**(Universidad del Perú, DECANA DE AMÉRICA)**

**FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA**

**E.A.P. INGENIERÍA DE SISTEMAS**



**LENGUAJES Y COMPILADORES**

***TRADUCTOR PARA GRAMÁTICA VULCAN***

**PROFESOR**

[CARLOS AUGUSTO RUIZ DE LA CRUZ MELO](https://classroom.google.com/c/MzQyMjYzODc2ODI1)

**INTEGRANTES**

Bernabel Mancilla Axell Hernan 19200003

Chauca Herrera Roomel Fernando 19200152

Fernández Tolentino Kevin Kenyi 19200014

Paredes Carranza Nick Eugenio 19200114

**2021**

Introducción

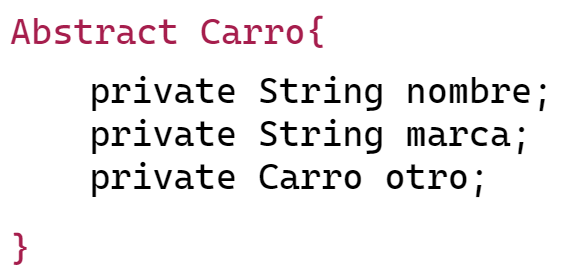
El siguiente documento sustenta todo el desarrollo del traductor para la gramática Vulcan. Este documento está divido en tres secciones: Análisis Léxico, Análisis Sintáctico y Análisis semántico. En cada sección se explicará el módulo con las clases correspondientes. Debemos precisar que este documento es un conglomerado de todos los documentos anteriores; sin embargo, está adaptado de tal forma que evita la redundancia.

También es importante decir que el software cuenta con dos módulos, es decir, dos partes. Un analizador léxico y un analizador sintáctico; el semántico se encuentra incluido dentro del sintáctico. Sin mas aclaraciones continuamos con la sustentación.

# Análisis Léxico

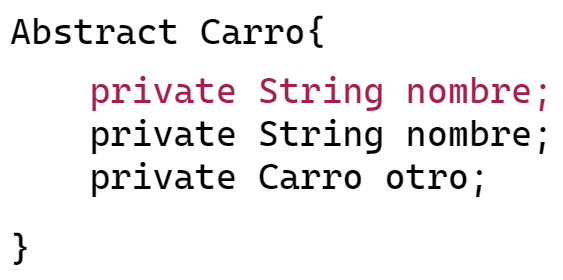
El análisis semántico es el proceso que se encarga de generar los Tokens y la Tabla de símbolos. Esto es posible porque el Analizador trabaja con un radio de lectura, este radio de lectura es necesario para poder reconocer los tipos definidos.

Por ejemplo cuando el analizador Léxico encuentra la cadena **Abstract ID { … }** automáticamente incluye ese ID a los tipos de datos definidos y cuando vuelva a encontrar un ID con este lexema lo reconocerá como TIPO\_DEFINIDO. Por están razón el analizador léxico debe tener un radio de conocimiento sobre un entorno.



Este radio de lectura que maneja el analizador léxico también es utilizado para generar la tabla de símbolos. La tabla de símbolos es generada en el proceso del análisis léxico.

Por ejemplos cuando reconoce la serie de lexemas

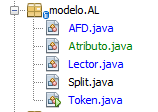


Automáticamente lo reconoce como una declaración y lo almacena en la tabla de símbolos. Este proceso también es el mismo para las funciones y clases. La tabla de símbolos nos permitirá llevar un registro de todas las variables que han sido declaradas y la posición en donde han sido declaradas, de esta forma al utilizar variables siempre se consultará a la tabla de símbolos si existe en esta.

La forma en la que hemos implementado la tabla de símbolos es con una **Tabla Hash** que soluciona las colisiones por encadenamiento. Escogimos esta estructura de datos porque el proceso de análisis puedes llegar a ser complejo y realizar consultar reiterativamente podría dilatar demasiado el proceso; sin embargo la Tabla Hash nos ofrece complejidad constante en todas sus operaciones, con lo que se podría decir que prácticamente la **Tabla Hash**  no afecta el tiempo de ejecución en lo absoluto. La desventaja de utilizar una Tabla Hash radica en la cantidad de memoria requerida para su implementación, ya que para almacenar, por ejemplo, 128 registros de declaraciones se necesita un Array de 2^10 (2 elevado a la 10); sin embargo las computadoras actuales cuentan con memoria de sobra para realizar estas implementaciones.

