MemoriaDescendente-Recursivo.pdf

javig23

Procesadores de Lenguajes

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Tabla de contenido

Diseño del Analizador Léxico

Definición de tokens

Para la creación del analizador léxico, lo primero a tener en cuenta es qué elementos vamos a considerar como tokens. En esta implementación, los elementos considerados como tokens son los siguientes:

ELEMENTO	CÓDIGO	ATRIBUTO
boolean	PalResboolean	-
function	PalResfunction	-
if	PalResif	-
input	PalResinput	-
int	PalResint	-
let	PalReslet	-
print	PalResprint	-
return	PalResreturn	-
String	PalResstring	-
while	PalReswhile	-
constante entera	cte	número
Cadena(“)	cad	lexema
Identificador	ID	posTS
=	asig	-
|=	oig	-
,	coma	-
;	puntocoma	-
(	ParIzq	-
)	ParDer	-
{	CorIzq	-
}	CorDer	-
+ (op aritmético)	OpArSuma	-
|| (op lógico)	OpLogO	-
== (op relacional)	OpRlg	-
eof	EOF	-

Definición de la gramática

El siguiente paso se corresponde con la construcción de la gramática, que toma el siguiente resultado:

0- S -> del S | d A | l B | / C | | E | “ F | = G| + | , | ; | (|) | { | } | eof
1- A -> d A | λ
2- B -> d B | l B | _ B| λ
3- C -> / D
4- D -> c D | cr S
5- E -> = | |
6- F -> c F | “
7- G -> = | λ

Autómata Finito Determinista

A continuación, se diseña el autómata finito determinista correspondiente al lenguaje:

The HackerWay
Profesionales de la seguridad informática

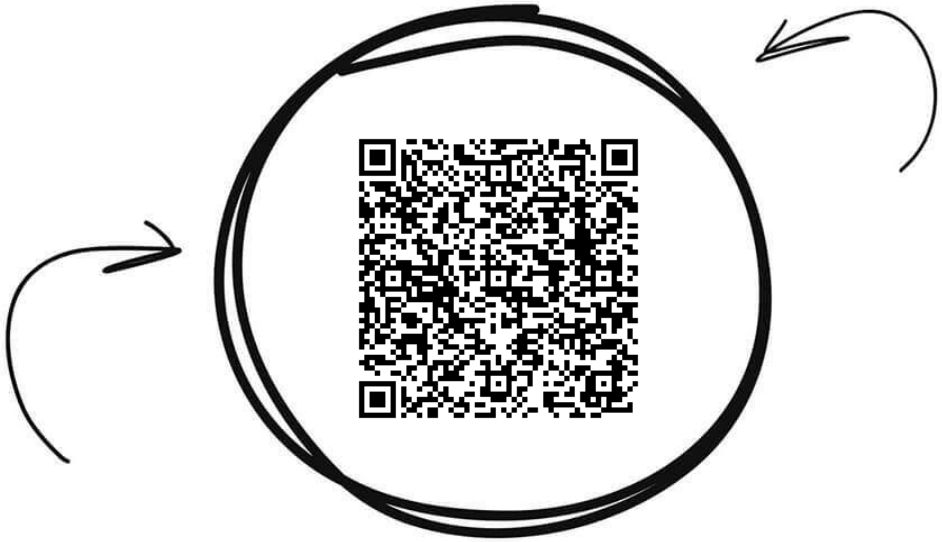
FORMACIÓN 100% PRÁCTICA EN CIBERSEGURIDAD

Estudia ahora y paga al encontrar trabajo. Accede al mundo laboral con nuestra formación de 6 meses en ciberseguridad

¡Transforma tu futuro en ciberseguridad!

