MEMORIA ENTREGA PARCIAL I

Grupo 20

Integrantes

Ignacio Simó García
Alejandro Torres Hernández
Álvaro Moneo Martínez de Azagra

Índice

Descripción del Diseño Final del Procesador

El diseño del procesador desarrollado sigue una arquitectura modular que implementa un enfoque descendente predictivo LL (1) basado en gramáticas factorizadas y sin recursividad por la izquierda. Los elementos claves del diseño incluyen:

1. Análisis Léxico

- Se definieron las **palabras reservadas** como boolean, int, function, get, var, input, output, if, return, string, void, entre otras.
- Las **acciones semánticas** asociadas al léxico gestionan la generación de tokens específicos y el almacenamiento de identificadores en la tabla de símbolos, verificando errores como cadenas demasiado largas o caracteres no válidos en identificadores estructura del léxico es robusta y evita bucles infinitos mediante el uso de autómatas finitos deterministas con transiciones específicas y detección de fin de archivo.

2. Análisis Sintáctico

- Se adoptó un análisis **predictivo descendente LL (1)**, apoyado en la creación de una **tabla sintáctica** que guía las decisiones de expansión de reglas en función del terminal de entrada y del símbolo no terminal actual.
- Se quitó la recursividad por la izquierda y se factorizaron las reglas para garantizar que las producciones cumplan con la condición LL (1) (conjuntos FIRST disjuntos).

3. Análisis Semántico

- El análisis semántico implementado en el procesador tiene como objetivo validar que los programas cumplan con las reglas semánticas del lenguaje y asegurar la coherencia de los tipos de datos y estructuras.

- Teniendo en cuenta cada caso, y verificando los tipos si son correctos como en los casos de funciones, if, for, inicializando variables etc. Y plasmándolo en la tabla de símbolos

4. Tabla de Símbolos

- La **tabla de símbolos** se organiza en niveles jerárquicos donde TABLA #0 almacena las variables globales y las funciones declaradas, mientras que las tablas auxiliares contienen los parámetros y variables locales de cada función.
- Cada entrada de laos como el tipo de dato, el número de parámetros (en caso de funciones), los tipos de los parámetros, el tipo de retorno y el desplazamiento en memoria.

5. Gestión de Errores

- Se crearon las verificaciones de errores semánticos y sintácticos, como "Número fuera de rango", "Token no reconocido" y "Caracteres no válidos en identificadores".
- Se implementó la gestión de errores léxicos como cadenas con más de 64 caracteres o identificadores con caracteres especiales, entre otros.

6. Casos de Prueba

- Se validó el procesador con que incluyen operaciones válidas e inválidas. Para los casos correctos, se presentan árboles sintácticos y volcados de la tabla de símbolos generados con herramientas como VASt.

1. Analizador Léxico

1.1. Tokens

Palabras reservadas:

boolean < BOOLEAN , >
int <INT, >,
function < FUNC , >
get < GET , >
constante entera < ENT , numero >
var < VAR , >
input < INPUT , >
output < OUTPUT , >
if < IF , >
return < RETURN , >
string < STRING , >
void < VOID , >
for < FOR , >
Cadena < CAD , ` lexema ´ >
Identificador < ID , posTS >,

Aritméticos: /= = , ; ( ) { }

< ASIGDIV , > ,< IG , >, < COMA , >, < PYC , >, < PARIZQ , >, < PARDER , >,
< LLAVIZQ, >, < LLAVDER, >

Para los operadores opcionales que son: / ! !=
< DIV , >, < NEG , >, < DIST , >, < EOF , >

1.2. Gramática

S--> lA | dB | /C | 'E | delS | !D | (|) | { | } | ; | , | EOF

A--> lA | dA | _A | λ

B--> dB | λ

C--> = | /F | λ

D--> = | λ

E--> c1E | '

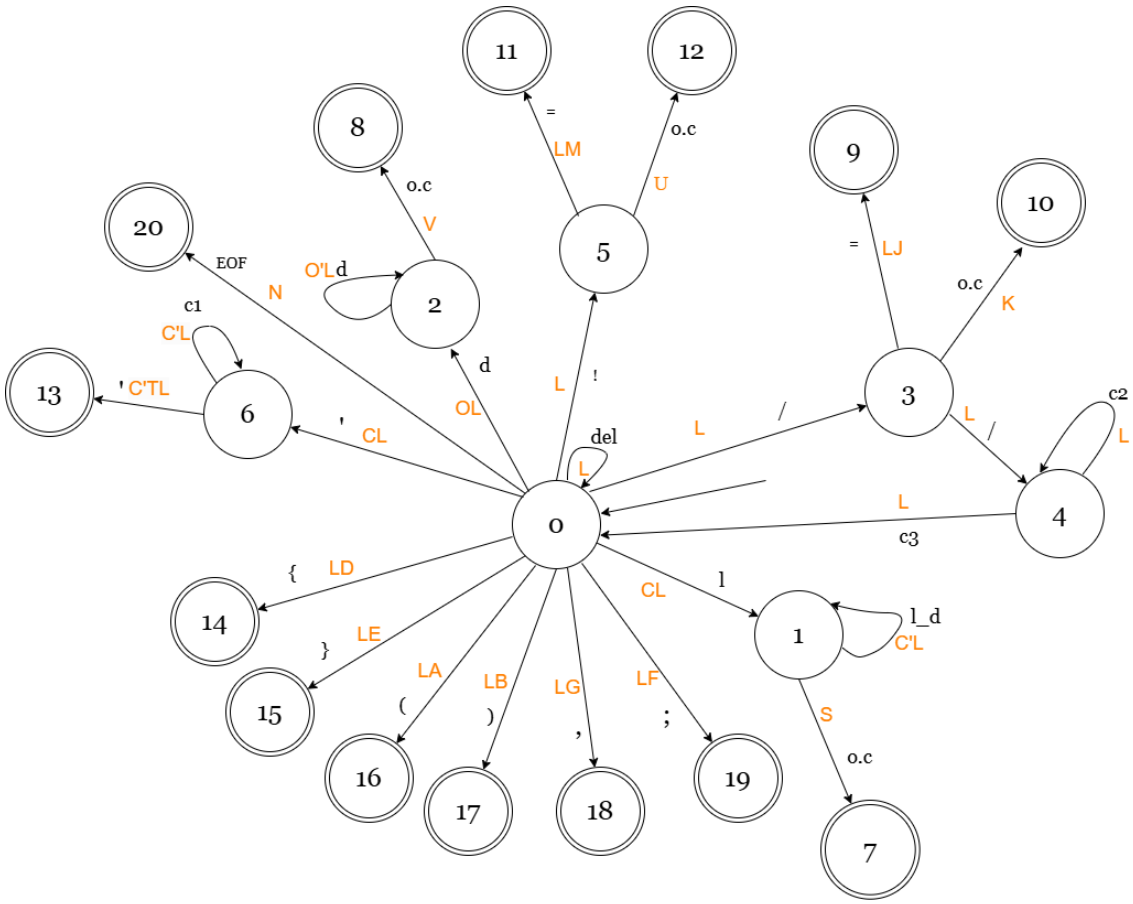
F--> c2F | c3S

c1: cualquier caracter menos '

c2: cualquier carácter de nuestro alfebeto

c3: donde c3 es un salto de línea

1.3. Autómata Finito Determinista



1.4. Acciones semánticas

LEER

L: Leer

GENERAR TOKEN

A: Gen_Token <PARIZQ, ->

B: Gen_Token<PARDER, ->

D: Gen_Token <LLAVIZQ, ->

E: Gen_Token<LLAVDER, ->

F: Gen_Token<PYC, ->

G: Gen_Token<COMA, ->

I: Gen_Token<IG, ->

J: Gen_Token<ASIGDIV, ->

K: Gen_Token<DIV, ->

U: Gen_Token<NEG, ->

M: Gen_Token<DIST, ->

N: Gen_Token<EOF, ->

T: Gen_Token<CAD, lex>

V: if(num > 32767) the Error (“Número demasiado grande”)