## Clases que conforman el Analizador Léxico.

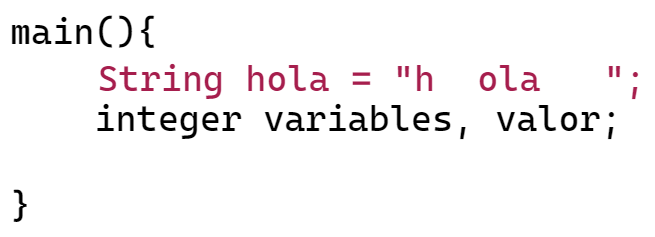
Las clases correspondientes al Analizador Léxico (AL) se encuentran en un submodulo de la carpeta modelo (modelo.AL).



Esta clase cuenta con 5 clases necesarias para su funcionamiento. Las explicaremos de menor a mayor complejidad.

#### Clase Split

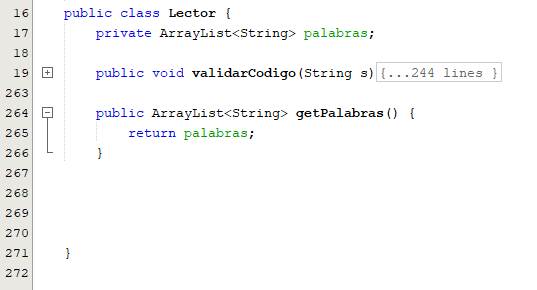
La clase Split es necesario para separar las palabras por los espacios, java ya ofrece este método; sin embargo la volvimos a crear porque este método split debía funcionar cuando se trataban las cadenas, por ejemplo.



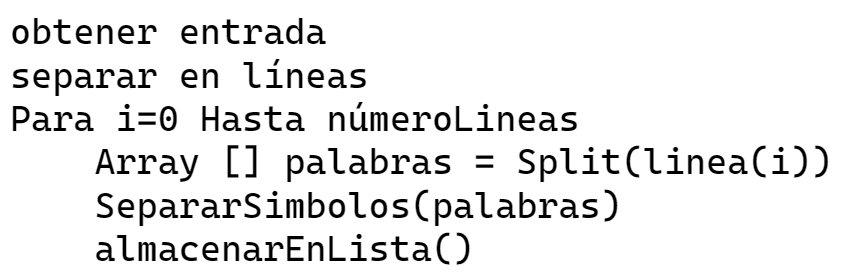
Si realizamos un split en la línea en rojo perderíamos la información de las cadenas, es decir una cadena con espacios no puede ser spliteada porque las cadenas se debe tomar literalmente como están escritas; sin embargo en el resto del código los espaciones son irrelevantes, por esta razón tuvimos que crear la clase Split nuevamente.

#### Clase Lector

La clase Lector corresponde a la clase que se encargará de segregar los lexemas, cuando hablamos de segregar nos referimos a separar ya que la entrada recibida siempre es una cadena; sin embargo, después de este proceso tenemos varias cadenas.



