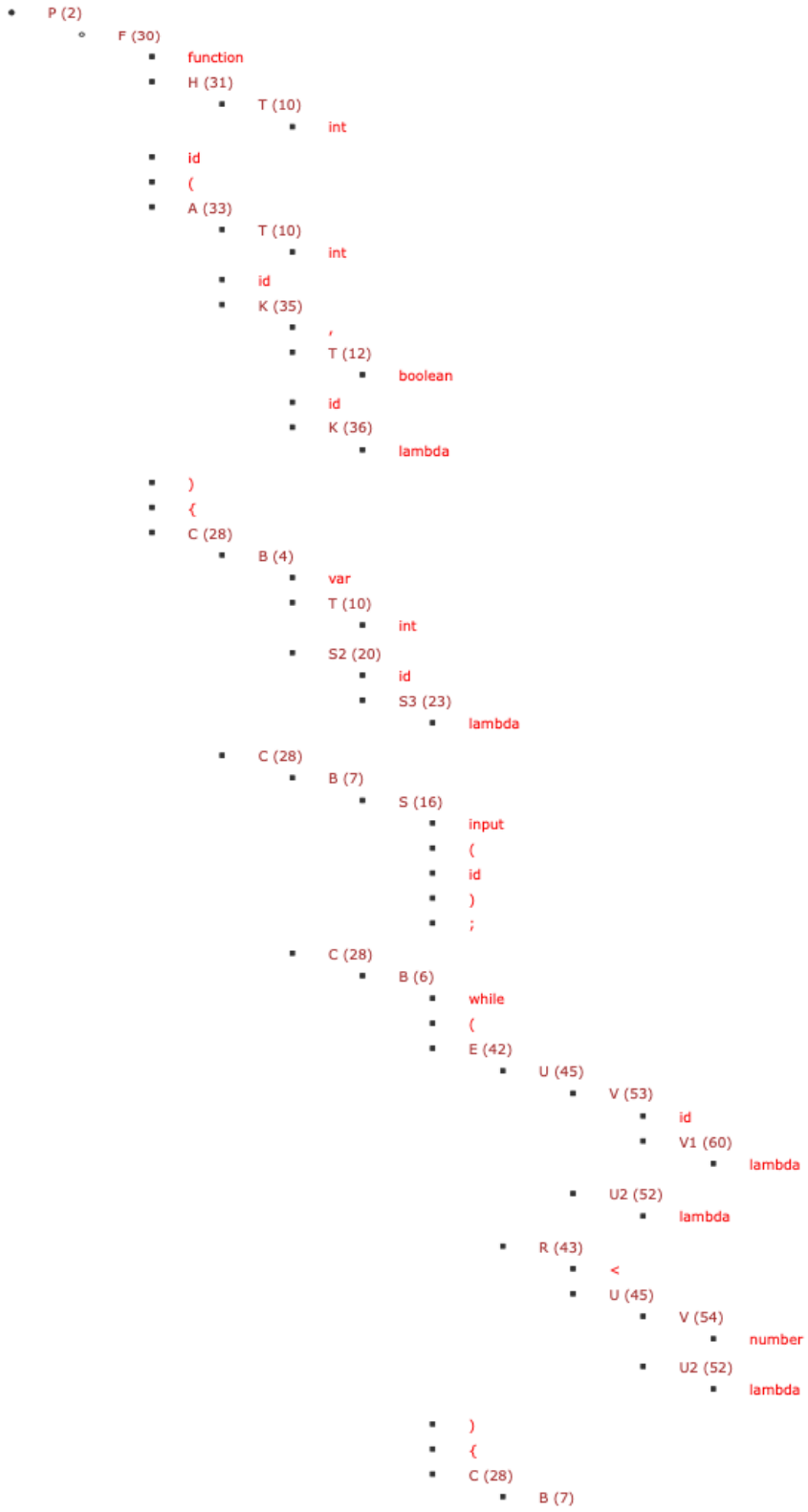
<function, >
<int, >
<id, 0>
<(, >
<int, >
<id, 1>
<,, >
<boolean, >
<id, 2>
<), >
<{, >
<var, >
<int, >
<id, 3>
<;, >
<input, >
<(, >
<id, 3>
<), >
<;, >
<while, >
<(, >
<id, 1>
<<, >
<number, 4>
<), >
<{, >
<id, 3>
<-=, >
<id, 1>
<;, >
<if, >
<(, >
<id, 3>
<<, >
<number, 8>
<), >
<id, 2>
<=, >
<true, >
<;, >
<}, >
<var, >
<string, >
<id, 4>
<=, >
<cadena, 'Hola caracola'>
<;, >
<print, >
<(, >
<id, 4>
<), >
<;, >
<return, >
<id, 3>
<;, >
<}, >

