PROCESADOR DE LENGUAJES

GRUPO XXX

Índice

- Introducción ……2
- Diseño del Analizador Léxico ……2
- Diseño del Analizador Sintáctico ……5
- Diseño del Analizador Semántico ……
- Diseño de la Tabla de Símbolos ……16
- Anexo ……17

Introducción

Hemos realizado este procesador usando el lenguaje Go de Google, una alternativa a C más actual y muy agradable de programar, sin el uso de ninguna librería externa. Todo el grupo hemos trabajado desde Linux y usando github para mayor comodidad.

Diseño del Analizador Léxico

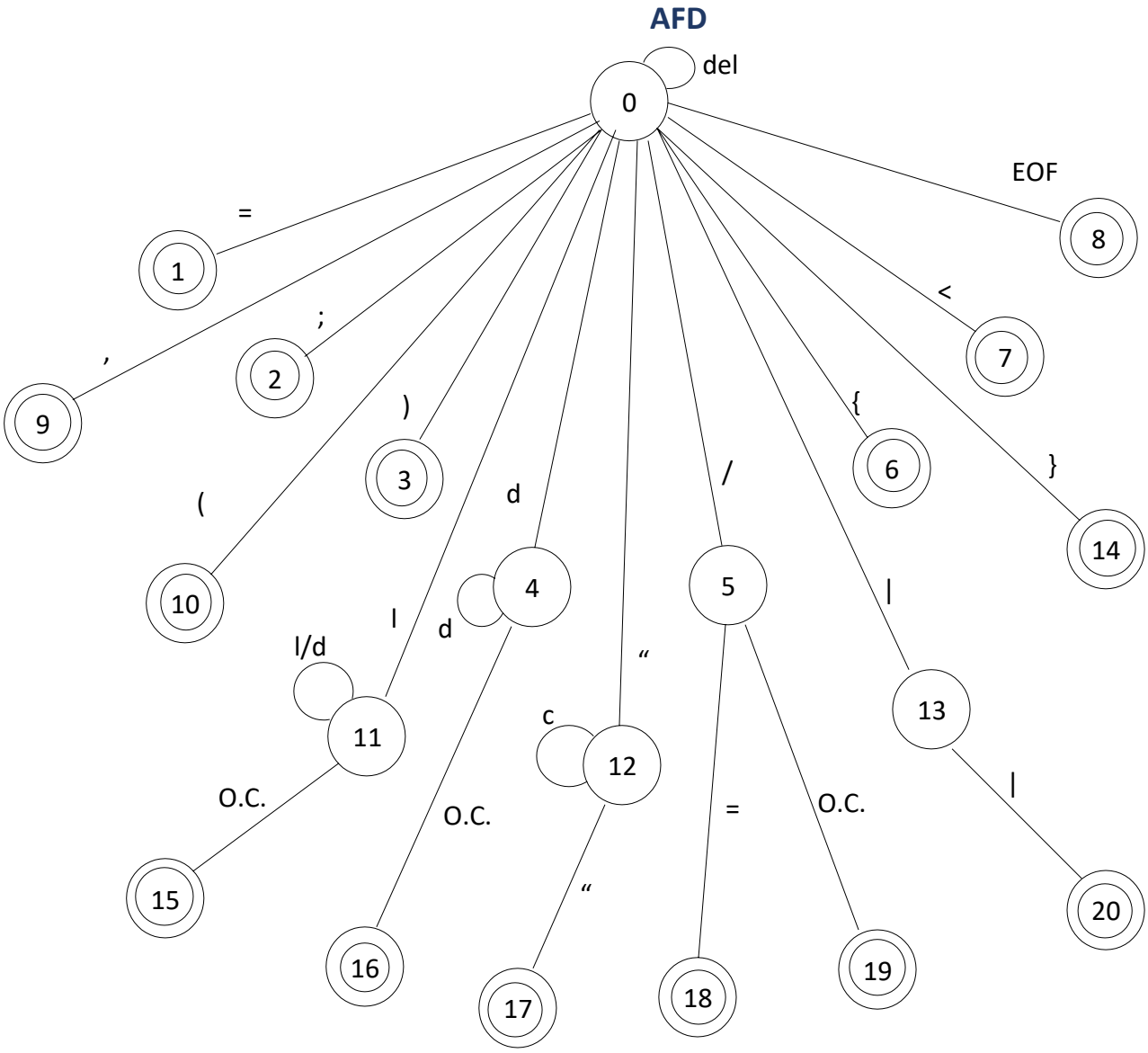
Tokens

<TYPEBOOL,- >
<LOOPDOW,- >
<DECLFUNC,- >
<CONDIF,- >
<IOIN,- >
<TYPEINT,- >
<IOOUT,- >
<RET,- >
<TYPESTR,- >
<DECLVAR,- >
<LOOPWHL,- >
<ID,num >
<OPASSIGN,- >
<SEPCOM,- >
<SEPSCOM,- >
<SEPLPAR,- >
<SEPRPAR,- >
<SEPLCURL,- >
<SEPRCURL,- >
<OPLT,- >
<VALINT,num>
<SEPSTR,-"c*">
<OPDIVASSIGN,- >
<OPDIV,- >
<OPOR,- >
<DELEOF,- >

Gramática

S → lA | dB | "C | /D | |E | = | , | ; | (|) | { | } | < | EOF | delS
A → lA | dA |λ
B → dB | λ
C → cC | "
D → = | λ
E → |

Autómata


AFD
del
0
EOF
8
=
1
<
7
;
2
,
9
{
6
)
3
}
14
(
10
d
4
/
5
|
13
l
d
l/d
11
“
c
12
|
20
O.C.
15
O.C.
16
“
17
=
18
O.C.
19

Acciones Semánticas

```
0:0     leer()
0:1     GenToken(OPASSIGN, -)
0:9     GenToken(SEPCOM, -)
0:2     GenToken(SEPSCOM, -)
0:10    GenToken(SEPLPAR, -)
0:3     GenToken(SEPRPAR, -)
0:6     GenToken(SEPLCURL, -)
0:14    GenToken(SEPRCURL, -)
0:7     GenToken(OPLT, -)
0:8     GenToken(DELEOF, -)
0:4     num = d; leer()
4:4     num = num *10 + d; leer()
4:16    GenToken(VALINT, num)
0:12    cadena = “
12:12   cadena = cadena + c; leer()
12:17   cadena = cadena + c; GenToken(SEPSTR, cadena)
0:5     leer()
5:18    GenToken(OPDIVASSIGN, -)
5:19    GenToken(OPDIV, -)
0:13    leer()
13:20   GenToken(OPOR, -)
0:11    lexema = l; leer()
11:11   lexema = lexema + l/d; leer()
11:15   p := buscarTS(lexema)
            If (lexema = “boolean”) Then GenToken(TYPEBOOL, -)
            else If (lexema = “do”) Then GenToken(LOOPDOW, -)
            else If (lexema = “function”) Then GenToken(DECLFUNC, -)
            else If (lexema = “if”) Then GenToken(CONDIF, -)
            else If (lexema = “input”) Then GenToken(IOIN, -)
            else If (lexema = “int”) Then GenToken(TYPEINT, -)
            else If (lexema = “print”) Then GenToken(IOOUT, -)
            else If (lexema = “return”) Then GenToken(RET, -)
            else If (lexema = “string”) Then GenToken(TYPESTR, -)
            else If (lexema = “var”) Then GenToken(DECLVAR, -)
            else If (lexema = “while”) Then GenToken(LOOPWHL, -)
            else If (p = NULL) Then insertarTS(lexema) GenToken(ID, p)
```

