2022

PRÁCTICA PROCESADORES DEL LENGUAJE

Procesador de Lenguaje JavaScript-PdL

GRUPO 13

INTEGRANTES:

- Mario López Estaire - 160107
- Andrés Bravo Francos - 160203
- Grettell Umpierrez Sardiñas – 210401

UNIVERSIDAD POLITÉCNICA DE MADRID

Índice

OBJETIVOS

La Práctica consistirá en el diseño y construcción de un procesador de lenguaje JavaScript-PdL.

Objetivos comunes implementados

- Un programa está compuesto por funciones, declaraciones y sentencias.
- Las funciones admiten recursividad, pero no anidamiento.
- Tipos de datos: entero, lógicos y cadenas.
- Variables y su declaración. Los identificadores sin declarar son variables globales enteras.
- Expresiones
- Sentencias de asignación, llamada a una función, retorno de una función, condicional simple
- Sentencias de entrada/salida por terminal (print e input)
- Operadores
    - Aritméticos: +, *
    - Relacionales: ==
    - Lógicos: &&
    - Asignación: =

Objetivos específicos implementados

- Sentencias: sentencia de selección múltiple (switch-case) incluyendo la sentencia break.
- Comentarios: comentario de bloque (/**/)
- Cadenas: con comillas dobles (“ ”)
- Operadores especiales: Asignación con multiplicación (*=)
- Técnica de Análisis Sintáctico: Descendente con tablas

EL PROYECTO

Nuestro procesador está implementado en lenguaje Python y consta de los módulos o clases:

- *analizadorLexico.py* y su directorio asociado *objetosAL* correspondientes al Analizador Léxico.
- *analizadorSintactico.py* y sus directorios asociados *objetosASintac* y *objetosASSemantico* correspondientes al Analizador Sintáctico y Semántico debido al método de Análisis Semántico utilizado (Traducción Dirigida por la Sintaxis).
- *analizadorSemantico.py* que inicia el análisis sintáctico y semántico.
- *GestorError.py* en el directorio *objetosGenerales* para la gestión de errores léxicos, sintácticos y semánticos y su volcado en el fichero *errores.txt*.
- *Reader.py* en el directorio *objetosGenerales* para leer del fichero fuente o escribir en los ficheros de salida.
- Directorio *TablaSimbolos* correspondiente a la Tabla de Símbolos.
- Directorio *Ficheros Fuente* donde se encuentra el *fichero_fuente.txt* donde se debe ‘escribir’ el código a procesar.
- Directorio *Ficheros Salida* donde se encuentran los ficheros de salida *errores.txt*, *parse.txt*, *tabla.txt* y *tokens.txt*.
- *procesador.py* que hace de main del procesador.

  Para la ejecución del procesador se debe ejecutar procesador.py.

ANALIZADOR LÉXICO

TOKENS

Generales	***Operadores***	***Valores de Datos***	**Declaraciones**
<abrirParentesis, > <cerrarParentesis, > <abrirCorchete, > <cerrarCorchete, > <coma, > <ptoComa, > <dosPuntos, > <eof, >	<opAritmetico,1 > <opAritmetico, 2 > <opRelacional,1 > <opLogico,1 > <asigMultiplicacion, - > <asignacion, - > Op_aritmético 1: * 2: + Op_relacional 1: == Op_lógico 1: &&	<cteEntera, valor> <cadena, “lexema”> < true, > < false, >	<id, posTs> <let, >
Tipos de Datos *(Palabras reservadas)*	***Funciones*** *(Palabras reservadas)*	***Sentencias*** *(Palabras reservadas)*	***E/S*** *(Palabras reservadas)*
<int, > <boolean, > <string, >	<function, > <return, >	<if, > <switch, > <case, > <default, > <break, >	<input, > <print, >

GRAMÁTICA

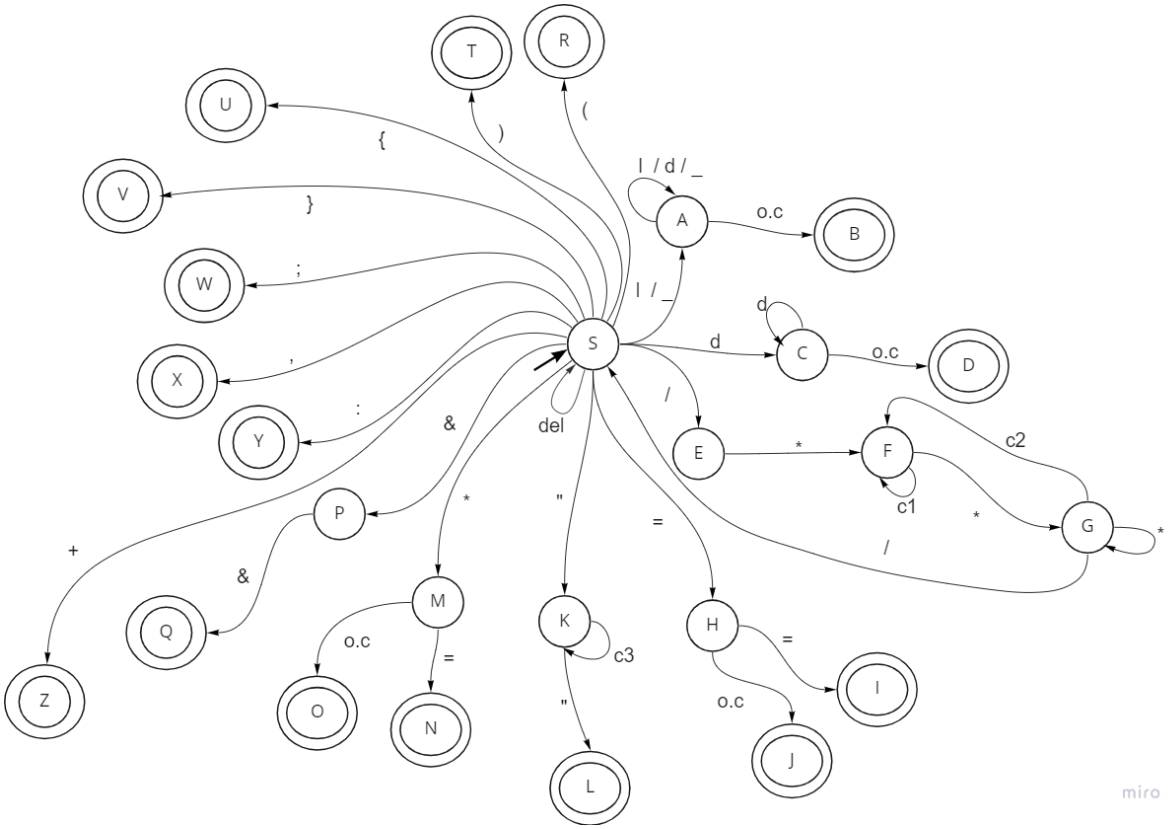
$S \rightarrow delS \mid lA \mid \_A \mid dB \mid = C \mid “D \mid * E \mid \&F \mid /$
$G \mid (\mid) \mid \{ \mid \} \mid ; \mid , \mid + \mid :$

$A \rightarrow lA \mid dA \mid \_A \mid o.c$
$B \rightarrow dB \mid o.c$
$C \rightarrow = \mid o.c$
$D \rightarrow c3D \mid “$
$E \rightarrow = \mid o.c$
$F \rightarrow \&$
$G \rightarrow * H$
$H \rightarrow c1H \mid * H’$
$H’ \rightarrow c2H \mid * H’ \mid /S$

LEYENDA:

l: {a-z,A_Z}
d: {0 - 9}
$o.c$: cualquier carácter no contemplado en el nodo
$c1$: todos los caracteres – {*}
$c2$: todos los caracteres – {*,/}
$c3$: todos los caracteres – {“}
del: tab, espacio, eol

AUTÓMATA



ACCIONES SEMÁNTICAS

1. **S→S**: leer();
2. **S→A**: lexema = c; leer();
3. **A→A**: lexema = lexema + c; leer();
4. **A→B**:
   if(lexema == palabraReservada) then generarToken(lexema, -);
   else{
   pos=BuscarLugarTS(lexema);
   if(pos ¡= null) then generarToken(id, pos);
   else{
   pos=insertarIdTS(lexema);
   generarToken(id,pos);
   }
5. **S→C**: valor = char_int(d); leer();
6. **C→C**: valor = valor * 10 + char_int(d); leer();
18. **S→H**: leer();
19. **H→I**: generarToken(opRelacional, 1);leer();
20. **H→J**: generarToken(asignación, -);
21. **S→M**: leer();
22. **M→N**: generarToken(asigMultiplicacion, -);leer();
23. **M→O**: generarToken(opAritmetico, 1);
24. **S→P**: leer();
25. **P→Q**: generarToken(opLogico, 1); leer();
26. **S→R**: generarToken(abrirParentesis, -

7. **C→D**: if (valor > 32767) then error (60);
   else generarToken(cte_entera, valor);
8. **S→E**: leer();
9. **E→F**: leer();
10. **F→F**: leer();
11. **F→G**: leer();
12. **G→F**: leer();
13. **G→G**: leer();
14. **G→S**: leer();
15. **S→K**: lexema = " "; contador=0; leer();
16. **K→K**: lexema = lexema + c; leer();contador++;
17. **K→L**: if(contador > 64) then error (61);
    else generarToken(cadena, lexema);

);leer();

27. **S→T**: generarToken(cerrarParentesis, -); leer();
28. **S→U**: generarToken(abrirCorchete, -); leer();
29. **S→V**: generarToken(cerrarCorchete, -); leer();
30. **S→W**: generarToken(ptoComa, -); leer();
31. **S→X**: generarToken(coma, -); leer();
32. **S→Y**: generarToken(dosPuntos, -); leer();
33. **S→Z**: generarToken(opAritmetico, 2); leer();

MATRIZ AFD

	l	d	_	c1	c2	c3	/	*	=	"	&	(	)	{	}	;	,	:	+	del	o.c
S	A 2	C 5	A 2				E 8	M 21	H 18	K 15	P 24	R 26	T 27	U 28	V 29	W 30	X 31	Y 32	Z 33	S 1	50
A	A 3	A 3	A 3																		B 4
B																					
C		C 6																			D 7
D																					
E								F 9													51
F				F 10				G 11													52
G					F 12		S 14	G 13													53
H									I 19												J 20
I																					
J																					
K						K 16				L 17											54
L																					
M								N 22													O 23
N																					
O																					
P											Q 25										55
Q																					
R																					
T																					
U																					
V																					
W																					
X																					
Y																					
Z																					

*Los estados encerrados EN VERDE representan estados finales (y por tanto no reciben nada). Los números EN ROJO representan ERRORES.

TABLA DE SÍMBOLOS

La Tabla de Símbolos es implementada utilizando tres clases EntradaTablaSimbolo, TablaSimbolo y

GestorTablaSimbolo. La instancia de GestorTablaSimbolo se ocupará de gestionar la creación y eliminación de tablas, así como la búsqueda o inserción en las tablas de símbolos de entradas u atributos de estas como tipo. Tiene, a grandes rasgos, dos estructuras de tipo dictionary de Python que contendrán los id (key) de cada tabla asociado al id de la tabla anterior en uno de los casos y a la instancia de TablaSimbolo en el otro. Cada instancia de TablaSimbolo en el gestor tendrá a su vez otras dos estructuras dictionary donde se asocia el lexema (key) de cada entrada a su id en uno de los casos, y el id (key) de la entrada a la instancia de EntradaTablaSimbolo en el otro. Se ocupa de las tareas de búsqueda o inserción de sus entradas.

Las estructuras dictionary mencionadas son una colección ordenada, modificable y que no admite duplicados que almacenan los valores de datos en pares clave:valor. En estas los elementos tienen un orden definido y se puede hacer referencia a ellos mediante el nombre de la clave. Estas estructuras son determinantes para lograr que la búsqueda e inserción de tablas o entradas en nuestra Tabla de Símbolos sea eficiente.

Cada entrada de un identificador es de tipo EntradaTablaSimbolo y tiene todos los atributos posibles inicializados a valores descartables en caso de no utilizarse según el tipo al que pertenece:

Lexema	Tipo	Despl	NumParam	TipoRetorno	TipoParametros	ModoParametros	EtiqFuncion	Tabla (id)

Las ‘columnas’ TipoParametros y ModoParametros son listas de tipos que podrían representarse con subtablas

Por tanto, la tabla es homogénea en atributos dado que siempre tiene los mismos sin importar el tipo.