Escanea el QR para descubrir cómo empezar

Procesadores de Lenguajes

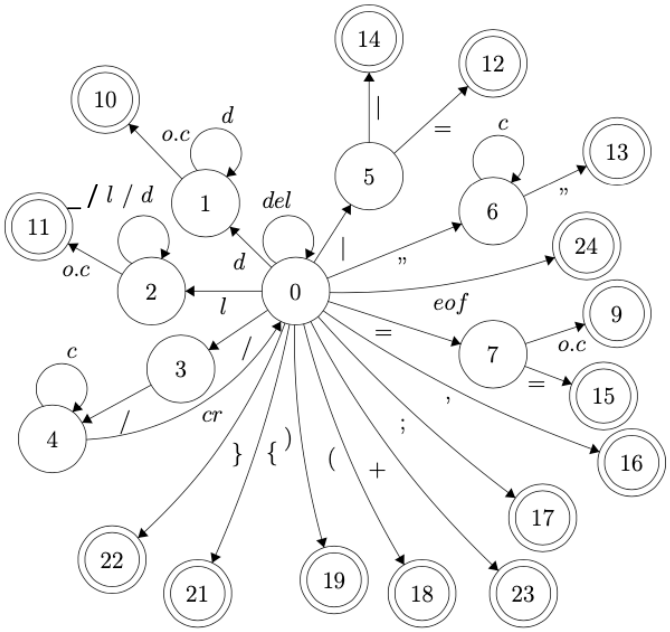


Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR



Acciones semánticas

Las acciones semánticas resultan ser las siguientes:

LEER(): todas las transiciones menos las de o.c.
0-24: GENTOKEN(EOF,-)
0-23: GENTOKEN(OpArSuma,-)
0-22: GENTOKEN(CorDer,-)
0-21: GENTOKEN(CorIzq,-)
0-19: GENTOKEN(ParDer,-)
0-18: GENTOKEN(ParIzq,-)
0-17: GENTOKEN(puntocoma,-)
0-16: GENTOKEN(coma,-)
8-15: GENTOKEN(OpRIg,-)
7-9: GENTOKEN(asig,-)
0-6: lexema:=null; cont:=0
6-6: lexema:=lexema+c; cont++
6:13: if(cont>64) then Error("Cadena fuera de rango")
else GENTOKEN(cad, lexema)
5-12: GENTOKEN(oig,-)
5-14: GENTOKEN(OpLogO,-)
0-1: numero:=d
1-1: numero:=numero*10 + d
1-10: if(numero>32767) then Error("Entero fuera de rango")
else GENTOKEN(cte, numero)
0-2: lex:=l
2-2: lex:=lex + l/d
2-11: if(lex ∈ PalRes) then GenToken(PalRes,-)
else p := buscarTS(lex)
if(p=null) then p := insertarTS(lex); GenToken(id,p)
else Error("Identificador ya declarado")

Errores

Todas las transiciones que no aparecen en el AFD producen un error.

Diseño de la Tabla de Símbolos.

Se trata de una estructura de datos donde se almacena toda la información relevante sobre los identificadores del programa. Todos los módulos van a tener acceso a ella. Está compuesta por una entrada para cada identificador con una serie de campos para atributos:

TABLA DE SÍMBOLOS #Nº
* lexema : '…'
  + tipo :
  + despl :
  + numParam :
  + tipoParamXX :
  + tipoRetorno :
  + etiqFuncion :

Diseño del Analizador Sintáctico

Gramática LL(1)

Axioma = P

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

Terminales = { let if while function int boolean string return input ( ) { } ; , = |= == id entero cadena + || print eof }
Producciones = {

1. P -> B P
2. P -> F P
3. P -> eof

4. B -> let T id ;
5. B -> if (E) S
6. B -> S
7. B -> while (E) { C }

8. T -> int
9. T -> boolean
10. T -> string

11. S -> id S1
12. S -> return X ;
13. S -> print (E) ;
14. S -> input (id) ;

15. S1 -> = E ;
16. S1 -> (L) ;
17. S1 -> |= E ;

18. X -> E
19. X -> lambda

20. C -> B C
21. C -> lambda

22. L -> E Q
23. L -> lambda

24. Q -> , E Q
25. Q -> lambda

26. F -> function id H (A) { C }

27. H -> T
28. H -> lambda

29. A -> T id K
30. A -> lambda

31. K -> , T id K
32. K -> lambda

33. E -> R E1

34. E1 -> == R E1
35. E1 -> lambda

36. R -> U R1

37. R1 -> + U R1
38. R1 -> lambda

39. U -> || U
40. U -> V

41. V -> id V1
42. V -> (E)
43. V -> entero
44. V -> cadena

45. V1 -> (L)
46. V1 -> lambda

}

Demostración de que la gramática es LL(1)