Diseño del Analizador Sintáctico

Gramática

P -> D P	////1
P -> F P	////2
P -> S P	////3
P -> eof	////4
D -> var T id ;	////5
T -> int	////6
T -> string	////7
T -> boolean	////8
F -> function J id (A) { C }	////9
J -> T	////10
J -> lambda	////11
A -> lambda	////12
A -> T id AI	////13
AI -> lambda	////14
AI -> , T id AI	////15
C -> D C	////16
C -> S C	////17
C -> lambda	////18
S -> id E ;	////19
S -> if (E) B	////20
B -> S	////21
B -> { H }	////22
H -> D H	////23
S -> print (E) ;	////24
S -> input (E) ;	////25
S -> return E ;	////26
L -> lambda	////27
L -> , id LI	////28
LI -> lambda	////29
LI -> , id LI	////30
E -> lambda	////31
SC -> do { C } W	////32
W -> while (E) ;	////33
E -> Z EI	////34
EI -> lambda	////35
EI -> || Z EI	////36
Z -> U ZI	////37
ZI -> lambda	////38
ZI -> < U ZI	////39
U -> O UI	////40
UI -> lambda	////41
UI -> / O UI	////42

O -> M OI ////43
OI -> lambda ////44
OI -> /= M OI ////45
M -> V MI ////46
MI -> lambda ////47
MI -> = V MI ////48
V -> id VI ////49
V -> entero ////50
V -> cadena ////51
V -> (E) ////52
VI -> (L) ////53
VI -> lambda ////54

Demostración de que la gramática es adecuada

Para que la gramática propuesta sea adecuada debe cumplir una serie de condiciones:

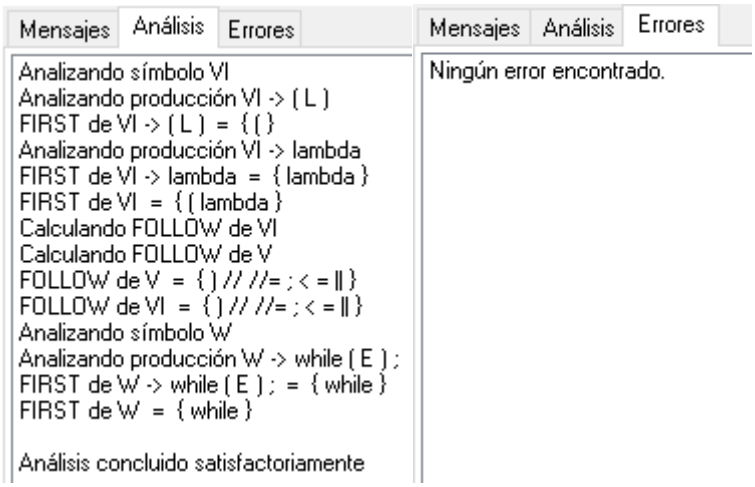
1. No estar factorizada
2. No ser recursiva por la izquierda
3. La gramática ha de ser LL(1)

Para que una gramática sea LL(1) además se debe cumplir lo siguiente:

Para cualquier par de producciones $A \rightarrow \alpha$, $A \rightarrow \beta$ se cumple:

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. $Si\ \lambda \in FIRST(\alpha), entonces\ FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

Usamos la herramienta SDGLL1 proporcionada por el departamento para extraer conjuntos First y Follow y analizar automáticamente si la gramática cumple las condiciones para ser LL(1). A continuación, adjuntamos una imagen del resultado de ejecutar el programa sobre nuestra gramática:



Procedimientos (Análisis descendente recursivo)