A continuación, se realiza una descripción detallada:

Clase Tipo

Atributos estáticos de la clase:

- UNDEFINED : tiene como valor la cadena “undefined”.
- ENTERO : tiene como valor la cadena “tipo_entero”
- CADENA : tiene como valor la cadena “tipo_cadena”
- LOGICO : tiene como valor la cadena “tipo_logico”
- FUNCION : tiene como valor la cadena “tipo_funcion”
- VACIO : tiene como valor la cadena “tipo_vacio”
- OK : tiene como valor la cadena “tipo_ok”
- ERROR : tiene como valor la cadena “tipo_error”

Clase EntradaTablaSímbolos

Atributos estáticos de la clase:

- ENTRADA_ID : empieza en 0 y aumenta en 1 con cada objeto de la clase instanciado.
- NUM_ETIQUETA : empieza en 1 y aumenta en 1 con cada objeto de la clase instanciado que llame al método setEtiqueta(). Solo se utiliza en los atributos de tipo *tipo_funcion*.

Cada entrada de nuestra TS tiene los siguientes atributos:

- **lexema**: lexema del identificador que se obtiene a través del Analizador Léxico
- **id**: se refiere a un valor numérico se obtiene a partir de un atributo estático ENTRADA_ID sin importar la tabla. Es el número que identifica a cada entrada en todas las Tablas de Símbolos existentes. Es equivalente a posición.
- **tipo**: es el tipo del identificador. Su valor está asociado a alguno de los valores de los atributos estáticos

de la clase Tipo. Es inicializado en el valor *undefined* pero actualizado por el Analizador Semántico a *tipo_entero*, *tipo_cadena*, *tipo_logico* o *tipo_funcion*.

- **despl**: es el valor de desplazamiento de los identificadores de tipo entero, cadena o lógico en su tabla de símbolos. Es inicializado en -1 y actualizado por el Analizador Semántico para estos tipos de identificadores. Si el identificador es de *tipo_funcion* el valor de este atributo permanecerá en -1.
- **numParametros**: es el número de parámetros de la función. Es inicializado en -1 y actualizado por el Analizador Semántico cuando el tipo del identificador es *tipo_funcion* según la cantidad de parámetros declarados.
- **tipoParametros**: es un atributo de tipo lista. Es inicializado como una lista vacía y actualizado por el Analizador Semántico cuando el identificador es de *tipo_funcion* según el tipo de los parámetros declarados. La lista puede estar vacía indicando que no se ha actualizado porque el identificador no es *tipo_funcion* o que no se declararon parámetros. Solo se tiene en cuenta su valor para los identificadores *tipo_funcion* por lo que esta ambigüedad no tiene consecuencias indeseadas.
- **modoParametros**: es un atributo de tipo lista. Es inicializado como una lista vacía y actualizado por el Analizador Semántico cuando el identificador es de *tipo_funcion* según el modo de paso de los parámetros declarados. En este lenguaje sólo existe el tipo de paso por valor (1). La lista puede estar vacía indicando que no se ha actualizado porque el identificador no es *tipo_funcion* o que no se declararon parámetros. Solo se tiene en cuenta su valor para los identificadores *tipo_funcion* por lo que esta ambigüedad no tiene consecuencias indeseadas.
- **tipoDevuelto**: es el tipo devuelto por los identificadores de *tipo_funcion*. Su valor está asociado a alguno de los valores de los atributos estáticos de la clase Tipo. Es inicializado en el valor *undefined* pero actualizado por el Analizador Semántico a *tipo_entero*, *tipo_cadena* o *tipo_logico* dependiendo del tipo declarado o en su defecto a *tipo_vacio* si no se declara tipo de retorno. Si el identificador no es *tipo_funcion* el valor de este atributo permanecerá en *undefined*.
- **etiqueta** : es un identificador que se inicializa como una cadena vacía y que es actualizado por el Analizador Semántico cuando el identificador es de *tipo_funcion* utilizando el siguiente formato:
    - "Et" + **lexema** + NUM_ETIQUETA

      Cada vez que se actualiza el tipo de un identificador a *tipo_funcion* se obtiene la etiqueta con este formato utilizando el método setEtiqueta() de la clase. En este también se actualiza NUM_ETIQUETA aumentando en 1 con cada llamada.
- **tabla** : es el id de la Tabla a la que pertenece la entrada. Se inicializa en 0 pero es actualizado por la clase TablaSimbolos cuando es llamado el método insertarValor() por el método insertarEntrada() o insertarEntradaTG() del GestorTablaSimbolos que utiliza el Analizador Léxico.

Cada entrada de nuestra TS tiene los siguientes métodos:

- Métodos getter y setter de cada parámetro antes mencionado. La lógica es la misma exceptuando los siguientes casos:
    - **setTipo**(*tipo*) : antes de modificar el tipo del identificador a *tipo* comprueba si es *tipo_funcion* y en caso afirmativo inicializa a 0 **numParametros** y llama a **setEtiqueta**()
    - **setTipoParametros**(tp) : antes de agregar el tipo *tp* a la lista **tipoParametros** aumenta el **numParametros** en 1.
    - **setModoParametros**(*metodo*) : agregar el método a la lista *modoParametros*.
    - **setEtiqueta**() : cambia el valor de etiqueta al formato antes explicado y luego aumenta en 1 el atributo estático NUM_ETIQUETA.
- **toString**(): devuelve una cadena de salida con el formato que debe mostrarse en el fichero tabla.txt. En todos los casos devuelve lexema y de los atributos tipo. El resto de atributos dependería del tipo del identificador.

Clase TablaSímbolos

Atributo estático de la clase:

- NTABLAS: empieza en 0 y aumenta en 1 con cada objeto de la clase instanciado.

Cada tabla instancia de TablaSimbolos tiene los siguientes atributos:

- **id**: obtenido a partir de NTABLA actualizado antes de dar valor a este atributo (por tanto, el id de la primera tabla, la tabla Global, es 1).
- **indices**: es una estructura de Python de tipo dict (equivalente a HashMap de java) que contiene cada lexema (key) de las entradas de la tabla asociado a su id (value). Se utiliza para obtener el id de determinado lexema de la tabla para casos como obtener la entrada utilizando el id (key) en la estructura **entradas**.
- **entradas**: es una estructura de Python de tipo dict (equivalente a HashMap de java) que contiene cada id (key) de las entradas de la tabla asociado al objeto de EntradaTablaSimbolos correspondiente.
- **ultimoDespl** : es el valor del último desplazamiento en la tabla. Su valor es modificado por el Analizador Semántico cuando la entrada es de tipo_entero, tipo_cadena o tipo_logico sumando el tamaño del tipo. Inicializado en 0, cada vez que se inserta una entrada de estos tipos se utiliza este atributo para actualizar su atributo **despl** y luego se realiza la actualización de **ultimoDespl** como fue explicado anteriormente.

Cada tabla instancia de TablaSimbolos tiene los siguientes métodos:

- Métodos getter de los parámetros **id** y **ultimoDespl** y setter de este último.
- **toString**() : devuelve una cadena de salida con el formato que debe mostrarse en el fichero tabla.txt de la tabla. Concatena todas las salidas del método **toString**() de cada entrada de la estructura **entradas**.
- **insertarValor**(*lex*, *idTabla*) : crea una instancia de EntradaTablaSimbolo con lexema *lex*. Utiliza el método **setTabla** de la entrada para establecer su atributo **tabla** a *idTabla*. Por último, añade la entrada a **indice** y **entrada**.
- **buscarTSNombre**(*lexema*) : busca el id de la entrada en **indice** utilizando *lexema*, y con este obtiene la entrada en **entradas**. Si *lexema* no está en **indice** devuelve False.
- **buscarEntradaID**(id) : busca el *id* en **entradas**. Si no está devuelve None (null)

Clase GestorTablaSímbolos

El gestor instancia de GestorTablaSímbolos tiene los siguientes atributos:

- **currentTablaID** : es el id de la última tabla creada. Se inicializa en -1. Cada vez que se crea una tabla toma el valor del atributo **id** de esta. Si la tabla es 'borrada' toma el valor de la anterior tabla. En las especificaciones de este lenguaje no hay anidamiento por tanto solo habrán máximo dos tablas activas (global y local) y el valor de currentTablaID volverá siempre a 1 (id primera tabla TG) una vez se elimine la tabla local, y a -1 cuando se elimine la TG.
- **writerFichero** : es una instancia de la clase Reader para escribir cada tabla en el fichero tabla.txt según el formato indicado.
- **bloqueTS** : es una estructura de Python de tipo dict (equivalente a HashMap de java) que contiene cada id (key) de las tablas asociado al id (value) de la tabla anterior. Esta estructura es utilizada por el método removeTable() para actualizar **currentTablaID** a la tabla anterior antes de eliminar una tabla.
- **listaTS** : es una estructura de Python de tipo dict (equivalente a HashMap de java) que contiene cada id (key) de las tablas asociado a la instancia de TablaSimbolos (value).
- **ultimaTS** : es la última instancia de TablaSimbolos creada (tabla actual).

El gestor instancia de GestorTablaSímbolos tiene los siguientes métodos:

- **crearTabla**() : crea una instancia de TablaSimbolos. Añade la tabla a **listaTS** y **currentTablaID** a **bloqueTS** con el **id** de la nueva tabla como key en ambos casos. Actualiza **currentTablaID** al valor del atributo **id** de la nueva tabla.
- **getTablaActual**() : devuelve la entrada de **listaTS** asociada a **currentTablaID**.
- **removeTable**() : llama al metodo **toString**() de la tabla actual obtenida con **getTablaActual**() y escribe la salida en el fichero tabla.txt con **writerFichero**.
- **buscarEntradaPorID**(*entId*) : busca la entrada en la **ultimaTS** llamando a **buscarEntradaID**() con *entId*. Si no obtiene la entrada la busca en todas las tablas activas (solo la tabla global en este caso porque no hay anidamiento). Devuelve la entrada de **id** *entId* o None (null) si no la encuentra.
- **buscarEntradaPorLexema**(*lexema*) : busca la entrada en la **ultimaTS** llamando a **buscarTSNombre**() con *lexema*. Si no obtiene la entrada la busca en todas las tablas activas (solo la tabla global en este caso porque no hay anidamiento). Devuelve la entrada de **id** *entId* o None (null) si no la encuentra.
- **buscarEntradaTablaActuaLexema**(*lexema*) : busca la entrada en la **ultimaTS** llamando a **buscarTSNombre**() con *lexema*. Si no obtiene la entrada devuelve False.
- **insertarEntrada**(*lexema*) : llama al método **insertarValor**() de la tabla actual con *lexema* e **id** de la tabla actual.
- **insertarEntradaTG**(*lexema*) : llama al método **insertarValor**() de la tabla global (**id** = 1) con *lexema* e **id** de la tabla global.
- **insertarTipoTamTS**(*entID*, *tipo*, *tamanho*) : cambia el tipo de una entrada de **id** *entID* de *undefined* a *tipo* utilizando **setTipo**(). Esta instancia de EntradaTablaSimbolos es obtenida con el método **buscarEntradaPorID**() del gestor con *entID*. Además, si el parámetro *tamanho* no es False llama al método **setDespl**() de la entrada con el valor obtenido con **getUltimoDespl**() de la tabla de la entrada y actualiza el valor de próximo desplazamiento de esa tabla con *tamanho* sumado al desplazamiento actual.
- **insertarTipoParametros**(*entID*, *tipoLista*) : inserta cada elemento de *tipoLista* en la lista del atributo **tipoParametros** utilizando **setTipoParametros**() de la instancia de EntradaTablaSimbolos obtenida con el método **buscarEntradaPorID**() del gestor con *entID*.
- **insertarTipoDevuelto**(*entID*, *tipo*) : inserta *tipo* como atributo **tipoDevuelto** utilizando **setTipoDevuelto**() de la instancia de EntradaTablaSimbolos obtenida con el método **buscarEntradaPorID**() del gestor con *entID*.
- **buscarTipo**(*entID*) : busca y devuelve el **tipo** de la entrada instancia de EntradaTablaSimbolos obtenida con el método **buscarEntradaPorID**() del gestor con *entID*.
- **buscarTipoParametros**(*entID*) : busca y devuelve el **tipoDevuelto** de la entrada instancia de EntradaTablaSimbolos obtenida con el método **buscarEntradaPorID**() del gestor con *entID*.

