Proyecto PDL

JavaScript PL

GRUPO 44

Chica Manjarrez, Sergio
Querol Cruz, Anaïs
Sebastián González, Silvia

Memoria

1. Diseño

El procesador de la práctica se ha realizado en Java, se ha escogido debido a ser un POO el cual facilita muchas tareas, como por ejemplo la TS. El procesador final es un ejecutable .jar el cual se ejecuta a través de la línea de comandos. Una vez iniciado el ejecutable pide un fichero y al procesarlo generará cuatro ficheros, Tokens.txt , Parser.txt , TS.txt y Errores.txt.

1.1. Analizador léxico

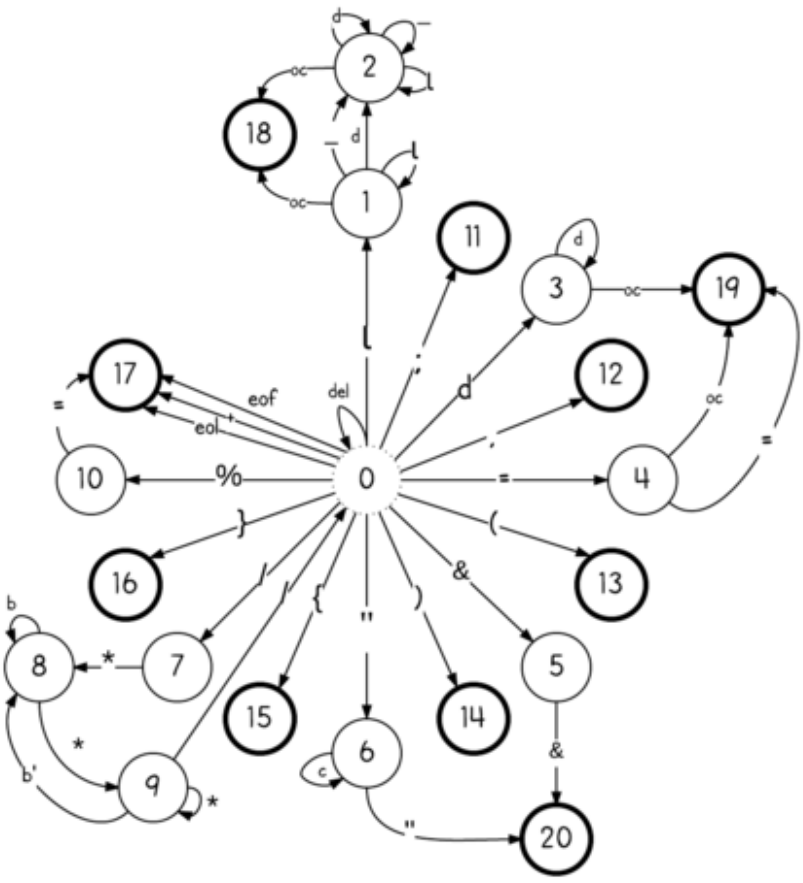
La principal tarea para el desarrollo del analizador sintáctico ha sido analizar los tokens que deseábamos incluir y su gramática. Concluimos no añadir tokens opcionales y ceñirnos a lo pedido en la práctica. De esta manera los tokens resultantes fueron:

<eol, >	<Coma,,>	<PtComa,;>	<OpRelEq,==>	<OpAsMod,%=>	<Num,valor>
<eof, >	<ParA,(>	<LlavA,{>	<OpLogAnd,&&>	<OpArSum,+>	<Cadena,sec>
<OpAs,=>	<ParC,)>	<LlavC,}>	<PalRes,pos>	<Id,posTS>	<Boolean,0:1>

A continuación se presenta la gramática empleada con su correspondiente autómata:

```
S -> ; | , | ( | )
      | { | } | +
      | eof | del S
      | l A | d D
      | = E | & F
      | " C | / B
      | % M
A -> l A | d A'
      | _ A' | λ
A' -> l A' | d A'
      | _ A' | λ
D -> d D | λ
E -> = | λ
F -> &
C -> c C | "
B -> * B'
B' -> b B' | * B"
B" -> / S | b' B' | *
B"
M -> =

del : espacio, tab,
eol
d : {0..9}
c : caracteres - {"}
b : caracteres - {*}
b': caracteres - {*/}
```



```
0:1
  if caracterSig == l
        lex = l
  else
        Error("Símb.No Rec.")

1:1
  if caracterSig == l
        lex = lex ⊕ l
  else
        Error("Símb.No Rec.")

1:2
  if caracterSig == d | '_'
        lex = lex ⊕ (d|_)
  else
        Error("Símb.No Rec.")

1:18
  if lex == buscaPS(lex)
/*Busca el lexema en una tabla de palabras reservadas, si la encuentra la devolverá*/
        GT(PalRes, lex )
  else if lex == "true"
       GT(Boolean, true )
  else if lex == "false"
       GT(Boolean, false )
  else
        GT (Id, lex )
```

```
0:11
  if caracterSig == ';'
        GT(PtComa, ; )
  else
        Error("Símb.No Rec.")

0:3
  if caracterSig == d
        valor = valorCaracter("d")
  else
        Error("Símb.No Rec.")

3:3
  if caracterSig == d
        valor = valor*10 +
valorCaracter("d")
  else
        Error("Símb.No Rec.")

3:19
  GT(Num, valor)

0:12
  if caracterSig == ','
        GT(Coma, , )
  else
        Error("Símb.No Rec.")
```

```
0:4
  if caracterSig == '='
        //Nothing
  else
        Error("Símb.No Rec.")

4:19
  if caracterSig == '='
        GT(OpRelacIgual, == )
  else
        GT(OpAsig, = )

0:13
  if caracterSig == '('
        GT(ParAb, ( )
  else
        Error("Símb.No Rec.")

0:5
  if caracterSig == '&'
        //Nothing
  else
        Error("Símb.No Rec.")

5:20
  if caracterSig == '&'
        GT(OpLogAnd, && )
  else
        Error("Símb.No Rec.")
```

```
0:14
  if caracterSig == ')'
        GT(ParCer, ) )
  else
        Error("Símb.No Rec.")

0:6
  if caracterSig == '"'
        lex = '"'
  else
        Error("Símb.No Rec.")

6:6
  if caracterSig == c
        lex = lex ⊕ c
  else
        Error("Símb.No Rec.")
```

```
7:8
  if caracterSig == '*'
        //Nothing
  else
        Error("Símb.No Rec.")

8:8
  if caracterSig == b
        //Nothing
  else
        Error("Símb.No Rec.")

8:9
  if caracterSig == '*'
        //Nothing
  else
        Error("Símb.No Rec.")
```

```
0:16
  if caracterSig == '}'
        GT(LlavCer, } )
  else
        Error("Símb.No Rec.")

0:10
  if caracterSig == '%'
        //Nothing
  else
        Error("Símb.No Rec.")

10:17
  if caracterSig == '='
        GT(OpAsigMod, %= )
  else
        Error("Símb.No Rec.")
```

```
6:20
  if caracterSig == '"'
        GT(Cadena, lex )
  else
        Error("Símb.No Rec.")

0:15
  if caracterSig == '{'
        GT(LlavAb, { )
  else
        Error("Símb.No Rec.")

0:7
  if caracterSig == '/'
        //Nothing
  else
        Error("Símb.No Rec.")
```

```
9:8
  if caracterSig == b'
        //Nothing
  else
        Error("Símb.No Rec.")

9:9
  if caracterSig == '*'
        //Nothing
  else
        Error("Símb.No Rec.")

9:0
  if caracterSig == '/'
        //Nothing
  else
        Error("Símb.No Rec.")
```

```
0:17
  if caracterSig == eof
        GT(eof, )
  else if caracterSig == eol
              GT(eol, )
  else if caracterSig == '+'
              GT(OpAritSum, + )
  else
        Error("Símb.No Rec.")
```