Análisis hecho con la herramienta SDGLL(1)

```
Analizando símbolo A
Analizando producción A -> T id K
Analizando símbolo T
Analizando producción T -> int
FIRST de T -> int  =  { int }
Analizando producción T -> boolean
FIRST de T -> boolean  =  { boolean }
Analizando producción T -> string
FIRST de T -> string  =  { string }
FIRST de T  =  { boolean int string }
FIRST de A -> T id K  =  { boolean int string }
Analizando producción A -> lambda
FIRST de A -> lambda  =  { lambda }
FIRST de A  =  { boolean int string lambda }
Calculando FOLLOW de A
FOLLOW de A  =  { ) }
Analizando símbolo B
Analizando producción B -> let T id ;
FIRST de B -> let T id ;  =  { let }
Analizando producción B -> if ( E ) S
FIRST de B -> if ( E ) S  =  { if }
Analizando producción B -> S
Analizando símbolo S
Analizando producción S -> id S1
FIRST de S -> id S1  =  { id }
Analizando producción S -> return X ;
FIRST de S -> return X ;  =  { return }
Analizando producción S -> print ( E ) ;
FIRST de S -> print ( E ) ;  =  { print }
Analizando producción S -> input ( id ) ;
FIRST de S -> input ( id ) ;  =  { input }
FIRST de S  =  { id input print return }
FIRST de B -> S  =  { id input print return }
Analizando producción B -> while ( E ) { C }
FIRST de B -> while ( E ) { C }  =  { while }
FIRST de B  =  { id if input let print return while }
Analizando símbolo C
Analizando producción C -> B C
FIRST de C -> B C  =  { id if input let print return while }
Analizando producción C -> lambda
FIRST de C -> lambda  =  { lambda }
FIRST de C  =  { id if input let print return while lambda }
Calculando FOLLOW de C
```

FOLLOW de C = { } }
Analizando símbolo E
Analizando producción E -> R E1
Analizando símbolo R
Analizando producción R -> U R1
Analizando símbolo U
Analizando producción U -> || U
FIRST de U -> || U = { || }
Analizando producción U -> V
Analizando símbolo V
Analizando producción V -> id V1
FIRST de V -> id V1 = { id }
Analizando producción V -> (E)
FIRST de V -> (E) = { (}
Analizando producción V -> entero
FIRST de V -> entero = { entero }
Analizando producción V -> cadena
FIRST de V -> cadena = { cadena }
FIRST de V = { (cadena entero id }
FIRST de U -> V = { (cadena entero id }
FIRST de U = { (cadena entero id || }
FIRST de R -> U R1 = { (cadena entero id || }
FIRST de R = { (cadena entero id || }
FIRST de E -> R E1 = { (cadena entero id || }
FIRST de E = { (cadena entero id || }
Analizando símbolo E1
Analizando producción E1 -> == R E1
FIRST de E1 -> == R E1 = { == }
Analizando producción E1 -> lambda
FIRST de E1 -> lambda = { lambda }
FIRST de E1 = { == lambda }
Calculando FOLLOW de E1
Calculando FOLLOW de E
Calculando FOLLOW de X
FOLLOW de X = { ; }
Analizando símbolo Q
Analizando producción Q -> , E Q
FIRST de Q -> , E Q = { , }
Analizando producción Q -> lambda
FIRST de Q -> lambda = { lambda }
FIRST de Q = { , lambda }
Calculando FOLLOW de Q
Calculando FOLLOW de L
FOLLOW de L = {) }
FOLLOW de Q = {) }
FOLLOW de E = {) , ; }
FOLLOW de E1 = {) , ; }
Analizando símbolo F
Analizando producción F -> function id H (A) { C }
FIRST de F -> function id H (A) { C } = { function }