Se tiene la función auxiliar comp(token) y errorSintaxis(msg) para el correcto análisis sintáctico. Permiten comparar tokens y notificar errores de sintaxis. Al parser se le pasa la lista completa de tokens devueltos por el lexer.

```
func P() {
	token := Tokens[CurrentToken].Tok
	for token == ERROR {
		line++
		CurrentToken++
		token = Tokens[CurrentToken].Tok
	}
	if token == "EOF" {
		result += " 4"
		// Termina
	} else if token == "DECLFUNC" {
		result += " 2"
		F()
		P()
	} else if token == "ID" || token == "CONDIF" || token == "IOIN"
|| token == "IOOUT" || token == "RET" {
		result += " 3"
		S()
		P()
	} else if token == "DECLVAR" {
		result += " 1"
		D()
		P()
	} else {
		ErrorSintaxis("")
	}
}

// F procedure that deals with functions
func F() {
	token := Tokens[CurrentToken].Tok
	if token == "DECLFUNC" {
		flagDecl = 1
		result += " 9"
		comp("DECLFUNC")
		FTipo := J()
		idFunc = Tokens[CurrentToken].Name
		posTS := comp("ID")
		flagDecl = 0
		setType(FTipo, posTS)
		flagFunc = 1
		flagTab = flagAux + 1
		flagAux = flagTab
		comp("SEPLPAR")
		params = 1
		A()
		setTypeParam(FTipo, posTS)
		setParam(FTipo, posTS)
		typeParam = make([]string, 0)
		params = 0
		comp("SEPRPAR")
		comp("SEPLCURL")
```

```
                C()
                comp("SEPRCURL")
                // params = -1
                idFunc = ""
                flagFunc = 0
                flagTab = 0
        } else {
                ErrorSintaxis("")
        }
}

// J Procedure that gives type to a function
func J() string {
        token := Tokens[CurrentToken].Tok
        var FTipo string
        if token == "TYPEBOOL" || token == "TYPEINT" || token ==
"TYPESTR" {
                result += " 10"
                FTipo = T()
        } else if token == "ID" {
                //FTipo = "VOID"
                result += " 11"
        } else {
                ErrorSintaxis("")
        }

        return FTipo
}

// A procedure that gives type to function arguments
func A() {
        token := Tokens[CurrentToken].Tok
        if token == "TYPEBOOL" || token == "TYPEINT" || token ==
"TYPESTR" {
                result += " 13"
                ATipo := T()
                typeParam = append(typeParam, ATipo)
                flagDecl = 1
                posTS := comp("ID")
                flagDecl = 0
                setType(ATipo, posTS)
                AI()
        } else if token == "SEPRPAR" {
                result += " 12"
        } else {
                ErrorSintaxis("")
        }
}

// C Procedure that deals with lower scopes (everything between {...})
func C() {
        token := Tokens[CurrentToken].Tok
        if token == "DECLVAR" {
                result += " 16"
                D()
                C()
        } else if token == "ID" || token == "CONDIF" || token == "IOIN"
|| token == "IOOUT" || token == "RET" {
                result += " 17"
                S()
```

```
                C()
        } else if token == "SEPRCURL" {
                result += " 18"
        } else {
                ErrorSintaxis("")
        }
}

// S Procedure: arithmeticological operations, if and return
statements, and IO
func S() {
        token := Tokens[CurrentToken].Tok
        if token == "ID" {
                result += " 19"
                E()
                comp("SEPSCOM")
        } else if token == "CONDIF" {
                result += " 20"
                comp("CONDIF")
                comp("SEPLPAR")
                E()
                comp("SEPRPAR")
                B()
        } else if token == "IOIN" {
                result += " 22"
                comp("IOIN")
                comp("SEPLPAR")
                E()
                comp("SEPRPAR")
                comp("SEPSCOM")
        } else if token == "IOOUT" {
                result += " 23"
                comp("IOOUT")
                comp("SEPLPAR")
                E()
                comp("SEPRPAR")
                comp("SEPSCOM")
        } else if token == "RET" {
                result += " 24"
                comp("RET")
                if Tokens[CurrentToken].Tok != SEPSCOM {
                        ErrorSintaxis("")
                }
                E()
                comp("SEPSCOM")
        } else {
                ErrorSintaxis("")
        }
}

// D proc: Variable Declaration
func D() {
        token := Tokens[CurrentToken].Tok
        if token == "DECLVAR" {
                result += " 5"
                comp("DECLVAR")
                DTipo := T()
                flagDecl = 1
                posTS := comp("ID")
                flagDecl = 0
```

```
                setType(DTipo, posTS)
                comp("SEPSCOM")
        } else {
                ErrorSintaxis("")
        }
}

// T proc: Handles static Typing
func T() string {
        token := Tokens[CurrentToken].Tok
        TTipo := token
        if token == "TYPEBOOL" {
                result += " 8"
                comp("TYPEBOOL")
        } else if token == "TYPESTR" {
                result += " 7"
                comp("TYPESTR")
        } else if token == "TYPEINT" {
                result += " 6"
                comp("TYPEINT")
        } else {
                ErrorSintaxis("")
        }

        return TTipo
}

// AI proc: additional function arguments after the first argument
func AI() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPCOM" {
                params++
                result += " 15"
                comp("SEPCOM")
                AITipo := T()
                typeParam = append(typeParam, AITipo)
                flagDecl = 1
                posTS := comp("ID")
                flagDecl = 0
                setType(AITipo, posTS)
                AI()
        } else if token == "SEPRPAR" {
                result += " 14"
        } else {
                ErrorSintaxis("")
        }
}

// B proc: IF statement {}
func B() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLCURL" {
                result += " 21"
                comp("SEPLCURL")
                C()
                comp("SEPRCURL")
        } else {
                ErrorSintaxis("")
        }
}
```

```
// L proc: function parameters
func L(idCall string) {
	token := Tokens[CurrentToken].Tok
	if token == "ID" {
		result += " 26"
		callFunc = 1
		tokName := Tokens[CurrentToken].Name
		comp("ID")
		typeParamCall = append(typeParamCall,
searchType(tokName))
		LI()
		compareTypesCall(idCall)
		typeParamCall = make([]string, 0)
		callFunc = 0
	} else if token == "SEPRPAR" {
		result += " 25"
	} else {
		ErrorSintaxis("")
	}
}

// LI proc: additional parameters after the first one
func LI() {
	token := Tokens[CurrentToken].Tok
	if token == "SEPCOM" {
		result += " 28"
		comp("SEPCOM")
		tokName := Tokens[CurrentToken].Name
		comp("ID")
		typeParamCall = append(typeParamCall,
searchType(tokName))
		LI()
	} else if token == "SEPRPAR" {
		result += " 27"
	} else {
		ErrorSintaxis("")
	}
}

// SC proc: Do While
func SC() {
	token := Tokens[CurrentToken].Tok
	if token == "LOOPDOW" {
		result += " 30"
		comp("LOOPDOW")
		comp("SEPLCURL")
		C()
		comp("SEPRCURL")
		W()
	} else {
		ErrorSintaxis("")
	}
}

// W proc: While loop
func W() {
	token := Tokens[CurrentToken].Tok
	if token == "LOOPWHL" {
		result += " 31"
```

```
                comp("LOOPWHL")
                comp("SEPLPAR")
                E()
                comp("SEPRPAR")
                comp("SEPSCOM")
        } else {
                ErrorSintaxis("")
        }
}

// E proc: Arithmeticological operations
func E() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLPAR" || token == "SEPSTR" || token == "VALINT"
|| token == "ID" {
                result += " 32"
                if token == ID {
                        opTypes = append(opTypes,
searchType(Tokens[CurrentToken].Name))
                } else if token != SEPLPAR {
                        opTypes = append(opTypes, token)
                }
                Z()
                EI()
                compareTypes(opTypes[0])
        } else if token == "SEPRPAR" || token == "SEPSCOM" {
                result += " 29"
        } else {
                ErrorSintaxis("")
        }
}

// EI proc: More operational stuff
func EI() {
        token := Tokens[CurrentToken].Tok
        if token == "OPOR" {
                result += " 34"
                comp("OPOR")
                Z()
                EI()
        } else if token == "SEPRPAR" || token == "SEPSCOM" {
                result += " 33"
        } else {
                ErrorSintaxis("")
        }
}

// Z proc: Less than
func Z() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLPAR" || token == "SEPSTR" || token == "VALINT"
|| token == "ID" {
                result += " 35"
                if token == ID {
                        opTypes = append(opTypes,
searchType(Tokens[CurrentToken].Name))
                } else if token != SEPLPAR {
                        opTypes = append(opTypes, token)
                }
                U()
```

```
                ZI()
        } else {
                ErrorSintaxis("")
        }
}

// ZI proc: Less than (2)
func ZI() {
        token := Tokens[CurrentToken].Tok
        if token == "OPLT" {
                result += " 37"
                comp("OPLT")
                U()
                ZI()
        } else if token == "SEPRPAR" || token == "SEPSCOM" || token ==
"OPOR" {
                result += " 36"
        } else {
                ErrorSintaxis("")
        }
}

// U proc: Div
func U() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLPAR" || token == "SEPSTR" || token == "VALINT"
|| token == "ID" {
                result += " 38"
                if token == ID {
                        opTypes = append(opTypes,
searchType(Tokens[CurrentToken].Name))
                } else if token != SEPLPAR {
                        opTypes = append(opTypes, token)
                }
                O()
                UI()
        } else {
                ErrorSintaxis("")
        }
}

// UI proc: more DIV stuff
func UI() {
        token := Tokens[CurrentToken].Tok
        if token == "OPDIV" {
                result += " 40"
                comp("OPDIV")
                O()
                UI()
        } else if token == "SEPRPAR" || token == "SEPSCOM" || token ==
"OPLT" || token == "OPOR" {
                result += " 39"
        } else {
                ErrorSintaxis("")
        }
}

// O proc: DIVASSIGN
func O() {
        token := Tokens[CurrentToken].Tok
```

```
        if token == "SEPLPAR" || token == "SEPSTR" || token == "VALINT"
|| token == "ID" {
                result += " 41"
                if token == ID {
                        opTypes = append(opTypes,
searchType(Tokens[CurrentToken].Name))
                } else if token != SEPLPAR {
                        opTypes = append(opTypes, token)
                }
                M()
                OI()
        } else {
                ErrorSintaxis("")
        }
}

// OI proc: more DIVASSIGN stuff
func OI() {
        token := Tokens[CurrentToken].Tok
        if token == "OPDIVASSIGN" {
                result += " 43"
                comp("OPDIVASSIGN")
                M()
                OI()
        } else if token == "SEPRPAR" || token == "OPDIV" || token ==
"SEPSCOM" || token == "OPLT" || token == "OPOR" {
                result += " 42"
        } else {
                ErrorSintaxis("")
        }
}

// M proc: OPASSIGN
func M() {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLPAR" || token == "SEPSTR" || token == "VALINT"
|| token == "ID" {
                result += " 44"
                if token == ID {
                        opTypes = append(opTypes,
searchType(Tokens[CurrentToken].Name))
                } else if token != SEPLPAR {
                        opTypes = append(opTypes, token)
                }
                V()
                MI()
        } else {
                ErrorSintaxis("")
        }
}

// MI proc: more OPASSIGN stuff
func MI() {
        token := Tokens[CurrentToken].Tok
        if token == "OPASSIGN" {
                result += " 46"
                comp("OPASSIGN")
                V()
                MI()
```

```
        } else if token == "SEPRPAR" || token == "OPDIV" || token ==
"OPDIVASSIGN" || token == "SEPSCOM" || token == "OPLT" || token ==
"OPOR" {
                result += " 45"
        } else {
                ErrorSintaxis("")
        }
}

// V proc: magical stuff
func V() {
        token := Tokens[CurrentToken].Tok
        idCall := Tokens[CurrentToken].Name
        if token == "ID" {
                result += " 47"
                opTypes = append(opTypes, searchType(idCall))
                comp("ID")
                VI(idCall)
        } else if token == "VALINT" {
                result += " 48"
                opTypes = append(opTypes, token)
                comp("VALINT")
        } else if token == "SEPSTR" {
                result += " 49"
                opTypes = append(opTypes, token)
                comp("SEPSTR")
        } else if token == "SEPLPAR" {
                result += " 50"
                comp("SEPLPAR")
                E()
                comp("SEPRPAR")
        } else {
                ErrorSintaxis("")
        }
}

// VI proc: more magical stuff
func VI(idCall string) {
        token := Tokens[CurrentToken].Tok
        if token == "SEPLPAR" {
                result += " 51"
                comp("SEPLPAR")
                L(idCall)
                comp("SEPRPAR")
        } else if token == "SEPRPAR" || token == "OPASSIGN" || token ==
"OPDIV" || token == "OPDIVASSIGN" || token == "SEPSCOM" || token ==
"OPLT" || token == "OPOR" {
                result += " 52"
        } else {
                ErrorSintaxis("")
        }
}
```