En lo que respecta a las acciones semánticas, debido al uso de la herramienta jflex para la generación del analizador léxico, estas acciones semánticas han sido reemplazadas por expresiones regulares, las cuales cumplen las mismas funciones.

Las expresiones regulares descritas en el fichero a partir del cual JFlex crea el Analizador Léxico (.flex), describen los pasos que debe seguir éste a la hora de reconocer los tokens.
Primeramente se han definido las propiedades que deben cumplir los identificadores, los cuales empiezan por letra y después pueden tener caracteres alfanuméricos o barra baja. Los números se han definido como caracteres numéricos.
Luego se ha creado una regla para cada símbolo que reconoce nuestra gramática, en este caso tenemos:

- Operador Aritmético suma: ‘+’
- Operador Asignación simple y asignación con módulo : ‘=’ , ‘%=’
- Operador Lógico AND: ‘&&’
- Operador Relacional, igualdad: ‘==’
- Paréntesis y llaves de apertura y cierre: ‘(‘, ‘)’, ‘{‘, ‘}’
- Coma y punto y coma: ‘,’ , ‘;’

También se han definido las reglas que deben tener los comentarios, los cuales empiezan por ‘/*’ y terminan por ‘*/’, cabe resaltar que aunque existen reglas para detectar comentarios, no existen reglas para generar tokens a partir de éstos, simplemente se ignoran.
Finalmente, las cadenas son reconocidas como cualquier conjunto de caracteres entre comillas, ‘ “ ‘ .

JFlex se encarga de generar los tokens a partir de dichas reglas, las palabras reservadas son diferenciadas de los identificadores por el orden de precedencia, las reglas que están definidas en primer lugar son las de las palabras reservadas y después de éstas, los identificadores.

Los errores que detecta este analizador se producen cuando se introduce un símbolo que no se ha implementado en la gramática (como sucede, por ejemplo, con la resta o el “OR” lógico). El error que se muestra por consola en este caso es:

"ERROR LÉXICO (línea): No se reconoce el símbolo en la gramática, después del símbolo (s)"

1.2. Analizador sintáctico

Este analizador representa la estructura principal del programa: coordina a los otros dos analizadores, solicitando al léxico que le envíe un token nuevo y propiciando que el semántico realice en el orden adecuado las inserciones en la tabla de símbolos, las equiparaciones de tipos, la coherencia de expresiones… para no generar compilaciones incorrectas.

Esta guía para lograr coordinarlos se consigue gracias a una gramática que, en nuestro caso, ha sido descendente recursiva:

```
P' -> P
P -> eol P | B eol P | F eol P | eof
Z -> eol Z | λ
B -> var T id | if ( E ) S | S | for ( I ; E ; A ) eol Z { eol Z C }
S -> id S' | return X | write ( E ) | prompt ( id )
S'-> = E | %= E | ( L )
T -> int | boolean | char
T'-> T | λ
X -> E | λ
L -> E L' | λ
L'-> , E L' | λ
F -> function T' id ( G ) eol Z { eol Z C }
C -> B eol Z C | λ
G -> T id G' | λ
G'-> , T id G' | λ
I -> id = E | λ
A -> id A' | λ
A' -> = E | %= E
E -> R E'
E' -> && R E' | λ
R -> U R'
R' -> == U | λ
U -> V U'
U' -> + V U' | λ
V -> id V'| ( E ) | numero | cadena | true | false
V' -> ( L ) | λ
```

Para comprobar si la gramática anteriormente mostrada es correcta, se ha procedido a analizar todos los first y follow para demostrar que se trata de una gramática reconocedora de un lenguaje LL(1). Dado que en la tabla no hay casillas con varias producciones, se demuestra que se sabe la regla a ejecutar en cada momento; es decir, es determinista.

	FIRST	FOLLOW
P’	eol eof var if for id return write prompt function	$
P	eol eof var if for id return write prompt function	$
Z	eol λ	var if for id return write prompt { }
B	var if for id return write prompt	eol
S	id return write prompt	eol
S’	= %= (	eol
T	int boolean char	id
T’	λ int boolean char	id
X	λ id (numero cadena true false	eol
L	λ id (numero cadena true false	)
L’	, λ	)
F	function	eol
C	λ var if for id return write prompt	}
G	λ int boolean char	)
G’	, λ	)
I	id λ	;
A	id λ	)
A’	= %=	)
E	id (num cadena true false	) , ; eol
E’	&& λ	) , ; eol
R	id (num cadena true false	&&) , ; eol
R’	== λ	&&) , ; eol
U	id (num cadena true false	== &&) , ; eol
U’	+ λ	== &&) , ; eol
V	id (num cadena true false	+ == &&) , ; eol
V’	(λ	+ == &&) , ; eol