FIRST de F = { function }
Analizando símbolo H
Analizando producción H -> T
FIRST de H -> T = { boolean int string }
Analizando producción H -> lambda
FIRST de H -> lambda = { lambda }
FIRST de H = { boolean int string lambda }
Calculando FOLLOW de H
FOLLOW de H = { (}
Analizando símbolo K
Analizando producción K -> , T id K
FIRST de K -> , T id K = { , }
Analizando producción K -> lambda
FIRST de K -> lambda = { lambda }
FIRST de K = { , lambda }
Calculando FOLLOW de K
FOLLOW de K = {) }
Analizando símbolo L
Analizando producción L -> E Q
FIRST de L -> E Q = { (cadena entero id || }
Analizando producción L -> lambda
FIRST de L -> lambda = { lambda }
FIRST de L = { (cadena entero id || lambda }
Analizando símbolo P
Analizando producción P -> B P
FIRST de P -> B P = { id if input let print return while }
Analizando producción P -> F P
FIRST de P -> F P = { function }
Analizando producción P -> eof
FIRST de P -> eof = { eof }
FIRST de P = { eof function id if input let print return while }
Analizando símbolo R1
Analizando producción R1 -> + U R1
FIRST de R1 -> + U R1 = { + }
Analizando producción R1 -> lambda
FIRST de R1 -> lambda = { lambda }
FIRST de R1 = { + lambda }
Calculando FOLLOW de R1
Calculando FOLLOW de R
FOLLOW de R = {) , ; == }
FOLLOW de R1 = {) , ; == }
Analizando símbolo S1
Analizando producción S1 -> = E ;
FIRST de S1 -> = E ; = { = }
Analizando producción S1 -> (L) ;
FIRST de S1 -> (L) ; = { (}
Analizando producción S1 -> |= E ;
FIRST de S1 -> |= E ; = { |= }
FIRST de S1 = { (= |= }
Analizando símbolo V1

Analizando producción V1 -> (L)
FIRST de V1 -> (L) = { (}
Analizando producción V1 -> lambda
FIRST de V1 -> lambda = { lambda }
FIRST de V1 = { (lambda }
Calculando FOLLOW de V1
Calculando FOLLOW de V
Calculando FOLLOW de U
FOLLOW de U = {) + , ; == }
FOLLOW de V = {) + , ; == }
FOLLOW de V1 = {) + , ; == }
Analizando símbolo X
Analizando producción X -> E
FIRST de X -> E = { (cadena entero id || }
Analizando producción X -> lambda
FIRST de X -> lambda = { lambda }
FIRST de X = { (cadena entero id || lambda }

Análisis concluido satisfactoriamente

Tabla Sintáctica

Tabla obtenida con la herramienta SDGLL(1)

Tabla sintáctica para Gram.ll1

	(	)	+	,	;	=	==	boolean	cadena	entero	eof	function	id	if	input	int	let	print	return	string	while	{	|=	||	}	$ (final de cadena)
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 T id ;	B → S	B → S	--	B → while (E) { C }	--	--	--	--	--
C	--	--	--	--	--	--	--	--	--	--	--	--	C → B C	C → B C	C → B C	--	C → B C	C → B C	C → B C	--	C → B C	--	--	--	C → 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	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
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 → E Q	L → lambda	--	--	--	--	--	--	L → E Q	L → E Q	--	--	L → E Q	--	--	--	--	--	--	--	--	--	--	L → E Q	--	--
P	--	--	--	--	--	--	--	--	--	--	P → eof	P → F P	P → B P	P → B P	P → B P	--	P → B P	P → B P	P → B P	--	P → B P	--	--	--	--	--
Q	--	Q → lambda	--	Q → , E Q	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
R	R → U R1	--	--	--	--	--	--	--	R → U R1	R → U R1	--	--	R → U R1	--	--	--	--	--	--	--	--	--	--	R → U R1	--	--
R1	--	R1 → lambda	R1 → + U R1	R1 → lambda	R1 → lambda	--	R1 → lambda	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
S	--	--	--	--	--	--	--	--	--	--	--	--	S → id S1	--	S → input (id) ;	--	--	S → print (E) ;	S → return X ;	--	--	--	--	--	--
S1	S1 → (L) ;	--	--	--	--	S1 → = E ;	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	S1 → = E ;	--	--	--
T	--	--	--	--	--	--	--	T → boolean	--	--	--	--	--	--	--	T → int	--	--	--	T → string	--	--	--	--	--	--
U	U → V	--	--	--	--	--	--	--	U → V	U → V	--	--	U → V	--	--	--	--	--	--	--	--	--	--	U → || U	--	--
V	V → (E)	--	--	--	--	--	--	--	V → cadena	V → entero	--	--	V → id V1	--	--	--	--	--	--	--	--	--	--	--	--	--
V1	V1 → (L)	V1 → lambda	V1 → lambda	V1 → lambda	V1 → lambda	--	V1 → lambda	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
X	X → E	--	--	--	X → lambda	--	--	--	X → E	X → E	--	--	X → E	--	--	--	--	--	--	--	--	--	--	X → E	--	--