Diseño de la Tabla de Símbolos

A nivel de implementación hemos usado slices dobles, similar a una matriz de arraylist.

Se estructura y organiza de la siguiente forma:

CONTENIDO DE LA TABLA #X:
---

* LEXEMA: '-'
* TIPO: '-'
* POS: '-'
* DESPLZ: '-'
* NUMPARAMS: '-'
* PARAM: '-'
* PARAMTYPES: '-'
* TIPORETORNO: '-'

++ SIGUIENTE ELEMENTO DE LA TABLA CON LA MISMA ESTRUCTURA ++

CONTENIDO DE LA TABLA #X+1:

++ MISMA ESTRUCTURA QUE ANTES, SE REPITE PARA CUANTAS TABLAS HAYA ++

Anexo

Prueba 1

```
   var string texto;
   function imprime (string msg)
   {
            print ("Mensage introducido:");
            print (msg);
   }
   function pideTexto ()
   {
            print ("Introduce un texto");
            input (texto);
   }
   pideTexto();
   var string textoAux;
   textoAux = texto;
imprime (textoAux);
```

Tokens:

```
   <DECLVAR, >
   <TYPESTR, >
   <ID,0>
   <SEPSCOM, >
   <DECLFUNC, >
   <ID,1>
   <SEPLPAR, >
   <TYPESTR, >
   <ID,0>
   <SEPRPAR, >
   <SEPLCURL, >
   <IOOUT, >
   <SEPLPAR, >
   <SEPSTR,"Mensage introducido:">
   <SEPRPAR, >
   <SEPSCOM, >
   <IOOUT, >
   <SEPLPAR, >
   <ID,>
   <SEPRPAR, >
   <SEPSCOM, >
   <SEPRCURL, >
   <DECLFUNC, >
   <ID,2>
   <SEPLPAR, >
```

<SEPRPAR, >
<SEPLCURL, >
<IOOUT, >
<SEPLPAR, >
<SEPSTR,"Introduce un texto">
<SEPRPAR, >
<SEPSCOM, >
<IOIN, >
<SEPLPAR, >
<ID,>
<SEPRPAR, >
<SEPSCOM, >
<SEPRCURL, >
<ID,2>
<SEPLPAR, >
<SEPRPAR, >
<SEPSCOM, >
<DECLVAR, >
<TYPESTR, >
<ID,3>
<SEPSCOM, >
<ID,3>
<OPASSIGN, >
<ID,0>
<SEPSCOM, >
<ID,1>
<SEPLPAR, >
<ID,3>
<SEPRPAR, >
<SEPSCOM, >
<EOF, >

