Memoria Práctica
Procesadores de Lenguajes

Construcción de un Compilador

Grupo 33:
Pedro Amaya Moreno
Juan López González
Pablo Rodríguez Beceiro

Estructuras del Lenguaje

Para realizar el diseño del analizador léxico, primero vamos a identificar todas las estructuras que va a tener que detectar.

Estructuras

- **Comentarios de bloque:** */* …*/*
- **Constantes:** ***Enteras(int) -> 16bits. -> valor < 32768***
  Cadenas(string) -> "…" -> longCadena <= 64
  Lógicas(boolean) -> true o false
- **Operadores:** ***Suma +*** ***Multiplicación ****
- **Operadores de relación:** ***Menor que <*** ***Mayor que: >***
- **Operadores lógicos:** ***And &&*** ***Or ||***
- **Operadores de asignación:** ***Asignación =*** ***Asig. con división: /=***

  Precedencia de los operadores:

  ```
  Producto (*) -> Suma (+) -> Mayor (>) o Menor (<) -> And (&&)
      -> OR (||)
  ```
- **Identificadores:** **(L| _)(L|d| _)***

  El identificador puede empezar con una letra o _ y se puede completar con letras, números y _.

- **Palabras reservadas:**

  let -> **let** var0 Type; / **let** var0 Type = exp.
  Tipo de variable (Type) -> **int / string / boolean**
  print -> **print** (exp); / **print** exp.
  input -> **input** var0;
  function -> **function** funName [Type] (args) {Sentencias **return** [val];}
  if -> **if** (Cond) sentencia.
  switch ->
  switch (var0) {
  case cte1: sentencias **break.**
  case cteN: sentencias
  default:
  }
- **Caracteres extra:**

  Paréntesis -> **( )**
  Llaves -> **{ }**
  Dos puntos -> **:**
  Punto y coma -> **;**
  Coma -> **,**

Descrita la estructura de lo que podemos detectar en el documento de texto, vamos a empezar el diseño del analizador léxico:

Diseño del Analizador Léxico

1) TOKENS

Nombre_Token	Codigo_Token	Atributo_Tk	Token
Cte_Entera	CTE_INT: 1	valorInt	<1, valorInt>
Cte_Cadena	CTE_STRING: 2	lexema	<2, lexema>
Cte_Logica (true/false)	CTE_BOOLEAN: 3	0(F) o 1(T)	<3, 0/1>
Suma (+)	SUM: 4	-	<4, >
Multiplicación (*)	MULT: 5	-	<5, >
Menor (<)	MEN: 6		<6, >
Mayor (>)	MAY: 7	-	<7, >
And (&&)	AND: 8	-	<8, >
Or (||)	OR: 9	-	<9, >
Asignación (=)	ASIG: 10	-	<10, >
AsignacionDiv (/=)	ASDIV: 11	-	<11, >
ParentesisAb (( )	PARAB: 12	-	<12, >
ParentesisCerr ( ))	PARC: 13	-	<13, >
LlaveAb ({)	LLAVAB: 14	-	<14, >
LlaveCerr (})	LLAVC: 15	-	<15, >
DosPuntos (:)	DOSP: 16	-	<16, >
Coma (,)	COMA: 17	-	<17, >
EndOfLine (;)	EOL: 18	-	<18, >
PR_let (let)	LET: 19	-	<19, >
PR_int (int)	INT: 20	-	<20, >
PR_string (string)	STRING: 21	-	<21, >
PR_boolean (boolean)	BOOLEAN: 22	-	<22, >
PR_print (print)	PRINT: 23	-	<23, >
PR_input (input)	IN: 24	-	<24, >
PR_function (function)	FUNC: 25	-	<25, >
PR_return (return)	RET: 26	-	<26, >
PR_if (if)	IF: 27	-	<27, >
PR_switch (switch)	SWITCH: 28	-	<28, >
PR_case (case)	CASE: 29	-	<29, >
PR_default (default)	DEF: 30	-	<30, >
PR_break (break)	BREAK: 31	-	<31, >
Identificador	ID: 32	posTabS	<32, posTabS>
EOF	EOF: 33		<33, >

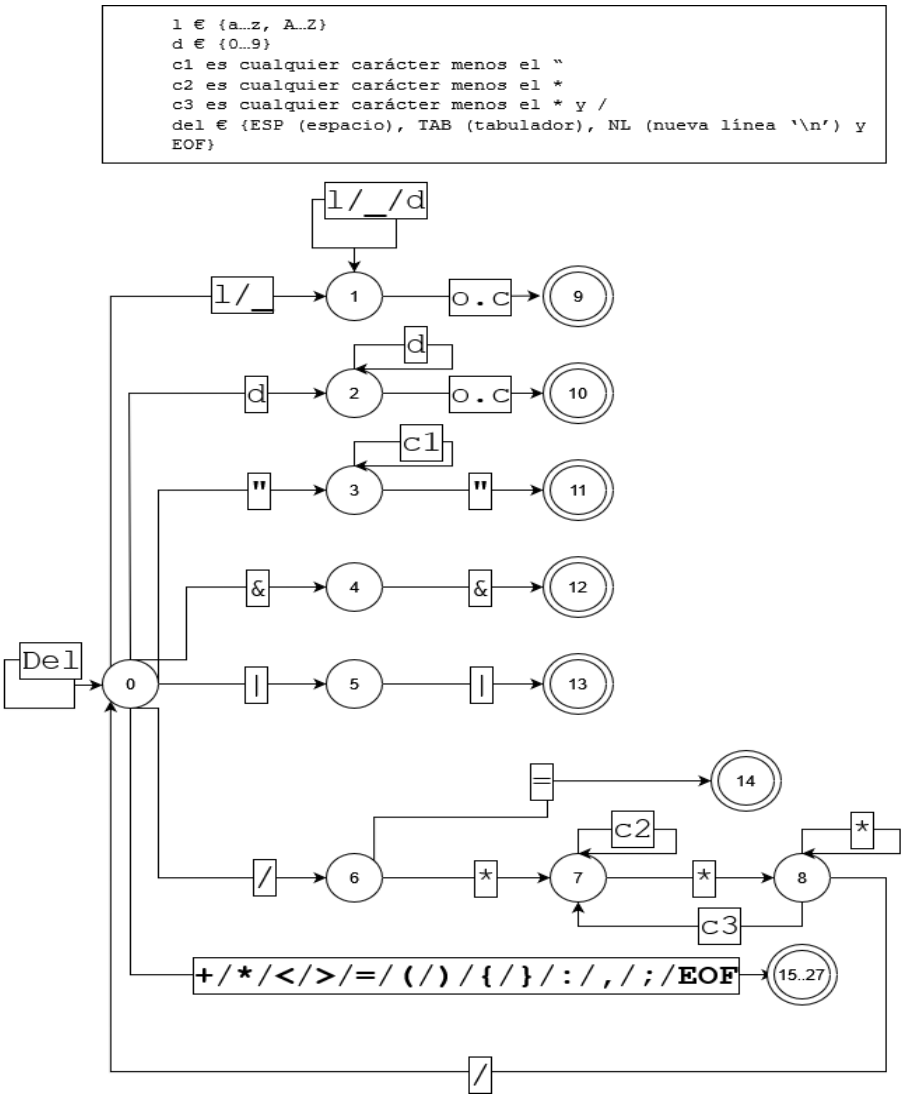
2) GRAMÁTICA REGULAR

GR = < N = {S, A, B, C, D, E, F, G, H},
T = cualquier carácter, P, S >
P = {

S -> lA | _A | dB | ”C | + | * | < | > | &D | |E |
= | (|) | { | } | : | , | ; | /F | del S | EOF
A -> lA | _A | dA |λ
B -> dB | λ
C -> c1C | “
D -> &
E -> |
F -> *G | =
G -> c2G | *H
H -> c3G | *H | /S