La Tabla de Símbolos diseñada se vuelca en el fichero tabla.txt con el siguiente formato:

Línea con el número de la TS:

Esta línea se utiliza como encabezado para comenzar cada TS. El formato de esta línea es:

"CONTENIDOS DE LA TABLA # " + *número_de_la_TS* + " :" + **\n\n**

- **\n** : Salto de línea
- *número_de_la_TS* : número correspondiente a la TS actual

Línea del lexema

"**\t*** LEXEMA :\t" + *lexema_del_id* + **\n**

- **\n** : Salto de línea
- **\t** : Tabulador

- *lexema_del_id* : nombre del identificador

Línea del atributo

"**\t**ATRIBUTOS : **\n**" + **\t** + "+ " + *nombre_atributo* + " :" + **\t** + *valor_atributo* + **\n** + **\t** + "---------- ----------"

- **\n** : Salto de línea
- **\t** : Tabulador
- *nombre_atributo* : puede tomar los siguiente valores:
  - o Despl
  - o Tipo
  - o numParam
  - o TipoPararmXX
  - o ModoParamXX
  - o TipoRetorno
  - o EtiqFuncion
- *valor_atributo* : valor que tiene el atributo *nombre_atributo*

Un ejemplo del contenido de una tabla generada por nuestro procesador:

CONTENIDOS DE LA TABLA # 1 :

* LEXEMA : 'a'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 0
---------- ----------

ANALIZADOR SINTÁCTICO

GRAMÁTICA

// Axioma general
1. $A \rightarrow B\ A$
2. $A \rightarrow FA$
3. $A \rightarrow eof$
// Sentencias compuestas
4. $B \rightarrow if\ (\ E\ )\ S$
5. $B \rightarrow let\ id\ T\ N$
6. $B \rightarrow S$
	19. $R \rightarrow U\ R'$
	20. $R' \rightarrow \lambda$
	21. $R' \rightarrow ==\ U\ R'$ //Operaciones relacionales
	22. $U \rightarrow V\ U'$
	23. $U' \rightarrow \lambda$
	24. $U' \rightarrow +\ V\ U'$ //Operaciones aritméticas (suma)

7. $B \rightarrow switch\ (E)\ \{\ Z\ \}$

// Sentencias simples

8. $S \rightarrow id\ S';$
9. $S \rightarrow print\ \ E;$
10. $S \rightarrow input\ id;$
11. $S \rightarrow return\ X;$
12. $S \rightarrow break;$
13. $S' \rightarrow = E$
14. $S' \rightarrow * = E$
15. $S' \rightarrow (\ L\ )$

// Expresiones

16. $E \rightarrow R\ E'$
17. $E' \rightarrow \lambda$
18. $E' \rightarrow \&\&\ R\ E'$ // Operaciones lógicas
25. $V \rightarrow P\ V'$
26. $V' \rightarrow \lambda$
27. $V' \rightarrow *\ P\ V'$ // Operaciones aritméticas (producto)

// Operandos

28. $P \rightarrow id\ P'$
29. $P \rightarrow (\ E\ )$
30. $P \rightarrow cteEntera$
31. $P \rightarrow cadena$
32. $P \rightarrow true$
33. $P \rightarrow false$
34. $P' \rightarrow \lambda$
35. $P' \rightarrow (\ L\ )$

// Argumentos de función

36. $L \rightarrow E\ Q$
37. $L \rightarrow \lambda$
38. $Q \rightarrow ,E\ Q$
39. $Q \rightarrow \lambda$

// Valor de retorno

40. $X \rightarrow E$
41. $X \rightarrow \lambda$

// Tipos de variables

42. $T \rightarrow int$
43. $T \rightarrow boolean$
44. $T \rightarrow string$

// Declaración de funciones

45. $F \rightarrow function\ id\ H\ (\ D\ )\ \{\ C\ \}$
49. $D \rightarrow \lambda$
50. $K \rightarrow ,T\ id\ K$
51. $K \rightarrow \lambda$
52. $C \rightarrow B\ C'$
53. $C' \rightarrow B\ C'$
54. $C' \rightarrow \lambda$

// Inicialización de identificadores

55. $N \rightarrow ;$
56. $N \rightarrow = E\ ;$
57. $N \rightarrow * = E;$

// Cuerpo de switch

58. $Z \rightarrow case\ cteEntera : O\ Z'$
59. $Z \rightarrow default : O$
60. $O \rightarrow B\ O$
61. $O \rightarrow \lambda$

46. $H \rightarrow T$ 47. $H \rightarrow \lambda$ 48. $D \rightarrow T\ id\ K$	62. $Z' \rightarrow Z$ 63. $Z' \rightarrow \lambda$

FIRST Y FOLLOW

	FIRST	FOLLOW
A	if | let | switch | id | print | input | return | function | eof	eof
B	if | let | switch | id | print | input | return | break	if | let | switch | id | print | input | return | break | function | eof | } | case | default
S	id | print | input | return | break	if | let | switch | id | print | input | return | break | function | eof | } | case | default
S'	= | *= | (	;
E	id | (| cteEntera | cadena | true | false	) | ; | ,
E'	&& | λ	) | ; | ,
R	id | (| cteEntera | cadena | true | false	&& |) | ; | ,
R'	== | λ	&& |) | ; | ,
U	id | (| cteEntera | cadena | true | false	== | && |) | ; | ,
U'	+ | λ	== | && |) | ; | ,
V	id | (| cteEntera | cadena | true | false	+ | == | && |) | ; | ,
V'	* | λ	+ | == | && |) | ; | ,
P	id | (| cteEntera | cadena | true | false	* | + | == | && |) | ; | ,
P'	(| λ	* | + | == | && |) | ; | ,
L	id | (| cteEntera | cadena | true | false | λ	)
Q	, | λ	)
X	id | (| cteEntera | cadena | true | false | λ	;
T	int | boolean | string	id | (| ; | = | *=
F	function	if | let | switch | id | print | input | return | break | function | eof
H	int | boolean | string | λ	(
D	int | boolean | string | λ	)
K	, | λ	)
C	if | let | switch | id | print | input | return | break	}

C'	*if	let	switch	id	print	input	return	break	λ*	*}*
N	*;	=	*=*	*if	let	switch	id	print	input	return	break	function	eof	}	case	default*
Z	*case	default*	*}*
O	*if	let	switch	id	print	input	return	λ*	*break	case	default	}*
O'	*break	λ*	*case	default	}*

Justificación

Verificación condición LL

Las reglas de A cumplen la condición LL porque:

- $First(BA) \cap First(FA) = \emptyset$ porque $\{ if \mid let \mid switch \mid id \mid print \mid input \mid return \mid break \} \cap \{ function \} = \emptyset$
- $First(FA) \cap First(eof) = \emptyset$ porque $\{ function \} \cap \{ eof \} = \emptyset$
- $First(BA) \cap First(eof) = \emptyset$ porque $\{ if \mid let \mid switch \mid id \mid print \mid input \mid return \mid break \} \cap \{ eof \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de B cumplen la condición LL porque:

- $First(if\ (E)\ S) \cap First(let\ id\ T\ N\ ) = \emptyset$ porque $\{ if \} \cap \{ let \} = \emptyset$
- $First(if\ (E)\ S) \cap First(S\ ) = \emptyset$ porque $\{ if \} \cap \{ id \mid print \mid input \mid return \} = \emptyset$
- $First(if\ (E)\ S) \cap First(switch\ (E)\ \{ Z \}\ ) = \emptyset$ porque $\{ if \} \cap \{ switch \} = \emptyset$
- $First(let\ id\ T\ N) \cap First(S\ ) = \emptyset$ porque $\{ let \} \cap \{ id \mid print \mid input \mid return \} = \emptyset$
- $First(let\ id\ T\ N) \cap First(\ switch\ (E)\ \{ Z \}\ ) = \emptyset$ porque $\{ let \} \cap \{ switch \} = \emptyset$
- $First(S) \cap First(\ switch\ (E)\ \{ Z \}\ ) = \emptyset$ porque $\{ id \mid print \mid input \mid return \mid break \} \cap \{ switch \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de S cumplen la condición LL porque:

- $First(id\ S';) \cap First(print\ E\ ;) = \emptyset$ porque $\{ id \} \cap \{ print \} = \emptyset$
- $First(id\ S';) \cap First(\ input\ id\ ;) = \emptyset$ porque $\{ id \} \cap \{ input \} = \emptyset$
- $First(id\ S';) \cap First(return\ X\ ;) = \emptyset$ porque $\{ id \} \cap \{ return \} = \emptyset$
- $First(id\ S';) \cap First(break\ ;) = \emptyset$ porque $\{ id \} \cap \{ break \} = \emptyset$
- $First(print\ E;) \cap First(input\ id\ ;) = \emptyset$ porque $\{ print \} \cap \{ input \} = \emptyset$
- $First(print\ E;) \cap First(\ return\ X\ ;) = \emptyset$ porque $\{ print \} \cap \{ return \} = \emptyset$
- $First(print\ E;) \cap First(\ break\ ;) = \emptyset$ porque $\{ print \} \cap \{ break \} = \emptyset$
- $First(\ input\ id\ ;) \cap First(\ return\ X\ ;) = \emptyset$ porque $\{ input \} \cap \{ return \} = \emptyset$
- $First(\ input\ id\ ;) \cap First(\ break\ ;) = \emptyset$ porque $\{ input \} \cap \{ break \} = \emptyset$
- $First(\ return\ X\ ;) \cap First(\ break\ ;) = \emptyset$ porque $\{ return \} \cap \{ break \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de S' cumplen la condición LL porque:

- $First(= E) \cap First(*= E) = \emptyset$ porque $\{ = \} \cap \{ *= \} = \emptyset$

- $First(= E) \cap First(\ (\ L\ )\ ) = \emptyset$ porque $\{=\} \cap \{\ (\ \} = \emptyset$
- $First(*= U) \cap First(\ (\ L\ )\ ) = \emptyset$ porque $\{*=\} \cap \{\ (\ \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de E cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de E' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ \&\&\ R\ E')\cap Follow(\ E'\ ) = \emptyset$ porque $\{\ \&\&\} \cap \{\ )\ |\ ;\ |\ ,\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de R cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de R' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(==\ U\ R')\cap Follow(\ R'\ ) = \emptyset$ porque $\{==\} \cap \{\ \&\&\ |\ )\ |\ ;\ |\ ,\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de U cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de U' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(+\ V\ U')\cap Follow(\ U'\ ) = \emptyset$ porque $\{+\} \cap \{==\ |\ \&\&\ |\ )\ |\ ;\ |\ ,\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de V cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de V' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(*\ P\ V')\cap Follow(\ V'\ ) = \emptyset$ porque $\{*\} \cap \{+\ |\ ==\ |\ \&\&\ |\ )\ |\ ;\ |\ ,\ \} = \emptyset$ por tanto se

cumple la condición LL.

Las reglas de P cumplen la condición LL porque:

- $First(\ id\ P'\ ) \cap First(\ (\ E\ )\ ) = \emptyset$ porque $\{\ id\ \} \cap \{\ (\ \} = \emptyset$
- $First(\ id\ P'\ ) \cap First(\ cteEntera\ ) = \emptyset$ porque $\{\ id\ \} \cap \{\ cteEntera\ \} = \emptyset$
- $First(id\ P') \cap First(\ true\ ) = \emptyset$ porque $\{\ id\ \} \cap \{\ true\ \} = \emptyset$
- $First(id\ P') \cap First(\ false\ ) = \emptyset$ porque $\{\ id\ \} \cap \{\ false\ \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de P' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ (\ L)\ ) \cap Follow(\ P'\ ) = \emptyset$ porque $\{\ (\ \} \cap \{\ *\ |\ +\ |\ ==\ |\ \&\&\ |\ )\ |\ ;\ |\ ,\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de L cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(E\ Q) \cap Follow(\ L\ ) = \emptyset$ porque $\{\ id\ |\ (\ |\ cteEntera\ |\ cadena\ |\ true\ |\ false\ \} \cap \{\ )\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de Q cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ ,E\ Q\ ) \cap Follow(\ Q\ ) = \emptyset$ porque $\{\ ,\ \} \cap \{\ )\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de X cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ E\ ) \cap Follow(\ X\ ) = \emptyset$ porque $\{\ id\ |\ (\ |\ cteEntera\ |\ cadena\ |\ true\ |\ false\ \} \cap \{\ ;\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de T cumplen la condición LL porque:

- $First(\ int\ ) \cap First(\ boolean\ ) = \emptyset$ porque $\{\ int\ \} \cap \{\ boolean\ \} = \emptyset$
- $First(\ int\ ) \cap First(\ string\ ) = \emptyset$ porque $\{\ int\ \} \cap \{\ string\ \} = \emptyset$
- $First(\ boolean\ ) \cap First(\ string\ ) = \emptyset$ porque $\{\ boolean\ \} \cap \{\ string\ \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de F cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de H cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos

terminales en común

- Como una de las reglas es λ
  - $First(\ T\ ) \cap Follow(\ H\ ) = \emptyset$ porque $\{\ int \mid boolean \mid string\ \} \cap \{\ ;\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de D cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ T\ \ id\ K\ ) \cap Follow(\ D\ ) = \emptyset$ porque $\{\ int \mid boolean \mid string\ \} \cap \{\ )\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de K cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ ,T\ \ id\ K\ ) \cap Follow(\ K\ ) = \emptyset$ porque $\{\ ,\ \} \cap \{\ )\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de C cumplen la condición LL porque:

- Solamente tiene una regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de C' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ B\ C\ ') \cap Follow(\ C'\ ) = \emptyset$ porque $\{\ if \mid let \mid switch \mid id \mid print \mid input \mid break\ \} \cap \{\ \}\ \} = \emptyset$ por tanto se cumple la condición LL.