Gen_Token<ENT, num>

CONCATENAR:

C: lex := ∅

C’: lex := lex ⊕ car

CALCULAR NÚMERO

O: num := valor(d)

O’: num:= num*10 + valor(d)

INSERTARTS

S: p:= BuscarTS(lex)

if(p ∈ PR) then Gen_Token(PR, p) else

if(cont>64) then Error (“Cadena demasiado larga”) else

if(p=Null) then p:= InsertaTS(palabra)

Gen_Token<ID, p>

1.5. Errores

Error 1: Salto de línea en medio de una cadena.

Error 2: Número fuera de rango. (3000 max)

Error 3: Cadena fuera de rango. (64 car max)

Error 4: Caracteres no válidos en identificadores.

Error 5: “Tokens no reconocido” ej: ==, +

2. Analizador Sintáctico

2.1. Gramática final

Gramática factorizada y eliminada la recursividad por la izquierda:

Terminales = { ( ) { } ; , /= = ! / != id for boolean string input output return function void if EOF var ent cad int }

NoTerminales = { P B D S S1 E E1 C C1 T A L K Q X H R R1 U U1 M F AUX AUX1 }

Axioma = P

Producciones = {

P -> B P | F P | EOF

B -> for (D ; E ; AUX) { C } | var T id ; | if (E) S | S

D -> var T id | id = E | lambda

T -> int | boolean | string

S -> id S1 | input id ; | output E ; | return X ;

S1 -> = E ; | /= E ; | (L) ;

L -> E Q | lambda

Q -> , E Q | lambda

X -> E | lambda

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

H -> T | void

A -> T id K | void

K -> , T id K | lambda

C -> B C1

C1 -> C | lambda

E -> R E1

E1 -> != R | lambda

R -> M R1

R1 -> / R | lambda

M -> ! M | U

U -> id U1 | (E) | ent | cad

U1 -> (L) | lambda

AUX -> id AUX1 | lambda

AUX1 -> = E | /= E | (L)

}

2.2. Condición LL1

Para que una gramática cumpla con la condición LL(1), es necesario que el conjunto de predicción de cada producción en la gramática sea disjunto (Ø)

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 -> void

FIRST de A -> void = { void }

FIRST de A = { boolean int string void }

Analizando símbolo AUX

Analizando producción AUX -> id AUX1

FIRST de AUX -> id AUX1 = { id }

Analizando producción AUX -> lambda

FIRST de AUX -> lambda = { lambda }

FIRST de AUX = { id lambda }

Calculando FOLLOW de AUX

FOLLOW de AUX = {) }

Analizando símbolo AUX1

Analizando producción AUX1 -> = E

FIRST de AUX1 -> = E = { = }

Analizando producción AUX1 -> //= E

FIRST de AUX1 -> //= E = { //= }

Analizando producción AUX1 -> (L)

FIRST de AUX1 -> (L) = { (}

FIRST de AUX1 = { (//= = }

Analizando símbolo B

Analizando producción B -> for (D ; E ; AUX) { C }

FIRST de B -> for (D ; E ; AUX) { C } = { for }

Analizando producción B -> var T id ;

FIRST de B -> var T id ; = { var }

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 -> input id ;

FIRST de S -> input id ; = { input }

Analizando producción S -> output E ;

FIRST de S -> output E ; = { output }

Analizando producción S -> return X ;

FIRST de S -> return X ; = { return }

FIRST de S = { id input output return }

FIRST de B -> S = { id input output return }

FIRST de B = { for id if input output return var }

Analizando símbolo C

Analizando producción C -> B C1

FIRST de C -> B C1 = { for id if input output return var }

FIRST de C = { for id if input output return var }

Analizando símbolo C1

Analizando producción C1 -> C

FIRST de C1 -> C = { for id if input output return var }

Analizando producción C1 -> lambda

FIRST de C1 -> lambda = { lambda }

FIRST de C1 = { for id if input output return var lambda }

Calculando FOLLOW de C1

Calculando FOLLOW de C

FOLLOW de C = { } }

FOLLOW de C1 = { } }

Analizando símbolo D

Analizando producción D -> var T id

FIRST de D -> var T id = { var }

Analizando producción D -> id = E

FIRST de D -> id = E = { id }

Analizando producción D -> lambda

FIRST de D -> lambda = { lambda }

FIRST de D = { id var lambda }

Calculando FOLLOW de D

FOLLOW de D = { ; }

Analizando símbolo E

Analizando producción E -> R E1

Analizando símbolo R

Analizando producción R -> M R1

Analizando símbolo M

Analizando producción M -> ! M

FIRST de M -> ! M = { ! }

Analizando producción M -> U

Analizando símbolo U

Analizando producción U -> id U1

FIRST de U -> id U1 = { id }

Analizando producción U -> (E)

FIRST de U -> (E) = { (}

Analizando producción U -> ent

FIRST de U -> ent = { ent }

Analizando producción U -> cad

FIRST de U -> cad = { cad }

FIRST de U = { (cad ent id }

FIRST de M -> U = { (cad ent id }

FIRST de M = { ! (cad ent id }

FIRST de R -> M R1 = { ! (cad ent id }

FIRST de R = { ! (cad ent id }

FIRST de E -> R E1 = { ! (cad ent id }

FIRST de E = { ! (cad ent id }

Analizando símbolo E1

Analizando producción E1 -> != R

FIRST de E1 -> != R = { != }

Analizando producción E1 -> lambda

FIRST de E1 -> lambda = { lambda }

FIRST de E1 = { != lambda }

Calculando FOLLOW de E1

Calculando FOLLOW de E

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 = {) }

Calculando FOLLOW de X

FOLLOW de X = { ; }

Calculando FOLLOW de AUX1

FOLLOW de AUX1 = {) }

FOLLOW de E = {) , ; }

FOLLOW de E1 = {) , ; }

Analizando símbolo F

Analizando producción F -> function H id (A) { C }

FIRST de F -> function H id (A) { C } = { function }

FIRST de F = { function }

Analizando símbolo H

Analizando producción H -> T

FIRST de H -> T = { boolean int string }

Analizando producción H -> void

FIRST de H -> void = { void }

FIRST de H = { boolean int string void }

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

Calculando FOLLOW de A

FOLLOW de A = {) }

FOLLOW de K = {) }

Analizando símbolo L

Analizando producción L -> E Q

FIRST de L -> E Q = { ! (cad ent id }

Analizando producción L -> lambda

FIRST de L -> lambda = { lambda }

FIRST de L = { ! (cad ent id lambda }

Analizando símbolo P

Analizando producción P -> B P

FIRST de P -> B P = { for id if input output return var }

Analizando producción P -> F P

FIRST de P -> F P = { function }

FIRST de S1 = { (//= = }

Analizando símbolo U1

Analizando producción U1 -> (L)