}

Siendo: l ∈ {a…z, A…Z} d ∈ {0…9}
c1 ∈ T \{“} c2 ∈ T \{ * EOF } c3 ∈ T \ { * / EOF }
del ∈ {ESP (espacio), TAB (tabulador), NL (nueva línea ‘\n’)}



3) AUTÓMATA FINITO DETERMINISTA

4) ACCIONES SEMÁNTICAS

```
Leer: el siguiente carácter del fichero
                car := leer(); <-> leer;
Concatenar: la secuencia de caracteres de una cadena o de un
     identificador. Realizado sobre la variable lex.
Valor: obtiene el valor numérico del carácter. valor(car)
Genera Token (GenToken(codToken, valAtributo))
Buscar en la Tabla de Palabras Reservadas ( BuscarTPR(palabra) )
Buscar en la Tabla de Símbolos ( BuscarTSL(palabra) en la local y
                                 BuscarTSG(palabra) en la global)
Añadir a la Tabla de Símbolos ( AñadirTS(palabra) )
Acciones:
0:0 {leer;} <-> A
0:1 lex: = car; leer.
1:1 lex: = lex (+) car; leer.
1:9 pos: = BuscarTPR(lex);
    if (pos != Null)
     then if (pos == FALSE) then GenToken (CTE_BOOLEAN, 0)
          elif (pos == TRUE) then GenToken (CTE_BOOLEAN, 1)
          else GenToken (pos, -).
          //FALSE y TRUE son codigos para las palabras reservadas
          //'true' y 'false'. Si es false(0) o true(1)
    else
     pos: = BuscarTS(lex);
     pos := BuscarTSL(lex);
     if (zona_dec = false && pos == 0)
          pos := BuscarTSG(lex);
     if(pos != 0)
          then GenToken(ID, pos);
     else
          if(zona_dec == true)
               pos := AñadirTSActual(lex);
          else
               pos := AñadirTSGlobal(lex);
          GenerarToken(ID, pos);
          //Tk generado de Id (añadido a la TS Actual)
0:2  val := valor(car); leer;
2:2  val := val*10 + valor(car); leer;
2:10 if (val < 32768) then GenToken (CTE_INT, val); else error();
0:3  lex := ""; cont = 0; leer;
3:3  lex := lex (+) car; cont++; leer;
3:11 leer;
     if (cont <= 64) then GenToken (CTE_STRING, lex) else error();
0:4   A
4:12  leer; GenToken (AND -);
0:5   A
5:13  leer; GenToken (OR -);
0:6   A
6:14  leer; GenToken (ASDIV, -).
6:7 / 7:7/ 7:8 / 8:7 / 8:8 / 8:0 A
0:15  leer; GenToken (SUM, -).
0:16  leer; GenToken (MULT, -).
```

```
0:17  leer; GenToken (MEN, -).
0:18  leer; GenToken (MAY, -).
0:19  leer; GenToken (ASIG, -).
0:20  leer; GenToken(PARAB, -);
0:21  leer; GenToken(PARC, -);
0:22  leer; GenToken(LLAVAB, -);
0:23  leer; GenToken(LLAVC, -);
0:24  leer; GenToken(DOSP, -);
0:25  leer; GenToken(COMA, -);
0:26  leer; GenToken(EOL, -);
0:27  GenToken(EOF, -);
```

Tabla Transiciones:

	del		letra U {_}		dígito		"		&		|		/		+		*		<		>		=		(		)		{		}		:		,		;		EOF		O. C.	
0	0	A	1	0:1	2	0:2	3	0:3	4	A	5	A	6	A	15	0:15	16	0:16	17	0:17	18	0:18	19	0:19	20	0:20	21	0:21	22	0:22	23	0:23	24	0:24	25	0:25	26	0:26	27	0:27		40
1	9	1:9	1	1:1	1	1:1	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9	9	1:9
2	10	2:10	10	2:10	2	2:2	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10	10	2:10
3	3	3:3	3	3:3	3	3:3	11	3:11	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3	3	3:3		41	3	3:3
4		42		42		42		42	12	4:12		42		42		42		42		42		42		42		42		42		42		42		42		42		42		45		42
5		43		43		43		43		43	13	5:13		43		43		43		43		43		43		43		43		43		43		43		43		43		45		43
6		44		44		44		44		44		44		44		44	7	A		44		44	14	6:14		44		44		44		44		44		44		44		45		44
7	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	8	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A		46	7	A
8	7	A	7	A	7	A	7	A	7	A	7	A	0	A	7	A	8	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A	7	A		46	7	A
(9)																																											
...																																										
(27)																																										

5) ERRORES LÉXICOS

Error Léxico 40:
Ha llegado el carácter 'c' no reconocido por el procesador en la linea 1

Error Léxico 41:
Finalización de fichero inesperada sin cerrar la cadena 'lex'

Error Léxico 42:
Esperando '&', ha recibido 'c' en la línea 1

Error Léxico 43:
Esperando '|', ha recibido 'c' en la línea 1

Error Léxico 44:
Esperando '=' o '*', ha recibido 'c' en la línea 1

Error Léxico 45:
Finalización de fichero inesperada esperando un carácter

Error Léxico 46:
Finalización de fichero inesperada dentro de un comentario

Error Léxico 47:
El valor = 'val' supera el valor maximo de 32767 en la línea 1

Error Léxico 48:
En la cadena 'lex' se excede la cantidad maxima de 64 caracteres en la linea 1

Error Léxico 49:
La cadena '%s' termina inesperadamente con un salto de linea, en la linea 1

Diseño de la Tabla de Símbolos

Al no tener hecho el Analizador Semántico, las únicas operaciones que podemos realizar con la tabla es la de inserción y búsqueda de un identificador.

Tablas:

Las tablas son los objetos donde vamos almacenando los identificadores que se van detectando. La organización de la tabla es usando una estructura con tablas hash.
Esta estructura consta de un espacio de almacenamiento para los identificadores y una tabla hash que asocia los lexemas de los identificadores con su localización de almacenamiento.
Su espacio de almacenamiento es dinámico, comenzando con un tamaño para 16 identificadores y doblando su tamaño cada vez que se queda sin espacio, es decir, para la primera expansión reserva 16*2. Sus celdas van numeradas de la 1 hasta la n.
La tabla hash que usamos es la Hash Table, clase de Java, que comienza con un tamaño de 16 y factor de carga de 0.75.
La función de inserción de la tabla introduce el nuevo identificador en la última celda sin ocupar (si llena todas las celdas duplica su tamaño) e introduce en el hash table la asociación entre el nuevo lexema y su posición en la tabla.
La función de búsqueda busca su celda con el hash table y su lexema. Si no está en el hash table, devuelve 0, indicando que no lo ha encontrado o su posición si lo ha hecho.
Además, las tablas tienen una función para la inserción del tipo y desplazamiento en los identificadores, una para solo la inserción del tipo (usada para la inserción de funciones que no tienen desplazamiento) y otra para la inserción de la etiqueta de función. También implementamos una función que nos devolviera el nombre de los identificadores para poder mandárselos al gestor de errores en caso de que haya algún error.
Las tablas también controlan el desplazamiento que hay que introducir en los identificadores.

Identificadores:

Los identificadores almacenan su nombre, su tipoy su desplazamiento o si hiciera falta la etiqueta de función. Un identificador no sabe a qué tabla pertenece, solo almacena su información.

Tabla de Símbolos:

La tabla de símbolos es la que maneja la gestión de la tabla global, la actual y la local. Gestiona la creación, destrucción, búsqueda e inserción en las tablas. Además de la zona de declaración (se modifica con las acciones semánticas).
La primera creación de una tabla hace que esta sea la tabla global y la actual en ese momento.
Todos los identificadores que pertenezcan a esta tabla tienen posiciones positivas, es decir, en la generación de un token identificador si su posición en la tabla de símbolos es positiva, entonces pertenece a la global.
Como solo se pueden tener dos contextos (en una función (local) y fuera de una función (global)), las posiciones son o positivas, global, o negativas, pertenecientes a la tabla local.

