2022

PRÁCTICA PROCESADORES DEL LENGUAJE

Procesador de Lenguaje JavaScript-PdL

GRUPO 13

INTEGRANTES:

* Mario López Estaire
* Andrés Bravo Francos
* Grettell Umpierrez Sardiñas – 210401



2022

Índice

**[Objetivos](#Obj)** [………………………………………………………………………………………………………………………….. 2](#Obj)

[Objetivos Comunes……………………………………………………………………………………………….. 2](#Obj_Comunes)

[Objetivos Específicos……………………………………………………………………………………….……. 2](#Obj_Esp)

[**El Proyecto** ……………………………………………………………………………………………………………………….. 2](#Proyecto)

[**Analizador Léxico** ………………………………………………………………………………………………………….. 3](#AL)

[Tokens………………………………………………………………………………………………………..………….. 3](#Tokens)

[Gramática del Lenguaje ……….………………………………………………………………………….……… 3](#GramaticaAL)

[Autómata Finito Determinista ……….…………….………………………………………………….……… 4](#AutomataAL)

[Acciones Semánticas ……….……………………………………………..……………………………….……… 4](#A_Sem_AL)

[Matriz de AFD …………..……….……………………………………………..……………………………….…… 5](#MatrizAFD)

[**Tabla de Símbolos** ……….……………………………………………..…………………………………………..….……… 6](#TS)

[**Analizador Sintáctico** ……….……………………………………………..……………………………….………………… 10](#AS)

[Gramática LL …………….………….………………………………………………………………………….……… 10](#gramaticaAS)

[First y Follow …………………………………………………………………………………………….……………… 12](#FirstFollow)

[Justificación Condición LL ………………………………………………………………………………………... 13](#Justif)

[Tabla Descendente LL ……..…………………………………………..………………………………………….. 17](#TASDescLL)

[**Analizador Semántico** …….………………………………………………………………….………………….……………. 17](#ASemantico)

[Gramática de Atributos con TDS …….………………………………………………………………….……… 17](#GramaticaASema)

[**Errores** …………………………………………………………………………………………………………………………………. 27](#Errores)

[Léxicos …………………………………………..……………………………………………………………….………… 27](#E_Lex)

[Sintácticos ……………………………………………………………………………………….……………………….. 27](#E_Sin)

[Semánticos ……………………………………………………………………………………….……………………….. 28](#E_Sem)

[**Anexos** ……………………………………………………………………………………….…………………………………….... 30](#Anexos)

[Código para Vast……………………………………………………………..…………….……………………….. 30](#VAST)

# *[OBJETIVOS](#Indice)*

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](#Indice)*

# 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 hace la función del main o programa principal del procesador.

# 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.

# *[ANALIZADOR LÉXICO](#Indice)*

**TOKENS**

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

# GRAMÁTICA

|  |  |
| --- | --- |
| | : | **LEYENDA:**  : {a-z,A\_Z}  : {0 - 9}  : cualquier carácter no contemplado en el nodo  : todos los caracteres – {\*}  : todos los caracteres – {\*,/}  : todos los caracteres – {“}  : tab, espacio, eol |

# AUTÓMATA

# 

# ACCIONES SEMÁNTICAS

|  |  |
| --- | --- |
| S→S: leer();S→A: lexema = c; leer();A→A: lexema = lexema + c; leer();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);}S→C: valor = char\_int(d); leer();C→C: valor = valor \* 10 + char\_int(d); leer();C→D: if (valor > 32767) then error (60);else generarToken(cte\_entera, valor);S→E: leer();E→F: leer();F→F: leer();F→G: leer();G→F: leer();G→G: leer();G→S: leer();S→K: lexema = “ ”; contador=0; leer();K→K: lexema = lexema + c; leer();contador++;K→L: if(contador > 64 ) then error (61);else generarToken(cadena, lexema); | S→H: leer();H→I: generarToken(opRelacional, 1);leer();H→J: generarToken(asignación, -);S→M: leer();M→N: generarToken(asigMultiplicacion, - );leer();M→O: generarToken(opAritmetico, 1 );S→P: leer();P→Q: generarToken(opLogico, 1 ); leer();S→R: generarToken(abrirParentesis, - );leer();S→T: generarToken(cerrarParentesis, - ); leer();S→U: generarToken(abrirCorchete, - ); leer();S→V: generarToken(cerrarCorchete, - ); leer();S→W: generarToken(ptoComa, - ); leer();S→X: generarToken(coma, - ); leer();S→Y: generarToken(dosPuntos, - ); leer();S→Z: generarToken(opAritmetico, 2 ); leer(); |

# MATRIZ AFD

# 

\*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 a 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 es 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 solo 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 es 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

# “\tATRIBUTOS : \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 # 2 :

# \* LEXEMA : 'a'

# ATRIBUTOS :

# + tipo : tipo\_entero

# + despl : 0

# ----------- ----------

# *[ANALIZADOR SINTÁCTICO](#Indice)*

# GRAMÁTICA

|  |  |
| --- | --- |
| // Axioma general// Sentencias compuestas     // Sentencias simples | // Expresiones   1. // Operaciones lógicas 3. //Operaciones relacionales 4. //Operaciones aritméticas (suma) 5. // Operaciones aritméticas (producto)   // Operandos |

|  |  |
| --- | --- |
| // Argumentos de función       // Valor de retorno      // Tipos de variables      // Declaración de funciones | // Inicialización de identificadores  // Cuerpo de switch |