	eol	eof	var	id	if	(	)	for
P'	*P'* → *P*	*P'* → *P*	*P'* → *P*	*P'* → *P*	*P'* → *P*			*P'* → *P*
P	*P* → **eol** *P*	*P* → **eof**	*P* → *B* **eol** *P*	*P* → *B* **eol** *P*	*P* → *B* **eol** *P*			*P* → *B* **eol** *P*
Z	*Z* → **eol** *Z*		*Z* → ε	*Z* → ε	*Z* → ε			*Z* → ε
B			*B* → **var** *T* **id**	*B* → *S*	*B* → **if** **(** *E* **)** *S*			*B* → **for** **(** *I* **;** *E* **;** *A* **)** **eol** *Z* **{** **eol** *Z C* **}**
S				*S* → **id** *S'*				
S'						*S'* → **(** *L* **)**		
T								
T'				*T'* → ε				
X	*X* → ε			*X* → *E*		*X* → *E*		
L				*L* → *E L'*		*L* → *E L'*	*L* → ε	
L'							*L'* → ε	
F								
C			*C* → *B* **eol** *Z C*	*C* → *B* **eol** *Z C*	*C* → *B* **eol** *Z C*			*C* → *B* **eol** *Z C*
G							*G* → ε	
G'							*G'* → ε	
I				*I* → **id** **=** *E*				
A				*A* → **id** *A'*			*A* → ε	
A'								
E				*E* → *R E'*		*E* → *R E'*		
E'	*E'* → ε						*E'* → ε	
R				*R* → *U R'*		*R* → *U R'*		
R'	*R'* → ε						*R'* → ε	
U				*U* → *V U'*		*U* → *V U'*		
U'	*U'* → ε						*U'* → ε	
V				*V* → **id** *V'*		*V* → **(** *E* **)**		
V'	*V'* → ε					*V'* → **(** *L* **)**	*V'* → ε	

	;	{	}	return	write	prompt	=	%=
P'				$P' \rightarrow P$	$P' \rightarrow P$	$P' \rightarrow P$		
P				$P \rightarrow B$ **eol** P	$P \rightarrow B$ **eol** P	$P \rightarrow B$ **eol** P		
Z		$Z \rightarrow \varepsilon$	$Z \rightarrow \varepsilon$	$Z \rightarrow \varepsilon$	$Z \rightarrow \varepsilon$	$Z \rightarrow \varepsilon$		
B				$B \rightarrow S$	$B \rightarrow S$	$B \rightarrow S$		
S				$S \rightarrow$ **return** X	$S \rightarrow$ **write** **(** E **)**	$S \rightarrow$ **prompt** **(** **id** **)**		
S'							$S' \rightarrow$ **=** E	$S' \rightarrow$ **%=** E
T								
T'								
X								
L								
L'								
F								
C			$C \rightarrow \varepsilon$	$C \rightarrow B$ **eol** $Z\,C$	$C \rightarrow B$ **eol** $Z\,C$	$C \rightarrow B$ **eol** $Z\,C$		
G								
G'								
I	$I \rightarrow \varepsilon$							
A								
A'							$A' \rightarrow$ **=** E	$A' \rightarrow$ **%=** E
E								
E'	$E' \rightarrow \varepsilon$							
R								
R'	$R' \rightarrow \varepsilon$							
U								
U'	$U' \rightarrow \varepsilon$							
V								
V'	$V' \rightarrow \varepsilon$							

	int	boolean	char	,	function	&&	==
P'					$P' \rightarrow P$		
P					$P \rightarrow F$ **eol** P		
Z							
B							
S							
S'							
T	$T \rightarrow$ **int**	$T \rightarrow$ **boolean**	$T \rightarrow$ **char**				
T'	$T' \rightarrow T$	$T' \rightarrow T$	$T' \rightarrow T$				
X							
L							
L'				$L' \rightarrow$, E L'			
F					$F \rightarrow$ **function** T' **id** (G) **eol** Z { **eol** Z C }		
C							
G	$G \rightarrow T$ **id** G'	$G \rightarrow T$ **id** G'	$G \rightarrow T$ **id** G'				
G'				$G' \rightarrow$, T **id** G'			
I							
A							
A'							
E							
E'				$E' \rightarrow \varepsilon$		$E' \rightarrow$ **&&** R E'	
R							
R'				$R' \rightarrow \varepsilon$		$R' \rightarrow \varepsilon$	$R' \rightarrow$ **==** U R'
U							
U'				$U' \rightarrow \varepsilon$		$U' \rightarrow \varepsilon$	$U' \rightarrow \varepsilon$
V							
V'				$V' \rightarrow \varepsilon$		$V' \rightarrow \varepsilon$	$V' \rightarrow \varepsilon$

	+	numero	cadena	true	false	$
P'						
P						
Z						
B						
S						
S'						
T						
T'						
X		$X \rightarrow E$	$X \rightarrow E$	$X \rightarrow E$	$X \rightarrow E$	
L		$L \rightarrow E\,L'$	$L \rightarrow E\,L'$	$L \rightarrow E\,L'$	$L \rightarrow E\,L'$	
L'						
F						
C						
G						
G'						
I						
A						
A'						
E		$E \rightarrow R\,E'$	$E \rightarrow R\,E'$	$E \rightarrow R\,E'$	$E \rightarrow R\,E'$	
E'						
R		$R \rightarrow U\,R'$	$R \rightarrow U\,R'$	$R \rightarrow U\,R'$	$R \rightarrow U\,R'$	
R'						
U		$U \rightarrow V\,U'$	$U \rightarrow V\,U'$	$U \rightarrow V\,U'$	$U \rightarrow V\,U'$	
U'	$U' \rightarrow + V\,U'$					
V		$V \rightarrow$ **numero**	$V \rightarrow$ **cadena**	$V \rightarrow$ **true**	$V \rightarrow$ **false**	
V'	$V' \rightarrow \varepsilon$					