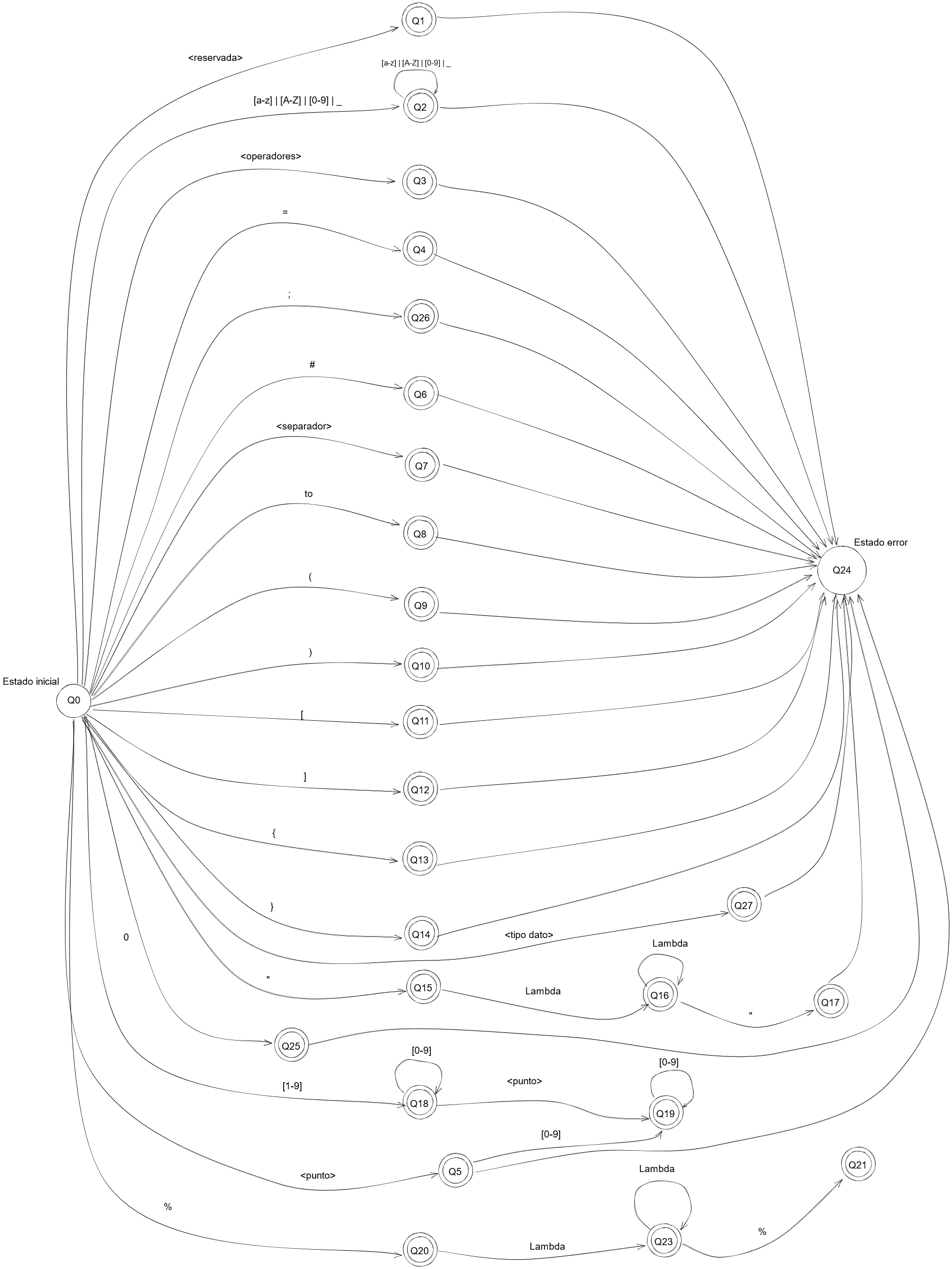
El método que se encarga de este proceso es validarCodigo que recibe como parámetro una cadena. Todas las cadenas separadas se almacenan en un ArrayList, estas cadenas separadas reciben el nombre de palabras. El algoritmo que se utiliza para este proceso es muy engorroso con lo cual adjuntaremos en pseudo código.



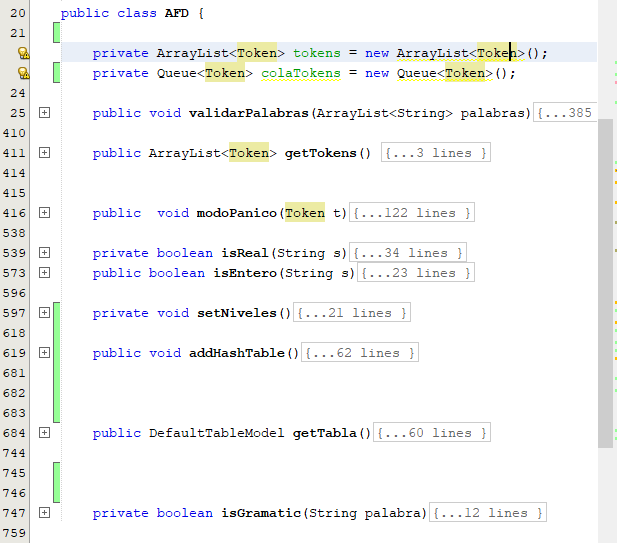
Este es el algoritmo utilizado para generar las palabras. Lo presentamos de esta forma porque leerlo directamente en código puede resultar tedioso.