Las reglas de N cumplen la condición LL porque:

- $First(\ ;\ ) \cap First(= E\ ) = \emptyset$ porque $\{\ ;\ \} \cap \{= \} = \emptyset$
- $First(\ ;\ ) \cap First(*= E\ ) = \emptyset$ porque $\{\ ;\ \} \cap \{ *= \} = \emptyset$
- $First(= E\ ) \cap First(\ *= E\ ) = \emptyset$ porque $\{ = \} \cap \{ *= \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de Z cumplen la condición LL porque:

- $First(\ case\ cteEntera : O\ Z') \cap First(default : O\ \ ) = \emptyset$ porque $\{\ case\ \} \cap \{\ default\ \} = \emptyset$
- No hay regla λ por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

Las reglas de O cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ B\ O\ ) \cap Follow(\ O\ ) = \emptyset$ porque $\{\ if \mid let \mid switch \mid id \mid print \mid input \mid return \mid break\ \} \cap \{\ case \mid default \mid\ \}\ \} = \emptyset$

Las reglas de Z' cumplen la condición LL porque:

- Solamente hay dos reglas y una de ellas es λ por tanto no hay que comprobar si los first tienen símbolos terminales en común
- Como una de las reglas es λ
  - $First(\ Z\ ) \cap Follow(\ Z'\ ) = \emptyset$ porque $\{\ case \mid default\ \} \cap \{\ \}\ \} = \emptyset$

La gramática obtenida para el Analizador Sintáctico Descendente LL(1) con Tablas es LL(1) porque:

- Para cada no terminal para el que haya más de una regla de producción, dichas reglas no derivan un mismo terminal (la gramática está factorizada).
- Cumple la condición LL
- No existe recursividad por la izquierda

TABLA A.S DESCENDENTE (LL)

	if	let	switch	id	print	input	return	function	break	cteEntera	cadena	true	false	==	&&	;	,	(	)	{	}	+	*	=	*=	int	boolean	string	case	default	:	eof
A	1	1	1	1	1	1	1	2	1	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	101	3
B	4	5	7	6	6	6	6	102	6	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102	102
S	103	103	103	8	9	10	11	103	12	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103	103
S'	104	104	104	104	104	104	104	104	104	104	104	104	104	104	104	104	104	15	104	104	104	104	104	13	14	104	104	104	104	104	104	104
E	105	105	105	15	105	105	105	105	105	16	16	16	16	105	105	105	105	16	105	105	105	105	105	105	105	105	105	105	105	105	105	105
E'	106	106	106	106	106	106	106	106	106	106	106	106	106	106	18	17	17	106	17	106	106	106	106	106	106	106	106	106	106	106	106	106
R	107	107	107	18	107	107	107	107	107	19	19	19	19	107	107	107	107	19	107	107	107	107	107	107	107	107	107	107	107	107	107	107
R'	108	108	108	108	108	108	108	108	108	108	108	108	108	21	20	20	20	108	20	108	108	108	108	108	108	108	108	108	108	108	108	108
U	109	109	109	22	109	109	109	109	109	22	22	22	22	109	109	109	109	22	109	109	109	109	109	109	109	109	109	109	109	109	109	109
U'	110	110	110	110	110	110	110	110	110	110	110	110	110	23	23	23	23	110	23	110	110	24	110	110	110	110	110	110	110	110	110	110
V	111	111	111	24	111	111	111	111	111	25	25	25	25	111	111	111	111	25	111	111	111	111	111	111	111	111	111	111	111	111	111	111
V'	112	112	112	112	112	112	112	112	112	112	112	112	112	26	26	26	26	112	26	112	112	26	27	112	112	112	112	112	112	112	112	112
P	113	113	113	28	113	113	113	113	113	30	31	32	33	113	113	113	113	29	113	113	113	113	113	113	113	113	113	113	113	113	113	113
P'	114	114	114	114	114	114	114	114	114	114	114	114	114	34	34	34	34	35	34	114	114	34	34	114	114	114	114	114	114	114	114	114
L	115	115	115	36	115	115	115	115	115	36	36	36	36	115	115	115	115	36	37	115	115	115	115	115	115	115	115	115	115	115	115	115
Q	116	116	116	116	116	116	116	116	116	116	116	116	116	116	116	116	38	116	39	116	116	116	116	116	116	116	116	116	116	116	116	116
X	117	117	117	40	117	117	117	117	117	40	40	40	40	117	117	41	117	40	117	117	117	117	117	117	117	117	117	117	117	117	117	117
T	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	42	43	44	118	118	118	118
F	119	119	119	119	119	119	119	45	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119
H	120	120	120	120	120	120	120	120	120	120	120	120	120	120	120	120	120	47	120	120	120	120	120	120	120	46	46	46	120	120	120	120
D	121	121	121	121	121	121	121	121	121	121	121	121	121	121	121	121	121	121	49	121	121	121	121	121	121	48	48	48	121	121	121	121
K	122	122	122	122	122	122	122	122	122	122	122	122	122	122	122	122	50	122	51	122	122	122	122	122	122	122	122	122	122	122	122	122
C	52	52	52	52	52	52	52	123	52	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123	123
C'	53	53	53	53	53	53	53	124	53	124	124	124	124	124	124	124	124	124	124	124	54	124	124	124	124	124	124	124	124	124	124	124
N	125	125	125	125	125	125	125	125	125	125	125	125	125	125	125	55	125	125	125	125	125	125	125	56	57	125	125	125	125	125	125	125
Z	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	126	58	59	126	126	126
O	60	60	60	60	60	60	60	127	60	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127	127
Z'	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	63	128	128	128	128	128	128	62	62	128	128	

		Error Sintáctico

ANALIZADOR SEMÁNTICO

GRAMÁTICA DE ATRIBUTOS CON TDS

1. $A' \rightarrow \{1.1\}\ A\ \{1.2\}$	1.1)
	`TSG := crearTabla();`
	`desplG = 0;`

	1.2)
	`LiberarTabla(TSG);`
	`desplG = null;`

2. $A \rightarrow B\ A\ \{\ \}$	

3. $A \rightarrow FA\ \{\ \}$	
4. $A \rightarrow eof\ \{\ \}$	
5. $B \rightarrow if\ (\ E\ )\ \{5.1\}\ S\ \{5.2\}$	5.1) S.funcion := B.funcion S.switch := B.switch
	5.2) B.tipo := if (E.tipo == 'tipo_logico') then S.tipo else tipo_error (201); B.tipoRet := S.tipoRet
6. $B \rightarrow let\ \{6.1\}\ id\ T\ \{6.2\}\ N\ \{6.3\}$	6.1) zona_decl = true

	6.2)
	añadeTipo(id.pos,T.tipo);
	if(TSL == null)
	then añadeDespl(id.pos, desplG);
	desplG += T.tamaño;
	else añadeDespl(id.pos, desplL);
	desplL += T.tamaño);
	zona_decl = false;

	6.3) B.tipo := if (N.tipo == 'tipo_vacio' or buscarTipo(id.pos) == N.tipo) then 'tipo_ok' else 'tipo_error' (202)
7. $B \rightarrow \{\ 7.1\ \}\ S\ \{\ 7.2\ \}$	7.1) S.funcion = B.funcion S.switch := B.switch
	7.2) B.tipo := S.tipo; B.tipoRet := S.tipoRet;
8. $B \rightarrow switch\ (E)\ \{\ \ \{8.1\}\ Z\ \}\ \{\ 8.2\ \}$	8.1) Z.funcion := B.funcion; Z.switch := true;
	8.2) B.tipo := if (E.tipo == 'tipo_entero') then if (Z.tipo = 'tipo_ok')

	then 'tipo_ok'; else 'tipo_error'; (203) else 'tipo_error'; (204) B.tipoRet := Z.tipoRet;
9. $S \rightarrow \{9.1\}\ id\ \{9.2\}\ S'\ ;\ \{9.3\}$	9.1) decl_impl = true;

	9.2)
	if (buscarTipo(id.pos) == False)
	
	then añadeTipo(id.pos, 'tipo_entero');
	
	if(TSL == null)
	
	then añadeDespl(id.pos, desplG);
	desplG += 2;
	
	else añadeDespl(id.pos, desplL);
	desplL += 2;
	
	decl_impl = false;

	9.3) S.tipo := if (buscarTipo(id.pos) == 'tipo_funcion') then if(buscarTipoParametros(id.pos) == S'.tipo) then 'tipo_ok' else 'tipo_error' (205) else if (buscarTipo(id.pos) == S'.tipo) then 'tipo_ok' else 'tipo_error' (206)
10. $S \rightarrow print\ E\ ;\{10.1\}$	10.*1)* S.tipo := if(E.tipo == 'tipo_entero' or E.tipo == 'tipo_cadena') then 'tipo_ok' else if (E.tipo == 'tipo_funcion') if(buscarTipoDevuelto(E.pos) == 'tipo_entero' or buscarTipoDevuelto(E.pos) == 'tipo_cadena') then 'tipo_ok' else 'tipo_error' (207) else 'tipo_error' (207)
11. $S \rightarrow input\ \{11.1\}\ id\ \{11.2\}\ ;\{11.3\}$	11.1) decl_impl = true;

	11.2)
	if (buscarTipo(id.pos) == False)
	
	then añadeTipo(id.pos, 'tipo_entero');
	if(TSL == null)
	then añadeDespl(id.pos, desplG);
	desplG += 2;
	else añadeDespl(id.pos, desplL);
	desplL += 2;
	
	decl_impl = false;

	11.3) S.tipo := if (buscarTipo(id.pos) == 'tipo_entero' or buscarTipo(id.pos) == 'tipo_cadena') then 'tipo_ok' else 'tipo_error' (208)
12. $S \rightarrow return\ X\ ; \{12.1\}$	12.1) S.tipo := if (X.tipo != 'tipo_error') then if (S.funcion == true) then'tipo_ok' else 'tipo_error' (209) else 'tipo_error' (210) S.tipoRet := X.tipo
13. $S \rightarrow break\ ; \{13.1\}$	13.1) S.tipo := if (S.switch == true) then 'tipo_ok' else 'tipo_error' (211)
14. $S' \rightarrow = E\ \{14.1\}$	14.1) S'.tipo := E.tipo
15. $S' \rightarrow * = E\ \{15.1\}$	15.1) S'.tipo := if (E.tipo == 'tipo_entero') then U.tipo else 'tipo_error' (212)
16. $S' \rightarrow (\ L\ ) \{16.1\}$	16.1) S'.tipo := L.tipo
17. $E \rightarrow R\ E'\ \{17.1\}$	17.1) E.tipo := if (E'.tipo != 'tipo_vacio') then if (R.tipo == E'.tipo or E'.tipo ==

	'tipo_logico') then R.tipo else 'tipo_error' (213) else if (E'.tipo == 'tipo_vacio' and R.tipo != 'tipo_error') then R.tipo else 'tipo_error' (214)
18. $E' \rightarrow \lambda$ {18.1}	18.1) E'.tipo := 'tipo_vacio'
19. $E' \rightarrow \&\& R\ E'1$ {19.1}	19.1) E'.tipo := if (R.tipo == 'tipo_logico' and (E'1.tipo == 'tipo_vacio' or E'1.tipo == 'tipo_logico')) then R.tipo else 'tipo_error' (215)
20. $R \rightarrow U\ R'$ {20.1}	20.1) R.tipo := if (R'.tipo != 'tipo_vacio') then if(U.tipo == R'.tipo and R'.tipo == 'tipo_entero') then 'tipo_logico' else 'tipo_error' (216) else if (U.tipo != 'tipo_error') then U.tipo else 'tipo_error' (217)
21. $R' \rightarrow \lambda$ {21.1}	21.1) R'.tipo := 'tipo_vacio'
22. $R' \rightarrow == U\ R'1$ {22.1}	22.1) R'.tipo := if (U.tipo == R'1.tipo or R'1.tipo == 'tipo_vacio') and (R'1.tipo != 'tipo_error') and U.tipo != 'tipo_error') then U.tipo else 'tipo_error' (218)
23. $U \rightarrow V\ U'$ {23.1}	23.1) U.tipo := if ((V.tipo == U'.tipo or U'.tipo == 'tipo_vacio') and U'.tipo != 'tipo_error' and V.tipo != 'tipo_error') then V.tipo else 'tipo_error' (219)
24. $U' \rightarrow \lambda$ {24.1}	24.1) U'.tipo := 'tipo_vacio'
25. $U' \rightarrow +\ V\ U'1$ {25.1}	25.1) U'.tipo := if ((V.tipo == 'tipo_entero' and (U'1.tipo == 'tipo_vacio' or U'1.tipo == 'tipo_entero')) and U'1.tipo != 'tipo_error' and V.tipo != 'tipo_error')

	then V.tipo else 'tipo_error' (220)
26. $V \rightarrow P\ V'\ \{26.1\}$	26.1) V.tipo := if ((P.tipo == V'.tipo or V'.tipo == 'tipo_vacio') and V'.tipo != 'tipo_error' and P.tipo != 'tipo_error') then P.tipo else 'tipo_error' (221)
27. $V' \rightarrow \lambda\ \{27.1\}$	27.1) V'.tipo := 'tipo_vacio'
28. $V' \rightarrow *\ P\ V'1\ \{28.1\}$	27.1) V'.tipo := if ((P.tipo == 'tipo_entero' and (V'1.tipo == 'tipo_vacio' or V'1.tipo == 'tipo_entero')) and V'1.tipo != 'tipo_error') then P.tipo else 'tipo_error' (222)
29. $P \rightarrow \{29.1\}\ id\ \{29.2\}\ P'\ \{29.3\}$	29.1) decl_impl = true