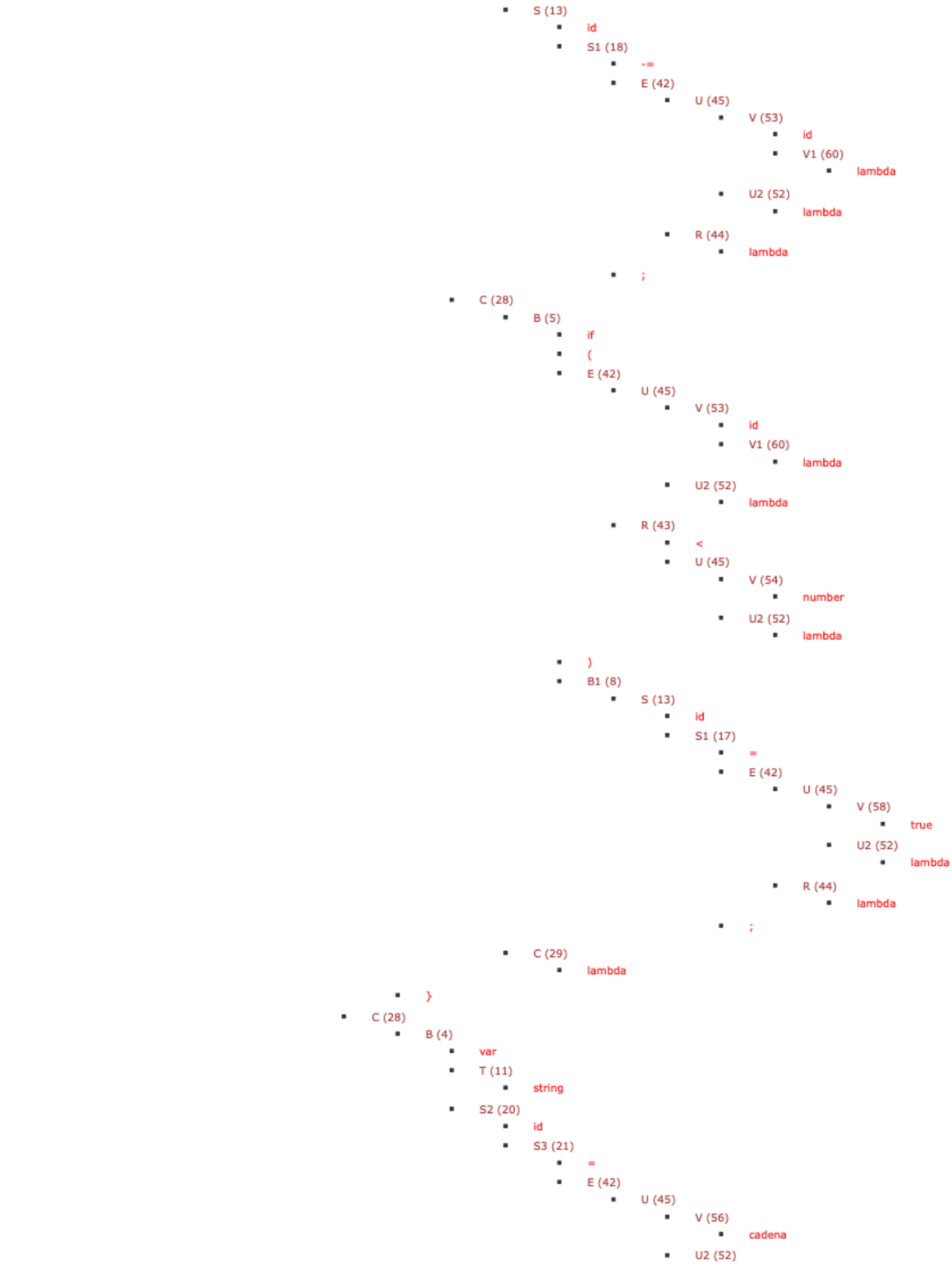
TABLA DE SIMBOLOS

TABLA DE IDENTIFICADORES #1:

```
* LEXEMA : 'test1'
  ATRIBUTOS :
  + tipo: 'funcion'
  + tipoRetorno: 'entero'
  + tipoParametros: 'entero, logico'
  + numParametros: '2'
  + id: 0
------------------------
* LEXEMA : 'i'
  ATRIBUTOS :
  + tipo: 'entero'
  + id: 1
------------------------
* LEXEMA : 'ok'
  ATRIBUTOS :
  + tipo: 'logico'
  + id: 2
------------------------
* LEXEMA : 'num'
  ATRIBUTOS :
  + tipo: 'entero'
  + id: 3
------------------------
* LEXEMA : 'hola'
  ATRIBUTOS :
  + tipo: 'cadena'
  + id: 4
------------------------
```