TS:

CONTENIDO DE LA TABLA #1:

---

* LEXEMA: 'texto'
* TIPO: 'TYPESTR'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'imprime'
* TIPO: ''
* POS: '2'
* DESPLZ: '1'

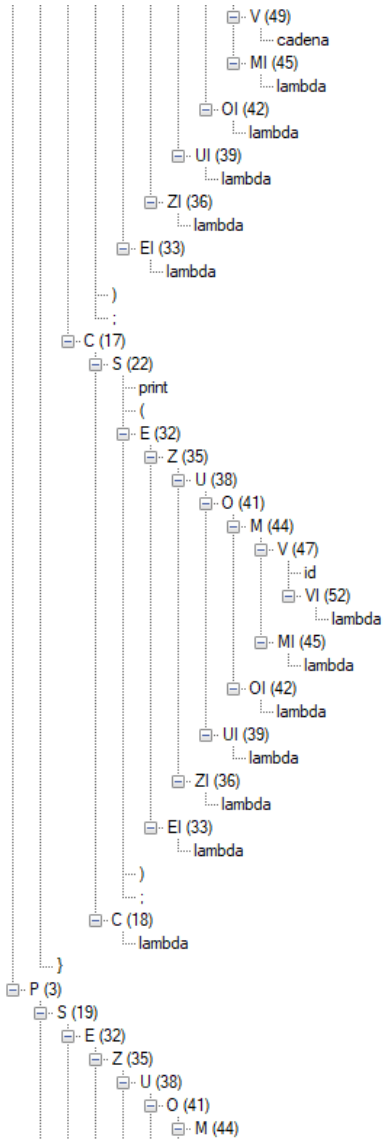
```
    * NUMPARAMS: '1'
    * PARAMTYPES: '[TYPESTR]'
    ------------------------
    * LEXEMA: 'pideTexto'
    * TIPO: ''
    * POS: '3'
    * DESPLZ: '2'
    * NUMPARAMS: '1'
    * PARAMTYPES: '[]'
    ------------------------
    * LEXEMA: 'textoAux'
    * TIPO: 'TYPESTR'
    * POS: '4'
    * DESPLZ: '3'
    * NUMPARAMS: '0'
    * PARAMTYPES: '[]'
    ------------------------
    CONTENIDO DE LA TABLA #2:
    ----------------
    * LEXEMA: 'msg'
    * TIPO: 'TYPESTR'
    * POS: '1'
    * DESPLZ: '0'
    * NUMPARAMS: '0'
    * PARAMTYPES: '[]'
    ------------------------
    CONTENIDO DE LA TABLA #3:
----------------
```

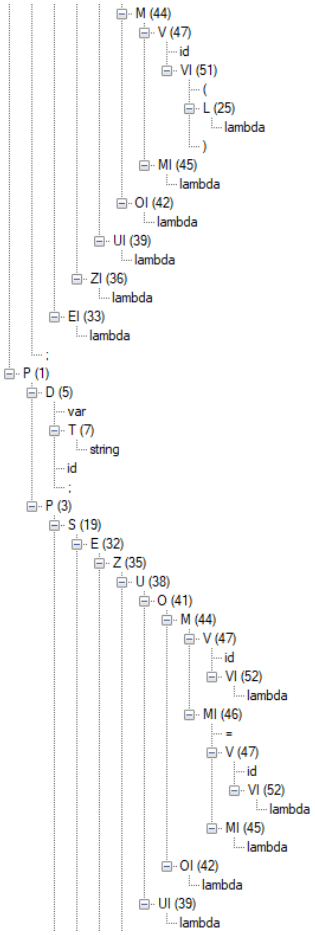
Parse:

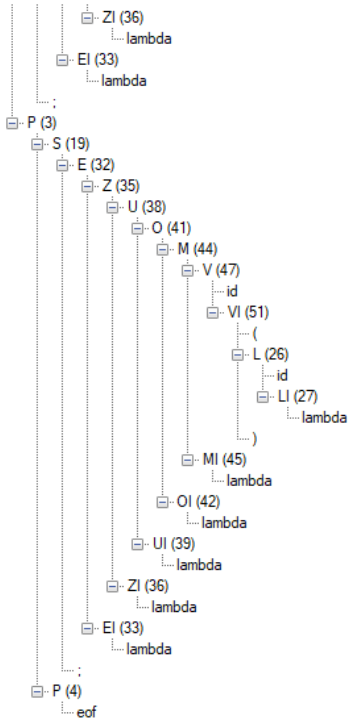
Descendente 1 5 7 2 9 11 13 7 14 17 23 32 35 38 41 44 49 45 42 39 36 33 17 23 32 35 38 41 44 47 52 45 42 39 36 33 18 2 9 11 12 17 23 32 35 38 41 44 49 45 42 39 36 33 17 22 32 35 38 41 44 47 52 45 42 39 36 33 18 3 19 32 35 38 41 44 47 51 25 45 42 39 36 33 1 5 7 3 19 32 35 38 41 44 47 52 46 47 52 45 42 39 36 33 3 19 32 35 38 41 44 47 51 26 27 45 42 39 36 33 4

Árbol:


P (1)
D (5)
var
T (7)
string
id
;
P (2)
F (9)
function
J (11)
lambda
id
(
A (13)
T (7)
string
id
AI (14)
lambda
)
{
C (17)
S (23)
input
(
E (32)
Z (35)
U (38)
O (41)
M (44)
V (49)
cadena
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda
ZI (36)
lambda
EI (33)
lambda
)
;
C (17)
C (17)
S (23)
input
(
E (32)
Z (35)
U (38)
O (41)
M (44)
V (47)
id
VI (52)
lambda
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda
ZI (36)
lambda
EI (33)
lambda
)
;
C (18)
lambda
}
P (2)
F (9)
function
J (11)
lambda
id
(
A (12)
lambda
)
{
C (17)
S (23)
input
(
E (32)
Z (35)
U (38)
O (41)
M (44)
V (49)


V (49)
cadena
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda
ZI (36)
lambda
EI (33)
lambda
)
;
C (17)
S (22)
print
(
E (32)
Z (35)
U (38)
O (41)
M (44)
V (47)
id
VI (52)
lambda
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda
ZI (36)
lambda
EI (33)
lambda
)
;
C (18)
lambda
}
P (3)
S (19)
E (32)
Z (35)
U (38)
O (41)
M (44)


M (44)
V (47)
id
VI (51)
(
L (25)
lambda
)
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda
ZI (36)
lambda
EI (33)
lambda
;
P (1)
D (5)
var
T (7)
string
id
;
P (3)
S (19)
E (32)
Z (35)
U (38)
O (41)
M (44)
V (47)
id
VI (52)
lambda
MI (46)
=
V (47)
id
VI (52)
lambda
MI (45)
lambda
OI (42)
lambda
UI (39)
lambda