	29.2)
	if (buscarTipo(id.pos) == False)
	then añadeTipo(id.pos, 'tipo_entero')
	if(TSL == null)
	then añadeDespl(id.pos, desplG);
	desplG += 2;
	else añadeDespl(id.pos, desplL);
	desplL += 2;
	
	decl_impl = false;

	29.3) P.tipo := if(buscarTipo(id.pos) == 'tipo_funcion') then if(buscarTipoParametros(id.pos) == P'.tipo) then buscarValorDevuelto(id.pos) else 'tipo_error' (223) else if P'.tipo == 'tipo_vacio' and P'.esfuncion == False then buscarTipo(id.pos) else 'tipo_error' (224)
30. $P \rightarrow (\ E\ )\ \{30.1\}$	30.1) P.tipo := E.tipo

31. $P \rightarrow cteEntera$ {31.1}	31.1) P.tipo := 'tipo_entero'
32. $P \rightarrow cadena$ {32.1}	32.1) P.tipo := 'tipo_cadena'
33. $P \rightarrow true$ {33.1}	33.1) P.tipo := 'tipo_logico'
34. $P \rightarrow false$ {34.1}	34.1) P.tipo := 'tipo_logico'
35. $P' \rightarrow \lambda$ {35.1}	35.1) P'.tipo := 'tipo_vacio' P'.esfuncion := False
36. $P' \rightarrow (L)$ {36.1}	36.1) P'.tipo := L.tipo P'.esfuncion := True
37. $L \rightarrow E\ Q$ {37.1}	37.1) L.tipo:= if(Q.tipo != 'tipo_vacio' and Q.tipo != 'tipo_error') then if(E.tipo != 'tipo_error') then E.tipo x Q.tipo else 'tipo_error' (225) else if(E.tipo != 'tipo_error' and Q.tipo != 'tipo_error') then E.tipo else 'tipo_error' (226)
38. $L \rightarrow \lambda$ {38.1}	38.1) L.tipo := 'tipo_vacio'
39. $Q \rightarrow ,E\ Q1$ {39.1}	39.1) Q.tipo := if(Q1.tipo != 'tipo_vacio' and Q1.tipo != 'tipo_error') then if(E.tipo != 'tipo_error') then E.tipo x Q1.tipo else 'tipo_error' (227) else if(E.tipo != 'tipo_error' and Q1.tipo != 'tipo_error') then E.tipo else 'tipo_error' (228)
40. $Q \rightarrow \lambda$ {40.1}	40.1) Q.tipo := 'tipo_vacio'
41. $X \rightarrow E$ {41.1}	41.1) X.tipo := E.tipo
42. $X \rightarrow \lambda$ {42.1}	42.1) X.tipo := 'tipo_vacio'
43. $T \rightarrow int$ {43.1}	43.1) T.tipo := 'tipo_entero'; T.tamaño := 2;

44. $T \rightarrow boolean$ {44.1}	44.1) T.tipo := 'tipo_logico'; T.tamaño := 2;
45. $T \rightarrow string$ {45.1}	45.1) T.tipo := 'tipo_cadena'; T.tamaño := 128;
46. $F \rightarrow$ $function$ {46.1} id {46.2} H (D) {46.3} { C } {46.4}	46.1) zona_decl = true
	46.2) TSL := crearTabla(); desplL = 0;
	46.3) zona_decl = false; añadeTipo(id.pos, 'tipo_funcion'); añadeTipoParametros(D.tipo); añadeTipoDevuelto(H.tipo); zona_decl = false

	46.4)
	if (C.tipoRet != H.tipo)
	then if((C.tipoRet == undefined and H.tipo == 'tipo_vacio') == False)
	then error()(229)
	
	liberarTabla(TSL);
	desplL = null;

47. $H \rightarrow T$ {47.1}	47.1) H.tipo := T.tipo
48. $H \rightarrow \lambda$ {48.1}	48.1) H.tipo := 'tipo_vacio'
49. $D \rightarrow T\ id\ K$ {49.1}	49.1) D.tipo := if(K.tipo != 'tipo_vacio' and K.tipo != 'tipo_error') then T.tipo x K.tipo; else if(K.tipo != 'tipo_error') then T.tipo ; else 'tipo_error'; (230) añadeTipo(id.pos, T.tipo); añadeDespl(id.pos,desplL); desplL += T.tamaño; if K.listaTT # Comprobar si existe then # ese atributo en K for sublista in K.listaTT añadeTipo(sublista[0], sublista[1]); añadeDespl(sublista[0], desplL);

	`desplL += sublista[2];`
50. $D \rightarrow \lambda \{50.1\}$	`50.1)` `D.tipo := 'tipo_vacio'`

51. $K \rightarrow , T\ id\ K1\ \{51.1\}$	`51.1)`
	`K.tipo := if(K1.tipo != 'tipo_vacio' and K1.tipo != 'tipo_error')`
	`then T.tipo x K1.tipo;`
	
	`else if(K1.tipo != 'tipo_error')`
	`then T.tipo ;`
	`else 'tipo_error'; (231)`
	
	`K.listaTT := [[id.pos, T.tipo, T.tamanho]]`
	
	`if K1.listaTT # Comprobar si existe ese`
	`then # atributo en K1`
	`for sublista in K1.listaTT`
	`K.listaTT += sublista`

52. $K \rightarrow \lambda \{52.1\}$	`52.1)` `K.tipo := 'tipo_vacio'`
53. $C \rightarrow \{53.1\}\ B\ C'\ \{53.2\}$	`53.1)` `B.funcion := true`

	`53.2)`
	`C.tipo := if(B.tipo == C'.tipo == 'tipo_ok')`
	`then 'tipo_ok';`
	`else 'tipo_error'; (232)`
	
	`C.tipoRet := if( B.tipoRet != undefined and B.tipoRet != 'tipo_error' )`
	`then if (B.tipoRet == C'.tipoRet)`
	`then B.tipoRet`
	`else if (C'.tipoRet == undefined)`
	`then B.tipoRet`
	`else 'tipo_error' (233)`
	
	`else if(C'.tipoRet != undefined and C'.tipoRet != 'tipo_error')`
	`then C'.tipoRet`
	`else if (C'.tipoRet != 'tipo_error' or B.tipoRet != 'tipo_error')`
	`then 'tipo_error' (234)`
	`else undefined`

54. $C' \rightarrow \{54.1\}\ B\ C'1\ \{54.2\}$	`54.1)` `B.funcion := true`

	`54.2)`
	`C'.tipo := if(B.tipo == C'1.tipo == 'tipo_ok')`
	`then 'tipo_ok';`
	`else 'tipo_error'; (235)`
	
	`C'.tipoRet := if( B.tipoRet != undefined and B.tipoRet != 'tipo_error' )`
	`then if ( B.tipoRet == C'1.tipoRet)`
	`then B.tipoRet`

	else if (C'1.tipoRet == undefined) then B.tipoRet else 'tipo_error' (236) else if(C'1.tipoRet != undefined and C'1.tipoRet != 'tipo_error') then C'1.tipoRet else if (C'1.tipoRet == 'tipo_error' or B.tipoRet == 'tipo_error') then 'tipo_error' (237) else undefined
55. $C' \rightarrow \lambda \{55.1\}$	55.1) C'.tipo := 'tipo_ok'; C'.tipoRet == undefined;
56. $N \rightarrow ; \{56.1\}$	56.1) N.tipo := 'tipo_vacio'
57. $N \rightarrow = E ; \{57.1\}$	57.1) N.tipo := E.tipo
58. $N \rightarrow * = E ; \{58.1\}$	58.1) N.tipo := if (E.tipo == 'tipo_entero') then E.tipo else 'tipo_error' (238)
59. $Z \rightarrow case\ cteEntera : \{59.1\}\ O\ \{59.2\}\ Z'\ \{59.3\}$	59.1) O.funcion = Z.funcion; O.switch := Z.switch
	59.2) Z'.funcion = Z.funcion; Z'.switch := Z.switch
	59.3) Z.tipo := if (O.tipo == Z'.tipo == 'tipo_ok') then 'tipo_ok' else 'tipo_error' (239) Z.tipoRet := if(O.tipoRet != undefined and O.tipoRet != 'tipo_error') then if (O.tipoRet == Z'.tipoRet) then O.tipoRet else if (Z'.tipoRet == undefined) then O.tipoRet else 'tipo_error' (236) else if(Z'.tipoRet != undefined and Z'.tipoRet != 'tipo_error') then Z'.tipoRet else if (Z'.tipoRet == 'tipo_error' or O.tipoRet == 'tipo_error') then 'tipo_error' (237) else undefined

60. $Z \rightarrow default : \{60.1\}\ O\ \{60.2\}$	60.1) O.funcion = Z.funcion O.switch := Z.switch
	60.2) Z.tipo := if (O.tipo == 'tipo_ok') then 'tipo_ok' else 'tipo_error' (240) Z.tipoRet := if O.tipoRet != 'tipo_error' then O.tipoRet else 'tipo_error' (237)
61. $O \rightarrow \{61.1\}\ B\ \{61.2\}\ O1\ \{61.3\}$	61.1) B.funcion = O.funcion B.switch := O.switch
	61.2) O1.funcion = O.funcion O1.switch := O.switch
	61.3) O.tipo := if (B.tipo == O1.tipo == 'tipo_ok') then 'tipo_ok' else 'tipo_error' (241) O.tipoRet := if(B.tipoRet != undefined and B.tipoRet != 'tipo_error') then if (B.tipoRet == O1.tipoRet) then B.tipoRet else if (O1.tipoRet == undefined) then B.tipoRet else 'tipo_error' (236) else if(O1.tipoRet != undefined and O1.tipoRet != 'tipo_error') then O1.tipoRet else if (O1.tipoRet == 'tipo_error' or B.tipoRet == 'tipo_error') then 'tipo_error' (237) else undefined
62. $O \rightarrow \lambda\ \{62.1\}$	62.1) O.tipo := 'tipo_ok'
63. $Z' \rightarrow \{63.1\}\ Z\ \{63.2\}$	63.1) Z.switch := Z'.switch Z.funcion = Z'.funcion
	63.2) Z'.tipo := if (Z.tipo == 'tipo_ok') then 'tipo_ok' else 'tipo_error' (242) Z'.tipoRet := if Z.tipoRet != 'tipo_error'

	then Z.tipoRet else 'tipo_error' (237)
64. $Z'' \rightarrow \lambda$ {64.1}	64.1) Z''.tipo := 'tipo_ok'

ERRORES

Léxicos

Código 50: Fue introducido un carácter no esperado.

Código 51: Fue introducido un carácter no válido. Se esperaba *.

Código 52: Fue introducido un carácter no esperado.

Código 53: Fue introducido un carácter no esperado.

Código 54: Fue introducido un carácter no esperado en la cadena.

Código 55: Fue introducido un carácter no válido. Se esperaba &.

Código 60: El número entero introducido está fuera de rango (es mayor que 32767 o menor que -32768).

Código 61: La cadena supera los 64 caracteres.

Sintácticos

Código 100: Token *siguiente_token* no esperado. Se esperaba *cima_pila*.

Código 101: El token *siguiente_token* no pertenece al primer elemento de una sentencia válida, una función o es fin de fichero.

Código 102: El token *siguiente_token* no pertenece al primer elemento de una sentencia válida.

Código 103: El token *siguiente_token* no pertenece al primer elemento de una sentencia simple válida.

Código 104: El token *siguiente_token* no es una asignación válida o la llamada a una función.

Código 105: El token *siguiente_token* no pertenece a una expresión válida.

Código 106: El token *siguiente_token* no pertenece a una expresión lógica u otra expresión válida.

Código 107: El token *siguiente_token* no pertenece a una expresión válida.