ARBOL VAST :


P (2)
F (30)
function
H (31)
T (10)
int
id
(
A (33)
T (10)
int
id
K (35)
,
T (12)
boolean
id
K (36)
lambda
)
{
C (28)
B (4)
var
T (10)
int
S2 (20)
id
S3 (23)
lambda
C (28)
B (7)
S (16)
input
(
id
)
;
C (28)
B (6)
while
(
E (42)
U (45)
V (53)
id
V1 (60)
lambda
U2 (52)
lambda
R (43)
<
U (45)
V (54)
number
U2 (52)
lambda
)
{
C (28)
B (7)


S (13)
id
S1 (18)
-=
E (42)
U (45)
V (53)
id
V1 (60)
lambda
U2 (52)
lambda
R (44)
lambda
;
C (28)
B (5)
if
(
E (42)
U (45)
V (53)
id
V1 (60)
lambda
U2 (52)
lambda
R (43)
<
U (45)
V (54)
number
U2 (52)
lambda
)
B1 (8)
S (13)
id
S1 (17)
=
E (42)
U (45)
V (58)
true
U2 (52)
lambda
R (44)
lambda
;
C (29)
lambda
}
C (28)
B (4)
var
T (11)
string
S2 (20)
id
S3 (21)
=
E (42)
U (45)
V (56)
cadena
U2 (52)


lambda
R (44)
lambda
S4 (25)
lambda
C (28)
B (7)
S (15)
print
(
L (37)
E (42)
U (45)
V (53)
id
V1 (60)
lambda
U2 (52)
lambda
R (44)
lambda
Q (40)
lambda
)
;
C (28)
B (7)
S (14)
return
X (26)
E (42)
U (45)
V (53)
id
V1 (60)
lambda
U2 (52)
lambda
R (44)
lambda
;
C (29)
lambda
}
P (3)
Eof

(*Prueba2*) - Segundo caso de pruebas sin errores :

```
var int a = 2, b = 3;
function int suma(int i, int j) {
    return i + j;
}

var int c = suma(a, b);
```

(*Prueba3*) - Tercer caso de prueba sin errores :

```
var string num;
num = 'hola';
var int a = 7;
while(10 < a){
    a -= 1;
    print(num);
}
```

(*Prueba4*) - Cuarto caso de prueba sin errores :

```
function boolean main (string hello, int world){
    var int a = 1;
    var int b = 2;
    var int c = 3;
    var int d = 4;
    var boolean t = true;
    var boolean f = false;
    while(0 < d){
        c = c*b+a;
    }
    var boolean r = f;
    if(c < 50) r = t;
    return r;
}
```

(*Prueba5*) - Quinto caso de prueba sin errores :

```
function boolean test5 (string name, int age, boolean isClient){
    if (!isClient) isClient = true;
    var string  u = 'underage';
    while (age < 18){
        print(name);
```

```
        print(u);
        age = age + 1;
    }
    return isClient;
}
```

(*Prueba6*) - Primer caso de prueba con errores :

```
var int a = 2, b = 3;
function int suma(int i, int j) {
    return i + ;
}

c = suma(a, b);
```

ERRORES : Syntax error. State V received ; token.

(*Prueba7*) - Segundo caso de prueba con errores :

```
function int test2(int i, boolean ok){
    var int num;
    input(num);
    while(i < 'patata'){
        num -= i;
        if (num < ok) ok = true;
    }
    var string hola = 'Hola caracola';
    print(hola);
    return ok;
}
```

ERRORES : Semantic error: type mismatch. Expected INT found STRING.

(*Prueba8*) - Tercer caso de prueba con errores :

```
var string num;
num = 'hola';
a = 7;
while(10){
    a -= 1;
    print(num);
}
```

ERRORES : Semantic error: a is not defined.

(*Prueba9*) - Cuarto caso de prueba con errores :

```
function boolean main (string hello, int world){
    var int a = 1;
    var int b = 2;
    var int c = 3;
    var int d = 4;
    var boolean t = true;
    var boolean f = false;
    while(0 < d){
        f = c*b+a;
    }
    var boolean r = f;
    if(c < 50) r = t;
    return r;
}
```

ERRORES : Semantic error: type mismatch.

(*Prueba10*) - Quinto caso de prueba con errores :

```
function test5 (string name, int age, boolean isClient){
    if (!isClient) isClient = true;
    var string = 'underage'
    while (age < 18){
        print(name);
        print(u);
        age = age + string;
    }
}
```

ERRORES : Syntax error. State S2 received = token.