Procedimientos

```
Proc P() {
        Begin
                if (sig_token ∈ {let,if,while,id,return,print,input}) then
                        B()
                else if (sig_token ∈{function}) then
                        F()
                else if (sig_token ∉ {$}) then
                        Error()
        End
```

```
}

Proc B() {
        Begin
                if (sig_token = {let}) then
                        equipara(<let>)
                        T()
                        equipara(<id>)
                        equipara(<;>)
                else if (sig_token = {if}) then
                        equipara(<if>)
                        equipara(<(>)
                        E()
                        equipara(<)>)
                        S()
                else if (sig_token ∈ {id,return,print,input}) then
                        S()
                else if (sig_token = {while})
                        equipara(<while>)
                        equipara(<(>)
                        E()
                        equipara(<)>)
                        equipara(<{>)
                        C()
                        equipara(<}>)
                else Error()
        End
}

Proc T() {
        Begin
                if (sig_token = {int}) then
                        equipara(<int>)
                else if (sig_token = {boolean}) then
                        equipara(<boolean>)
                else if (sig_token = {string}) then
                        equipara(<string>)
                else Error()
        End
}

Proc S() {
        Begin
                if (sig_token = {id}) then
                        equipara(<id>)
                        S1()
                else if (sig_token = {return}) then
                        equipara(<return>)
                        X()
                        equipara(<;>)
                else if (sig_token = {print}) then
                        equipara(<print>)
                        equipara(<(>)
```

```
                E()
                equipara(<)>)
                equipara(<;>)
        else if (sig_token = {input}) then
                equipara(<input>)
                equipara(<(>)
                equipara(<id>)
                equipara(<)>)
                equipara(<;>)
        else Error()
    End
}

Proc S1() {
    Begin
        if (sig_token = {=}) then
                equipara(<=>)
                E()
                equipara(<;>)
        else if (sig_token = {(}) then
                equipara(<(>)
                L()
                equipara(<)>)
                equipara(<;>)
        else if (sig_token = {|=}) then
                equipara(<|=>)
                E()
                equipara(<;>)
        else Error()
    End
}

Proc X() {
    Begin
        E()
        if (sig_token ∈ {;}) then { }
        else Error()
    End
}

Proc C() {
    Begin
        if (sig_token ∈ {let,if,while,id,return,print,input}) then
                B()
        else if (sig_token = { } }) then { }
        else Error()
    End
}

Proc L() {
    Begin
        if (sig_token ∈ {id, (,entero,cadena}) then
                E()
```

```
                Q()
            else if (sig_token ∈ {)} then { }
            else Error()
        End
}

Proc Q() {
        Begin
            if (sig_token = {,}) then
                equipara(<,>)
                E()
            else if (sig_token ∈ {)}) then { }
            else Error()
        End
}

Proc F() {
        Begin
            if (sig_token = {function}) then
                equipara(<function>)
                equipara(<id>)
                H()
                equipara(<(>)
                A()
                equipara(<)>)
                equipara(<{>)
                C()
                equipara(<}>)
            else Error()
        End
}

Proc H(){
        Begin
            T()
            if (sig_token ∈ {({}) then { }
            else Error()
        End
}

Proc A() {
        Begin
            T()
            equipara(<id>)
            K()
            if (sig_token ∈ {)}) then { }
            else Error()
        End
}

Proc K() {
        Begin
            if (sig_token = {,}) then
```

```
                    equipara(<,>)
                    T()
                    equipara(<id>)
                    K()
            else if (sig_token ∈ {)}) then { }
            else Error()
        End
}

Proc E() {
        Begin
            if(sig_token ∈ {(,cadena,entero,id,||}) then
                    R()
                    E1()
            else Error()
        End
}

Proc E1() {
        Begin
            if (sig_token = {==}) then
                    equipara(<==>)
                    R()
                    E1()
            else if (sig_token ∈ {), , , ;}) then { }
            else Error()
        End
}

Proc R() {
        Begin
            if(sig_token ∈ {(,cadena,entero,id,||}) then
                    U()
                    R1()
            else Error()
        End
}

Proc R1(){
        Begin
            if (sig_token = {+}) then
                    equipara(<+>)
                    U()
                    R1()
            else if (sig_token ∈ {, , ),==, ;}) then { }
            else Error()
        End
}

Proc U() {
        Begin
            if (sig_token ={||} then
                    equipara(<||>)
```

```
                U()
        else if (sig_token ∈ {(,cadena,entero,id}) then
                V()
        else Error()

    End
}

Proc V() {
    Begin
        if(sig_token = {id}) then
            equipara(<id>)
            V1()
        else if(sig_token = {(}) then
            equipara(<(>)
            E()
            equipara(<)>)
        else if(sig_token = {entero}) then
            equipara(<entero>)
        else if(sig_token = {cadena}) then
            equipara(<cadena>)
        else Error()
    End
}

Proc V1() {
    Begin
        if (sig_token = {(}) then
            equipara(<(>)
            L()
            equipara(<)>)
        else if (sig_token ∈ {, , ), ==,+,||, ;}) then { }
        else Error()
    End
}
```