Código 108: El token *siguiente_token* no pertenece a una expresión relacional u otra expresión válida.

Código 109: El token *siguiente_token* no pertenece a una expresión válida.

Código 110: El token *siguiente_token* no pertenece a una expresión aritmética de suma u otra expresión válida.

Código 111: El token *siguiente_token* no pertenece a una expresión válida.

Código 112: El token *siguiente_token* no pertenece a una expresión aritmética de multiplicación u otra expresión válida.

Código 113: El token *siguiente_token* no pertenece a una expresión válida.

Código 114: El token *siguiente_token* no es un identificador de variable o la llamada a una función

Código 115: El token *siguiente_token* no es un argumento de función válido porque no pertenece a una expresión válida

Código 116: El token *siguiente_token* no es un argumento de función válido porque no pertenece a una expresión válida

Código 117: El token *siguiente_token* no es un valor de retorno de función válido porque no pertenece a una

expresión válida

Código 118: El token *siguiente_token* no es un tipo de variable válido

Código 119: El token *siguiente_token* no pertenece a una declaración válida de una función

Código 120: El token *siguiente_token* no es un tipo de valor de retorno de función válido

Código 121: El token *siguiente_token* no es una declaración válida de un argumento de una función

Código 122: El token *siguiente_token* no es una declaración válida de un argumento de una función

Código 123: El token *siguiente_token* no pertenece a una sentencia válida

Código 124: El token *siguiente_token* no pertenece a una sentencia válida

Código 125: El token *siguiente_token* no es una asignación válida

Código 126: El token *siguiente_token* no es un cuerpo válido para la condicional múltiple switch

Código 127: El token *siguiente_token* no pertenece a una sentencia válida

Código 128: El token *siguiente_token* no pertenece a una sentencia válida

- *siguiente_token*: último token devuelto por el Analizador Léxico
- *cima_pila:* último token apilado dado el algoritmo del Analizador Sintáctico Descendente por Tablas.

Semánticos

Código 200: El identificador *lexema* ya ha sido declarado.

Código 201: La expresión evaluada como condición no es de tipo lógico. Es tipo *tipoExpresion*

Código 202: Se le debe asignar una expresión de tipo *tipoCorrecto* al identificador. La expresión asignada es de tipo *tipoExpresion*.

Código 203: Existencia de sentencias no válidas en el cuerpo del switch.

Código 204: La expresión que se evalúa debe ser de tipo entero. La expresión es de tipo *tipoExpresion.*

Código 205: Los parámetros pasados a la función deben ser de tipo *tipoCorrecto*. Los parámetros pasados son de tipo *tipoParamentros.*

Código 206: La expresión asignada al identificador es de tipo *tipoExpresion* cuando debe ser el mismo tipo que este (*tipoCorrecto*).

Código 207: La sentencia PRINT solo 'imprime' expresiones de tipo cadena o entero. La expresión es de tipo *tipoExpresion.*

Código 208: La sentencia INPUT solo admite identificadores de tipo cadena o entero. El identificador *lexema* es de tipo *tipoExpresion*

Código 209: La sentencia RETURN solo puede ser declarada en el cuerpo de una función.

Código 210: La expresión de retorno es incorrecta.

Código 211: La sentencia BREAK solo puede declararse dentro del CASE O DEFAULT de un SWITCH.

Código 212: La expresión debe ser de tipo entero. Es de tipo *tipoExpresion.*

Código 213: Las expresiones operando con && deben ser de tipo lógico. Son de tipo *tipoExpresion* y *tipoExpresion* respectivamente.

Código 214: Expresión incorrecta.

Código 215: El operador lógico && se aplica sobre expresiones de tipo lógico. La(s) expresion(es) declarada(s) es(son) de tipo *tipoExpresion* (y *tipoExpresion*).

Código 216: Las expresiones a comparar con == deben ser de tipo entero. Son de tipo *tipoExpresion* y *tipoExpresion* respectivamente.

Código 217: Expresión incorrecta.

Código 218: Las expresiones son de tipos diferentes. Son de tipo *tipoExpresion* y *tipoExpresion* respectivamente.

Código 219: Las expresiones son de tipos diferentes. Son de tipo *tipoExpresion* y *tipoExpresion* respectivamente.

Código 220: Las expresiones no son de tipo entero. La(s) expresion(es) declarada(s) es(son) de tipo tipoExpresion (y tipoExpresion).

Código 221: Las expresiones son de tipos diferentes. Son de tipo *tipoExpresion* y *tipoExpresion* respectivamente.

Código 222: Las expresiones no son de tipo entero. La(s) expresion(es) declarada(s) es(son) de tipo tipoExpresion (y tipoExpresion).

Código 223: Los parámetros pasados a la función deben ser de tipo *tipoCorrecto*. Los parámetros pasados son de tipo *tipoParamentros.*

Código 224: El identificador *lexema* no fue declarado como una función.

Código 225: Expresión incorrecta en el primer argumento de la llamada a la función.

Código 226: Expresión incorrecta en los argumentos de la llamada a la función.

Código 227: Expresión incorrecta en los argumentos de la llamada a la función.

Código 228: Expresión incorrecta en los argumentos de la llamada a la función.

Código 229: El tipo del valor de retorno *tipoRetorno* no coincide con el tipo valor de retorno declarado *tipoCorrecto.*

Código 230: Sentencia incorrecta en la declaración de los parámetros de la función.

Código 231: Sentencia incorrecta en la declaración de los parámetros de la función.

Código 232: Sentencia incorrecta en el cuerpo de la función.

Código 233: Valores de retorno de la función son de un tipo distinto: *tipoRetorno* y tipoRetorno.

Código 234: Expresión de declaración del valor de retorno de la función incorrecta.

Código 235: Sentencia incorrecta en el cuerpo de la función.

Código 236: Valores de retorno de la función son de un tipo distinto: *tipoRetorno* y *tipoRetorno*.

Código 237: Expresión de declaración del valor de retorno de la función incorrecta.

Código 238: La expresión asignada a multiplicar debe ser de tipo entero, sin embargo, es de tipo *tipoExpresion.*

Código 239: Sentencia incorrecta en el cuerpo del CASE.

Código 240: Sentencia incorrecta en el cuerpo del DEFAULT del switch.

Código 241: Sentencia incorrecta.

Código 242: Sentencia incorrecta.

- *lexema*: lexema del identificador
- *tipoCorrecto:* tipo que sería correcto en determinado caso.
- *tipoExpresion*: tipo de la expresión que no es correcto.
- *tipoParamentros*: tipo de los parámetros de la función que no son correctos.
- tipoRetorno: tipo de retorno de la función que no es correcto.

Cada error en nuestro procesador tendrá la siguiente estructura en el fichero “errores.txt”:
“Línea ” + *num_línea* + “ – Código de error ” + *cod_error* + “ :” + **\n\t** + *mensaje_error*

- **\n** : Salto de línea
- **\t** : Tabulador
- *num_línea* : el número de línea en el fichero fuente donde se encuentra el error.
- *cod_error* : código de error según lo especificado anteriormente
- *mensaje_error* : mensaje específico del error en el formato siguente
  "ERROR " + tipo_error + " – " + *mensaje_descriptivo*

- *tipo_error : LÉXICO || SINTÁCTICO || SEMÁNTICO*
- *mensaje_descriptivo:* mensaje descriptivo del error que incluye el carácter, cadena o valor incorrecto, incluyendo el carácter esperado en determinados casos.

Un ejemplo del contenido de un error generado por nuestro procesador:
Línea 1 - Código de error 51:
ERROR LÉXICO - carácter CAR: 'l' no válido. Se esperaba *.

En nuestro procesador la compilación para con errores sintácticos o semánticos, sin embargo, detecta todos los errores lógicos porque el token no se genera y pasa al siguiente, que de estar bien sintácticamente y semánticamente no da problemas de este tipo, llegando hasta el final del fichero.

ANEXOS

CODIGO VAST

```
/////// GRAMATICA ////////////

Terminales = { if let switch id print input return function break
cteEntera cadena true false == && ; , ( ) { } + * = *= int boolean
string case default : eof }
NoTerminales = { Ap A B S Sp E Ep R Rp U Up V Vp P Pp L Q X T
F H D K C Cp N Z Zp O }
Axioma = Ap
Producciones = {

        Ap -> A              //// Axioma
        A  -> B A
        A -> F A
        A -> eof
        B -> if ( E ) S      //// Sentencias compuestas
        B -> let id T N
        B -> S
        B -> switch ( E ) { Z }
        S -> id Sp ;         //// Sentencias simples
        S -> print E ;
        S -> input id ;
        S -> return X ;
        S -> break ;
        Sp -> = E
        Sp -> *= E
        Sp -> ( L )
        E -> R Ep            //// Expresiones
        Ep -> lambda
        Ep -> && R Ep
        R -> U Rp
        Rp -> lambda
        Rp -> == U Rp
        U -> V Up
        Up -> lambda
        Up -> + V Up
        V -> P Vp
        Vp -> lambda
        Vp -> * P Vp
        P -> id Pp           //// Operandos
        P -> ( E )
        P -> cteEntera
        P -> cadena
        P -> true
        P -> false
        Pp -> lambda
        Pp -> ( L )
        L -> E Q             //// Argumentos de función
        L -> lambda
        Q -> , E Q
        Q -> lambda
        X -> E               //// Valor de retorno
        X -> lambda
        T -> int             //// Tipos de variables
        T -> boolean
        T -> string
        F -> function id H ( D ) { C }  //// Declaración de
        funciones
        H -> T
        H -> lambda
        D -> T id K
        D -> lambda
        K -> , T id K
        K -> lambda
        C -> B Cp
        Cp -> B Cp
        Cp -> lambda
        N -> ;      //// Inicialización de identificadores
        N -> = E ;
        N -> *= E ;
        Z -> case cteEntera : O Zp   //// Cuerpo del switch
        Z -> default : O
        O -> B O
        O -> lambda
        Zp -> Z
        Zp -> lambda
}
```

CASOS DE PRUEBA

Casos correctos:

CASO 1

fichero_fuente.txt

```
/*
    CASO DE PRUEBA #1: Probando el switch y el desplazamiento en T.S.
*/

let cadena string;

let x int;
print "Introduce un numero del 1 al 5 :";
input  x;

switch (x) {

    case 1:
        print "Tienes un 1!!";
        break;
    case 2:
    case 3:
    case 4:
        print "Tienes un 2, un 3 o un 4!!";
        break;
    default:
        print "Tienes un 5!!";
}

print "Escribe tu nombre: ";
input cadena;
print "Hola: ";
print cadena;

let edad int;

print "Escribe tu edad: ";
input edad;
print "Tienes: ";
print edad;
```

tokens.txt:

< let , >	< break , >

```
< id  , 0 >
< string , >
< ptoComa , >
< let , >
< id  , 1 >
< int , >
< ptoComa , >
< print , >
< cadena , "Introduce un numero del 1 al 5 :" >
< ptoComa , >
< input , >
< id  , 1 >
< ptoComa , >
< switch , >
< abrirParentesis , >
< id  , 1 >
< cerrarParentesis , >
< abrirCorchete , >
< case , >
< cteEntera , 1 >
< dosPuntos , >
< print , >
< cadena , "Tienes un 1!!" >
< ptoComa , >
< break , >
< ptoComa , >
< case , >
< cteEntera , 2 >
< dosPuntos , >
< case , >
< cteEntera , 3 >
< dosPuntos , >
< case , >
< cteEntera , 4 >
< dosPuntos , >
< print , >
< cadena , "Tienes un 2, un 3 o un 4!!" >
< ptoComa , >
```

```
< ptoComa , >
< default , >
< dosPuntos , >
< print , >
< cadena , "Tienes un 5!!" >
< ptoComa , >
< cerrarCorchete , >
< print , >
< cadena , "Escribe tu nombre: " >
< ptoComa , >
< input , >
< id  , 0 >
< ptoComa , >
< print , >
< cadena , "Hola: " >
< ptoComa , >
< print , >
< id  , 0 >
< ptoComa , >
< let , >
< id  , 2 >
< int , >
< ptoComa , >
< print , >
< cadena , "Escribe tu edad: " >
< ptoComa , >
< input , >
< id  , 2 >
< ptoComa , >
< print , >
< cadena , "Tienes: " >
< ptoComa , >
< print , >
< id  , 2 >
< ptoComa , >
< eof , >
```

tabla.txt

```
CONTENIDOS DE LA TABLA # 1 :

        * LEXEMA :      'cadena'
        ATRIBUTOS :
        + Tipo : 'tipo_cadena'
        + Despl :       0
        ---------- ----------
```

```
* LEXEMA :       'x'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl :         64
---------- ----------
* LEXEMA :       'edad'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl :         65
---------- ----------
```

parse.txt:

```
Descendente
1 2 6 45 56 2 6 43 56 2 7 10 17 20
23 26 32 27 24 21
18 2 7 11 2 8 17 20 23 26 29 35 27 24
21 18 59 61 7 10
17 20 23 26 32 27 24 21 18 61 7 13 62 63
59 62 63 59 62 63
59 61 7 10 17 20 23 26 32 27 24 21 18 61
7 13 62 63 60 61
7 10 17 20 23 26 32 27 24 21 18 62 2 7
10 17 20 23 26 32
27 24 21 18 2 7 11 2 7 10 17 20 23 26
32 27 24 21 18 2
7 10 17 20 23 26 29 35 27 24 21 18 2 6
43 56 2 7 10 17
20 23 26 32 27 24 21 18 2 7 11 2 7 10
17 20 23 26 32 27
24 21 18 2 7 10 17 20 23 26 29 35 27 24
21 18 4
```