Prueba 2

```
    var int a;
    var int b;
    a = 3;
    b = a;
    var boolean c;
    c = a < b;
    if (c) {b = 1;}
    if (b < a){ b /= 4;}
    a = a / b;
    print (a);
print (b);
```

Tokens:

```
    <OPASSIGN, >
    <VALINT,3>
    <SEPSCOM, >
    <ID,1>
    <OPASSIGN, >
    <ID,0>
    <SEPSCOM, >
    <DECLVAR, >
    <TYPEBOOL, >
    <ID,2>
    <SEPSCOM, >
    <ID,2>
    <OPASSIGN, >
```

<ID,0>
<OPLT, >
<ID,1>
<SEPSCOM, >
<CONDIF, >
<SEPLPAR, >
<ID,2>
<SEPRPAR, >
<SEPLCURL, >
<ID,>
<OPASSIGN, >
<VALINT,1>
<SEPSCOM, >
<SEPRCURL, >
<CONDIF, >
<SEPLPAR, >
<ID,1>
<OPLT, >
<ID,0>
<SEPRPAR, >
<SEPLCURL, >
<ID,>
<OPDIVASSIGN, >
<VALINT,4>
<SEPSCOM, >
<SEPRCURL, >
<ID,0>
<OPASSIGN, >
<ID,0>
<OPDIV, >
<ID,1>
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<ID,0>
<SEPRPAR, >
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<ID,1>
<SEPRPAR, >
<SEPSCOM, >
<EOF, >

TS:

CONTENIDO DE LA TABLA #1:

---

* LEXEMA: 'a'
* TIPO: 'TYPEINT'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'b'
* TIPO: 'TYPEINT'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'c'
* TIPO: 'TYPEBOOL'
* POS: '3'
* DESPLZ: '2'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

Parse:

Descendente 1 5 6 1 5 6 3 19 32 35 38 41 44 47 52 46 48 45 42 39 36 33 3 19 32 35 38 41 44 47 52 46 47 52 45 42 39 36 33 1 5 8 3 19 32 35 38 41 44 47 52 46 47 52 45 42 39 37 38 41 44 47 52 45 42 39 36 33 3 20 32 35 38 41 44 47 52 45 42 39 36 33 21 17 19 32 35 38 41 44 47 52 46 48 45 42 39 36 33 18 3 20 32 35 38 41 44 47 52 45 42 39 37 38 41 44 47 52 45 42 39 36 33 21 17 19 32 35 38 41 44 47 52 45 43 44 48 45 42 39 36 33 18 3 19 32 35 38 41 44 47 52 46 47 52 45 42 40 41 44 47 52 45 42 39 36 33 3 23 32 35 38 41 44 47 52 45 42 39 36 33 3 23 32 35 38 41 44 47 52 45 42 39 36 33 4

Prueba 3

```
var int a;
var int b;
var int c;
print ("Introduce el primer operando");
input (a);
print ("Introduce el segundo operando");
input (b);
function int divide (int num1, int num2)
{
        return num1/num2;
}
```

```
    c = divide (a, b);
print (c);
```

Tokens:

```
    <DECLVAR, >
    <TYPEINT, >
    <ID,0>
    <SEPSCOM, >
    <DECLVAR, >
    <TYPEINT, >
    <ID,1>
    <SEPSCOM, >
    <DECLVAR, >
    <TYPEINT, >
    <ID,2>
    <SEPSCOM, >
    <IOOUT, >
    <SEPLPAR, >
    <SEPSTR,"Introduce el primer operando">
    <SEPRPAR, >
    <SEPSCOM, >
    <IOIN, >
    <SEPLPAR, >
    <ID,0>
    <SEPRPAR, >
    <SEPSCOM, >
    <IOOUT, >
    <SEPLPAR, >
    <SEPSTR,"Introduce el segundo operando">
    <SEPRPAR, >
    <SEPSCOM, >
    <IOIN, >
    <SEPLPAR, >
    <ID,1>
    <SEPRPAR, >
    <SEPSCOM, >
    <DECLFUNC, >
    <TYPEINT, >
    <ID,3>
    <SEPLPAR, >
    <TYPEINT, >
    <ID,0>
    <SEPCOM, >
    <TYPEINT, >
    <ID,1>
    <SEPRPAR, >
```

<SEPLCURL, >
<RET, >
<ID,>
<OPDIV, >
<ID,>
<SEPSCOM, >
<SEPRCURL, >
<ID,2>
<OPASSIGN, >
<ID,3>
<SEPLPAR, >
<ID,0>
<SEPCOM, >
<ID,1>
<SEPRPAR, >
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<ID,2>
<SEPRPAR, >
<SEPSCOM, >
<EOF, >

TS:

CONTENIDO DE LA TABLA #1:

---

* LEXEMA: 'a'
* TIPO: 'TYPEINT'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'b'
* TIPO: 'TYPEINT'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'c'
* TIPO: 'TYPEINT'
* POS: '3'
* DESPLZ: '2'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

```
   ------------------------
   * LEXEMA: 'divide'
   * TIPO: 'TYPEINT'
   * POS: '4'
   * DESPLZ: '3'
   * NUMPARAMS: '2'
   * PARAMTYPES: '[TYPEINT TYPEINT]'
   ------------------------
   CONTENIDO DE LA TABLA #2:
   ---------------
   * LEXEMA: 'num1'
   * TIPO: 'TYPEINT'
   * POS: '1'
   * DESPLZ: '0'
   * NUMPARAMS: '0'
   * PARAMTYPES: '[]'
   ------------------------
   * LEXEMA: 'num2'
   * TIPO: 'TYPEINT'
   * POS: '2'
   * DESPLZ: '1'
   * NUMPARAMS: '0'
   * PARAMTYPES: '[]'
------------------------
```

Parse:

Descendente 1 5 6 1 5 6 1 5 6 3 23 32 35 38 41 44 49 45 42 39 36 33 3 22 32 35 38 41 44 47 52 45 42 39 36 33 3 23 32 35 38 41 44 49 45 42 39 36 33 3 22 32 35 38 41 44 47 52 45 42 39 36 33 2 9 10 6 13 6 15 6 14 17 24 32 35 38 41 44 47 52 45 42 40 41 44 47 52 45 42 39 36 33 18 3 19 32 35 38 41 44 47 52 46 47 51 26 28 27 45 42 39 36 33 3 23 32 35 38 41 44 47 52 45 42 39 36 33 4