Diseño del analizador semántico

Este es el esquema de traducción en el que se basa el analizador semántico:

P' ->	{TSG := Crear_TS() ; DespG := 0} P {DestruyeTS(TSG)}
P -> B P_1	{}

P -> F P_1	{}
P -> eof	{}
B -> let T id ;	{ZonaDeclaracion = true; InsertarTipoTS(id.pos, T.tipo); InsertarDespTS(id.pos, desp); Desp = Desp + T.ancho); ZonaDeclaracion = false}
B -> if (E) S	{if(E.tipo==boolean) then S.tipo else error("Condición del if tiene que ser booleana")}
B -> S	{B.tipo := S.tipo}
B -> while (E) { C }	{B.tipo := if(E.tipo == boolean) then C.tipo else error("Condicion del while tiene que ser booleana") C.bucle = true;}
T -> int	{T.tipo := entero T.ancho := 1}
T -> boolean	{T.tipo := logico T.ancho := 1}
T -> string	{T.tipo := cadena T.ancho := 64}
S -> id S1	{(id.tipo = buscaTipoTS(id.pos) if (id.tipo == null) then error else if (id.tipo == S1.tipo) then S.tipo := S1.tipo else S.tipo := error}
S -> return X ;	{S.tipoRet := X.tipo}
S -> print (E) ;	{if (E.tipo == cadena || E.tipo == entero) then S.tipo := tipo_ok else S.tipo := error("No se pueden imprimir booleanos") }
S -> input (id) ;	{(id.tipo = buscaTipoTS(id.pos) if (id.tipo == cadena || id.tipo == entero) then S.tipo := tipo_ok else S.tipo:= error("No se puede hacer input de booleanos") }
S1 -> = E ;	{S1.tipo := E.tipo}
S1 -> (L) ;	{S1.tipo := vacio S1.tipoParam := L.tipoParam}
S1 -> |= E ;	{S1.tipo := E.tipo}
X -> E	{X.tipo := E.tipo}
X -> lambda	{X.tipo := vacio}
C -> B C_1	{if(B.tipo == tipo_ok && C_1.tipo == tipo_ok) then C.tipo := tipo_ok else C.tipo := B.tipo }