#### Clase AFD

La clase AFD es la encargada de recibir las palabras y analizarlas para generar los Tokens, esta clase es un Automata Finito Determinista. Adjuntamos el grafo del Automáta. Con este Automata es que se generan los Tokens.



Los métodos de la clase AFD son los siguientes.

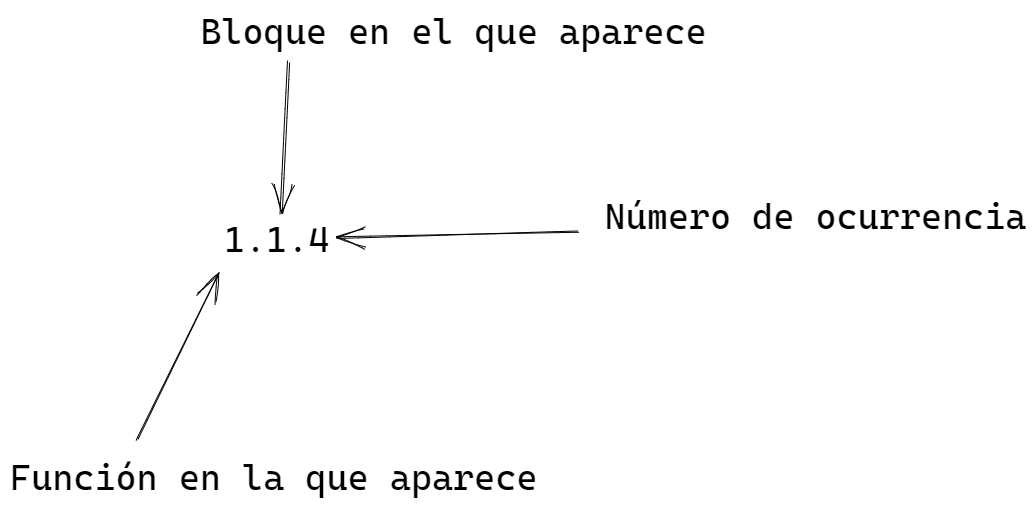


El método validarPalabras es el encargado de generar los Tokens, si este método detecta un error a través del Automáta entra a una condicional donde valida si es posible validar el modo pánico o no. En caso se pueda aplicar realizar el modo pánico.

El método modoPanico es el encargado de realizar la recuperación del error.

Los métodos isReal e isEntero son pequeños autómatas que ayudan al modoPanico a realizar sus tareas, estos autimatas reconocen números reales y números enteros respectivamente.

El método setNiveles es el método encargado de establecer los niveles en el que fueron reconocidos los Tokens. El nivel está representando de la siguiente forma



Esta notación para el nivel de un Token nos permite registrar la información necesaria de un Token para más adelante realizar las validaciones respecto a si una declaración ha sido validada y su ámbito.

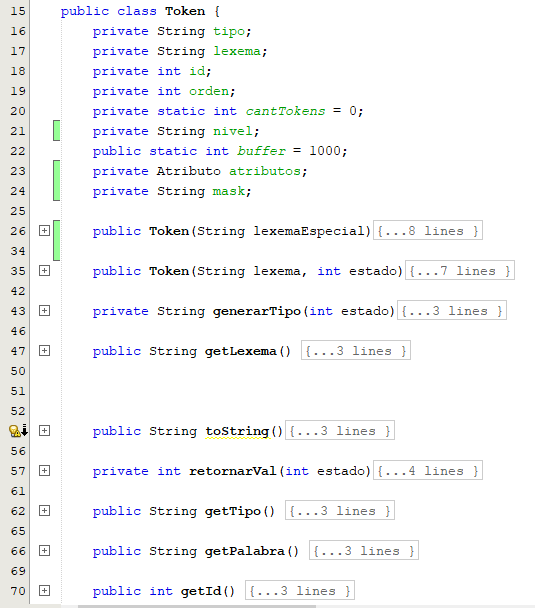
La función isGramatic es la función que determina si un lexema pertenece a la gramática.

La función addHashTable es la función que registra todas las declaraciones de variables, funciones y clases en la HashTable que hemos implementado.

Por último tenemos el método getTabla que es el encargado de obtener la tabla de los Tokens para poder visualizarlo.

#### Clase Token

Por último tenemos la clase Token que es la clase que nos permite guardar toda la información respecto a los lexemas.