Árbol generado: https://drive.google.com/file/d/1p-fHyPPXP6fF9hVlyp437-3omsIiPxOT/view?usp=share_link

CASO 2

fichero_fuente.txt

```
/*
    CASO DE PRUEBA #2: Probando declaracion explicita e implicita y sentencia if
*/

a = b + c;  /* Sumando variables con declaración implicita */

if (a == b)
    print "Iguales";

let cadena string = "HEY";
let x boolean = true;
let y boolean = true;

/* Usando variables globales dentro de la funcion */
function printCadena () {
    print cadena;
}

if (x&&y&&x&&y&&y)
    printCadena();
```

tokens.txt:

< id , 0 >	< boolean , >
< asignacion , >	< asignacion , >
< id , 1 >	< true , >
< opAritmetico , 2 >	< ptoComa , >
< id , 2 >	< function , >
< ptoComa , >	< id , 6 >
< if , >	< abrirParentesis , >
< abrirParentesis , >	< cerrarParentesis , >
< id , 0 >	< abrirCorchete , >
< opRelacional , 1 >	< print , >
< id , 1 >	< id , 3 >
< cerrarParentesis , >	< ptoComa , >
< print , >	< cerrarCorchete , >
< cadena , "Iguales" >	< if , >
< ptoComa , >	< abrirParentesis , >
< let , >	< id , 4 >
< id , 3 >	< opLogico , 1 >
< string , >	< id , 5 >
< asignacion , >	< opLogico , 1 >
< cadena , "HEY" >	< id , 4 >
< ptoComa , >	< opLogico , 1 >
< let , >	< id , 5 >
< id , 4 >	< opLogico , 1 >
< boolean , >	< id , 5 >
< asignacion , >	< cerrarParentesis , >

< true , > < ptoComa , > < let , > < id , 5 >	< id , 6 > < abrirParentesis , > < cerrarParentesis , > < ptoComa , > < eof , >

tabla.txt:

CONTENIDOS DE LA TABLA # 2 : CONTENIDOS DE LA TABLA # 1 : * LEXEMA : 'a' ATRIBUTOS : + Tipo : 'tipo_entero' + Despl : 0 ---------- ---------- * LEXEMA : 'b' ATRIBUTOS : + Tipo : 'tipo_entero' + Despl : 1 ---------- ---------- * LEXEMA : 'c' ATRIBUTOS : + Tipo : 'tipo_entero' + Despl : 2 ---------- ----------	* LEXEMA : 'cadena' ATRIBUTOS : + Tipo : 'tipo_cadena' + Despl : 3 ---------- ---------- * LEXEMA : 'x' ATRIBUTOS : + Tipo : 'tipo_logico' + Despl : 67 ---------- ---------- * LEXEMA : 'y' ATRIBUTOS : + Tipo : 'tipo_logico' + Despl : 68 ---------- ---------- * LEXEMA : 'printCadena' ATRIBUTOS : + Tipo : 'tipo_funcion' + numParam : 0 + TipoRetorno : 'tipo_vacio' + EtiqFuncion : 'EtprintCadena1' ---------- ----------

parse.txt

Descendente
1 2 7 9 14 17 20 23 26 29 35 27 25 26
29 35 27 24 21 18
2 5 17 20 23 26 29 35 27 24 22 23 26 29
35 27 24 21 18 10
17 20 23 26 32 27 24 21 18 2 6 45 57 17
20 23 26 32 27 24
21 18 2 6 44 57 17 20 23 26 33 27 24 21
18 2 6 44 57 17
20 23 26 33 27 24 21 18 3 46 48 50 53 7
10 17 20 23 26 29
35 27 24 21 18 55 2 5 17 20 23 26 29 35
27 24 21 19 20 23
26 29 35 27 24 21 19 20 23 26 29 35 27 24
21 19 20 23 26 29
35 27 24 21 19 20 23 26 29 35 27 24 21 18 9
16 38 4

Árbol generado: https://drive.google.com/file/d/1HBJ5pE55LSOhCmnRI-pX1QPM6bI1TXSe/view?usp=share_link

CASO 3

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #3 : Probando if, switch y function
*/

let XX int = 9;
let YY boolean = true;

if(XX==9 && YY&&true)
    print "Hola";

function fun string (boolean x, int y, string cad) {
    switch (y) {
        case 1:
            return "y es 1";
        case 2:
        case 3:
            break;
        case 9:
            return "y es 9";
            break;
    }

    if (y == 8)
        return "y es 8";

    if(x)
        return cad;

    let cadena2 string = "Buenas";
    return cadena2;
}

print fun(YY,10,"Hola");
```

tokens.txt:

< let , >	< id , 5 >	< cerrarParentesis , >
< id , 0 >	< cerrarParentesis , >	< return , >
< int , >	< abrirCorchete , >	< cadena , "y es 8" >
< asignacion , >	< switch , >	< ptoComa , >
< cteEntera , 9 >	< abrirParentesis , >	< if , >
< ptoComa , >	< id , 4 >	< abrirParentesis , >
< let , >	< cerrarParentesis , >	< id , 3 >
< id , 1 >	< abrirCorchete , >	< cerrarParentesis , >
< boolean , >	< case , >	< return , >
< asignacion , >	< cteEntera , 1 >	< id , 5 >
< true , >	< dosPuntos , >	< ptoComa , >
< ptoComa , >	< return , >	< let , >
< if , >	< cadena , "y es 1" >	< id , 6 >

< abrirParentesis , >	< ptoComa , >	< string , >
< id , 0 >	< case , >	< asignacion , >
< opRelacional , 1 >	< cteEntera , 2 >	< cadena , "Buenas" >
< cteEntera , 9 >	< dosPuntos , >	< ptoComa , >
< opLogico , 1 >	< case , >	< return , >
< id , 1 >	< cteEntera , 3 >	< id , 6 >
< opLogico , 1 >	< dosPuntos , >	< ptoComa , >
< true , >	< break , >	< cerrarCorchete , >
< cerrarParentesis , >	< ptoComa , >	< print , >
< print , >	< case , >	< id , 2 >
< cadena , "Hola" >	< cteEntera , 9 >	< abrirParentesis , >
< ptoComa , >	< dosPuntos , >	< id , 1 >
< function , >	< return , >	< coma , >
< id , 2 >	< cadena , "y es 9" >	< cteEntera , 10 >
< string , >	< ptoComa , >	< coma , >
< abrirParentesis , >	< break , >	< cadena , "Hola" >
< boolean , >	< ptoComa , >	< cerrarParentesis , >
< id , 3 >	< cerrarCorchete , >	< ptoComa , >
< coma , >	< if , >	< eof , >
< int , >	< abrirParentesis , >	
< id , 4 >	< id , 4 >	
< coma , >	< opRelacional , 1 >	
< string , >	< cteEntera , 8 >	

tabla.txt:

CONTENIDOS DE LA TABLA # 2 :

* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'tipo_logico'
+ Despl : 0
---------- ----------
* LEXEMA : 'y'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 1
---------- ----------
* LEXEMA : 'cad'
ATRIBUTOS :
+ Tipo : 'tipo_cadena'
+ Despl : 2
---------- ----------
* LEXEMA : 'cadena2'
ATRIBUTOS :
+ Tipo : 'tipo_cadena'
+ Despl : 66
---------- ----------

CONTENIDOS DE LA TABLA # 1 :

* LEXEMA : 'XX'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 0
---------- ----------
* LEXEMA : 'YY'
ATRIBUTOS :
+ Tipo : 'tipo_logico'
+ Despl : 1
---------- ----------
* LEXEMA : 'fun'
ATRIBUTOS :
+ Tipo : 'tipo_funcion'
+ numParam : 3
+ TipoParam1 : 'tipo_logico'
+ TipoParam2 : 'tipo_entero'
+ TipoParam3 : 'tipo_cadena'
+ TipoRetorno : 'tipo_cadena'
+ EtiqFuncion : 'Etfun1'
---------- ----------

parse.txt

```
Descendente
1 2 6 43 57 17 20 23 26 31 27 24 21 18 2
6 44 57 17 20
23 26 33 27 24 21 18 2 5 17 20 23 26 29
35 27 24 22 23 26
31 27 24 21 19 20 23 26 29 35 27 24 21 19
20 23 26 33 27 24
21 18 10 17 20 23 26 32 27 24 21 18 3 46
47 45 49 44 51 43
51 45 52 53 8 17 20 23 26 29 35 27 24 21
18 59 61 7 12 41
17 20 23 26 32 27 24 21 18 62 63 59 62 63
59 61 7 13 62 63
59 61 7 12 41 17 20 23 26 32 27 24 21 18
61 7 13 62 64 54
5 17 20 23 26 29 35 27 24 22 23 26 31 27
24 21 18 12 41 17
20 23 26 32 27 24 21 18 54 5 17 20 23 26
29 35 27 24 21 18
12 41 17 20 23 26 29 35 27 24 21 18 54 6
45 57 17 20 23 26
32 27 24 21 18 54 7 12 41 17 20 23 26 29
35 27 24 21 18 55
2 7 10 17 20 23 26 29 36 37 17 20 23 26
29 35 27 24 21 18
39 17 20 23 26 31 27 24 21 18 39 17 20 23
26 32 27 24 21 18
40 27 24 21 18 4
```

Árbol generado: https://drive.google.com/file/d/1W3NRNBuzO2U8kUwy8xeB37GnLpT4vUT2/view?usp=share_link

CASO 4

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #4: Probando varias funciones al mismo tiempo y llamada a funcion
en el if
*/

let num int = 5;
let b boolean = true;

function Suma int (int a, int b) {

    j= a + b;
    return j;
    /* La función finaliza y devuelve el valor entero de la expresión */
}

if (num == 5)
    print "Bien!!";

print "Introduce un número para sumarle 18: ";
input num;

print "El resultado es: ";

if (b)
    print Suma(num,18);

function Multiplicacion boolean (int a) {
    a *= 100;
    if (a == 1000)
        return true;
    return false;
}

if(Multiplicacion(10))
    j = num + 40;
```

tokens.txt:

< let , >	< abrirParentesis , >	< int , >
< id , 0 >	< id , 0 >	< id , 7 >
< int , >	< opRelacional , 1 >	< cerrarParentesis , >
< asignacion , >	< cteEntera , 5 >	< abrirCorchete , >
< cteEntera , 5 >	< cerrarParentesis , >	< id , 7 >
< ptoComa , >	< print , >	< asigMultiplicacion , >
< let , >	< cadena , "Bien!!" >	< cteEntera , 100 >
< id , 1 >	< ptoComa , >	< ptoComa , >
< boolean , >	< print , >	< if , >
< asignacion , >	< cadena , "Introduce un número	< abrirParentesis , >
< true , >	para sumarle 18: " >	< id , 7 >
< ptoComa , >	< ptoComa , >	< opRelacional , 1 >

```
< function , >
< id , 2 >
< int , >
< abrirParentesis , >
< int , >
< id , 3 >
< coma , >
< int , >
< id , 4 >
< cerrarParentesis , >
< abrirCorchete , >
< id , 5 >
< asignacion , >
< id , 3 >
< opAritmetico , 2 >
< id , 4 >
< ptoComa , >
< return , >
< id , 5 >
< ptoComa , >
< cerrarCorchete , >
< if , >
```

```
< input , >
< id , 0 >
< ptoComa , >
< print , >
< cadena , "El resultado es: " >
< ptoComa , >
< if , >
< abrirParentesis , >
< id , 1 >
< cerrarParentesis , >
< print , >
< id , 2 >
< abrirParentesis , >
< id , 0 >
< coma , >
< cteEntera , 18 >
< cerrarParentesis , >
< ptoComa , >
< function , >
< id , 6 >
< boolean , >
< abrirParentesis , >
```

```
< cteEntera , 1000 >
< cerrarParentesis , >
< return , >
< true , >
< ptoComa , >
< return , >
< false , >
< ptoComa , >
< cerrarCorchete , >
< if , >
< abrirParentesis , >
< id , 6 >
< abrirParentesis , >
< cteEntera , 10 >
< cerrarParentesis , >
< cerrarParentesis , >
< id , 5 >
< asignacion , >
< id , 0 >
< opAritmetico , 2 >
< cteEntera , 40 >
< ptoComa , >
< eof , >
```