C -> lambda	{C.tipo := vacio}
L -> E Q	{L.tipo := E.tipo x Q.tipo}
L -> lambda	{}
Q -> , E Q_1	{Q.tipo := E.tipo x Q_1.tipo}
Q -> lambda	{}
F -> function id H	{TSL := crearTS() TSActual := TSL ZonaDeclaracion = true InsertaEtTS(id.pos, nuevaEt()) ZonaDeclaracion = false DespL := 0}
(A) { C }	if(C.tipoRet != H.tipo) then error("Retorno incorrecto") DestruyeTS(TSL) TSActual := TSG}
H -> T	{H.tipo := T.tipo}
H -> lambda	{H.tipo := vacio}
A -> T id K	{ZonaDeclaracion = true InsertarTipoTS(id.pos, T.tipo); InsertarDespTS(id.pos, desp); Desp = Desp + T.ancho; ZonaDeclaracion = false}
A -> lambda	{}
K -> , T id K_1	{ZonaDeclaracion = true InsertarTipoTS(id.pos, T.tipo); InsertarDespTS(id.pos, desp); Desp = Desp + T.ancho); ZonaDeclaracion = false}
K -> lambda	{}
E -> R E1	{if(E1.tipo == tipo_ok) then E.tipo := R.tipo else E.tipo := boolean if(E1.tipo != tipo_ok && R.tipo !=entero) then error("Op no válido")
E1 -> == R $E1_1$	{if(E_1.tipo == tipo_ok) then E1.tipo := R.tipo else if (R.tipo != entero || $E1_1$.tipo != entero) then error ("Los tipos han de ser enteros")}
E1 -> lambda	{E1.tipo := vacio}

R -> U R1 {if(R1.tipo == tipo_ok) then R.tipo := U.tipo
else if (R1.tipo != entero || U.tipo != entero) then error (“Los tipos han de ser enteros”)}

R1 -> + U $R1_1$ {if (R1.tipo != entero || U.tipo != entero) then error (“Los tipos han de ser enteros”)
R1.tipo := U.tipo}

R1 -> lambda {R1.tipo := vacio}

U -> || U_1 {if(U_1.tipo != boolean) then error(“Operador no válido”)
U.tipo := U_1.tipo}

U -> V {U.tipo := V.tipo}

V -> id V1 {V.tipo := buscaTipoTS(id.pos)}

V -> (E) {V.tipo := E.tipo}

V -> entero {V.tipo := entero
V.ancho := 1}

V -> cadena {V.tipo := cadena
V.ancho := 64}

V1 -> (L) {V1.tipo := L.tipo}

V1 -> lambda {}

Anexo: Casos de prueba

Caso 1: Prueba sin errores

```
let int   a   ;
let int    b  ;
let int number;
print ( "Introduce el primer operando" );
input (a);
print ("Introduce el segundo operando");input(b);
```

```
function  operacion int(int num_1, int num_2)
{
        return  num_1 + num_2+77;
}

number = 0;
print(operacion(b,a));
```

Tokens

```
<PalReslet,>
<PalResint,>
<ID,0>
<puntocoma,>
<PalReslet,>
<PalResint,>
<ID,1>
<puntocoma,>
<PalReslet,>
<PalResint,>
<ID,2>
<puntocoma,>
<PalResprint,>
<ParIzq,>
<cad,"Introduce el primer operando">
<ParDer,>
<puntocoma,>
<PalResinput,>
<ParIzq,>
<ID,0>
<ParDer,>
<puntocoma,>
<PalResprint,>
<ParIzq,>
<cad,"Introduce el segundo operando">
<ParDer,>
<puntocoma,>
<PalResinput,>
<ParIzq,>
<ID,1>
<ParDer,>
<puntocoma,>
<PalResfunction,>
<ID,3>
<PalResint,>
<ParIzq,>
<PalResint,>
<ID,4>
<coma,>
```

<PalResint,>
<ID,5>
<ParDer,>
<CorIzq,>
<PalResreturn,>
<ID,4>
<OpArSuma,>
<ID,5>
<OpArSuma,>
<cte,77>
<puntocoma,>
<CorDer,>
<ID,2>
<asig,>
<cte,0>
<puntocoma,>
<PalResprint,>
<ParIzq,>
<ID,3>
<ParIzq,>
<ID,1>
<coma,>
<ID,0>
<ParDer,>
<ParDer,>
<puntocoma,>
<EOF,>