A continuación se muestran todos los procedimientos correspondientes al analizador sintáctico recursivo descendente. Para ello se han utilizado unas funciones auxiliares llamadas **st=*siguienteToken()***, que llama al analizador léxico para que le entregue el siguiente token del fichero, y **compToken=*comprobarToken()***, que comprueba si el siguiente token es el esperado, en cuyo caso avanza el puntero.

```
procedure Pp
begin
   if ( st==eol | st==eof
      | st==var | st==if | st==for
      | st==id | st==return
      | st==write | st==prompt
      | st==function )
         P
   else
         error()
end

procedure B
begin
   if ( st==var )
         compToken(var)
         T
         compToken(id)
   else if ( st==if )
         compToken(if)
         compToken( ( )
         E
         comptoken( ) )
         S
   else if ( st==id | st==return
      | st==write | st==prompt )
         S
   else if ( st==for )
         compToken(for)
         compToken( ( )
         I
         compToken(;)
         E
         compToken(;)
         A
         compToken( ) )
         compToken(eol)
         Z
         compToken( { )
         compToken(eol)
         Z
         C
         compToken( } )
   else
         error()
end

procedure Z
begin
   if ( st==eol )
         compToken(eol)
         Z
   else if ( st==var | st==if
      | st==for | st==id | st=={
      | st==} | st==write
      | st==prompt | st==return )
         /*Nada*/
   else
         error()
end
```

```
procedure P
begin
   if ( st==var | st==if | st==id
      | st==return | st==write
      | st==prompt  )
         B
         compToken(eol)
         Z
         P
   else if ( st==function )
         F
         compToken(eol)
         P
   else if ( st==eol )
         compToken(eol)
         P
   else if ( st==eof )
         compToken(eof)
   else
         error()
end

procedure S
begin
   if ( st==id )
         compToken(id)
         Sp
   else if ( st==return )
         compToken(return)
         X
   else if ( st==write )
         compToken(write)
         compToken( ( )
         E
         compToken( ) )
   ele if ( st==prompt )
         compToken(prompt)
         compToken( ( )
         compToken(id)
         compToken( ) )
   else
         error()
end

procedure Sp
begin
   if ( st== = )
         compToken(=)
         E
   else if ( st== %= )
         compToken(%=)
         E
   else if ( st==( )
         compToken( ( )
         L
         compToken( ) )
   else
         error()
end
```

```
procedure T
begin
   if ( st==int )
        compToken(int)
   else if ( st==boolean )
        compToken(boolean)
   else if ( st==char )
        compToken(char)
   else
        error()
end

procedure X
begin
   if ( st==id | st==( | st==num
      | st==true | st==false
      | sp==cadena )
        E
   else if ( st==eol )
        /*Nada*/
   else
        error()
end

procedure L
begin
   if ( st==id | st==( | st==num
      | st==true | st==false
      | sp==cadena)
        E
        Lp
   else if ( st==) )
        /*Nada*/
   else
        error()
end

procedure Lp
begin
   if ( st==, )
        compToken(,)
        E
        Lp
   else if ( st==) )
        /*Nada*/
   else
        error()
end

procedure G
begin
   if ( st==int | st==boolean
      | st==char )
        T
        compToken(id)
        Gp
   else if ( st==) )
        /*Nada*/
   else
        error()
end
```

```
procedure Tp
begin
   if ( st==int | st==boolean
      | st==char )
        T
   else if ( st==id )
        /*Nada*/
   else
        error()
end

procedure F
begin
   if ( st==function )
        compToken(function)
        Tp
        compToken(id)
        compToken( ( )
        G
        compToken( ) )
        compToken(eol)
        Z
        compToken( { )
        compToken(eol)
        Z
        C
        compToken( } )
   else
        error()
end

procedure C
begin
   if ( st==var | st==if | st==for
      | st==id | st==return
      | st==write | st==prompt )
        B
        compToken(eol)
        Z
        C
   else if ( st==} )
        /*Nada*/
   else
        error()
end

procedure Gp
begin
   if ( st==, )
        compToken(,)
        T
        compToken(id)
        Gp
   else if ( st==) )
        /*Nada*/
   else
        error()
end
```

```
procedure I
begin
   if ( st==id )
        compToken(id)
        compToken(=)
        E
   else if ( st==; )
        /*Nada*/
   else
        error()
end

procedure A
begin
   if ( st==id )
        compToken(id)
        Ap
   else if ( st==) )
        /*Nada*/
   else
        error()
end

procedure Ap
begin
   if ( st== = )
        compToken(=)
        E
   else if ( st== %= )
        compToken(%=)
        E
   else
        error()
end

procedure E
begin
   if ( st==id | st==( | st==num
      | st==true | st==false
      | st==cadena )
        R
        Ep
   else
        error()
end

procedure Ep
begin
   if ( st==&& )
        compToken(&&)
        R
        Ep
   else if ( st==) | st==, | st==;
      | st==eol )
        /*Nada*/
   else
        error()
end

procedure R
begin
   if ( st==id | st==( | st==num
      | st==true | st==false
      | st==cadena )
        U
        Rp
   else
        error()
end

procedure Rp
begin
   if ( st== == )
        compToken(==)
        U
   else if ( st==&& | st==)
      | st==, | st==; | st==eol )
        /*Nada*/
   else
        error()
end

procedure U
begin
   if ( st==id | st==( | st==num
      | st==true | st==false
      | st==cadena )
        V
        Up
   else
        error()
end

procedure Up
begin
   if ( st==+ )
        compToken(+)
        V
        Up
   else if ( st== == | st==&&
      | st==) | st==, | st==;
      | st==eol )
        /*Nada*/
   else
        error()
end

procedure Vp
begin
   if ( st==( )
        compToken( ( )
        L
        compToken( ) )
   else if ( st==+ | st== ==
      | st==&& | st==) | st==,
      | st==; | st==eol )
        /*Nada*/
   else
        error()
end

procedure V
begin
   if ( st==id )
        compToken(id)
        Vp
   else if ( st==( )
        compToken( ( )
        E
        compToken( ) )
   else if ( st==num )
        compToken(num)
   else if ( st==cadena )
        compToken(cadena)
   else if ( st==true )
        compToken(true)
   else if ( st==false )
        compToken(false)
   else
        error()
end
```