Cuando la tabla de símbolos vuelve a crear otra tabla, la asigna a la local y cambia la actual por la local.
Al destruir una tabla, se destruye siempre la actual, si no hay ninguna tabla local, entonces se cierra la global sino la local siempre.
La función de inserción en la tabla de símbolos inserta según la zona de declaración. Las declaraciones explícitas siempre se introducen en la tabla actual (zona_dec = true) y las implícitas siempre en la tabla global.
La función de búsqueda de la posición en la tabla de símbolos por el nombre del identificador siempre busca primero en la actual, y después en la global. Devuelven su posición respecto al criterio que hemos descrito antes.
La función de inserción de tipo y desplazamiento o la de solo tipo o la de insertar la etiqueta, primero identifican la tabla en la que hay que realizarlo y le pasa la información necesaria a esa tabla para que realice la operación.
La función de búsqueda de tipo hace una cosa similar a las anteriores, encontrando la tabla y delegando la tarea de devolver su tipo a la tabla.

Diseño del Analizador Sintáctico

1) Gramática LL (1)

```
Terminales = { Cte_int Cte_str Cte_bool + * < > && || = /= ( ) { } : ,
; let int string boolean print input function return if switch case
default break Id Lambda $ }
NoTerminales = { PE P F SentS SentC SentSw ExpDec Ig Args ArgsE Vargs
VargsE BloqF BloqSw Type TypeE Val ValRet ValE And AndE Comp CompE Sum
SumE Prod ProdE Unit UnitE
}
Axioma = PE
Producciones = {
PE -> P ////1
P -> SentC P //// 2
P -> F P
P -> Lambda
SentS -> print Val ; //// 5
SentS -> input Id ;
SentS -> return ValRet ;
SentS -> break ;
SentS -> Id ExpDec ;
ExpDec -> = Val        //// 10
ExpDec -> ( Vargs )
ExpDec -> /= Val
Vargs -> Val VargsE    //// 13
Vargs -> Lambda
VargsE -> , Val VargsE ////15
VargsE -> Lambda
SentC -> let Id Type Ig ; //// 17
SentC -> if ( Val ) SentS
SentC -> switch ( Val ) { BloqSw }
SentC -> SentS
Ig -> = Val      //// 21
Ig -> Lambda
Type -> int      //// 23
Type -> string
Type -> boolean
BloqSw -> case Val : SentSw BloqSw //// 26
```

```
BloqSw -> default : SentSw
BloqSw -> Lambda
SentSw -> SentC SentSw //// 29
SentSw -> Lambda
F -> function Id TypeE ( Args ) { BloqF } //// 31
TypeE -> Type          //// 32
TypeE -> Lambda
Args -> Type Id ArgsE //// 34
Args -> Lambda
ArgsE -> , Type Id ArgsE //// 36
ArgsE -> Lambda
BloqF -> SentC BloqF   //// 38
BloqF -> Lambda
ValRet -> Val               //// 40
ValRet -> Lambda
Val -> And ValE             //// 42 a || b
ValE -> || And ValE         //// 43
ValE -> Lambda
And -> Comp AndE       //// 45 a && b
AndE -> && Comp AndE   //// 46
AndE -> Lambda
Comp -> Sum CompE      //// 48 a < b o a > b
CompE -> < Sum CompE   //// 49
CompE -> > Sum CompE
CompE -> Lambda
Sum -> Prod SumE       //// 52 a + b
SumE -> + Prod SumE         //// 53
SumE -> Lambda
Prod -> Unit ProdE          //// 55 a * b
ProdE -> * Unit ProdE  //// 56
ProdE -> Lambda
Unit -> ( Val )             //// 58
Unit -> Cte_int
Unit -> Cte_bool
Unit -> Cte_str
Unit -> Id UnitE
UnitE -> ( Vargs )          //// 63
UnitE -> Lambda             //// 64
}
```

2) Comprobación de la gramática LL (1)

Antes de realizar la comprobación de la gramática, vamos a realizar los First y Follow de los símbolos no terminales, para facilitar después la comprobación.

NTerminales	FIRST	FOLLOW
PE	function let if switch print input return break Id λ	$
P	function let if switch print input return break Id λ	$
F	function	function let if switch print input return break Id $
SentS	print input return break Id	function let if switch print input return break Id case default } $
SentC	let if switch print input return break Id	function let if switch print input return break Id case default } $
SentSw	let if switch print input return break Id λ	case default }
ExpDec	= (/=	;

Ig	= λ	;
Args	int string boolean λ	)
ArgsE	, λ	)
Vargs	(Cte_int Cte_bool Cte_str Id λ	)
VargsE	, λ	)
BloqF	let if switch print input return break Id λ	}
BloqSw	case default λ	}
Type	int string boolean	Id (= ;
TypeE	int string boolean λ	(
ValRet	(Cte_int Cte_bool Cte_str Id λ	;
Val	(Cte_int Cte_bool Cte_str Id	) ; : ,
ValE	|| λ	) ; : ,
And	(Cte_int Cte_bool Cte_str Id	||) ; : ,
AndE	&& λ	||) ; : ,
Comp	(Cte_int Cte_bool Cte_str Id	&& ||) ; : ,
CompE	< > λ	&& ||) ; : ,
Sum	(Cte_int Cte_bool Cte_str Id	< > && ||) ; : ,
SumE	+ λ	< > && ||) ; : ,
Prod	(Cte_int Cte_bool Cte_str Id	+ < > && ||) ; : ,
ProdE	* λ	+ < > && ||) ; : ,
Unit	(Cte_int Cte_bool Cte_str Id	* + < > && ||) ; : ,
UnitE	(λ	* + < > && ||) ; : ,