FIRST de U1 -> (L) = { (}

Analizando producción U1 -> lambda

FIRST de U1 -> lambda = { lambda }

FIRST de U1 = { (lambda }

Calculando FOLLOW de U1

Calculando FOLLOW de U

Calculando FOLLOW de M

FOLLOW de M = { !=) , // ; }

FOLLOW de U = { !=) , // ; }

FOLLOW de U1 = { !=) , // ; }

Analizando símbolo X

Analizando producción X -> E

FIRST de X -> E = { ! (cad ent id }

Analizando producción X -> lambda

FIRST de X -> lambda = { lambda }

Analizando producción P -> EOF

FIRST de P -> EOF = { EOF }

FIRST de P = { EOF for function id if input output return var }

Analizando símbolo R1

Analizando producción R1 -> // M R1

FIRST de R1 -> // M 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 -> //= E ;

FIRST de S1 -> //= E ; = { //= }

Analizando producción S1 -> (L) ;

FIRST de S1 -> (L) ; = { (}

FIRST de X = { ! (cad ent id lambda }

Análisis concluido satisfactoriamente

2.3. T. Sintáctica

La tabla sintáctica es una representación que se utiliza en los analizadores sintácticos predictivos para decidir qué regla aplicar en función del símbolo no terminal actual (fila) y el símbolo terminal que se encuentra en la entrada (columna).

	!	!=	(	)	,	//	//=	;	=	EOF	boolean	cad	ent	for	function	id	if	input	int	output	return	string	var	void	{	}	$ (final de cadena)
A	--	--	--	--	--	--	--	--	--	--	A → T id K	--	--	--	--	--	--	--	A → T id K	--	--	A → T id K	--	A → void	--	--	--
AUX	--	--	--	AUX → lambda	--	--	--	--	--	--	--	--	--	--	--	AUX → id AUX1	--	--	--	--	--	--	--	--	--	--	--
AUX1	--	--	AUX1 → (L)	--	--	--	AUX1 → //= E	--	AUX1 → = E	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
B	--	--	--	--	--	--	--	--	--	--	--	--	--	B → for (D ; E ; AUX) { C }	--	B → S	B → if (E) S	B → S	--	B → S	B → S	--	B → var T id ;	--	--	--	--
C	--	--	--	--	--	--	--	--	--	--	--	--	--	C → B C1	--	C → B C1	C → B C1	C → B C1	--	C → B C1	C → B C1	--	C → B C1	--	--	--	--
C1	--	--	--	--	--	--	--	--	--	--	--	--	--	C1 → C	--	C1 → C	C1 → C	C1 → C	--	C1 → C	C1 → C	--	C1 → C	--	--	C1 → lambda	--
D	--	--	--	--	--	--	--	D → lambda	--	--	--	--	--	--	--	D → id = E	--	--	--	--	--	--	D → var T id	--	--	--	--
E	E → R E1	--	E → R E1	--	--	--	--	--	--	--	--	E → R E1	E → R E1	--	--	E → R E1	--	--	--	--	--	--	--	--	--	--	--
E1	--	E1 → != R	--	E1 → lambda	E1 → lambda	--	--	E1 → lambda	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
F	--	--	--	--	--	--	--	--	--	--	--	--	--	--	F → function H id (A) { C }	--	--	--	--	--	--	--	--	--	--	--
H	--	--	--	--	--	--	--	--	--	--	H → T	--	--	--	--	--	--	--	H → T	--	--	H → T	--	H → void	--	--	--
K	--	--	--	K → lambda	K → , T id K	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
L	L → E Q	--	L → E Q	L → lambda	--	--	--	--	--	--	--	L → E Q	L → E Q	--	--	L → E Q	--	--	--	--	--	--	--	--	--	--	--
M	M → ! M	--	M → U	--	--	--	--	--	--	--	--	M → U	M → U	--	--	M → U	--	--	--	--	--	--	--	--	--	--	--
P	--	--	--	--	--	--	--	--	--	P → EOF	--	--	--	P → B P	P → F 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 → M R1	--	R → M R1	--	--	--	--	--	--	--	--	R → M R1	R → M R1	--	--	R → M R1	--	--	--	--	--	--	--	--	--	--	--
R1	--	R1 → lambda	--	R1 → lambda	R1 → lambda	R1 → // M R1	--	R1 → lambda	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
S	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	S → id S1	--	S → input id ;	--	S → output E ;	S → return X ;	--	--	--	--	--	--
S1	--	--	S1 → (L) ;	--	--	--	S1 → //= E ;	--	S1 → = E ;	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
T	--	--	--	--	--	--	--	--	--	--	T → boolean	--	--	--	--	--	--	--	T → int	--	--	T → string	--	--	--	--	--
U	--	--	U → (E)	--	--	--	--	--	--	--	--	U → cad	U → ent	--	--	U → id U1	--	--	--	--	--	--	--	--	--	--	--
U1	--	U1 → lambda	U1 → (L)	U1 → lambda	U1 → lambda	U1 → lambda	--	U1 → lambda	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
X	X → E	--	X → E	--	--	--	--	X → lambda	--	--	--	X → E	X → E	--	--	X → E	--	--	--	--	--	--	--	--	--	--	--

3. Analizador Semántico

P -> B P

P -> F P

P -> EOF {}

B -> for (D ; E ; AUX) { C } {B.tipo := if E.tipo = logico && !C.error then tipo_ok else tipo_error}

B -> var T id ; {AñadeTipoTS (id.pos, T.tipo)}

B -> if (E) S {B.tipo := if E.tipo = lógico then S.tipo else error}

B -> S {B.tipo := S.tipo}

D -> var T id {AñadeTipoTS(id.pos, T.tipo)}

D -> id = E {D.tipo := if BuscaTipoTS(id.pos) = E.tipo then tipo_ok else tipo_error}

D -> lambda {}

T -> int {T.tipo := entero}

T -> Boolean {T.tipo := logico}

T -> string {T.tipo := cadena}

S -> id { S1.tipo := BuscaTipoTS(id.pos)} S1

S -> input id ;

S -> output E ;

S -> return X { S.tipo := X.tipo} ;

S1 -> = E ; { if S1.tipo=E.tipo then tipo_ok else error}

S1 -> /= E ; {if S1.tipo= entero && S1.tipo=E.tipo then tipo_ok else error}

S1 -> ({if S1.tipo= función then tipo_ok else error} L) ;

L -> E Q

L -> lambda {}

Q -> , E Q

Q -> lambda {}

X -> E { X.tipo := E.tipo}

X -> lambda {}

F -> function H {AñadeTipoRetTS (id.pos, H.tipo)} id {AñadeTipoTS (id.pos, funcion)}
(A) { C {if C.tipo = H.tipo then tipo_ok else error} }