1.3. Analizador semántico

A continuación se muestra el esquema de traducción (EdT) que recoge las acciones semánticas del analizador semántico, así como el lanzamiento de errores de tipo semántico *Error()*. En aquellas donde no se especifica el contenido de algunos campos, se consideran vacíos *null*.

```
P' -> { crearTS(despl=0) = TSG } P
P -> eol P
P -> B eol P
P -> F { P.tipo := if (F.tipo != null) err & Error() } eol P
P -> eof { generarTS(); borrarTS()}

Z -> eol Z
Z -> λ

B -> var T id {B.tipo:= if (BuscaTSL(TS,id.ent)!=null) err & Error()
                 else AñadirTS(TS,id.ent,addDespl(),T.tipo, T.tam)}
B -> if ( E )S {crearTS(), B.tipo=if(E.tipo==boolean & S.tipo!=err)
                 then S.tipo & B.ret=S.ret else err & Error()}
B -> S { B.tipo := S.tipo, B.ret=S.ret }
B -> for ( I ; E ; A ) eol Z { eol Z C } { if (I.tipo==err |
     E.tipo!=boolean | A.tipo==err) Error() else B.ret=C.ret }

S -> id S'{ if (BuscaTS(id.ent)==null) then
                 AñadeTG(id.ent, addDesp(), int, 2 )
            if ( S'.tipo==fun & BuscaTipoTS(id.ent)!=Sp.tipo )
               S.tipo = err & Error()
            else if ( S'.tipo==fun & BuscaTipoTS(id.ent)!=S'.tipo )
               S.tipo = err & Error()
            else if ( BuscaTipoTS(id.ent)==fun &
                      BuscaParamTS(id.ent)!=S'.param )
               S.tipo = err & Error()
            else S.tipo = ok
           }
S -> return X { S.tipo = ok, S.ret = X.ret }
S -> write ( E ) { S.tipo = ok }
S -> prompt ( id ){if (buscaTS(id)== null)
                 AñadeTG(id.ent, addDespl(), int, 2); S.tipo = ok }

S' -> = E { S'.tipo = E.tipo }
S' -> %= E { S'.tipo = if ( E.tipo!=int ) E.tipo & Error() else int}
S' -> ( L ) { S'.tipo = fun , S'.param = L.param }

T -> int { T.tipo = int, T.tam = 2 }
T -> boolean { T.tipo = boolean, T.tam = 1 }
```

```
T -> char { T.tipo = char, T.tam = 4 }
T' -> T { T'.tam = T.tam, T'.ret = T.tipo }
T' -> λ

X -> E { X.tam=E.tam, X.ret=E.tipo }
X -> λ

L -> E L' { L.param = E.tipo + L'.param }
L -> λ
L' -> , E L1' { L'.param = E.tipo + L1'.param }
L' -> λ

F -> function T' id ( G ) eol Z { eol Z C }
      { if (BuscaTS(id)!=null) F.tipo = err & Error() ; crearTS();
        F.tipo = if (C.tipo == T'.tipo == ok) ok else err
      }

C -> B eol Z C1 { if ( B.ret!=null & C1.ret!=null )
                      if (B.ret!=C1.ret) C.tipo = err & Error()
                      else C.ret = B.ret
                  else if ( B.ret!=null & C1.ret==null )
                      C.ret = B.ret
                  else if ( B.ret==null & C1.ret!=null )
                      C.ret = C1.ret }
C -> λ

G -> T id G' { AñadeTL(id.ent,addDesp(),T.tipo,T.tam),
               G.param = T.tipo + G'.param }
G -> λ
G' -> , T id G1' { AñadeTL(id.ent,addDesp(),T.tipo,T.tam),
                   G'.param = T.tipo + G1'.param }
G' -> λ

I -> id = E { if (BuscaTipoTS(id.ent) == null)
                AñadeTG(id.ent, addDespl(), int, 2)
              I.tipo = if (BuscaTipoTS(id.ent)!= E.Tipo}
                      err & Error() else E.tipo & I.nom=id }

I -> λ { I.tipo = ok }

A -> id A'{ if (BuscaTS(id.ent)==null)
                AñadeTG(id.ent,addDespl(),int,2)
            if ( BuscaTipoTS(id.ent)!=A'.tipo ) A.tipo=err & Error()
            else A.tipo=ok
A -> λ { A.tipo = ok }
```

```
A' -> = E {A'.tipo = E.tipo}
A' -> %= E { A'.tipo = if (E.tipo!=int) E.tipo & Error() else int }
E -> R E' { if (E'.tipo==null) E.tipo=R.tipo
           else if ( R.tipo!=E'.tipo & E'.tipo!=err & R.tipo!=err )
                E.tipo=err & Error()
           else if ( R.tipo!=E'.tipo &
                     (E'.tipo==boolean | R.tipo==boolean ))
                E.tipo=err & Error()
           else E.tipo=boolean }
E' -> && R E1' { E'.tipo= if ((E1.tipo!=null & E1.tipo!=boolean) |
                R.tipo!=boolean ) err && Error() else boolean }
E' -> λ
R -> U R'{ if (R'.tipo==null) R.tipo=U.tipo
           else if ( U.tipo!=R'.tipo) R.tipo=err && Error()
           else R.tipo=boolean }

R' -> == U { R'.tipo = U.tipo }
R' -> λ
U -> V U'{ U.tipo = if (U'.tipo!=null & V.tipo!=U'.tipo)
                  then err & Error() else V.tipo }
U' -> + V U' { U'.tipo = if (U1'.tipo!=null & V.tipo!=U1'.tipo)
                           then err & Error() else V.tipo }
U' -> λ

V -> id V'{ if BuscaTS(id.ent)!=null & BuscaTipoTS(id.ent)==fun
              if ( paramTS(id.ent)!=V'.param )
                   V.tipo=BuscaRetTS(id.ent) & Error()
              else V.tipo=BuscaRetTS(id.ent)
            else if ( BuscaTS(id.ent)!=null &
                 BuscaTipoTS(id.ent)!=fun )
              if ( V'.param!=null ) V.tipo=err & Error()
              else V.tipo=BuscaTipoTS(id.ent)
            else if ( BuscaTS(id.ent)==null )
              AñadeTG(id.ent, addDespl(), int, 2)
              if ( V'.param!=null ) V.tipo=err & Error()
              else V.tipo=int
           }
V -> ( E ){ V.tipo=E.tipo }
V -> entero { V.tipo= if (entero>32767) err & Error() else int }
V -> cadena { V.tipo=char }
V -> true { V.tipo=boolean }
V -> false { V.tipo=boolean }

V' -> ( L ){ V'.param = L.param }
V' -> λ
```

1.4. Tabla de símbolos

Para una mayor facilidad en la implementación, se ha decidido usar la **estructura de ArrayList**. En una variable de ese tipo llamada "tablas" se incluyen todas las tablas de símbolos que genera el programa. La elección de esta estructura nos ha resultado idónea ya que permite tenerlas ordenadas en función de su aparición. De esta manera, cuando se quiera imprimir el contenido de todas las tablas, se puede realizar una indexación sencilla. Dentro de cada tabla de símbolos, se incluyen entradas.

Las **entradas** son objetos de la clase Entrada que permite almacenar los diferentes atributos que puede contener cualquier token asociado (nombre, tipo, desplazamiento…). Se debe resaltar que existen atributos que no se aplican en algunos identificadores (por ejemplo, el tipo de los parámetros de entrada sólo se mostrará en las funciones), por lo que almacenará "null" en dicho campo.