Ahora vamos a realizar la comprobación de la gramática.

```
//1 (Producción añadida para la creación y destrucción de la TSG)
PE -> P
      First(P) => { function let if switch print input return break Id λ }
            λ pertenece a First -> U Follow(PE) = {$}
//Un solo argumento, ya cumple la condición.

//2
P -> SentC P | F P| λ
      First( SentC P ) => { let if switch print input return break Id  }
            First(SentC) = { let if switch print input return break Id  }
                  λ no pertenece a First -> fin
      First( F P ) => { function }
            First(F) = { function } λ no pertenece a First -> fin
      First(λ) => { $ }
            First(λ) = { λ } λ pertenece a First ->
                  U Follow(P) = { $ }
First( SentC P ) ^ First( F P ) ^ {First(λ)\{ λ } U Follow(P)} = vacío

//En los casos siguiente si λ no pertenece a First no vamos a indicarlo y
//vamos asumir el fin de ese First. Si el primer simbolo de una produccion
//es un terminal lo indicamos directamente.
//5
SentS-> print Val ; | input Id ; | return ValRet ; | break ;
      | Id ExpDec ;
      First( print Val ; ) => { print }
      First( input Id ; ) => { input }
      First( return ValRet ; ) => { return }
      First( break ; ) => { break }
      First( Id ExpDec ; ) => { Id }
```

```
First( print Val ; ) ^ First( Id ; ) ^ First( return ValRet ; )
      ^ First( break ; ) ^ First( Id ExpDec ; ) = vacío

//10
ExpDec -> = Val | ( VArgs ) | /= Val
      First( = Val ) => { = }
      First( ( VArgs ) ) => { ( }
      First( /= Val ) => { /= }
First( = Val ) ^ First( ( Vargs ) ) ^ First( /= Val ) = vacío

//13
Vargs -> Val VargsE   | λ
      First( Val VargsE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Val) = { ( Cte_int Cte_bool Cte_str Id }
      First(λ) => { ) }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(Vargs) = { ) }
First( Val VargsE ) ^ {First(λ)\{ λ } U Follow(Vargs)} = vacío
//15
VargsE -> , Val VargsE | λ
      First( , Val VargsE ) => { , }
      First(λ) => { ) }
            First(λ) = { λ } λ pertenece a First ->
                  U Follow(VargsE) = { ) }
First( , Val VargsE ) ^ {First(λ)\{ λ } U Follow(VargsE)} = vacío

//17
SentC -> let Id Type Ig ; | if ( Val ) SentS
      | switch ( Val ) { BloqSw } | SentS
      First( let Id Type Ig ; ) => { let }
      First( if ( Val ) SentS ) => { if }
      First( switch ( Val ) { BloqSw } ) => { switch }
      First( SentS ) => { print input return break Id }
            First(SentS) = { print input return break Id }
First( let Id Type Ig ;) ^ First( if ( Val ) SentS )
      ^ First( switch ( Val ) { BloqSw } ) ^ First( SentS ) = vacío

//21
Ig -> = Val | λ
      First( = Val ) => { = }
      First(λ) => { ) }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(Ig) = { ; }
First( = Val ) ^ {First(λ)\{ λ } U Follow(Ig)} = vacío

//23
Type -> int | string | boolean
      First( int ) => { int}
      First( string ) => { string }
      First( boolean ) => { boolean }
First( int ) ^ First( string ) ^ First( boolean ) = vacío

//26
BloqSw -> case Val : SentSw BloqSw | default : SentSw | λ
      First( case Val : SentSw BloqSw ) => { case }
      First( default : SentSw ) => { default }
      First(λ) => { } }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(BloqSw) = { } }
```

```
First( case Val : SentSw BloqSw ) ^ First( default : SentSw )
      ^ {First(λ)\{ λ } U Follow(BloqSw)} = vacío

//29
SentSw -> SentC SentSw | λ
      First( SentC SentSw ) => { let if switch print input return break Id }
            First(SentC) = { let if switch print input return break Id }
      First(λ) => { case default } }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(SentSw) = { case default } }
First( SentC SentSw ) ^ {First(λ)\{ λ } U Follow(SentSw)} = vacío

//31
F -> function Id TypeE ( Args ) { BloqF }
      First( function Id TypeE ( Args ) { BloqF } ) => { function }
//Un solo argumento, ya cumple la condición.

//32
TypeE -> Type | λ
      First( Type ) => { int string boolean }
            First(Type) = { int string boolean }
      First(λ) => { ( }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(TypeE) = { ( }
First( Type ) ^ {First(λ)\{ λ } U Follow(TypeE)} = vacío

//34
Args -> Type Id ArgsE | λ
      First( Type Id ArgsE ) => { int string boolean }
            First(Type) = { int string boolean }
      First(λ) => { ) }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(Args) = { ) }
First( Type Id ArgsE ) ^ {First(λ)\{ λ } U Follow(Args)} = vacío

//36
ArgsE -> , Type Id ArgsE | λ
      First( , Type Id ArgsE ) => { , }
      First(λ) => { ) }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(ArgsE) = { ) }
First( , Type Id ArgsE ) ^ {First(λ)\{ λ } U Follow(ArgsE)} = vacío

//38
BloqF -> SentC BloqF | λ
      First( SentC BloqF ) => { let if switch print input return break Id }
            First(SentC) = { let if switch print input return break Id }
      First(λ) => { } }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(BloqF) = { } }
First( SentC BloqF ) ^ {First(λ)\{ λ } U Follow(BloqF)} = vacío

//40
ValRet -> Val | λ
      First( Val ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Val) = { ( Cte_int Cte_bool Cte_str Id }

      First(λ) => { ; }
```

```
            First(λ) = { λ } λpertenece a First ->
                  U Follow(ValRet) = { ; }
First( Val ) ^ {First(λ)\{ λ } U Follow(ValRet)} = vacío

//42
Val -> And ValE
      First( And ValE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(And) = { ( Cte_int Cte_bool Cte_str Id }
//Un solo argumento, ya cumple la condición.

//43
ValE -> || And ValE | λ
      First( || And ValE ) => { || }
      First(λ) => { ) ; : , }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(ValE) = { ) ; : , }
First( || And ValE ) ^ {First(λ)\{ λ } U Follow(ValE)} = vacío
//45
And -> Comp AndE
      First( Comp AndE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Comp) = { ( Cte_int Cte_bool Cte_str Id }
//Un solo argumento, ya cumple la condición.

//46
AndE -> && Comp AndE | λ
      First( && Comp AndE ) => { && }
      First(λ) => { || ) ; : , }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(AndE) = { || ) ; : , }
First( && Comp AndE ) ^ {First(λ)\{ λ } U Follow(AndE)} = vacío

//48
Comp -> Sum CompE
      First( Sum CompE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Sum) = { ( Cte_int Cte_bool Cte_str Id }
//Un solo argumento, ya cumple la condición.

//49
CompE -> < Sum CompE | > Sum CompE | λ
      First( < Sum CompE ) => { < }
      First( > Sum CompE ) => { > }
      First(λ) => { && || ) ; : , }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(CompE) = { && || ) ; : , }
First( < Sum CompE) ^ First( > Sum CompE )
      ^ {First(λ)\{ λ } U Follow(CompE)} = vacío

//52
Sum -> Prod SumE
      First( Prod CompE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Prod) = { ( Cte_int Cte_bool Cte_str Id }
//Un solo argumento, ya cumple la condición.

//53
SumE -> + Prod SumE | λ
      First( + Prod SumE ) => { + }
      First(λ) => { < > && || ) ; : , }
            First(λ) = { λ } λpertenece a First ->
```

```
                  U Follow(SumE) = { < > && || ) ; : , }
First( + Prod SumE ) ^ {First(λ)\{ λ } U Follow(SumE)} = vacío

//55
Prod -> Unit ProdE
      First( Unit ProdE ) => { ( Cte_int Cte_bool Cte_str Id }
            First(Unit) = { ( Cte_int Cte_bool Cte_str Id }
//Un solo argumento, ya cumple la condición.

//56
ProdE -> * Unit ProdE | λ
      First( * Unit ProdE ) => { * }
      First(λ) => { + < > && || ) ; : , }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(ProdE) = { + < > && || ) ; : , }
First( * Unit ProdE ) ^ {First(λ)\{ λ } U Follow(ProdE)} = vacío

//57
Unit -> ( Val ) | Cte_int | Cte_bool | Cte_str | Id UnitE
      First( ( Val ) ) => { ( }
      First( Cte_int ) => { Cte_int }
      First( Cte_bool ) => { Cte_bool }
      First( Cte_str ) => { Cte_str }
      First( Id UnitE ) => { Id }
First( ( Val ) ) ^ First( Cte_int ) ^ First( Cte_bool ) ^ First( Cte_str )
      ^ First( Id UnitE ) = vacío

//63
UnitE -> ( Vargs )| λ
      First( ( Vargs ) ) => { ( }
      First(λ) => { * + < > && || ) ; : , }
            First(λ) = { λ } λpertenece a First ->
                  U Follow(UnitE) = { * + < > && || ) ; : , }
First( ( Vargs ) ) ^ {First(λ)\{ λ } U Follow(UnitE)} = vacío
```

La gramática G cumple las condiciones LL(1) necesaria, por lo que podemos asegurarnos que es una gramática válida para un Analizador Descendente LL(1).

3) Tabla Descendiente LL(1)

Debido al tamaño de la tabla la hemos dividido en varias más pequeñas.