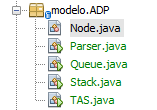
Todos estos atributos es la información que corresponde a un solo Lexema, es decir un lexema está asociado a toda esta información. Esta información es de vital importancia en la fase de análisis semántico que es el proceso que más información necesita. También implementamos los respectivos getter y setters.

# Análisis sintáctico

Para implementar el Analizador sintáctico escogimos un Analizador Descendente Predictivo debido a que el tiempo necesario para realizar uno en comparación con los LR es poco. La tabla que utiliza este ADP está adjunta en el resto de los archivos con el programa junto con la gramática y los conjuntos PRIMERO-SIGUIENTE.

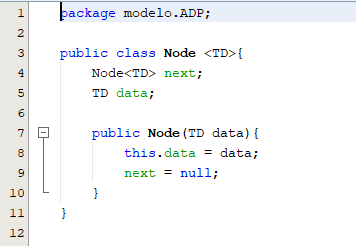
El ADP se encuentra en el submódulo modelo.ADP.

La carpeta ADP contiene cinco clases importantes: Node, Parser, Queue, Stack y TAS. Explicaremos de menor a mayor relevancia.



#### Clase Node

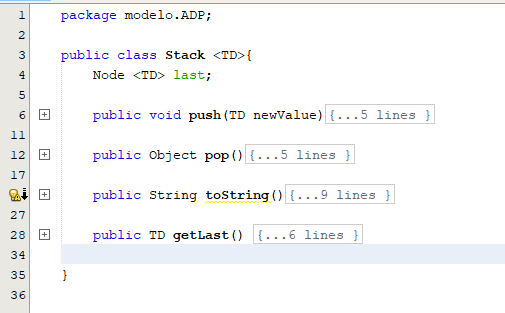
Esta clase Node, en español Nodo, representa la unidad básica de las estructuras de datos como la Pila, la Cola, Arboles, etc. Cada vez que alguien desee implementar una de estas estructuras primero debe implementar esta clase.



Esta clase como se observa en el código es genérica, ya que sería ineficiente crear una Clase nodo para guardar cadenas, otra para guardar enteros, etc. Al crear una clase genérica nos permite reutilizarla en cualquier ámbito para cualquier estructura de dato de cualquier tipo.

#### Clase Stack

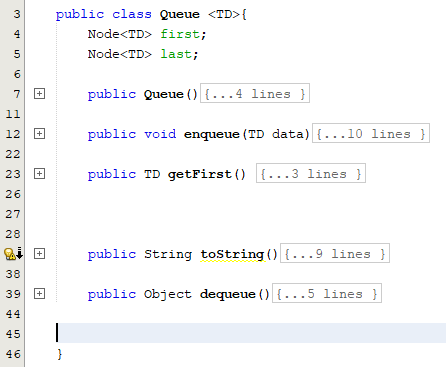
La clase Stack, en español Pila, es una estructura de datos que nos permitirá almacenar nuestras producciones para compararlo con la entrada.



En esta clase hemos implementado los métodos inherentes a cualquier Clase de Pila, push y pop. El método push sirve para añadir un elemento a la Pila y pop para retornar un elemento, pero sacándolo de la Pila. El método getLast nos sirve para consultar quien fue el último en ingresar a la Pila y por último el método toString que nos ayudará a visualizar la información de la Pila. Esta clase utiliza un Nodo genérico para almacenar los datos.

#### C lase Queue

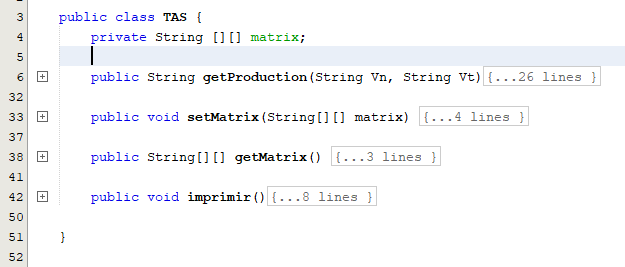
La Clase Queue, en español Cola, es una clase que hemos implementado para almacenar toda la entrada de Tokens que el Analizador Léxico nos retornará. Debe ser una cosa porque los Tokens deben salir de acuerdo con como ingresan, es decir, el primer Token leído es el primero que debe ser analizado, es decir, la estructura que mejor se adapta para almacenar los Tokens es la Clase Cola.