H -> T {H.tipo := T.tipo}

H -> void {H.tipo := void}

A -> T id {AñadeTipoTS (id.pos, T.tipo)} K

A -> void {A.tipo := void}

K -> , T id {AñadeTipoTS (id.pos, T.tipo)} K

K -> lambda {}

C -> B {C.tipo := B.tipo} C1

C1 -> C {C1.tipo := C.tipo}

C1 -> lambda {}

E -> R {E1.tipo:=R.tipo} E1 {E.tipo := R.tipo}

E1 -> != R {if E1.tipo=R.tipo then tipo_ok else error}

E1 -> lambda {}

R -> M {R.tipo:=M.tipo} {R1.tipo:=M.tipo} R1

R1 -> / R {if R1.tipo= entero && R1.tipo=R.tipo then tipo_ok else error}

R1 -> lambda {}

M -> ! M {if M.tipo= logico then tipo_ok else error}

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

U -> id U1 {U.tipo := BuscaTipoTS(id.pos) }

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

U -> ent { U.tipo := entero}

U -> cad { U.tipo := cadena }

U1 -> (L)

U1 -> lambda {}

AUX -> id AUX1 {AUX.tipo := BuscaTipoTS(id.pos) }

AUX -> lambda {}

AUX1 -> = E { if AUX1.tipo=E.tipo then tipo_ok else error}

AUX1 -> /= E {if AUX1.tipo= entero && S1.tipo=E.tipo then tipo_ok else error}

AUX1 -> (L)

4. Tabla de Símbolos

La tabla de símbolos permitirá almacenar y consultar toda la información relacionada con los identificadores necesarios durante la ejecución del programa.

La tabla contendrá una entrada por cada identificador, y cada entrada tendrá los siguientes atributos:

- **Posición en la tabla de símbolos:** número que referencia la posición en la tabla de símbolos, comenzando desde 1 y aumentando en uno por cada nueva entrada.
- **Tipo:** tipo asociado al identificador de la entrada, que podrá ser entero, lógico, cadena, función o vacío.
    - En caso de ser una **función**, se añadirán los siguientes atributos adicionales:
        - **Número de parámetros:** contiene la cantidad de parámetros que se utilizan en la llamada a la función. Además, se registra el tipo de cada uno de estos parámetros.
        - **Tipo de retorno:** especifica el tipo de dato devuelto por la función.
    - En caso de ser de tipo entero, lógico o cadena, los atributos asociados serán los siguientes:
        - **Desplazamiento:** indica la posición de la variable en memoria. El desplazamiento comienza en 0 y se incrementa en función del tipo: se suma 1 para los lógicos, 4 para los enteros y 64 para las cadenas.

Además, la estructura contará con una **tabla de símbolos principal**, que almacenará las funciones y las variables globales (que no son parámetros). Para cada función, se creará una tabla específica que contendrá la información correspondiente a sus parámetros.

La implementación de la tabla de símbolos se ha realizado mediante una estructura HashMap<String, Entrada>, donde la clave es el identificador y el valor es un objeto de tipo Entrada, el cual almacena todos los atributos descritos anteriormente.

Un ejemplo de la organización de una tabla de símbolos es el siguiente:

```
TABLA #0 :
 * Lexema : 'c'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'i'
   Atributos :
   + tipo : 'logico'
   + despl : 4
---------------------------------
 * Lexema : 'suma'
   Atributos :
   + tipo : 'funcion'
     + numParam : '2'
       + TipoParam1 : 'entero'
       + TipoParam2 : 'entero'
       + TipoRetorno : 'entero'
     + EtiqFuncion : 'suma'
---------------------------------
 * Lexema : 'resta'
   Atributos :
   + tipo : 'funcion'
     + numParam : '1'
       + TipoParam1 : 'entero'
       + TipoRetorno : 'logico'
     + EtiqFuncion : 'resta'
---------------------------------
```

```
TABLA #1 (Variables locales y parametros de la funcion 'suma'):
 * Lexema : 'a'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'b'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
 * Lexema : 'h'
   Atributos :
   + tipo : 'entero'
   + despl : 8
---------------------------------


TABLA #2 (Variables locales y parametros de la funcion 'resta'):
 * Lexema : 'g'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------

TABLA #3 (Variables locales y parametros de la funcion 'mult'):
 * Lexema : 'o'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
```

5. Gestor de errores

En la implementación del código se gestionan los posibles errores que puedan surgir, ya sean de tipo léxico, sintáctico o semántico. Cada uno de estos errores está numerado, lo que facilita identificar rápidamente el tipo de fallo ocurrido. Además, se indica la línea específica en la que se produce el error y el tipo de analizador implicado.

Errores léxicos

Se describe el tipo de error léxico encontrado junto con la línea en la que se produjo. Por ejemplo, se puede informar si una cadena excede la longitud máxima permitida o si aparece un carácter que no se reconoce como un token válido.

```
Error Léxico 5 en la linea 3: el caracter + no se reconoce como un token
Error Léxico 3 en la linea 4: La cadena aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa supera el tamaño máximo
Error Léxico 2 en la linea 5: Número fuera de rango en la línea 5
```

Errores sintácticos

Este tipo de error indica que el token recibido no es el esperado. Se muestra qué tokens eran válidos en ese contexto y cuál se ha recibido en su lugar.

```
Error Sintáctico en línea 2: se esperaba PYC, se encontró VAR
```

```
Error Sintáctico en línea 2: se esperaba INT|BOOLEAN|STRING, se encontró ID
```

```
Error Sintáctico en línea 5: se esperaba LLAVDER, se encontró EOF
```

Errores semánticos

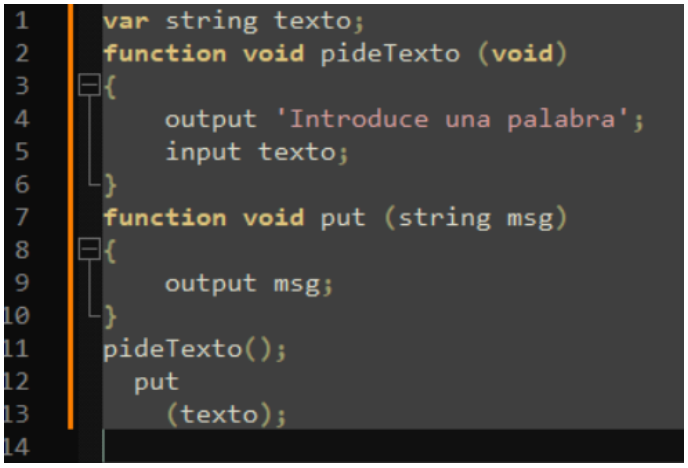
Se informa del tipo de error semántico producido, por ejemplo, cuando hay un tipo de dato incorrecto en una instrucción, junto con la línea donde ocurre.

```
Error Semántico en línea 3: Tipos incompatibles en asignación
Error Semántico en línea 4: División solo permitida entre enteros
Error Semántico en línea 7: El tipo de retorno no coincide con la función
```

6. Anexo (casos de prueba)

En este anexo se incluirán 10 casos de prueba de los cuales 5 serán correctos y 5 darán error en el procesador. Para los casos correctos, se incluirá el listado de tokens, el árbol sintáctico utilizando la herramienta VASt y el volcado de la tabla de símbolos.