	Cte_int	Cte_str	Cte_bool	+	*	<
PE						
P						
F						
SentS						
SentC						
SentSW						
ExpDec						
Ig						
Args						
ArgsE						
Vargs	Vargs -> Val VargsE	Vargs -> Val VargsE	Vargs -> Val VargsE			
VargsE						
BloqF						
BloqSw						
Type						
TypeE						
Val	Val -> And ValE	Val -> And ValE	Val -> And ValE			
ValRet	ValRet -> Val	ValRet -> Val	ValRet -> Val			
ValE						
And	And -> Comp AndE	And -> Comp AndE	And -> Comp AndE			
AndE						
Comp	Comp -> Sum CompE	Comp -> Sum CompE	Comp -> Sum CompE			
CompE						CompE -> < Sum CompE
Sum	Sum -> Prod SumE	Sum -> Prod SumE	Sum -> Prod SumE			
SumE				SumE -> + Prod SumE		SumE -> λ
Prod	Prod -> Unit ProdE	Prod -> Unit ProdE	Prod -> Unit ProdE			
ProdE				ProdE -> λ	ProdE -> * Unit ProdE	ProdE -> λ
Unit	Unit -> Cte_int	Unit -> Cte_str	Unit -> Cte_bool			
UnitE				UnitE -> λ	UnitE -> λ	UnitE -> λ

	>	&&	||	=	/=	(
PE						
P						
F						
SentS						
SentC						
SentSW						
ExpDec				ExpDec -> = Val	ExpDec -> /= Val	ExpDec -> (VArgs)
Ig				Ig -> = Val		
Args						
ArgsE						
Vargs						Vargs -> Val VargsE
VargsE						
BloqF						
BloqSw						
Type						
TypeE						TypeE -> λ
Val						Val -> And ValE
ValRet						ValRet -> Val
ValE			ValE -> || And ValE			
And						And -> Comp AndE
AndE		AndE -> && Comp AndE	AndE -> λ			
Comp						Comp -> Sum CompE
CompE	CompE -> > Sum CompE	CompE -> λ	CompE -> λ			
Sum						Sum -> Prod SumE
SumE	SumE -> λ	SumE -> λ	SumE -> λ			
Prod						Prod -> Unit ProdE
ProdE	ProdE -> λ	ProdE -> λ	ProdE -> λ			
Unit						Unit -> (Val)
UnitE	UnitE -> λ	UnitE -> λ	UnitE -> λ			UnitE -> (Vargs)

	)	{	}	:	,	;
PE						
P						
F						
SentS						
SentC						
SentSW			SentSw -> λ			
ExpDec						
Ig						Ig -> λ
Args	Args -> λ					
ArgsE	ArgsE -> λ				ArgsE -> , Type Id ArgsE	
Vargs	Vargs -> λ					
VargsE	VargsE -> λ				VargsE -> , Val VargsE	
BloqF			BloqF -> λ			
BloqSw			BloqSw -> λ			
Type						
TypeE						
Val						
ValRet						ValRet -> λ
ValE	ValE -> λ			ValE -> λ	ValE -> λ	ValE -> λ
And						
AndE	AndE -> λ			AndE -> λ	AndE -> λ	AndE -> λ
Comp						
CompE	CompE -> λ			CompE -> λ	CompE -> λ	CompE -> λ
Sum						
SumE	SumE -> λ			SumE -> λ	SumE -> λ	SumE -> λ
Prod						
ProdE	ProdE -> λ			ProdE -> λ	ProdE -> λ	ProdE -> λ
Unit						
UnitE	UnitE -> λ			UnitE -> λ	UnitE -> λ	UnitE -> λ

	let	int	string	boolean	print	input
PE	PE -> P				PE -> P	PE -> P
P	P -> SentC P				P -> SentC P	P -> SentC P
F						
SentS					SentS -> print Val ;	SentS -> intro Id ;
SentC	SentC -> let Id Type Ig ;				SentC -> SentS	SentC -> SentS
SentSW	SentSw -> SentC SentSw				SentSw -> SentC SentSw	SentSw -> SentC SentSw
ExpDec						
Ig						
Args		Args -> Type Id ArgsE	Args -> Type Id ArgsE	Args -> Type Id ArgsE		
ArgsE						
Vargs						
VargsE						
BloqF	BloqF -> SentC BloqF				BloqF -> SentC BloqF	BloqF -> SentC BloqF
BloqSw						
Type		Type -> int	Type -> string	Type -> boolean		
TypeE		TypeE -> Type	TypeE -> Type	TypeE -> Type		
Val						
ValRet						
ValE						
And						
AndE						
Comp						
CompE						
Sum						
SumE						
Prod						
ProdE						
Unit						
UnitE						

	function	return	if	switch	case
PE	PE -> P	PE -> P	PE -> P	PE -> P	
P	P -> F P	P -> SentC P	P -> SentC P	P -> SentC P	
F	F -> function Id TypeE (Args) { BloqF }				
SentS		SentS -> return ValRet ;			
SentC		SentC -> SentS	SentC -> if (Val) SentS	SentC -> switch (Id) { BloqSw }	
SentSW		SentSw -> SentC SentSw	SentSw -> SentC SentSw	SentSw -> SentC SentSw	SentSw -> λ
ExpDec					
Ig					
Args					
ArgsE					
Vargs					
VargsE					
BloqF		BloqF -> SentC BloqF	BloqF -> SentC BloqF	BloqF -> SentC BloqF	
BloqSw					BloqSw -> case Val : SentSw BloqSw
Type					
TypeE					
Val					
ValRet					
ValE					
And					
AndE					
Comp					
CompE					
Sum					
SumE					
Prod					
ProdE					
Unit					
UnitE					

	default	break	Id	$
PE		PE -> P	PE -> P	PE -> P
P		P -> SentC P	P -> SentC P	P -> λ
F				
SentS		SentS -> break ;	SentS -> Id ExpDec ;	
SentC		SentC -> SentS	SentC -> SentS	
SentSW	SentSw -> λ	SentSw -> SentC SentSw	SentSw -> SentC SentSw	
ExpDec				
Ig				
Args				
ArgsE				
Vargs			Vargs -> Val VargsE	
VargsE				
BloqF		BloqF -> SentC BloqF	BloqF -> SentC BloqF	
BloqSw	BloqSw -> default : SentSw			
Type				
TypeE				
Val			Val -> And ValE	
ValRet			ValRet -> Val	
ValE				
And			And -> Comp AndE	
AndE				
Comp			Comp -> Sum CompE	
CompE				
Sum			Sum -> Prod SumE	
SumE				
Prod			Prod -> Unit ProdE	
ProdE				
Unit			Unit -> Id UnitE	
UnitE				

4) Errores:

Error Sintactico 60:
Esperando 'token' ha recibido 'token', en la linea l
Error Sintactico 61:
Valor inesperado 'token', en la linea l

```
// El error 60 es el que ocurre cuando se compara un token de
// la pila con el token actual. El 61 cuando la celda en la
// linea del no terminal en la cima de la linea y la columna
// del token actual esta vacia
```

Diseño del Analizador Semántico

1) Esquema de Traducción de la Gramática LL (1)