La primera entrada de todas las tablas de símbolos, representa un **puntero a la tabla precedente**, ya que en el atributo "padre" se incluye la posición del ámbito inmediatamente superior en la variable "tablas". De esta manera, cada vez que se realice una búsqueda, podrá acceder a los elementos no sólo de su tabla local, sino que también accederá a través de ese puntero a las tablas de símbolos correspondientes a tablas que engloban al ámbito en el que nos encontramos.

Por tanto, la clase Entrada genera objetos con los siguientes atributos:

- **despl**: almacena los desplazamientos de las entradas en la tabla de símbolos. Como las funciones no generan un desplazamiento se copia el desplazamiento de la entrada anterior.
- **nom**: muestra el nombre de los identificadores para que la búsqueda en la tabla de símbolos pueda realizarse a partir de este nombre.
- **tipo**: almacena el tipo de las entradas para realizar las comparaciones pertinentes posteriormente.
- **tam**: se corresponde con el atributo tamaño, que permitirá saber cuánto ocupan los identificadores para poder conocer el desplazamiento de la entrada siguiente. En el caso de las funciones el tamaño introducido será igual a 0.
- **param**: únicamente se usará en el caso de que el elemento sea una función. Es un String que almacena de manera ordenada los tipos que recibe la función. En caso de no ser una función, este atributo valdrá "null".
- **ret**: como en el caso anterior, es un atributo específico de las funciones. Permite conocer el tipo de retorno que resulta tras llamar a la función. En caso de no ser una función, este atributo resultará "null".
- **padre**: únicamente tendrá valor en la primera entrada de la tabla. Se trata de un Integer que contiene el elemento que se debe indexar en el arraylist que almacena las tablas de símbolos para encontrar el ámbito desde el cual fue creada esta tabla de símbolos. El valor que toma en la tabla de símbolos global es **null** ya que es la primera tabla de todas y no ha sido creada por ninguna otra. A excepción de la primera entrada de cada tabla de símbolos, el valor de este atributo es "null".

Para saber en cada momento en qué tabla de símbolos se encuentra el programa, hemos creado una variable **tablaActua**l que contiene la posición de la tabla que contiene las entradas del

ámbito actual. Cada vez que se crea una tabla nueva, ésta se introduce al final del arraylist y "tablaActual" actualiza su valor al de esta posición. Al salir de ese ámbito, adopta el atributo padre de la primera entrada de dicha tabla para actualizar su posición.

Al finalizar la evaluación de la entrada, se procede a imprimir todas las tablas de símbolos que se habían generado mediante la llamada a la función "generarTS()". Así se consigue obtener un fichero de texto que muestra el contenido de las tablas que muestra el nombre de la tabla y sus entradas, incluyendo tan sólo el nombre, tipo y desplazamiento de cada elemento, excepto los casos de las funciones que incluyen el nombre, el tipo de retorno y los parámetros de entrada. La fórmula escogida para el estilo de esta tabla es **del$^+$ * del$^+$ LEXEMA del$^+$: del$^+$ 'nombre' del$^+$ RC** para incluir los nombres y **del$^+$ + del$^+$ atributo del$^+$: del$^+$ 'nombre' del$^+$ valor RC** para añadir los tipos, desplazamientos, parámetros o retornos dependiendo del caso.

A continuación, se muestran las funciones implementadas necesarias para el correcto funcionamiento de la Tabla de Símbolos:

- Entrada buscaTS (ArrayList <Entrada> TS, String nombre)

  Realiza la búsqueda de un elemento introduciendo su nombre en la tabla de símbolos. No busca sólo en su ámbito, sino que accede a aquellos otros en los que está contenido el estudiado. Devuelve la entrada si la ha encontrado o null en caso de no encontrarse.

- Entrada buscaTSL (ArrayList <Entrada> TS, String nombre)

  Busca una entrada a partir del lexema en una única tabla de símbolos. No accede a más ámbitos.

- void creaTS ()

  Genera una tabla de símbolos nueva y actualiza el valor de tablaActual a la posición que ocupa en "tablas".

- borraTS()

  Actualiza el valor de tablaActual a la posición de la tabla de símbolos que ha generado la tabla en la que se encuentra en ese momento.

- int addDesp(ArrayList <Entrada> TS)

  Suma desplazamiento y tamaño de la última entrada de una tabla de símbolos introducida como un parámetro para obtener el desplazamiento de la nueva entrada.

- generarTS()

  Imprime en un documento el contenido de todas las tablas de símbolos y son eliminadas. Se llama al finalizar el programa.

2. Casos de prueba

El tratamiento de los errores léxicos y sintácticos en este procesador se realiza mediante excepciones, por ello, cuando se detecta alguno de éstos el programa finaliza. Los errores del analizador semántico no es así, este permite señalizar los errores sin finalizar el programa, por lo que al final de la ejecución se obtendrán todos los errores semánticos.

A continuación se presentan 10 casos de prueba que demuestran el correcto funcionamiento del compilador. Las cinco primeras entradas son códigos correctos que analiza el sistema y permite generar una tabla de símbolos y el árbol mediante la herramienta VAST.

En cuanto a las cinco últimas, éstas muestran el correcto funcionamiento de los análisis de errores.

En una de estas pruebas se obtiene un error sintáctico, por lo tanto se ha procedido a volver a ejecutar el programa con el error corregido, de manera que podamos adjuntar el resto de errores que también tiene dicho fichero.

Si bien es cierto que se han generado los árboles de las herramientas VASt, éstos no entraban de una manera cómoda y legible en el documento que se presenta, por lo que se ha decidido incluirlos en el CD adjuntado.