Caso de prueba correcta 1:



```
var string texto;
function void pideTexto (void)
{
    output 'Introduce una palabra';
    input texto;
}
function void put (string msg)
{
    output msg;
}
pideTexto();
  put
    (texto);

```

- Listado de tokens:

```
< VAR ,  >
< STRING ,  >
< ID , 1 >
< PYC ,  >
< FUNC ,  >
< VOID ,  >
< ID , 2 >
< PARIZQ ,  >
< VOID ,  >
< PARDER ,  >
< LLAVIZQ ,  >
< OUTPUT ,  >
< CAD , "Introduce una palabra" >
< PYC ,  >
< INPUT ,  >
< ID , 1 >
< PYC ,  >
< LLAVDER ,  >
< FUNC ,  >
< VOID ,  >
< ID , 3 >
< PARIZQ ,  >
< STRING ,  >
< ID , 4 >
< PARDER ,  >
< LLAVIZQ ,  >
< OUTPUT ,  >
```

```
< ID , 4 >
< PYC ,  >
< LLAVDER ,  >
< ID , 2 >
< PARIZQ ,  >
< PARDER ,  >
< PYC ,  >
< ID , 3 >
< PARIZQ ,  >
< ID , 1 >
< PARDER ,  >
< PYC ,  >
< EOF ,  >
```

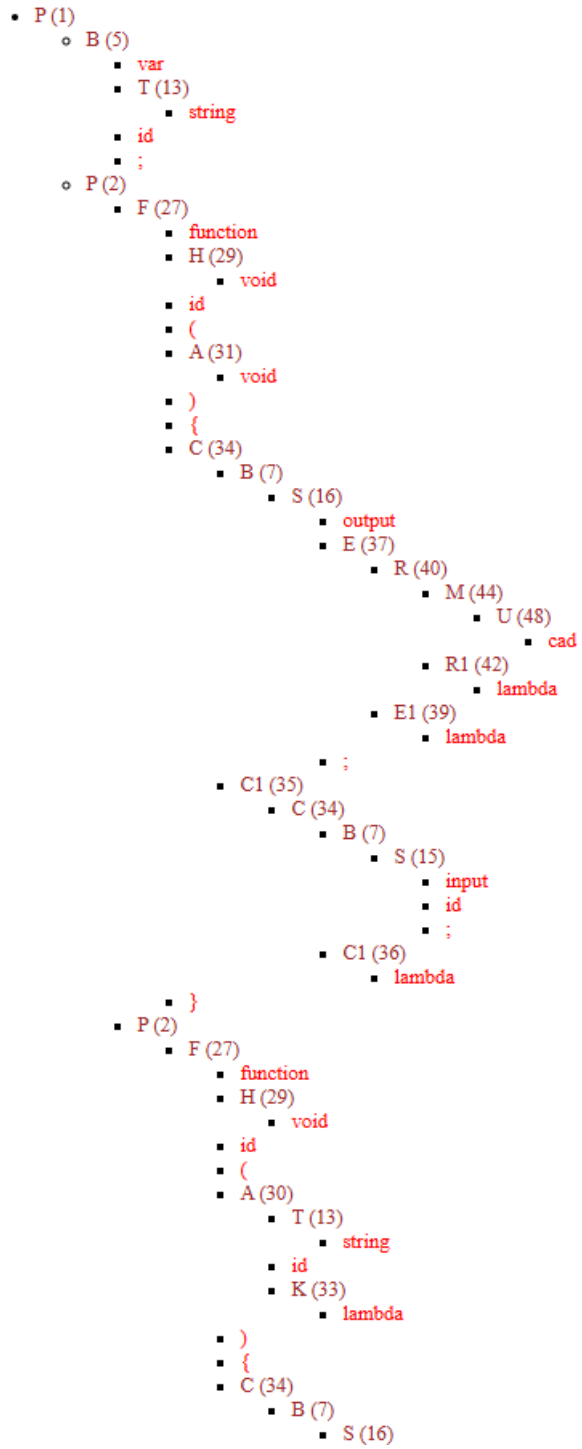
- Tabla de símbolos:

```
TABLA #0 :
 * Lexema : 'texto'
   Atributos :
   + tipo : 'cadena'
   + despl : 0
---------------------------------
 * Lexema : 'pideTexto'
   Atributos :
   + tipo : 'funcion'
     + numParam : '0'
       + TipoRetorno : 'void'
     + EtiqFuncion : 'pideTexto'
---------------------------------
 * Lexema : 'put'
   Atributos :
   + tipo : 'funcion'
     + numParam : '1'
       + TipoParam1 : 'cadena'
       + TipoRetorno : 'void'
     + EtiqFuncion : 'put'
---------------------------------

TABLA #1 :

TABLA #2 :
 * Lexema : 'msg'
   Atributos :
   + tipo : 'cadena'
   + despl : 0
---------------------------------
```

- Árbol sintáctico generado:



Caso de prueba correcta 2:

```
1  var int x;
2  var int y;
3  var int resultado;
4  output 'numero 1';
5  input x;
6  output 'numero 2';
7  input y;
8
9  function int calculo(int numA, int numB)
10 {
11     var int temp;
12     temp = 88 / numA / numB;
13     return temp;
14 }
15
16 resultado = calculo(x, y);
17 output resultado;
```

- Listado de tokens:

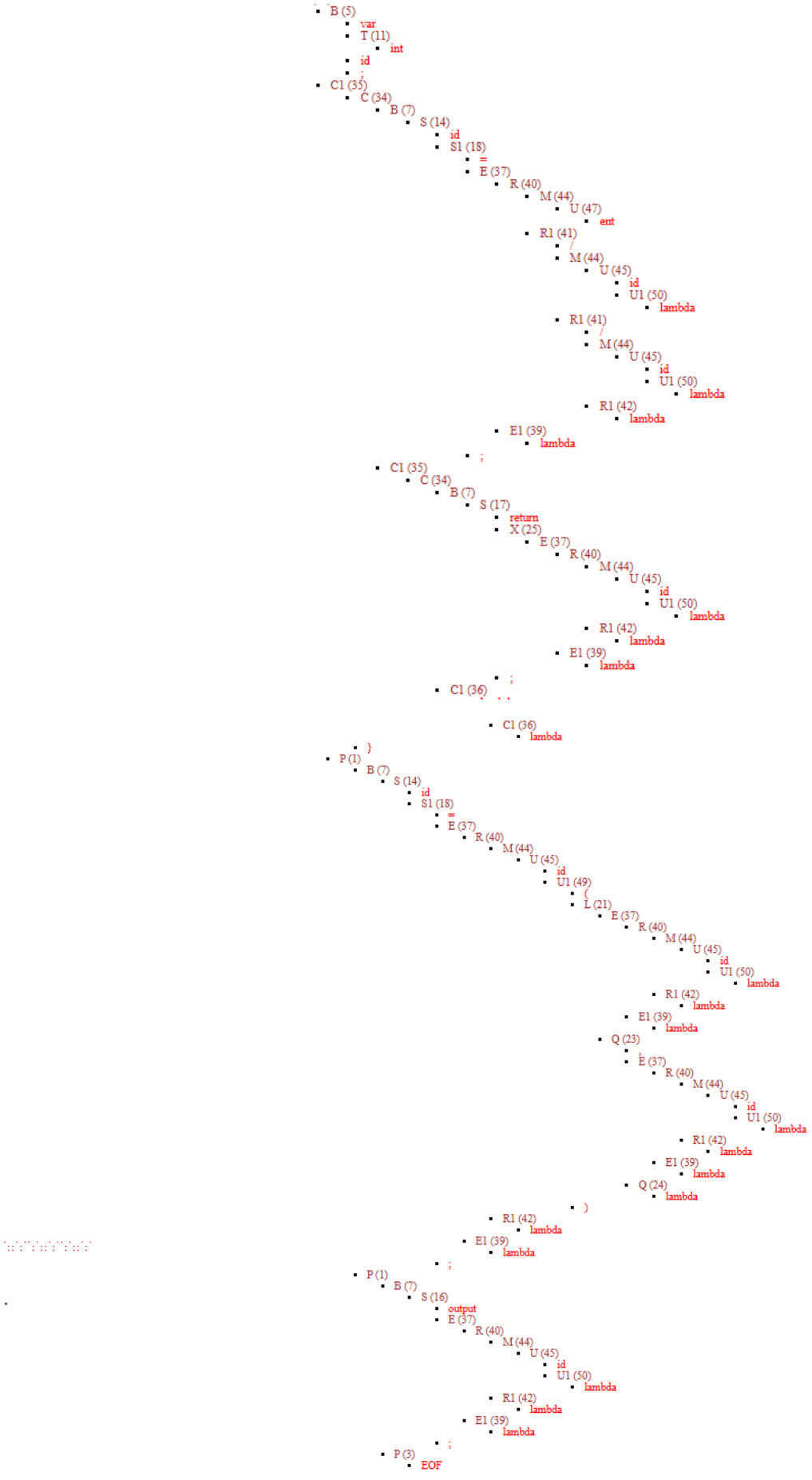
```
< VAR ,  >
< INT ,  >
< ID , 1 >
< PYC ,  >
< VAR ,  >
< INT ,  >
< ID , 2 >
< PYC ,  >
< VAR ,  >
< INT ,  >
< ID , 3 >
< PYC ,  >
< OUTPUT ,  >
< CAD , "numero 1" >
< PYC ,  >
< INPUT ,  >
< ID , 1 >
< PYC ,  >
< OUTPUT ,  >
< CAD , "numero 2" >
< PYC ,  >
< INPUT ,  >
< ID , 2 >
< PYC ,  >
< FUNC ,  >
< INT ,  >
< ID , 4 >
< PARIZQ ,  >
< INT ,  >
< ID , 5 >
< COMA ,  >
< INT ,  >
< ID , 6 >
< PARDER ,  >
< LLAVIZQ ,  >
< VAR ,  >
< INT ,  >
< ID , 7 >
< PYC ,  >
< ID , 7 >
< IG ,  >
< ENT , 88 >
< DIV ,  >
< ID , 5 >
< DIV ,  >
< ID , 6 >
< PYC ,  >
< RETURN ,  >
< ID , 7 >
< PYC ,  >
< LLAVDER ,  >
< ID , 3 >
< IG ,  >
< ID , 4 >
< PARIZQ ,  >
< ID , 1 >
< COMA ,  >
< ID , 2 >
< PARDER ,  >
< PYC ,  >
< OUTPUT ,  >
< ID , 3 >
< PYC ,  >
< EOF ,  >
```

- Tabla de símbolos:

```
TABLA #0 :
 * Lexema : 'x'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'y'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
 * Lexema : 'resultado'
   Atributos :
   + tipo : 'entero'
   + despl : 8
---------------------------------
 * Lexema : 'calculo'
   Atributos :
   + tipo : 'funcion'
     + numParam : '2'
       + TipoParam1 : 'entero'
       + TipoParam2 : 'entero'
       + TipoRetorno : 'entero'
     + EtiqFuncion : 'calculo'
---------------------------------

TABLA #1 :
 * Lexema : 'numA'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'numB'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
 * Lexema : 'temp'
   Atributos :
   + tipo : 'entero'
   + despl : 8
---------------------------------
```

- Árbol sintáctico generado:


B (5)
var
T (11)
int
id
;
C1 (35)
C (34)
B (7)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (47)
ent
R1 (41)
/
M (44)
U (45)
id
U1 (50)
lambda
R1 (41)
/
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
C1 (35)
C (34)
B (7)
S (17)
return
X (25)
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
C1 (36)
C1 (36)
lambda
}
P (1)
B (7)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (45)
id
U1 (49)
(
L (21)
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
Q (23)
,
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
Q (24)
lambda
)
R1 (42)
lambda
E1 (39)
lambda
;
P (1)
B (7)
S (16)
output
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
P (3)
EOF

Caso de prueba correcta 3:

```
1   var int num1;
2   var int num2;
3   var boolean check;
4   num1 = 3;
5   num2 = num1;
6
7   var boolean isDifferent;
8   isDifferent = num1 != num2;
9
10  if (isDifferent) num2 = 62;
11  num1 = num1 / num2;
12  output num1;
13  output num2;
```

- Listado de tokens:

```
< VAR ,  >
< INT ,  >
< ID , 1 >
< PYC ,  >
< VAR ,  >
< INT ,  >
< ID , 2 >
< PYC ,  >
< VAR ,  >
< BOOLEAN ,  >
< ID , 3 >
< PYC ,  >
< ID , 1 >
< IG ,  >
< ENT , 3 >
< PYC ,  >
< ID , 2 >
< IG ,  >
< ID , 1 >
< PYC ,  >
< VAR ,  >
< BOOLEAN ,  >
< ID , 4 >
< PYC ,  >
< ID , 4 >
< IG ,  >
< ID , 1 >
< DIST ,  >
< ID , 2 >
< PYC ,  >
< IF ,  >
< PARIZQ ,  >
< ID , 4 >
< PARDER ,  >
< ID , 2 >
< IG ,  >
< ENT , 62 >
< PYC ,  >
< ID , 1 >
< IG ,  >
< ID , 1 >
< DIV ,  >
< ID , 2 >
< PYC ,  >
< OUTPUT ,  >
< ID , 1 >
< PYC ,  >
< OUTPUT ,  >
< ID , 2 >
< PYC ,  >
< EOF ,  >
```

- Tabla de símbolos:

```
TABLA #0 :
 * Lexema : 'num1'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'num2'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
 * Lexema : 'check'
   Atributos :
   + tipo : 'logico'
   + despl : 8
---------------------------------
 * Lexema : 'isDifferent'
   Atributos :
   + tipo : 'logico'
   + despl : 9
---------------------------------
```

- Árbol sintáctico generado:

- ;
- P (1)
  - B (5)
    - var
    - T (12)
      - boolean
    - id
    - ;
  - P (1)
    - B (7)
      - S (14)
        - id
        - S1 (18)
          - =
          - E (37)
            - R (40)
              - M (44)
                - U (45)
                  - id
                  - U1 (50)
                    - lambda
              - R1 (42)
                - lambda
            - E1 (38)
              - !=
              - R (40)
                - M (44)
                  - U (45)
                    - id
                    - U1 (50)
                      - lambda
                - R1 (42)
                  - lambda
          - ;
    - P (1)
      - B (6)
        - if
        - (
        - E (37)
          - R (40)
            - M (44)
              - U (45)
                - id
                - U1 (50)
                  - lambda
            - R1 (42)
              - lambda
          - E1 (39)
            - lambda
        -)
        - S (14)
          - id
          - S1 (18)
            - =
            - E (37)
              - R (40)
                - M (44)
                  - U (47)
                    - ent
                - R1 (42)
                  - lambda
              - E1 (39)
                - lambda

Caso de prueba correcta 4:

```
function int devolverB(int a, int b)
{
    return b;
}

function int devolverD(int c, int d)
{
    return d;
}

```

- Listado de tokens:

```
< FUNC ,  >
< INT ,  >
< ID , 1 >
< PARIZQ ,  >
< INT ,  >
< ID , 2 >
< COMA ,  >
< INT ,  >
< ID , 3 >
```

```
< PARDER ,  >
< LLAVIZQ ,  >
< RETURN ,  >
< ID , 3 >
< PYC ,  >
< LLAVDER ,  >
< FUNC ,  >
< INT ,  >
< ID , 4 >
< PARIZQ ,  >
< INT ,  >
< ID , 5 >
< COMA ,  >
< INT ,  >
< ID , 6 >
< PARDER ,  >
< LLAVIZQ ,  >
< RETURN ,  >
< ID , 6 >
< PYC ,  >
< LLAVDER ,  >
< EOF ,  >
```

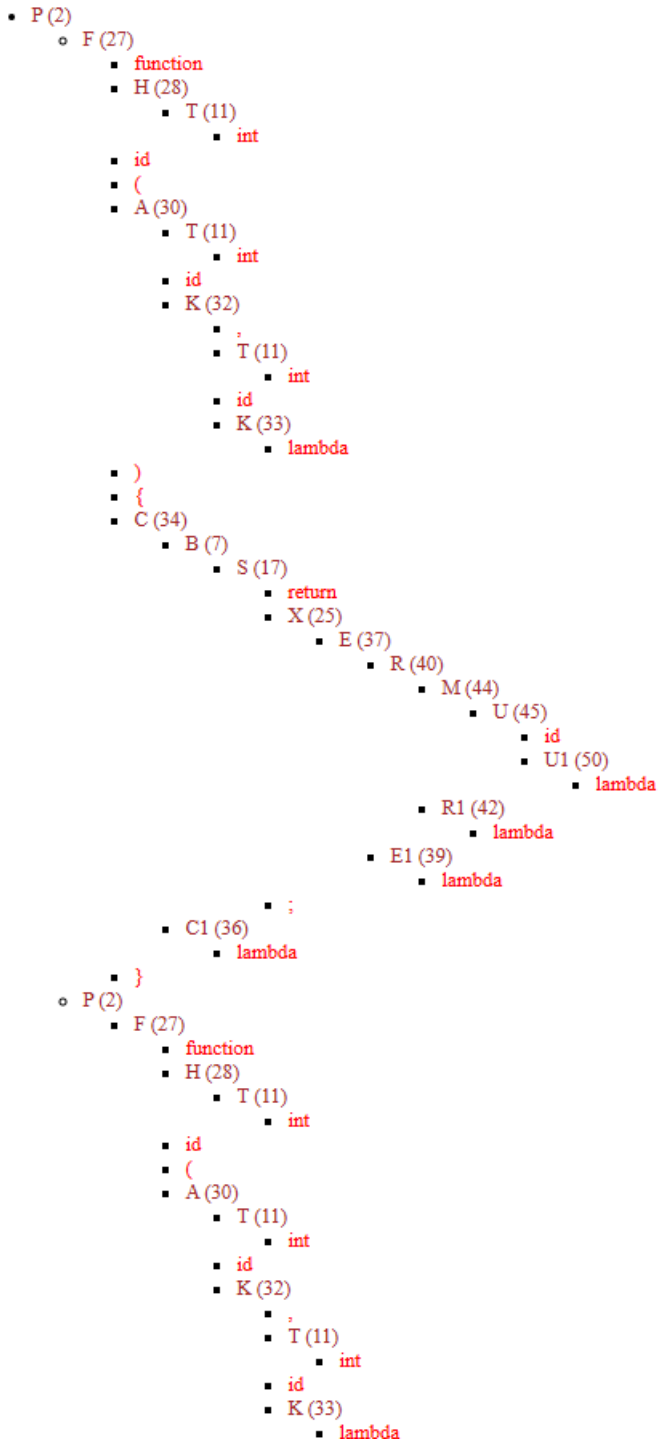
- Tabla de símbolos:

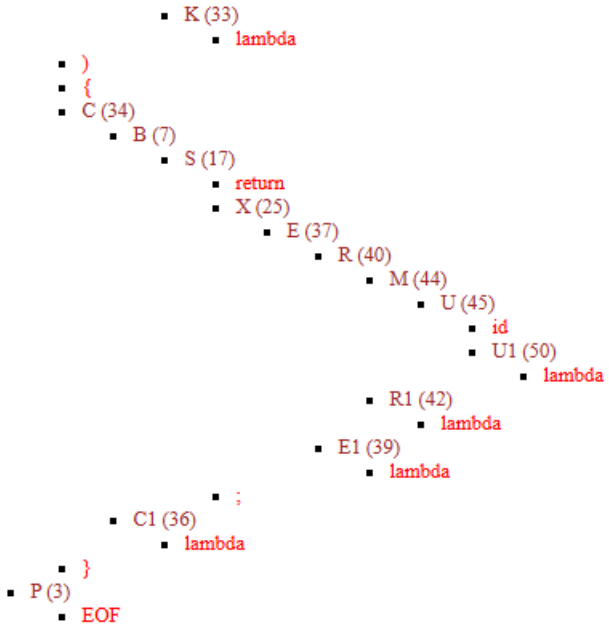
```
TABLA #0 :
 * Lexema : 'devolverB'
   Atributos :
   + tipo : 'funcion'
     + numParam : '2'
       + TipoParam1 : 'entero'
       + TipoParam2 : 'entero'
       + TipoRetorno : 'entero'
     + EtiqFuncion : 'devolverB'
---------------------------------
 * Lexema : 'devolverD'
   Atributos :
   + tipo : 'funcion'
     + numParam : '2'
       + TipoParam1 : 'entero'
       + TipoParam2 : 'entero'
       + TipoRetorno : 'entero'
     + EtiqFuncion : 'devolverD'
---------------------------------

TABLA #1 :
 * Lexema : 'a'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'b'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
```

```
TABLA #2 :
 * Lexema : 'c'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'd'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
```

- Árbol sintáctico generado:





Caso de prueba correcta 5:

```
1  var int x;
2  var int y;
3  var boolean flag;
4
5  x = 3;
6  y = x;
7
8  var boolean isEqual;
9  isEqual = x == y;
10
11 if (isEqual) y = y / 1;
12
13 isEqual = y == x;
14
15 if (isEqual) y = y / 4;
16
17 x = x / y;
18
19 output x;
20 output y;
```