Declaración de tipos:

```
      Tipos:      tipo_ok     tipo_err    vacio
                  entero      string      bool
                  funcion (TiposArgs -> TipoRet)
      Acciones:   crearTS()   destruirTS()
                  insertarTipoTS()
                  insertarTipoyDespTS()
                  buscarTipoTS()
                  insertarEtiquetaTS()
      Variables:  Zona_dec
                  TSG         DespG
                  TSL         DespL
                  TSAC
      Atributos:  .tipo       .tipoRet    .inSwitch
                  .inFunc     .ancho      .posTS
// Los errores semánticos los vamos a lanzar solo en el ámbito
// global al terminar el análisis y el ámbito de función.
// Cada vez que introduzcamos un nuevo tipo_err, lo
// registraremos en el gestor de errores para lanzar después
// todos los errores de los ámbitos en bloque.
Producciones = {
// 1
PE -> {TSG = crearTS(); DespG = 0; Zona_dec = false; TSAC = TSG;}1 P {
      if P.tipo = tipo_err then error(80); destruirTS(TSG);}2
// 2
P -> SentC P1 {
      P.tipo = if SentC.tipo = tipo_ok then P1.tipo else tipo_err;}3
P -> F P1 {P.tipo = if F.tipo = tipo_ok then P1.tipo else tipo_err;}4
P -> Lambda {P.tipo = tipo_ok}5

// 5
SentS -> print Val ; {
      SentS.tipo = if Val.tipo != tipo_err then tipo_ok else tipo_err;
      SentS.tipoRet = vacio;}6
SentS -> input Id {
      if buscarTipoTS(Id.posTS) = tipo_err then {
            anadirTipoyDespTS(Id.posTS, entero, despG);
            despG += 1;
      }80 ; {SentS.tipo = if buscaTipoTS(id.pos) in {int, string} then
      tipo_ok else tipo_err; SentS.tipoRet = vacio;}7
SentS -> return ValRet ; {
      SentS.tipo = if ValRet.tipo != err_tipo then tipo_ok
            else tipo_err;
      SentS.tipoRet = if SentS.inFunc = true then ValRet.tipo
            else tipo_err;}8
SentS -> break ; {SentS.tipo =
      if SentS.inSwitch = true then tipo_ok else tipo_err;
      SentS.tipoRet = vacio;}9
SentS -> Id {
      if buscarTipoTS(Id.posTS) = tipo_err then {
            anadirTipoyDespTS(Id.posTS, entero, despG);
            despG += 1;
      } ExpDec.posTS = Id.posTS;
```

```
      }10 ExpDec ; {SentS.tipo = ExpDec.tipo; SentS.tipoRet = vacio;}11

// 10
ExpDec -> = Val {ExpDec.tipo =
      if buscarTipoTS(ExpDec.posTS) = Val.tipo != err_tipo then tipo_ok
            else tipo_err;}12
ExpDec -> ( Vargs ) {
      ExpDec.tipo = if buscarTipoTS(ExpDec.posTS) = VArgs.tipo -> t
            then tipo_ok else tipo_err;}13
ExpDec -> /= Val {ExpDec.tipo =
      if buscarTipoTS(ExpDec.posTS) = Val.tipo = int then tipo_ok
            else tipo_err;}14

// 13
Vargs -> Val VargsE {
      Vargs.tipo = if VArgsE.tipo = vacio then Val.tipo
            else Val.tipo X VArgsE.tipo;}15
Vargs -> Lambda {Vargs.tipo = vacio}16

// 15
VargsE -> , Val VargsE1 {
      VargsE.tipo = if VArgsE1.tipo = vacio then Val.tipo
            else Val.tipo X VargsE1.tipo;}17
VargsE -> Lambda {VArgsE.tipo = vacio}18

// 17
SentC -> {zona_dec = true;}19 let Id Type {
      if buscarTipoTS(Id.posTS) != tipo_err{
            error(101);
      } else if TSG = TSAC then {
            insertarTipoyDespTS(Id.posTS, Type.tipo, despG);
            despG += Type.ancho;
      } else {
            insertarTipoyDespTS(Id.posTS, Type.tipo, despL);
            despL += Type.ancho;
      } Ig.posTS = Id.posTS; zona_dec = false}20 Ig ; {
      SentC.tipo = Ig.tipo; SentC.tipoRet = vacio;}21

SentC -> if ( Val ) {SentS.inSwitch = SentC.inSwitch; SentS.inFunc =
      SentC.inFunc;}22 SentS {
      SentC.tipo = if Val.tipo = boolean then SentS.tipo else tipo_err;
      SentC.tipoRet = SentS.tipoRet;}23

SentC -> switch ( Val ){{BloqSw.inFunc = SentC.inFunc}24 BloqSw }{
      SentC.tipo = if Val.tipo = int then BloqSw.tipo
                  else tipo_err;
      SentC.tipoRet = BloqSw.tipoRet}25

SentC -> {SentC.switch = SentC.inSwitch; SentS.func = SentC.inFunc}26
      SentS {SentC.tipo = SentS.tipo; SentC.tipoRet = SentS.tipoRet}27

// 21
Ig -> = Val {Ig.tipo = if BuscarTipoTS(Ig.posTS) = Val.tipo != err_tipo
      then tipo_ok else tipo_err}28
Ig -> Lambda {Ig.tipo = tipo_ok}29

// 23
Type -> int {Type.tipo = int; Type.ancho = 1;}30
```

```
Type -> string {Type.tipo = string; Type.ancho = 64;}31
Type -> boolean {Type.tipo = boolean; Type.ancho = 1;}32

// 26
BloqSw -> case Val : {SentSw.inFunc = BloqSw.inFunc}33 SentSw
      {BloqSw1.inFunc = BloqSw.inFunc}34 BloqSw1 {
      BloqSw.tipo = if Val.tipo = int and SentSw.tipo = tipo_ok then
            BloqSw1.tipo else tipo_err;
      BloqSw.tipoRet = if SentSw.tipoRet = BloqSw1.tipoRet or
            BloqSw1.tipoRet = vacio then SentSw.tipoRet
            elif SentSw.tipoRet = vacio then BloqSw1.tipoRet
            else tipo_err;}35
BloqSw -> default : {SentSw.inFunc = BloqSw.inFunc}36 SentSw {
      BloqSw.tipo = SentSw.tipo; BloqSw.tipoRet = SentSw.tipoRet;}37
BloqSw -> Lambda {BloqSw.tipo = tipo_ok; BloqSw.tipoRet = vacio;}38

// 29
SentSw -> {SentC.inSwitch = true; SentC.inFunc = SentSw.inFunc}39 SentC
            {SentSw1.inFunc = SentSw.inFunc}83 SentSw1 {
      SentSw.tipo = if SentC.tipo = tipo_ok then SentSw1.tipo
                  else tipo_err;
      SentSw.tipoRet = if SentC.tipoRet = SentSw1.tipoRet or
            SentSw1.tipoRet = vacio then SentC.tipoRet
            elif SentC.tipoRet = vacio then SentSw1.tipoRet
            else tipo_err;}40

SentSw -> Lambda {SentSw.tipo = tipo_ok; SentSw.tipoRet = vacio}41

// 31
F -> function Id {TSL = crearTS(); despL = 0; TSAC = TSL;
      zona_dec = true}42 TypeE ( Args ) {Zona_dec = false;
      insertarTipoTS(Id.posTS, Args.tipo -> TypeE.tipo);
      insertarEtiquetaTS(Id.posTS);}43 { BloqF } {
      F.tipo = if BloqF.tipoRet = TypeE.tipo then BloqF.tipo else
tipo_err;
      destruirTS(TSL); TSAC = TSG;}44
// 32
TypeE -> Type {TypeE.tipo = Type.tipo;}45
TypeE -> Lambda {TypeE.tipo = vacio;}46

// 35
Args -> Type Id {insertarTipoyDespTS(Id.posTS, Type.tipo, despL);
      despL += Type.ancho;}81 ArgsE {
      Args.tipo = if ArgsE.tipo = vacio then Type.tipo
            else Type.tipo x ArgsE.tipo}47
Args -> Lambda {Args.tipo = vacio}48

// 36
ArgsE -> , Type Id {insertarTipoyDespTS(Id.posTS, Type.tipo, despL);
      despL += Type.ancho;}82 ArgsE1 {
      Args.tipo = if ArgsE1.tipo = vacio then Type.tipo
            else Type.tipo x ArgsE1.tipo}49
ArgsE -> Lambda {ArgsE.tipo = vacio}50

// 38
BloqF -> {SentC.inFunc = true}51 SentC BloqF1 {
      BloqF.tipo = if SentC.tipo = tipo_ok then BloqF1.tipo
            else tipo_err;
```

```
      BloqF.tipoRet = if SentC.tipoRet = BloqF1.tipoRet or
            BloqF1.tipoRet = vacio then SentC.tipoRet
            elif SentC.tipoRet = vacio then BloqF1.tipoRet
            else tipo_err}52
BloqF -> Lambda {BloqF.tipo = tipo_ok; BloqF.tipoRet = vacio;}53

// 39
ValRet -> Val {ValRet.tipo = Val.tipo}54
ValRet -> Lambda {ValRet.tipo = vacio}55

// 42
Val -> And ValE {
      Val.tipo = if And.tipo = ValE.tipo or ValE.tipo = vacio
            then And.tipo else tipo_err;}56

// 43
ValE -> || And ValE1 {
      ValE.tipo = if And.tipo = boolean and
            ValE1.tipo in {vacio, boolean} then boolean
            else tipo_err;}57
ValE -> Lambda {ValE.tipo = vacio}58

// 45
And -> Comp AndE {
      And.tipo = if Comp.tipo = AndE.tipo or AndE.tipo = vacio
            then Comp.tipo else tipo_err}59
// 46
AndE -> && Comp AndE1 {
      AndE.tipo = if Comp.tipo = boolean and
            AndE1.tipo in {vacio, boolean} then boolean
            else tipo_err;}60
AndE -> Lambda {AndE.tipo = vacio}61


// 48
Comp -> Sum CompE {
      Comp.tipo = if Sum.tipo = CompE.tipo = int then boolean
            elif CompE.tipo = vacio then Sum.tipo
            else tipo_err;}62
// 49
CompE -> < Sum CompE1 {
      CompE.tipo = if Sum.tipo = int and CompE1.tipo = vacio then int
            else tipo_err;}63
CompE -> > Sum CompE1 {
      CompE.tipo = if Sum.tipo = int and CompE1.tipo = vacio then int
            else tipo_err;}64
CompE -> Lambda {CompE.tipo = vacio}65

// 52
Sum -> Prod SumE {
      Sum.tipo = if Prod.tipo = SumE.tipo or SumE.tipo in vacio then
            Prod.tipo else tipo_err;}66

// 53
SumE -> + Prod SumE1 {
      SumE.tipo = if Prod.tipo = int and SumE1.tipo in {vacio, int}
            then int else tipo_err;}67
SumE -> Lambda {SumE.tipo = vacio}68
```

```
// 55
Prod -> Unit ProdE {
      Prod.tipo = if Unit.tipo = ProdE.tipo or ProdE.tipo = vacio
            then Unit.tipo else tipo_err;}69

// 56
ProdE -> * Unit ProdE1 {
      ProdE.tipo = if Unit.tipo = int and ProdE1.tipo in {vacio, int}
            then int else tipo_err;}70
ProdE -> Lambda {ProdE.tipo = vacio}71

// 58
Unit -> ( Val ) {Unit.tipo = Val.tipo}72
Unit -> Cte_int {Unit.tipo = int}73
Unit -> Cte_bool {Unit.tipo = boolean}74
Unit -> Cte_str {Unit.tipo = string}75
Unit -> Id {
      if buscarTipoTS(Id.posTS) = tipo_err then {
            anadirTipoyDespTS(Id.posTS, int, despG);
            despG += 2;}
      ExpDec.posTS = Id.posTS;}76 UnitE {
            Unit.tipo = if UnitE.tipo = vacio then BuscaTipoTS(id.posTS)
                  else UnitE.tipo;}77
// 63
UnitE -> ( Vargs ) {
      UnitE.tipo = if buscarTipoTS(ExpDec.posTS) = VArgs.tipo -> t
            and t != vacio then t else tipo_err;}78
UnitE -> Lambda {UnitE.tipo = vacio}79
}
```