La clase Cola o Queue implementa los métodos inherentes a cualquier Cola, enqueue (encolar) y dequeue (decolar), luego también implementamos el método getFirst que nos permitirá consultar el primer elemento de la cola y por último el método toString que nos permitirá consultar el contenido de la Cola. Esta clase utiliza Nodos genéricos.

#### Clase TAS

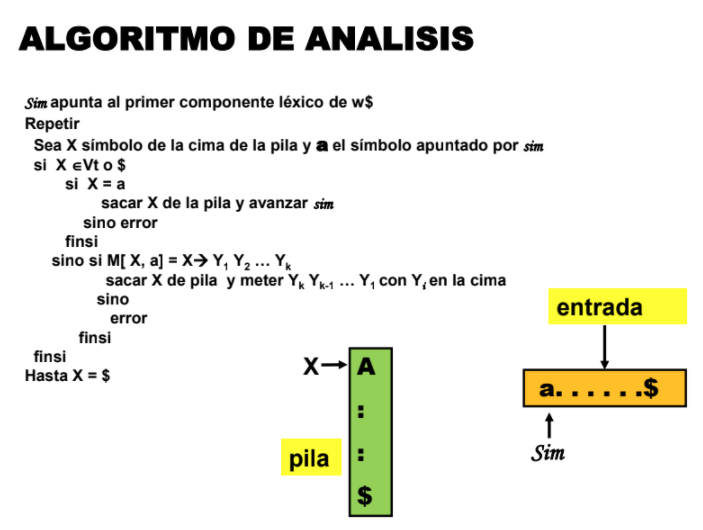
La clase TAS representa la clase para instanciar una Tabla de Análisis Sintáctico. Una tabla de análisis sintáctico es la encarga de decirnos que operación hacer cuando sucedan las intersecciones de los elementos de la Pila con la entrada.



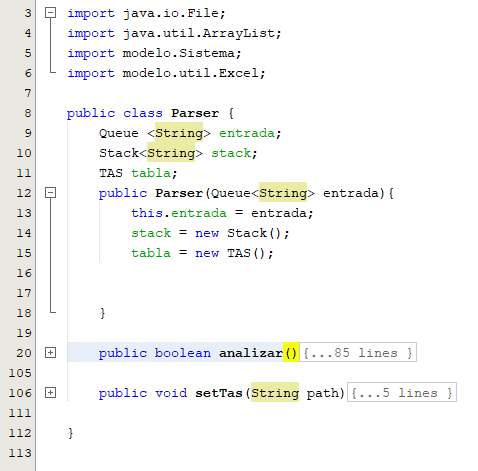
Para esta clase, hemos implementado el método getProduction que nos servirá para obtener la salida recibiendo como entrada la intersección de un Vn (Valor no termianl) y un Vt (Valor terminal), el método setMatrix que es el encargo de establecer la TAS a utilizar, getMatrix y por último el método imprimir que nos permitirá visualizar el contenido de la matriz. Esta Clase solo se instancia una vez y es utilizada de principio a fin por el Parser en su proceso de validación.

#### Clase Parser

La Clase Parser es la encargada de hacer la validación de la sintaxis. Para realizar este proceso de validación hemos utilizado el algoritmo mostrado en clase.

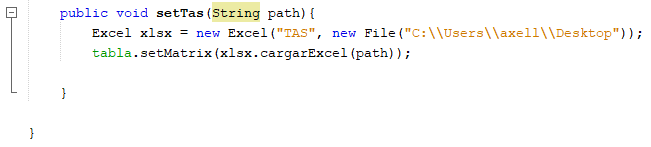


Este algoritmo está implementado en el método analizar. Como se observa esta Clase requiere una Cola (Queue) para almacenar la Entrada de Tokens, una Pila (Stack) para almacenar las producciones y una TAS. A continuación, explicaremos los métodos de esta clase.



##### Método setTas

Este método es el encargado de cargar la tabla de análisis sintáctico a la Clase TAS, este proceso lo hace a través de un intermediario que es la clase Excel. La clase Excel carga el contenido del Excel a una matriz y lo retorna.



##### Método analizar

El método analizar es la implementación del algoritmo enseñado en clase, hemos agregado los comentarios donde se visualiza la línea que hemos implementado del algoritmo en el código.