Árbol de análisis sintáctico

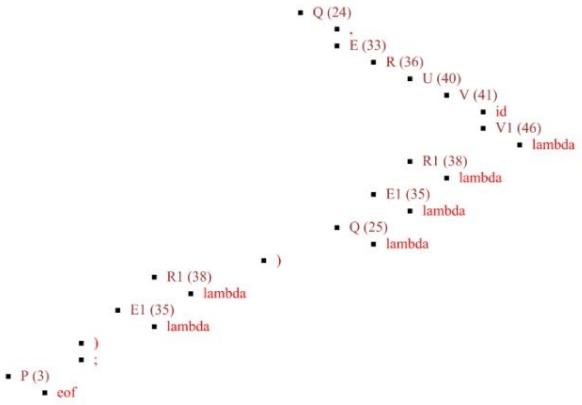


Tabla de símbolos

Tabla de simbolos #0:
* lexema: 'a'
 + tipo : 'entero'
 + despl : '0'

* lexema: 'b'
 + tipo : 'entero'
 + despl : '1'

* lexema: 'number'
 + tipo : 'entero'
 + despl : '2'

* lexema: 'operacion'
 + tipo : 'funcion'
 + numParam : '2'
 + tipoParam0 : 'entero'
 + tipoParam1 : 'entero'
 + tipoRetorno : 'entero'
 + etiqFuncion : 'etiq_operacion'

Tabla de simbolos #1:
* lexema: 'num_1'
 + tipo : 'entero'
 + despl : '0'

* lexema: 'num_2'
 + tipo : 'entero'
 + despl : '1'

Caso 2: prueba sin errores

```
let int a;
let   int  b;
let boolean bbb;
a = 3;
b=a;
if (a == b) b = 1;
a = a + b;
print (a) ;
print(b);
```

Caso 3: prueba sin errores

```
let int x;
let int y;
x=1;
y=1;
let int z;
z=x+y;

if(z==2) print("suma correcta");
function op int (int n){
return 10;
}
print(op(x));
let boolean aux;
while(aux){print("hola");}
//coment
print("fin");
```

Caso 4: prueba sin errores

```
let int x;
let boolean b;
let int z;
input (x);
print (x);
input (z);
print (x+z);
b=x==z;if (b)
x =
  x + 6
    + z
    + 1
    + (2
    + y
```

```
+ 7);
```

Caso 5: prueba sin errores

```
let string texto;
function pideTexto ()
{
        print ("Introduce una palabra");
        input (texto);
}
function alert (string msg)
{
        print (msg);
}
pideTexto();
let  string  textoAux;
textoAux = texto;
alert(textoAux);
```

Caso 6: prueba con errores

```
let int number1;
 let string cadena;
 let boolean logico1;let boolean logico2;
number1 = 0;

while (number1){ cadena = "hello";}
```

Errores

Error semantico [linea 5]: La condicion del while ha de ser booleana

Caso 7: prueba con errores

```
let int a;
let   int  b;
let boolean bbb;
a = 3;
b=a;
if (a == b) b = 1;
a += a +b;
print (a) ;
print(b);
```

Errores

Error sintactico [linea 7]: OpArSuma no esperado

Caso 8: prueba con errores

```
//Comentario válido
let int a;
a =1;
let int b;
b = 2;
/hacemos la suma
let int c;
c = a + b;
return c;
```

Errores

Error lexico [linea 6]: Comentario no válido

Caso 9: prueba con errores

```
let boolean x;
function oper int(int a, int b) {
        let int x;
        x = a + b;
        return x;
}
```

Errores

Error semantico [linea 3]: Variable 'x' repetida

Caso 10: prueba con errores

```
let int x;
function imprimir(string texto){
        print(texto);
}
imprimir(x);
```

Errores

Error semantico [linea 5]: Parametros de llamada a la funcion incorrectos

Se esperaba: [cadena] y se encontró: [entero]