Errores Semánticos:

Para el tratamiento y la gestión de errores semánticos, cada vez que emitimos un tipo_err nuevo, empujamos el error a una cola en el gestor, y la última acción semántica la vacía.

```
+ Error Semantico 80:
        El identificador '%s' usado para el input no es del tipo cadena o
        entero %s, antes de la linea %d
+ Error Semantico 81:
        El identificador '%s' usado para el input no es del tipo cadena o
        entero, antes de la linea %d
+ Error Semantico 82:
        Ha tratado de usar un break fuera de un switch, antes de la linea
        %d
+ Error Semantico 83:
        Tratando de asignar un valor al identificador '%s', que es una
        funcion, antes de la linea %d
+ Error Semantico 84:
        Tratando de asignar un valor de un tipo %s al identificador ‘%s’ de
        tipo %s, antes de la linea %d
+ Error Semantico 85:
        El identificador '%s' no es una funcion y se esta tratando como si
        lo fuera, antes de la linea %d
+ Error Semantico 86:
        La funcion '%s' (sus argumentos), esta recibiendo estos parametros
        %s que no se corresponden, antes de la linea %d
```

```
+ Error Semantico 87:
        Esperando un tipo booleano en la condicion del if, ha recibido un
        %s, antes de la linea %d
+ Error Semantico 88:
        En 'switch(Val)', Val debe ser un valor entero, sin embargo es un
        %s, antes de la linea %d
+ Error Semantico 89:
        En 'case Val:', Val debe ser un valor entero, sin embargo es un %s,
        antes de la linea %d
+ Error Semantico 90:
        returns inconsistentes en el switch, devolviendo %s al mismo
        tiempo, antes de la linea %d
+ Error Semantico 91:
        returns inconsistentes en los bloques de la funcion o del switch,
        devolviendo %s al mismo tiempo, antes de la linea %d
+ Error Semantico 92:
        En la funcion '%s' se espera que se devuelva %s, pero esta tratando
        de devolver %s, antes de la linea %d
+ Error Semantico 93:
        Con el operando ||, se ha tratado de hacer %s || bool, cuando solo
        se puede bool || bool, antes de la linea %d
+ Error Semantico 94:
        Con el operando &&, se ha tratado de hacer %s && bool, cuando solo
        se puede bool && bool, antes de la linea %d
+ Error Semantico 95:
        Con el operando < o >, se ha tratado de hacer %s < entero, cuando
        solo se puede entero < entero, antes de la linea %d
+ Error Semantico 96:
        Con el operando <, se ha tratado de hacer %s < entero, cuando solo
        se puede entero < entero, antes de la linea %d
+ Error Semantico 97:
        Con el operando >, se ha tratado de hacer %s > entero, cuando solo
        se puede entero > entero, antes de la linea %d
+ Error Semantico 98:
        Con el operando +, se ha tratado de hacer %s + entero, cuando solo
        se puede entero + entero, antes de la linea %d
+ Error Semantico 99:
        Con el operando *, se ha tratado de hacer %s * entero, cuando solo
        se puede entero * entero, antes de la linea %d
 + Error Semantico 100:
        El identificador '%s' es una funcion que no devuelve ningun valor y
        se esta tratando de usarla para asignar un valor, antes de la linea
        %d
+ Error Semantico 101:
        El identificador '%s' se esta tratando de redeclarar su tipo, antes
        de la linea %d
```

Anexo

Funcionamiento actual del gestor de errores:

Para poder continuar con al análisis sintáctico aunque hayamos encontrado algún error léxico o sintáctico, hemos incorporado algunas reglas para poder continuar con el análisis:

Errores Léxicos:

El gestor de errores almacena el código del último error léxico que se ha ejecutado y los maneja de tal forma:

- **40, 41, 45 y 46 (finalización de fichero inesperada)** ->
  Pide al analizador lexico otro token (El EOF)
- **42 (&& mal escrito):**
  Devuelve un token de &&.
- **43 (|| mal escrito):**
  Devuelve un token de ||.
- **44 (error de comentario o /=):**
  Como no se pueden diferenciar, se realiza soló el análisis léxico (lanzaría despues un error sintáctico)
- **47 (overflow del entero (val > 32767):**
  Devuelve un token de entero, como si hubiera sido válido (no tiene asignado ningún valor).
- **48 (overflow de la cadena (string > 64 car)):**
  Devuelve un token de cadena, como si hubiera sido válido (no tiene asignado ningún valor).
- **49 (cadena que termina en salto de linea):**
  Devuelve solo un token de EOL (;), lanzando despues un error sintáctico.

Lo usamos para que nunca el sintáctico se quede sin recibir un token.