2.1. Correctos

pruebas1.txt > Tokens

```
< eol ,  >
< eol ,  >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , boolean >
< Id , funcBoolean >
< ParAb , ( >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , var >
< PalRes , boolean >
< Id , dev >
```

```
< eol ,  >
< Id , dev >
< OpAsig , = >
< Boolean , false >
< eol ,  >
< PalRes , return >
< Id , funcBoolean >
< ParAb , ( >
< ParCer , ) >
< eol ,  >
< LlavCer , } >
< eol ,  >
< eol ,  >
< PalRes , var >
< PalRes , int >
```

```
< Id , variableGlobal >
< eol ,  >
< eol ,  >
< PalRes , write >
< ParAb , ( >
<  Cadena  ,  "\nIntroduce
un 'número'." >
< ParCer , ) >
< eol ,  >
< PalRes , prompt >
< ParAb , ( >
< Id , num >
< ParCer , ) >
< eol ,  >
< eof ,  >
```

pruebas1.txt > TS

```
CONTENIDO DE LA TABLA # 0:
  * LEXEMA    : 'funcBoolean'
    ATRIBUTOS :  (funcion)
    + tipo    : 'boolean'
    + param   : '' (no recibe parametros)

  * LEXEMA    : 'variableGlobal'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

  * LEXEMA    : 'num' :
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 2
-----------------------------------
```

```
CONTENIDO DE LA TABLA # 1:
  * LEXEMA    : 'dev'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 0

-----------------------------------
```

pruebas2.txt> Tokens

```
< PalRes , var >
< PalRes , int >
< Id , var1 >
< eol ,  >
< PalRes , var >
< PalRes , boolean >
< Id , var2 >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , int >
< Id , funcInt >
< ParAb , ( >
< PalRes , int >
< Id , n >
< Coma , , >
< PalRes , char >
< Id , msg >
< Coma , , >
< PalRes , boolean >
< Id , j >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
```

```
< eol ,  >
< Id , var1 >
< OpAsig , = >
< Id , funcInt >
< ParAb , ( >
< Num , 5 >
< Coma , , >
< Cadena , "prueba" >
< Coma , , >
< Id , var2 >
< ParCer , ) >
< eol ,  >
< eol ,  >
< PalRes , return >
< Id , var1 >
< eol ,  >
< LlavCer , } >
< eol ,  >
< eol ,  >
< PalRes , var >
< PalRes , char >
< Id , var3 >
< eol ,  >
< Id , var3 >
```

```
< OpAsig , = >
< Cadena , "texto" >
< eol ,  >
< PalRes , if >
< ParAb , ( >
< Id , funcInt >
< ParAb , ( >
< Num , 1 >
< Coma , , >
< Cadena , "" >
< Coma , , >
< Id , var2 >
< ParCer , ) >
< OpRelacIgual , == >
< Num , 3 >
< ParCer , ) >
< PalRes , write >
< ParAb , ( >
< Id , var3 >
< ParCer , ) >
< eol ,  >
< eof ,  >
```

pruebas2.txt> TS

```
CONTENIDO DE LA TABLA # 0:
  * LEXEMA    : 'var1'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

  * LEXEMA    : 'var2'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 2

  * LEXEMA    : 'funcInt'
    ATRIBUTOS :  (funcion)
    + tipo    : 'int'
    + param 1 : 'int'
    + param 2 : 'char'
    + param 3 : 'boolean'

  * LEXEMA    : 'var3'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 3

---------------------------------------
```

```
CONTENIDO DE LA TABLA # 1:
  * LEXEMA    : 'n'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

  * LEXEMA    : 'msg'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 2

  * LEXEMA    : 'j'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 6

---------------------------------------

CONTENIDO DE LA TABLA # 2:
---------------------------------------
```

pruebas3.txt> Tokens

```
< PalRes , var >
< PalRes , char >
< Id , var1 >
< eol ,  >
< PalRes , var >
< PalRes , boolean >
< Id , var2 >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , char >
< Id , funcChar >
< ParAb , ( >
< PalRes , int >
< Id , n >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< Id , var1 >
< OpAsig , = >
< Id , funcChar >
< ParAb , ( >
< Num , 13 >
< ParCer , ) >
< eol ,  >
< PalRes , return >
< Id , var1 >
< eol ,  >
< LlavCer , } >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , int >
< Id , funcInt >
< ParAb , ( >
< PalRes , char >
< Id , msg >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , if >
< ParAb , ( >
< ParAb , ( >
< Id , var1 >
< OpRelacIgual , == >
< Id , funcChar >
< ParAb , ( >
< Num , 3 >
< ParCer , ) >
< ParCer , ) >
< OpRelacIgual , == >
< Boolean , true >
< ParCer , ) >
< PalRes , return >
< Num , 5 >
< eol ,  >
< LlavCer , } >
< eol ,  >
< eol ,  >
< PalRes , write >
< ParAb , ( >
< Id , funcInt >
< ParAb , ( >
< Cadena , "prueba3" >
< ParCer , ) >
< ParCer , ) >
< eol ,  >
< eof ,  >
```

pruebas3.txt> TS

```
CONTENIDO DE LA TABLA # 0:
  * LEXEMA    : 'var1'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 0

  * LEXEMA    : 'var2'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 4

  * LEXEMA    : 'funcChar'
    ATRIBUTOS :  (funcion)
    + tipo    : 'char'
    + param 1 : 'int'

  * LEXEMA    : 'funcInt'
    ATRIBUTOS :  (funcion)
    + tipo    : 'int'
   + param 1 : 'char'
---------------------------------------

CONTENIDO DE LA TABLA # 1:
  * LEXEMA    : 'n'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

---------------------------------------

CONTENIDO DE LA TABLA # 2:
  * LEXEMA    : 'msg'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 0

---------------------------------------

CONTENIDO DE LA TABLA # 3:
---------------------------------------
```

pruebas4.txt> Tokens

```
< PalRes , var >
< PalRes , char >
< Id , var1 >
< eol ,  >
< PalRes , var >
< PalRes , boolean >
< Id , var2 >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , int >
< Id , funcInt >
< ParAb , ( >
< PalRes , char >
< Id , msg >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , return >
< Num , 10 >
< eol ,  >
< LlavCer , } >
< eol ,  >
< eol ,  >
< PalRes , function >
< PalRes , char >
< Id , funcChar >
< ParAb , ( >
< PalRes , int >
< Id , n >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , var >
< PalRes , int >
< Id , i >
< eol ,  >
< PalRes , if >
< ParAb , ( >
< Boolean , true >
< ParCer , ) >
< PalRes , return >
< Cadena , "prueba2" >
< eol ,  >
< PalRes , for >
< ParAb , ( >
< Id , i >
< OpAsig , = >
< Num , 0 >
< PtComa , ; >
< Id , funcInt >
< ParAb , ( >
< Cadena , "prueba1" >
< ParCer , ) >
< OpRelacIgual , == >
< Num , 5 >
< PtComa , ; >
< Id , i >
< OpAsigMod , %= >
< Id , funcInt >
< ParAb , ( >
< Cadena , "prueba2" >
< ParCer , ) >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , var >
< PalRes , char >
< Id , texto >
< eol ,  >
< Id , texto >
< OpAsig , = >
< Cadena , "pasa desde bucle" >
< eol ,  >
< Id , var2 >
< OpAsig , = >
< Id , var2 >
< OpLogAnd , && >
< Boolean , false >
< eol ,  >
< PalRes , return >
< Id , texto >
< eol ,  >
< LlavCer , } >
< eol ,  >
< PalRes , return >
< Cadena , "pasa la prueba" >
< eol ,  >
< LlavCer , } >
< eol ,  >
< Id , var1 >
< OpAsig , = >
< Id , funcChar >
< ParAb , ( >
< Num , 5 >
< ParCer , ) >
< eol ,  >
< eof ,  >
```