tabla.txt:

```
CONTENIDOS DE LA TABLA # 2 :

        * LEXEMA :      'a'
        ATRIBUTOS :
        + Tipo : 'tipo_entero'
        + Despl :       0
        ---------- ----------
        * LEXEMA :      'b'
        ATRIBUTOS :
        + Tipo : 'tipo_entero'
        + Despl :       1
        ---------- ----------

CONTENIDOS DE LA TABLA # 3 :

        * LEXEMA :      'a'
        ATRIBUTOS :
        + Tipo : 'tipo_entero'
        + Despl :       0
        ---------- ----------
CONTENIDOS DE LA TABLA # 1 :

        * LEXEMA :      'num'
        ATRIBUTOS :
        + Tipo : 'tipo_entero'
        + Despl :       0
        ---------- ----------
        * LEXEMA :      'b'
        ATRIBUTOS :
        + Tipo : 'tipo_logico'
        + Despl :       1
        ---------- ----------
        * LEXEMA :      'Suma'
        ATRIBUTOS :
        + Tipo : 'tipo_funcion'
                + numParam :   2
                        + TipoParam1 :  'tipo_entero'
                        + TipoParam2 :  'tipo_entero'
                        + TipoRetorno : 'tipo_entero'
                + EtiqFuncion :  'EtSuma1'
        ---------- ----------
        * LEXEMA :      'j'
        ATRIBUTOS :
        + Tipo : 'tipo_entero'
        + Despl :       2
        ---------- ----------
        * LEXEMA :      'Multiplicacion'
        ATRIBUTOS :
        + Tipo : 'tipo_funcion'
                + numParam :   1
                        + TipoParam1 :  'tipo_entero'
                        + TipoRetorno : 'tipo_logico'
                + EtiqFuncion :  'EtMultiplicacion2'
        ---------- ----------
```

parse.txt

```
Descendente
1 2 6 43 57 17 20 23 26 31 27 24 21 18 2
6 44 57 17 20
23 26 33 27 24 21 18 3 46 47 43 49 43 51
43 52 53 7 9 14
17 20 23 26 29 35 27 25 26 29 35 27 24 21
18 54 7 12 41 17
20 23 26 29 35 27 24 21 18 55 2 5 17 20
23 26 29 35 27 24
22 23 26 31 27 24 21 18 10 17 20 23 26 32
27 24 21 18 2 7
10 17 20 23 26 32 27 24 21 18 2 7 11 2 7
10 17 20 23 26
32 27 24 21 18 2 5 17 20 23 26 29 35 27
24 21 18 10 17 20
23 26 29 36 37 17 20 23 26 29 35 27 24 21
18 39 17 20 23 26
31 27 24 21 18 40 27 24 21 18 3 46 47 44
49 43 52 53 7 9
15 17 20 23 26 31 27 24 21 18 54 5 17 20
23 26 29 35 27 24
22 23 26 31 27 24 21 18 12 41 17 20 23 26
33 27 24 21 18 54
7 12 41 17 20 23 26 34 27 24 21 18 55 2 5
17 20 23 26 29
36 37 17 20 23 26 31 27 24 21 18 40 27 24
21 18 9 14 17 20
23 26 29 35 27 25 26 31 27 24 21 18 4
```

Árbol generado: https://drive.google.com/file/d/1dZL7vm4S23NzvKqQK4JbPO2_i744YDud/view?usp=share_link

CASO 5

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #5 : probando llamada a funcion y recursividad
*/

let saludo string;

function Multiplica int (string saludo) {
    print saludo;
    saludo  = "Vamos a multiplicar";
    let saludo2 string = saludo;
    print saludo2;
    j = a * b;
    return j;
}

function printResultado (int x) {
    if (cont == 0)
        printResultado(x);
    cont = cont + 1;
    print x;
}

switch (j){
    case 10:
        printResultado(Multiplica(saludo));
    default:
        break;
}
```

tokens.txt

< let , >	< id , 5 >	< cteEntera , 1 >
< id , 0 >	< opAritmetico , 1 >	< ptoComa , >
< string , >	< id , 6 >	< print , >
< ptoComa , >	< ptoComa , >	< id , 8 >
< function , >	< return , >	< ptoComa , >
< id , 1 >	< id , 4 >	< cerrarCorchete , >
< int , >	< ptoComa , >	< switch , >
< abrirParentesis , >	< cerrarCorchete , >	< abrirParentesis , >
< string , >	< function , >	< id , 4 >
< id , 2 >	< id , 7 >	< cerrarParentesis , >
< cerrarParentesis , >	< abrirParentesis , >	< abrirCorchete , >
< abrirCorchete , >	< int , >	< case , >
< print , >	< id , 8 >	< cteEntera , 10 >
< id , 2 >	< cerrarParentesis , >	< dosPuntos , >
< ptoComa , >	< abrirCorchete , >	< id , 7 >
< id , 2 >	< if , >	< abrirParentesis , >
< asignacion , >	< abrirParentesis , >	< id , 1 >
< cadena , "Vamos a multiplicar" >	< id , 9 >	< abrirParentesis , >

< ptoComa , > < let , > < id , 3 > < string , > < asignacion , > < id , 2 > < ptoComa , > < print , > < id , 3 > < ptoComa , > < id , 4 > < asignacion , >	< opRelacional , 1 > < cteEntera , 0 > < cerrarParentesis , > < id , 7 > < abrirParentesis , > < id , 8 > < cerrarParentesis , > < ptoComa , > < id , 9 > < asignacion , > < id , 9 > < opAritmetico , 2 >	< id , 0 > < cerrarParentesis , > < cerrarParentesis , > < ptoComa , > < default , > < dosPuntos , > < break , > < ptoComa , > < cerrarCorchete , > < eof , >

tabla.txt

CONTENIDOS DE LA TABLA # 2 :

* LEXEMA : 'saludo'
ATRIBUTOS :
+ Tipo : 'tipo_cadena'
+ Despl : 0
---------- ----------
* LEXEMA : 'saludo2'
ATRIBUTOS :
+ Tipo : 'tipo_cadena'
+ Despl : 64
---------- ----------

CONTENIDOS DE LA TABLA # 3 :

* LEXEMA : 'x'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 0
---------- ----------

CONTENIDOS DE LA TABLA # 1 :

* LEXEMA : 'saludo'
ATRIBUTOS :
+ Tipo : 'tipo_cadena'
+ Despl : 0
---------- ----------
* LEXEMA : 'Multiplica'
ATRIBUTOS :
+ Tipo : 'tipo_funcion'
+ numParam : 1
+ TipoParam1 : 'tipo_cadena'
+ TipoRetorno : 'tipo_entero'
+ EtiqFuncion : 'EtMultiplica1'
---------- ----------
* LEXEMA : 'j'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 64
---------- ----------
* LEXEMA : 'a'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 65
---------- ----------
* LEXEMA : 'b'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 66
---------- ----------
* LEXEMA : 'printResultado'
ATRIBUTOS :
+ Tipo : 'tipo_funcion'
+ numParam : 1
+ TipoParam1 : 'tipo_entero'
+ TipoRetorno : 'tipo_vacio'
+ EtiqFuncion : 'EtprintResultado2'
---------- ----------
* LEXEMA : 'cont'
ATRIBUTOS :
+ Tipo : 'tipo_entero'
+ Despl : 67
---------- ----------

parse.txt

```
Descendente
1 2 6 45 56 3 46 47 43 49 45 52 53 7
  10 17 20 23 26 29
35 27 24 21 18 54 7 9 14 17 20 23 26 32
  27 24 21 18 54 6
45 57 17 20 23 26 29 35 27 24 21 18 54 7
  10 17 20 23 26 29
35 27 24 21 18 54 7 9 14 17 20 23 26 29
  35 28 29 35 27 24
21 18 54 7 12 41 17 20 23 26 29 35 27 24
  21 18 55 3 46 48
49 43 52 53 5 17 20 23 26 29 35 27 24 22
  23 26 31 27 24 21
18 9 16 37 17 20 23 26 29 35 27 24 21 18
  40 54 7 9 14 17
20 23 26 29 35 27 25 26 31 27 24 21 18 54 7
  10 17 20 23 26
29 35 27 24 21 18 55 2 8 17 20 23 26 29
  35 27 24 21 18 59
61 7 9 16 37 17 20 23 26 29 36 37 17 20
  23 26 29 35 27 24
21 18 40 27 24 21 18 40 62 63 60 61 7 13
  62 4
```

Árbol generado: https://drive.google.com/file/d/1hEL0FbLH5o-zkbnASM0FVomKxylqoSFv/view?usp=share_link

Casos incorrectos:

CASO 6

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #6: Probando un break fuera del switch (semántico)
*/

let cadena string;

let x int;
print "Introduce un numero del 1 al 5 :";
input  x;

switch (x) {

    case 1:
        print "Tienes un 1!!";
        break;
    case 2:
    case 3:
    case 4:
        print "Tienes un 2, un 3 o un 4!!";
        break;
    default:
        print "Tienes un 5!!";
}

print "Escribe tu nombre: ";
input cadena;
print "Hola: ";
print cadena;

let edad int;

print "Escribe tu edad: ";
input edad;
print "Tienes: ";
print edad;

if(x==5){
    break;
}
```

errores.txt:

Línea 38 - Código de error 211:
ERROR SEMÁNTICO - La sentencia BREAK solo puede declararse dentro del CASE O DEFAULT de un SWITCH

CASO 7

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #7: Error de operador relacional
*/

a = b + c;  /* Sumando variables con declaración implicita */

if (a == b)
    print "Iguales";

let cadena string = "HEY";
let x boolean = true;
let y boolean = true;

/* Usando variables globales dentro de la funcion */
function printCadena () {
    print cadena;
}

if (x&&y&&x&&y&&y==x)
    printCadena();
```

errores.txt:

Línea 19 - Código de error 216:
ERROR SEMÁNTICO - Las expresiones a comparar con == deben ser de tipo entero. Son de tipo logico y logico respectivamente

CASO 8

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #8 : Probando error T.S
*/

let XX int = 9;
let YY boolean = true;

if(XX==9 && YY&&true)
    print "Hola";

function fun string (boolean x, int y, string cad) {
    switch (y) {
        case 1:
            return "y es 1";
        case 2:
        case 3:
            break;
        case 9:
            return "y es 1";
            break;
    }

    let XX boolean = false;
    let XX boolean = true;

    if (y == 8)
        return "y es 8";

    if(x)
        return cad;

    let cadena2 string = "Buenas";
    return cadena2;
}

print fun(YY,10,"Hola");
```

errores.txt:

Línea 24 - Código de error 200:
ERROR SEMÁNTICO - El identificador XX ya ha sido declarado

CASO 9

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #9 : valor de retorno distinto al declarado
*/

let num int = 5;
let b boolean = true;

function Suma int (int a, int b) {

    j= a + b;
    return j;
    /* La función finaliza y devuelve el valor entero de la expresión */
}

if (num == 5)
    print "Bien!!";

print "Introduce un número para sumarle 18: ";
input num;

print "El resultado es: ";

if (b)
    print Suma(num,18);

function Multiplicacion boolean (int a) {
    a *= 100;
    if (a == 1000)
        return a;
    return a;
}

if(Multiplicacion(10))
    j = num + 40;
```

errores.txt:

Línea 26 - Código de error 229:
ERROR SEMÁNTICO - El tipo del valor de retorno (entero) no coincide con el tipo valor de retorno declarado (logico)

CASO 10

fichero_fuente.txt:

```
/*
    CASO DE PRUEBA #10 : probando errores lexicos que no generan tokens en
expresiones sintacticas correctas
*/

let saludo [ string;

function Multiplica int (string saludo) {
    print saludo;
    saludo  = "Vamos a multiplicar";
    let saludo2 string = saludo;
    print saludo2;
    j = a * b;
    return j;
}

function - printResultado (int x) {
    if (cont == 0)
        printResultado(x);
    cont = cont + 1;
    print x;
}

switch (j){
    case 10:
        printResultado(Multiplica(saludo));
    default:
        break;
}

a = 90000;
```

errores.txt:

Línea 5 - Código de error 50:
ERROR LÉXICO - carácter CAR: '[' no esperado.

Línea 16 - Código de error 50:
ERROR LÉXICO - carácter CAR: '-' no esperado.

Línea 30 - Código de error 60:
ERROR LÉXICO - El número introducido está fuera de rango.
NUM: 90000 es mayor que 32767.

Línea 30 - Código de error 105:
ERROR SINTÁCTICO - Token ; no pertenece a una expresión válida