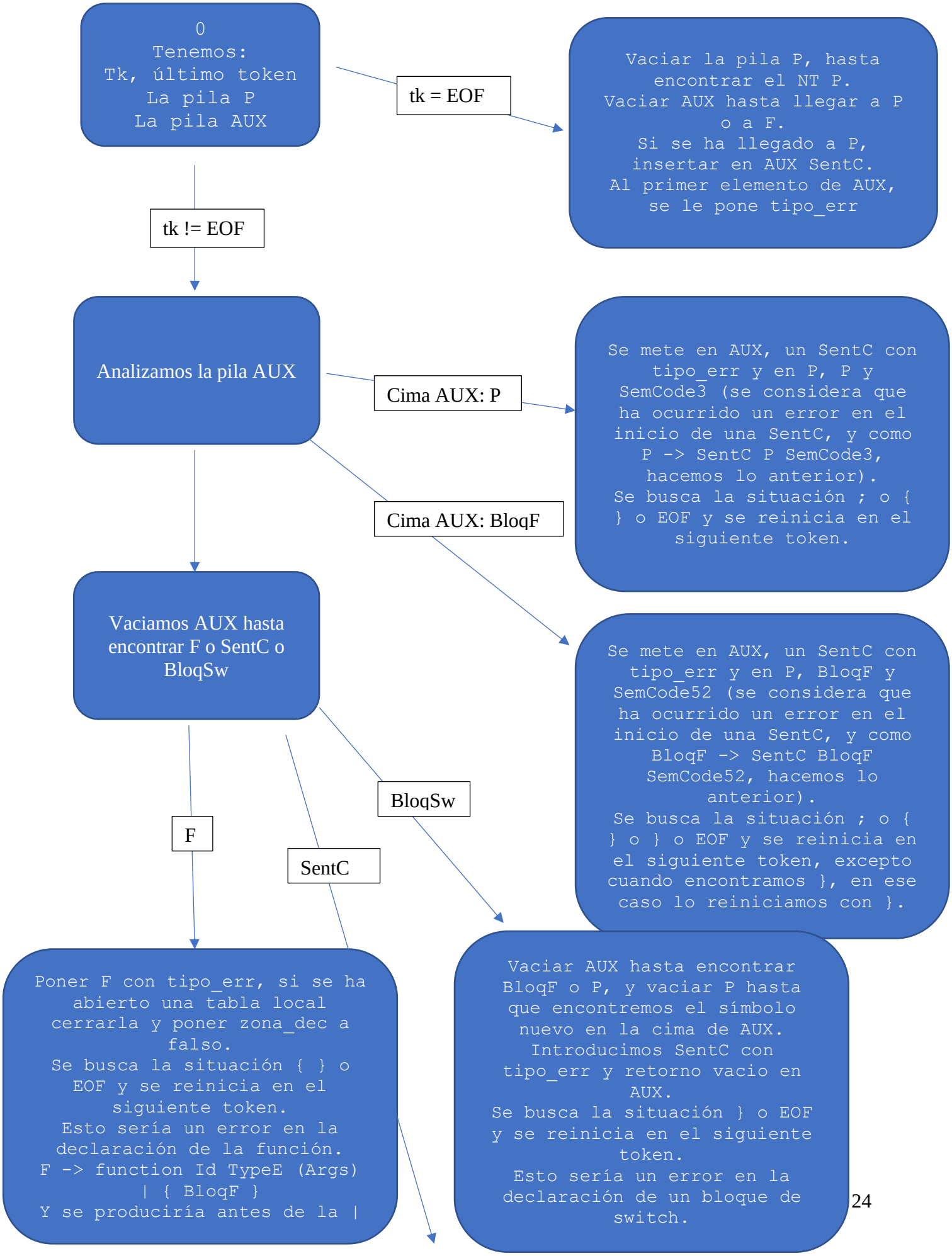
Errores Sintácticos:

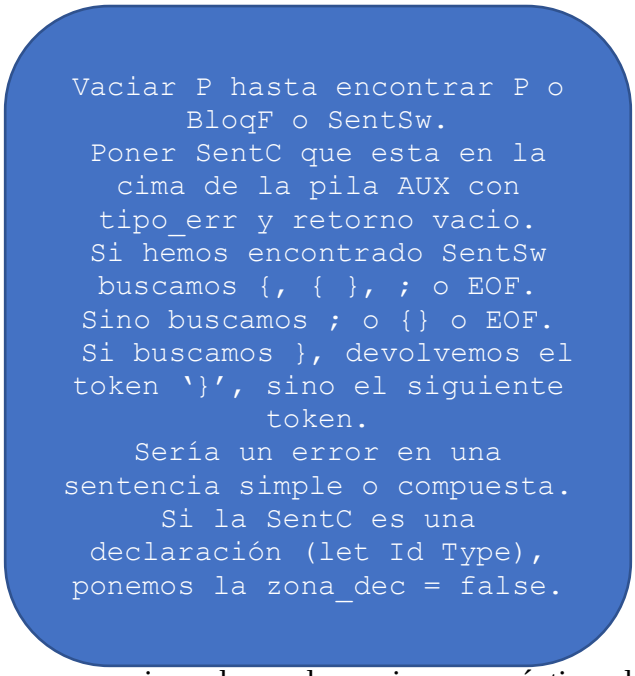
Los maneja haciendo un salto de lineas hasta un punto seguro donde retomar el análisis sintáctico. Durante el salto, continúa el análisis léxico y se pausa el semántico.

Para poder retomar bien el proceso de análisis, debemos manejar bien las pilas P y AUX.

Primero vamos a describir nomenclatura:

- ; , es que buscamos con el analizador léxico hasta encontrar un punto y coma.
- {}, buscamos encontrar una llave abierte y su su sucesiva cerrada (puede ocurrir la situación de { ...{, y lo que hacemos es cerrar primero la segunda y continuamos hasta cerrar la primera o llegar a EOF.
- }, buscamos solo la llave de cierre
  A continuación describimos en forma de grafo como hemos realizado el manejo de las pilas


0
Tenemos:
Tk, último token
La pila P
La pila AUX
tk = EOF
Vaciar la pila P, hasta encontrar el NT P.
Vaciar AUX hasta llegar a P o a F.
Si se ha llegado a P, insertar en AUX SentC.
Al primer elemento de AUX, se le pone tipo_err
tk != EOF
Analizamos la pila AUX
Cima AUX: P
Se mete en AUX, un SentC con tipo_err y en P, P y SemCode3 (se considera que ha ocurrido un error en el inicio de una SentC, y como P -> SentC P SemCode3, hacemos lo anterior).
Se busca la situación ; o { } o EOF y se reinicia en el siguiente token.
Cima AUX: BloqF
Vaciamos AUX hasta encontrar F o SentC o BloqSw
Se mete en AUX, un SentC con tipo_err y en P, BloqF y SemCode52 (se considera que ha ocurrido un error en el inicio de una SentC, y como BloqF -> SentC BloqF SemCode52, hacemos lo anterior).
Se busca la situación ; o { } o } o EOF y se reinicia en el siguiente token, excepto cuando encontramos }, en ese caso lo reiniciamos con }.
F
SentC
BloqSw
Poner F con tipo_err, si se ha abierto una tabla local cerrarla y poner zona_dec a falso.
Se busca la situación { } o EOF y se reinicia en el siguiente token.
Esto sería un error en la declaración de la función.
F -> function Id TypeE (Args) | { BloqF }
Y se produciría antes de la |
Vaciar AUX hasta encontrar BloqF o P, y vaciar P hasta que encontremos el símbolo nuevo en la cima de AUX.
Introducimos SentC con tipo_err y retorno vacio en AUX.
Se busca la situación } o EOF y se reinicia en el siguiente token.
Esto sería un error en la declaración de un bloque de switch.



Los errores semánticos se manejan solo con las acciones semánticas descritas antes.