pruebas4.txt> TS

```
CONTENIDO DE LA TABLA # 0:
  * LEXEMA    : 'var1'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 0

  * LEXEMA    : 'var2'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 4

  * LEXEMA    : 'funcInt'
    ATRIBUTOS :  (funcion)
    + tipo    : 'int'
    + param 1 : 'char'

  * LEXEMA    : 'funcChar'
    ATRIBUTOS :  (funcion)
    + tipo    : 'char'
    + param 1 : 'int'
```

```
------------------------------

CONTENIDO DE LA TABLA # 1:
  * LEXEMA    : 'msg'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 0
------------------------------

CONTENIDO DE LA TABLA # 2:
  * LEXEMA    : 'n'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

  * LEXEMA    : 'i'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 2
```

```
--------------------------------

CONTENIDO DE LA TABLA # 3:
--------------------------------

CONTENIDO DE LA TABLA # 4:
  * LEXEMA    : 'texto'
    ATRIBUTOS :
    + tipo    : 'char'
    + despl   : 0

--------------------------------
```

pruebas5.txt> Tokens

```
< PalRes , function >
< PalRes , boolean >
< Id , funcBoolean >
< ParAb , ( >
< PalRes , boolean >
< Id , bool >
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , var >
< PalRes , boolean >
< Id , masmas >
< eol ,  >
< eol ,  >
< PalRes , for >
< ParAb , ( >
< PtComa , ; >
< Id , masmas >
< OpRelacIgual , == >
< Boolean , false >
< PtComa , ; >
```

```
< ParCer , ) >
< eol ,  >
< LlavAb , { >
< eol ,  >
< PalRes , if >
< ParAb , ( >
< Id , bool >
< OpRelacIgual , == >
< Boolean , false >
< ParCer , ) >
< Id , masmas >
< OpAsig , = >
< Boolean , true >
< eol ,  >
< LlavCer , } >
< eol ,  >
< PalRes , return >
< Id , masmas >
< eol ,  >
< LlavCer , } >
< eol ,  >
< PalRes , if >
```

```
< ParAb , ( >
< Id , funcBoolean >
< ParAb , ( >
< Boolean , false >
< ParCer , ) >
< OpRelacIgual , == >
< Boolean , true >
< ParCer , ) >
< PalRes , prompt >
< ParAb , ( >
< Id , num >
< ParCer , ) >
< eol ,  >
< PalRes , write >
< ParAb , ( >
< Id , num >
< ParCer , ) >
< eol ,  >
< eof ,  >
```

pruebas5.txt> TS

```
CONTENIDO DE LA TABLA # 0:
  * LEXEMA    : 'funcBoolean'
    ATRIBUTOS :  (funcion)
    + tipo    : 'boolean'
    + param 1 : 'boolean'

  * LEXEMA    : 'num'
    ATRIBUTOS :
    + tipo    : 'int'
    + despl   : 0

------------------------------
```

```
CONTENIDO DE LA TABLA # 1:
  * LEXEMA    : 'bool'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 0

  * LEXEMA    : 'masmas'
    ATRIBUTOS :
    + tipo    : 'boolean'
    + despl   : 1

------------------------------
```

```
CONTENIDO DE LA TABLA # 2:
------------------------------

CONTENIDO DE LA TABLA # 3:
------------------------------

CONTENIDO DE LA TABLA # 4:
------------------------------
```

2.2. Erróneos

pruebas6.txt

```
ERROR SEMÁNTICO  (línea 13): El tipo de retorno de función no coincide con el
                             declarado.
```

pruebas7.txt

```
ERROR SEMÁNTICO  (línea 6): Función funcInt espera argumentos (int, char,
                            boolean).
ERROR SEMÁNTICO  (línea 9): El tipo de retorno de función no coincide con el
                            declarado.
ERROR SEMÁNTICO  (línea 13): Condicional espera tipo lógico.
```

pruebas8.txt

```
ERROR SEMÁNTICO  (línea 6): Sólo se pueden sumar tipos iguales.
ERROR SEMÁNTICO  (línea 6): Se asigna tipo err a una variable tipo char.
ERROR SEMÁNTICO  (línea 13): Se asigna tipo char a una variable tipo int.
ERROR SEMÁNTICO  (línea 14): El tipo de retorno de función no coincide con el
                             declarado.
```

pruebas9.txt

```
ERROR SEMÁNTICO  (línea 13): Se asigna un tipo boolean a una variable tipo int.
ERROR SEMÁNTICO  (línea 13): Sentencia incorrecta primera parte del for.
ERROR SEMÁNTICO  (línea 13): Sólo se pueden sumar tipos iguales.
ERROR SEMÁNTICO  (línea 13): Sentencia incorrecta segunda parte del for.
ERROR SEMÁNTICO  (línea 13): Se asigna tipo boolean en %= , sólo se permite int.
ERROR SEMÁNTICO  (línea 13): Se asigna un tipo boolean a una variable tipo int.
ERROR SEMÁNTICO  (línea 13): Sentencia incorrecta tercera parte del for.
ERROR SEMÁNTICO  (línea 21): Los return no son del mismo tipo.
ERROR SEMÁNTICO  (línea 21): El tipo de retorno de función no coincide con el
                             declarado.
```

pruebas10.txt

```
ERROR LÉXICO     (línea 1): No se reconoce el símbolo en la gramática, después del
                            símbolo "z"
ERROR SEMÁNTICO  (línea 9): Sólo se pueden comparar tipos iguales.
ERROR SEMÁNTICO  (línea 9): Se asigna tipo err a una variable tipo int.
ERROR SEMÁNTICO  (línea 10): Sólo se pueden sumar tipos iguales.
ERROR SEMÁNTICO  (línea 10): Se asigna tipo err a una variable tipo int.
ERROR SINTÁCTICO (línea 11): Token "return" no esperado.
```