Prueba 4:

```
   var string texto;
   var string texto;
   var int numero;

   function imprime (string msg)
   {
            var string pepe;
            print ("Mensage introducido:");
            print (msg);
   }

   var string nada;
```

```
function int pideTexto (int pitos, boolean flautas)
{
	var int alfonso;
	alfonso = alfonso / 4 / 6 ;
	var string alex;
	alex = "soy alex";

	print ("Introduce un texto");
	input (texto);

	nada = "eres nada";

	return numero;
}

var boolean hola;

pideTexto(numero, hola);
var string textoAux;
textoAux = texto;
imprime (textoAux);

var int mark;

return;

input(texto);
var boolean patata;
if ( mark || numero < 3 ) {
	print("adios");
}

numero /= 385 / mark;

var boolean kevin;

nada = "kevin y mark";

/*menudo comentario*/
```

Tokens:

<DECLVAR, >
<TYPESTR, >
<ID,0>
<SEPSCOM, >
<DECLVAR, >
<TYPESTR, >
<ID,0>
<SEPSCOM, >
<DECLVAR, >
<TYPEINT, >
<ID,1>
<SEPSCOM, >
<DECLFUNC, >
<ID,2>
<SEPLPAR, >
<TYPESTR, >
<ID,0>
<SEPRPAR, >
<SEPLCURL, >
<DECLVAR, >
<TYPESTR, >
<ID,>
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<SEPSTR,"Mensage introducido:">
<SEPRPAR, >
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<ID,>
<SEPRPAR, >
<SEPSCOM, >
<SEPRCURL, >
<DECLVAR, >
<TYPESTR, >
<ID,3>
<SEPSCOM, >
<DECLFUNC, >
<TYPEINT, >
<ID,4>
<SEPLPAR, >
<TYPEINT, >
<ID,0>
<SEPCOM, >

<TYPEBOOL, >
<ID,1>
<SEPRPAR, >
<SEPLCURL, >
<DECLVAR, >
<TYPEINT, >
<ID,>
<SEPSCOM, >
<ID,>
<OPASSIGN, >
<ID,>
<OPDIV, >
<VALINT,4>
<OPDIV, >
<VALINT,6>
<SEPSCOM, >
<DECLVAR, >
<TYPESTR, >
<ID,>
<SEPSCOM, >
<ID,>
<OPASSIGN, >
<SEPSTR,"soy alex">
<SEPSCOM, >
<IOOUT, >
<SEPLPAR, >
<SEPSTR,"Introduce un texto">
<SEPRPAR, >
<SEPSCOM, >
<IOIN, >
<SEPLPAR, >
<ID,>
<SEPRPAR, >
<SEPSCOM, >
<ID,>
<OPASSIGN, >
<SEPSTR,"eres nada">
<SEPSCOM, >
<RET, >
<ID,>
<SEPSCOM, >
<SEPRCURL, >
<DECLVAR, >
<TYPEBOOL, >
<ID,5>
<SEPSCOM, >

<ID,4>
<SEPLPAR, >
<ID,1>
<SEPCOM, >
<ID,5>
<SEPRPAR, >
<SEPSCOM, >
<DECLVAR, >
<TYPESTR, >
<ID,6>
<SEPSCOM, >
<ID,6>
<OPASSIGN, >
<ID,0>
<SEPSCOM, >
<ID,2>
<SEPLPAR, >
<ID,6>
<SEPRPAR, >
<SEPSCOM, >
<DECLVAR, >
<TYPEINT, >
<ID,7>
<SEPSCOM, >
<RET, >
<SEPSCOM, >
<IOIN, >
<SEPLPAR, >
<ID,0>
<SEPRPAR, >
<SEPSCOM, >
<DECLVAR, >
<TYPEBOOL, >
<ID,8>
<SEPSCOM, >
<CONDIF, >
<SEPLPAR, >
<ID,7>
<OPOR, >
<ID,1>
<OPLT, >
<VALINT,3>
<SEPRPAR, >
<SEPLCURL, >
<IOOUT, >
<SEPLPAR, >

<SEPSTR,"adios">
<SEPRPAR, >
<SEPSCOM, >
<SEPRCURL, >
<ID,1>
<OPDIVASSIGN, >
<VALINT,385>
<OPDIV, >
<ID,7>
<SEPSCOM, >
<DECLVAR, >
<TYPEBOOL, >
<ID,9>
<SEPSCOM, >
<ID,3>
<OPASSIGN, >
<SEPSTR,"kevin y mark">
<SEPSCOM, >
<EOF, >

TS:

CONTENIDO DE LA TABLA #1:

---

* LEXEMA: 'texto'
* TIPO: 'TYPESTR'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'numero'
* TIPO: 'TYPEINT'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'imprime'
* TIPO: ''
* POS: '3'
* DESPLZ: '2'
* NUMPARAMS: '1'
* PARAMTYPES: '[TYPESTR]'

---

* LEXEMA: 'nada'
* TIPO: 'TYPESTR'

* POS: '4'
* DESPLZ: '3'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'pideTexto'
* TIPO: 'TYPEINT'
* POS: '5'
* DESPLZ: '4'
* NUMPARAMS: '2'
* PARAMTYPES: '[TYPEINT TYPEBOOL]'
---
* LEXEMA: 'hola'
* TIPO: 'TYPEBOOL'
* POS: '6'
* DESPLZ: '5'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'textoAux'
* TIPO: 'TYPESTR'
* POS: '7'
* DESPLZ: '6'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'mark'
* TIPO: 'TYPEINT'
* POS: '8'
* DESPLZ: '7'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'patata'
* TIPO: 'TYPEBOOL'
* POS: '9'
* DESPLZ: '8'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'kevin'
* TIPO: 'TYPEBOOL'
* POS: '10'
* DESPLZ: '9'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

```
-------------------------
CONTENIDO DE LA TABLA #2:
----------------
* LEXEMA: 'msg'
* TIPO: 'TYPESTR'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
-------------------------
* LEXEMA: 'pepe'
* TIPO: 'TYPESTR'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
-------------------------
CONTENIDO DE LA TABLA #3:
----------------
* LEXEMA: 'pitos'
* TIPO: 'TYPEINT'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
-------------------------
* LEXEMA: 'flautas'
* TIPO: 'TYPEBOOL'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
-------------------------
* LEXEMA: 'alfonso'
* TIPO: 'TYPEINT'
* POS: '3'
* DESPLZ: '2'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
-------------------------
* LEXEMA: 'alex'
* TIPO: 'TYPESTR'
* POS: '4'
* DESPLZ: '3'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
```

Parse:

Descendente 1 5 7 1 5 7 1 5 6 2 9 11 13 7 14 16 5 7 17 23 32 35 38 41 44 49 45 42 39 36 33 17 23 32 35 38 41 44 47 52 45 42 39 36 33 18 1 5 7 2 9 10 6 13 6 15 8 14 16 5 6 17 19 32 35 38 41 44 47 52 46 47 52 45 42 40 41 44 48 45 42 40 41 44 48 45 42 39 36 33 16 5 7 17 19 32 35 38 41 44 47 52 46 49 45 42 39 36 33 17 23 32 35 38 41 44 49 45 42 39 36 33 17 22 32 35 38 41 44 47 52 45 42 39 36 33 17 19 32 35 38 41 44 47 52 46 49 45 42 39 36 33 17 24 32 35 38 41 44 47 52 45 42 39 36 33 18 1 5 8 3 19 32 35 38 41 44 47 51 26 28 27 45 42 39 36 33 1 5 7 3 19 32 35 38 41 44 47 52 46 47 52 45 42 39 36 33 3 19 32 35 38 41 44 47 51 26 27 45 42 39 36 33 1 5 6 3 24 29 3 22 32 35 38 41 44 47 52 45 42 39 36 33 1 5 8 3 20 32 35 38 41 44 47 52 45 42 39 36 34 35 38 41 44 47 52 45 42 39 37 38 41 44 48 45 42 39 36 33 21 17 23 32 35 38 41 44 49 45 42 39 36 33 18 3 19 32 35 38 41 44 47 52 45 43 44 48 45 42 40 41 44 47 52 45 42 39 36 33 1 5 8 3 19 32 35 38 41 44 47 52 46 49 45 42 39 36 33 4

Prueba 5

```
   var int num1;
   var boolean bool1;
   var string str1;

   function string saludo(int hola) {
      return "adios";
   }

   function int funcion2 (boolean uno) {
      return 2;
   }

   saludo(num1);
funcion2(bool1);
```

Tokens:

```
   <DECLVAR, >
   <TYPEINT, >
   <ID,0>
   <SEPSCOM, >
   <DECLVAR, >
   <TYPEBOOL, >
   <ID,1>
   <SEPSCOM, >
   <DECLVAR, >
   <TYPESTR, >
   <ID,2>
   <SEPSCOM, >
   <DECLFUNC, >
```

<TYPESTR, >
<ID,3>
<SEPLPAR, >
<TYPEINT, >
<ID,0>
<SEPRPAR, >
<SEPLCURL, >
<RET, >
<SEPSTR,"adios">
<SEPSCOM, >
<SEPRCURL, >
<DECLFUNC, >
<TYPEINT, >
<ID,4>
<SEPLPAR, >
<TYPEBOOL, >
<ID,0>
<SEPRPAR, >
<SEPLCURL, >
<RET, >
<VALINT,2>
<SEPSCOM, >
<SEPRCURL, >
<ID,3>
<SEPLPAR, >
<ID,0>
<SEPRPAR, >
<SEPSCOM, >
<ID,4>
<SEPLPAR, >
<ID,1>
<SEPRPAR, >
<SEPSCOM, >
<EOF, >

TS:

CONTENIDO DE LA TABLA #1:

---

* LEXEMA: 'num1'
* TIPO: 'TYPEINT'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

---

* LEXEMA: 'bool1'

* TIPO: 'TYPEBOOL'
* POS: '2'
* DESPLZ: '1'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'str1'
* TIPO: 'TYPESTR'
* POS: '3'
* DESPLZ: '2'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'
---
* LEXEMA: 'saludo'
* TIPO: 'TYPESTR'
* POS: '4'
* DESPLZ: '3'
* NUMPARAMS: '1'
* PARAMTYPES: '[TYPEINT]'
---
* LEXEMA: 'funcion2'
* TIPO: 'TYPEINT'
* POS: '5'
* DESPLZ: '4'
* NUMPARAMS: '1'
* PARAMTYPES: '[TYPEBOOL]'

CONTENIDO DE LA TABLA #2:
---
* LEXEMA: 'hola'
* TIPO: 'TYPEINT'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

CONTENIDO DE LA TABLA #3:
---
* LEXEMA: 'uno'
* TIPO: 'TYPEBOOL'
* POS: '1'
* DESPLZ: '0'
* NUMPARAMS: '0'
* PARAMTYPES: '[]'

Parse:

Descendente 1 5 6 1 5 8 1 5 7 2 9 10 7 13 6 14 17 24 32 35 38 41 44 49 45 42 39 36 33 18 2 9 10 6 13 8 14 17 24 32 35 38 41 44 48 45 42 39 36 33 18 3 19 32 35 38 41 44 47 51 26 27 45 42 39 36 33 3 19 32 35 38 41 44 47 51 26 27 45 42 39 36 33 4

Pasamos a los casos con errores:

Prueba 6

```
var string texto;
function imprime (string msg)
{
        print ("Mensage introducido:");
        print (msg);
}
function int pideTexto ()
{
        print ("Introduce un texto");
        input (texto);
}
pideTexto();
var string textoAux;
textoAux = texto;
imprime (textoAux);
```

ERROR en L12: falta el return de la función

Prueba 7:

```
var int a;
var int b;
a = 3;
b = a;
var boolean c;
c = a < b;
c = k;
if (c) {b = 1;}
if (b < a){ b /= 4;}
a = a / b;
print (a);
print (b);
```

ERROR en L7: no existe el id “k”

Prueba 8

```
var int a;
var int b;
var int c;
var string d;
print ("Introduce el primer operando");
input (a);
print ("Introduce el segundo operando");
input (b);
function int divide (int num1, int num2)
{
	return num1/num2;
}
d = a;
c = divide (a, b);
print (c);
```

ERROR en L13: Error en los tipos

Prueba 9:

```
var string texto;
var string texto;
var int numero;

function imprime (string msg)
{
	var string pepe;
	print ("Mensage introducido:");
	print (msg);
}

var string nada;

function int pideTexto (int pitos, boolean flautas)
{
	var int alfonso;
	alfonso = alfonso / 4 / 6 ;
	var string alex;
	alex = "soy alex";

	print ("Introduce un texto");
	input (texto);

	nada = "eres nada";

	return numero;
```

```
}

var boolean hola;

pideTexto(numero, nada);
var string textoAux;
textoAux = texto;
imprime (textoAux);

var int mark;

return;

input(texto);
var boolean patata;
if ( mark || numero < 3 ) {
        print("adios");
}

numero /= 385 / mark;

var boolean kevin;

nada = "kevin y mark";

/*menudo comentario*/
```

ERROR en L31: Error en los tipos de parametros

Prueba 10

```
var int num1;
var boolean bool1;
var string str1;
function string saludo(int hola) {
   return "adios";
}
function int funcion2 (boolean uno) {
   return 2;
}
saludo(num1);
funcion2(bool1);
return 800;
```

ERROR en L12: return fuera de funcion