# 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:

# porque

# porque

# porque

# No hay regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

# Las reglas de B cumplen la condición LL porque:

# porque

# porque

# porque

# porque

# porque

# porque

# No hay regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

# Las reglas de S cumplen la condición LL porque:

# porque

# porque

# porque

# porque

# porque

# porque

# porque

# porque

# porque

# porque

# No hay regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

# Las reglas de S’ cumplen la condición LL porque:

# porque

# porque

# porque

# No hay 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 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de P cumplen la condición LL porque:

# porque

# porque

# porque

# porque

# No hay regla 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de L cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de Q cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de X cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de T cumplen la condición LL porque:

# porque

# porque

# porque

# No hay regla 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de D cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor tanto se cumple la condición LL.

# Las reglas de K cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porquepor 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porque por tanto se cumple la condición LL.

# Las reglas de N cumplen la condición LL porque:

# porque

# porque

# porque

# No hay regla por tanto no se necesitan más comprobaciones para determinar que se cumple la condición LL

# Las reglas de Z cumplen la condición LL porque:

# porque

# No hay regla 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 por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porque

# Las reglas de Z’ cumplen la condición LL porque:

# Solamente hay dos reglas y una de ellas es por tanto no hay que comprobar si los first tienen símbolos terminales en común

# Como una de las reglas es

# porque

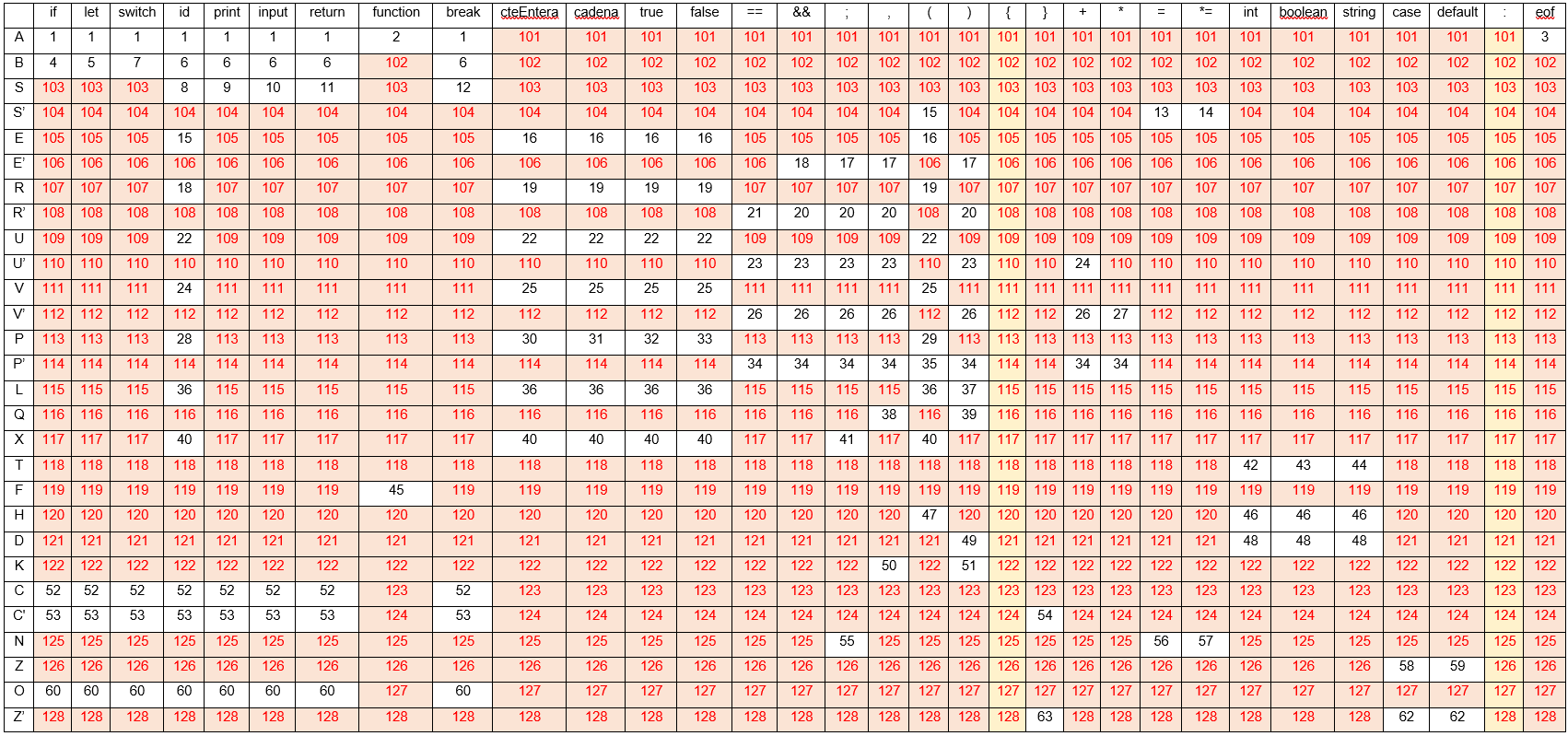
# La gramática obtenida para el Analizador Sintáctico Descendente LL(1) con Tablas es LL(1) porque:

# 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)**



|  |  |  |
| --- | --- | --- |
|  |  | Error Sintáctico |

# [ANALIZADOR SEMÁNTICO](#Indice)

# GRAMÁTICA DE ATRIBUTOS CON TDS

|  |  |
| --- | --- |
|  | 1.1)  TSG := crearTabla();  desplG = 0; |
|  | 1.2)  LiberarTabla(TSG);  desplG = null; |
|  |  |
|  |  |
|  |  |
|  | 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 |
|  | 6.1)  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) |
|  | 7.1)  S.funcion = B.funcion  S.switch := B.switch |
|  | 7.2)  B.tipo := S.tipo;  B.tipoRet := S.tipoRet; |
|  | 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; |
|  | 9.1)  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) |
|  | 10.1)  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) |
|  | 11.1)  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) |
|  | 12.1) 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 |
|  | 13.1) S.tipo := if (S.switch == true)      then 'tipo\_ok'  else 'tipo\_error' (211) |
|  | 14.1)  S'.tipo := E.tipo |
|  | 15.1)  S'.tipo := if (E.tipo == 'tipo\_entero')    then U.tipo    else 'tipo\_error' (212) |
|  | 16.1)  S'.tipo := L.tipo |
|  | 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) |
|  | 18.1)  E'.tipo := 'tipo\_vacio' |
|  | 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) |
|  | 20.1)  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) |
|  | 21.1)  R'.tipo := 'tipo\_vacio' |
|  | 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) |
|  | 23.1)  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) |
|  | 24.1)  U'.tipo := 'tipo\_vacio' |
|  | 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) |
|  | 26.1)  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) |
|  | 27.1)  V'.tipo := 'tipo\_vacio' |
|  | 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) |
|  | 29.1)  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) |
|  | 30.1)  P.tipo := E.tipo |
|  | 31.1)  P.tipo := 'tipo\_entero' |
|  | 32.1)  P.tipo := 'tipo\_cadena' |
|  | 33.1)  P.tipo := 'tipo\_logico' |
|  | 34.1)  P.tipo := 'tipo\_logico' |
|  | 35.1)  P'.tipo := 'tipo\_vacio'  P'.esfuncion := False |
|  | 36.1)  P'.tipo := L.tipo  P'.esfuncion := True |
|  | 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) |
|  | 38.1)  L.tipo := 'tipo\_vacio' |
|  | 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) |
|  | 40.1)  Q.tipo := 'tipo\_vacio' |
|  | 41.1)  X.tipo := E.tipo |
|  | 42.1)  X.tipo := 'tipo\_vacio' |
|  | 43.1)  T.tipo := 'tipo\_entero';  T.tamaño := 2; |
|  | 44.1)  T.tipo := 'tipo\_logico';  T.tamaño := 2; |
|  | 45.1)  T.tipo := 'tipo\_cadena';  T.tamaño := 128; |
| 1. {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; |
|  | 47.1)  H.tipo := T.tipo |
|  | 48.1)  H.tipo := 'tipo\_vacio' |
|  | 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]; |
|  | 50.1)  D.tipo := 'tipo\_vacio' |
|  | 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 |
|  | 52.1)  K.tipo := 'tipo\_vacio' |
|  | 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 |
|  | 54.1)  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 |
|  | 55.1)  C'.tipo := 'tipo\_ok';  C'.tipoRet == undefined; |
|  | 56.1)  N.tipo := 'tipo\_vacio' |
|  | 57.1)  N.tipo := E.tipo |
|  | 58.1)  N.tipo := if (E.tipo == 'tipo\_entero')  then E.tipo  else 'tipo\_error' (238) |
|  | 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) |
|  | 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) |
|  | 61.1)  B.funcion = O.funcion  B.switch := O.switch |
|  | 61.2)  O'.funcion = O.funcion  O'.switch := O.switch |
|  | 61.3)  O.tipo := if ( B.tipo == O'.tipo == 'tipo\_ok' )    then 'tipo\_ok'  else 'tipo\_error' (241) |
|  | 62.1)  O.tipo := 'tipo\_ok' |
|  | 63.1)  Z.switch := Z'.switch  Z.funcion = Z'.funcion  Z'.tipo := if (Z.tipo == 'tipo\_ok')  then 'tipo\_ok'  else 'tipo\_error' (242) |
|  | 64.1)  Z'.tipo := 'tipo\_ok' |

**[ERRORES](#Indice)**

# 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 \*.

**[ANEXOS](#Indice)**

***[CODIGO VAST](#Indice)***

/////// 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

}

aaaaaa