- Listado de tokens:

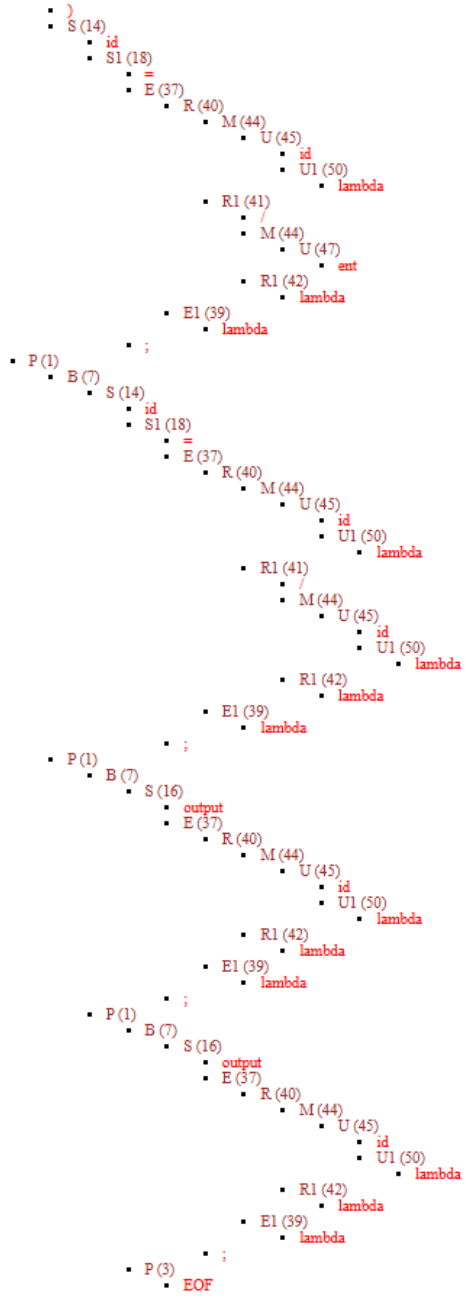
```
< VAR ,  >
< INT ,  >
< ID , 1 >
< PYC ,  >
< VAR ,  >
< INT ,  >
< ID , 2 >
< PYC ,  >
< VAR ,  >
< BOOLEAN ,  >
< ID , 3 >
< PYC ,  >
< ID , 1 >
< IG ,  >
< ENT , 3 >
< PYC ,  >
< ID , 2 >
< IG ,  >
< ID , 1 >
< PYC ,  >
< VAR ,  >
< BOOLEAN ,  >
< ID , 4 >
< PYC ,  >
< ID , 4 >
< IG ,  >
< ID , 1 >
< DIST ,  >
< ID , 2 >
< PYC ,  >
< IF ,  >
< PARIZQ ,  >
< ID , 4 >
< PARDER ,  >
< ID , 2 >
< IG ,  >
< ID , 2 >
< DIV ,  >
< ENT , 1 >
< PYC ,  >
< ID , 4 >
< IG ,  >
< ID , 2 >
< DIST ,  >
< ID , 1 >
< PYC ,  >
< IF ,  >
< PARIZQ ,  >
< ID , 4 >
< PARDER ,  >
< ID , 2 >
< IG ,  >
< ID , 2 >
< DIV ,  >
< ENT , 4 >
< PYC ,  >
< ID , 1 >
< IG ,  >
< ID , 1 >
< DIV ,  >
< ID , 2 >
< PYC ,  >
< OUTPUT ,  >
< ID , 1 >
< PYC ,  >
< OUTPUT ,  >
< ID , 2 >
< PYC ,  >
< EOF ,  >
```

- Tabla de símbolos:

```
TABLA #0 :
 * Lexema : 'x'
   Atributos :
   + tipo : 'entero'
   + despl : 0
---------------------------------
 * Lexema : 'y'
   Atributos :
   + tipo : 'entero'
   + despl : 4
---------------------------------
 * Lexema : 'flag'
   Atributos :
   + tipo : 'logico'
   + despl : 8
---------------------------------
 * Lexema : 'isEqual'
   Atributos :
   + tipo : 'logico'
   + despl : 9
---------------------------------
```

- Árbol sintáctico generado:


id
U1 (50)
lambda
R1 (42)
lambda
;
P (1)
B (6)
if
(
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (41)
/
M (44)
U (47)
ent
R1 (42)
lambda
E1 (39)
lambda
;
P (1)
B (7)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (38)
!=
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
;
P (1)
B (6)
if
(
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
)
S (14)


)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (41)
/
M (44)
U (47)
ent
R1 (42)
lambda
E1 (39)
lambda
;
P (1)
B (7)
S (14)
id
S1 (18)
=
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (41)
/
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
P (1)
B (7)
S (16)
output
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
P (1)
B (7)
S (16)
output
E (37)
R (40)
M (44)
U (45)
id
U1 (50)
lambda
R1 (42)
lambda
E1 (39)
lambda
;
P (3)
EOF

Caso de prueba error 1:

```
1  var int x;
2  var int y;
3
4  x = 2;
5  y = 3;
6
7  sumar(x,y);
```

```
Console
<terminated> main (2) [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe  (13 ene. 2025 11:53:45 – 11:53:45)
Error Semántico en línea 7: El identificador 'sumar' no es una función
```

Caso de prueba error 2:

```
1  var int x;
2  var int y;
3
4  x = 2;
5  y = 3;
6
7  for(var int x; x!=y x = 3){
8      contador = 1;
9  }
```

```
Console
<terminated> main (2) [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe  (13 ene. 2025 12:0
Error Sintáctico en línea 7: se esperaba PYC, se encontró ID
```

Caso de prueba error 3:

```
1  var int x;
2  var int y;
3
4  x = 2;
5  y = 3/2/3/5/3/2/9;
6
7  z = 6*4;
```

```
Error Léxico 5 en la linea 7: el caracter * no se reconoce como un token
Error Sintáctico en línea 7: se esperaba PYC, se encontró ENT
Error Sintáctico en línea 7: se esperaba FOR|VAR|IF|ID|INPUT|OUTPUT|RETURN|FUNC|EOF, se encontró ENT
```

Caso de prueba error 4:

```
1  var int resultado;
2  var boolean condicion;
3  var int x;
4
5  if (resultado) {  // Aquí se intenta evaluar un entero
6
7          output x;
8      }
9  resultado = resultado / 2;
```

```
Console ×
<terminated> main [Java Application] C:\Program Files\Java\jdk-11.0.11
Error Semántico en línea 5: Condicion no es logica
```

Caso de prueba error 5:

```
1  var int enteroLargo;
2  var string cadenaLarga;
3  cadenaLarga='_____________________________________________________+64';
4  enteroLargo=22222222222;
```

```
Console ×
<terminated> main [Java Application] C:\Program Files\Java\jdk-11.0.11\bin\javaw.exe  (13 ene. 2025 17:33:48 – 17:33:49)
Error Léxico 3 en la linea 3: La cadena _____________________________________________+64 supera el tamaño máximo permitido
Error Léxico 2 en la linea 4: Número fuera de rango en la línea 4
```