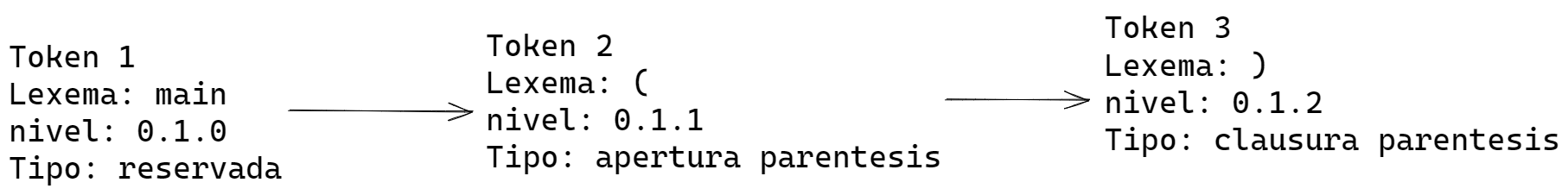


Las otras líneas que están fuera de los comentarios son necesarias para obtener la visualización requerida en las interfaces.

Debemos precisar que el método de análisis es realizado a nivel de Token y no de lexemas como lo hemos realizado en procesos anteriores, esto es dedibo a que existe un método oculto que es la llamada al semántico. Con lo cual nuestra entrada ya no es una serie de cadenas, sino, una serie de Tokens. Es decir, pasamos de tener esto:



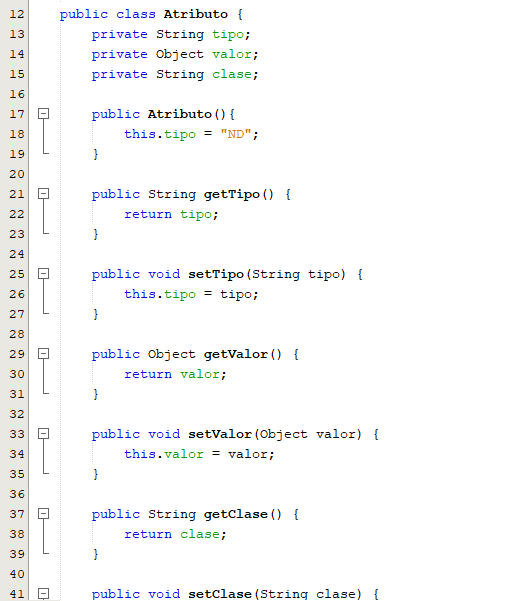
A tener esto:



# Análisis semántico

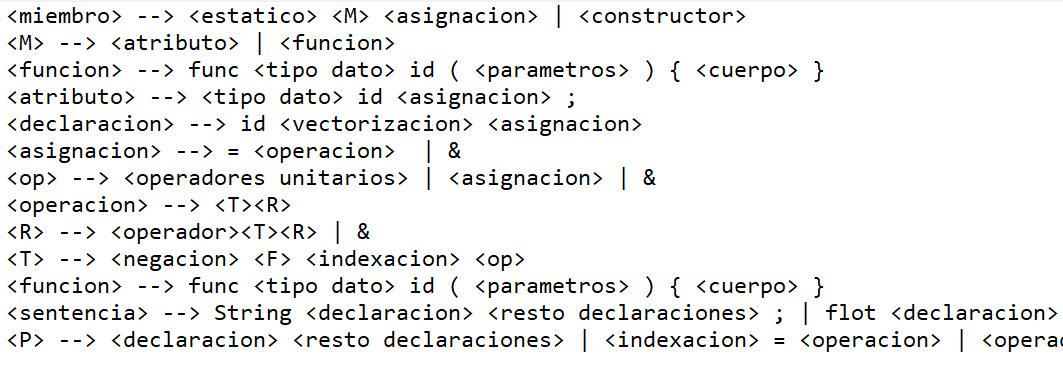
#### Clase Atributo

La clase Atributo es la necesaria para registrar toda la información que traen los Tokens de entrada consigo. Esta información es registrada en los Atributos y se validan en tiempo de ejecución si existe congruencia entre los atributos de un no terminal y otro valor no terminal. En esta clase también se implementaros los respectivos getters y setters. Debemos precisar que esta clase ha sido creada en el parque modelo.AL.



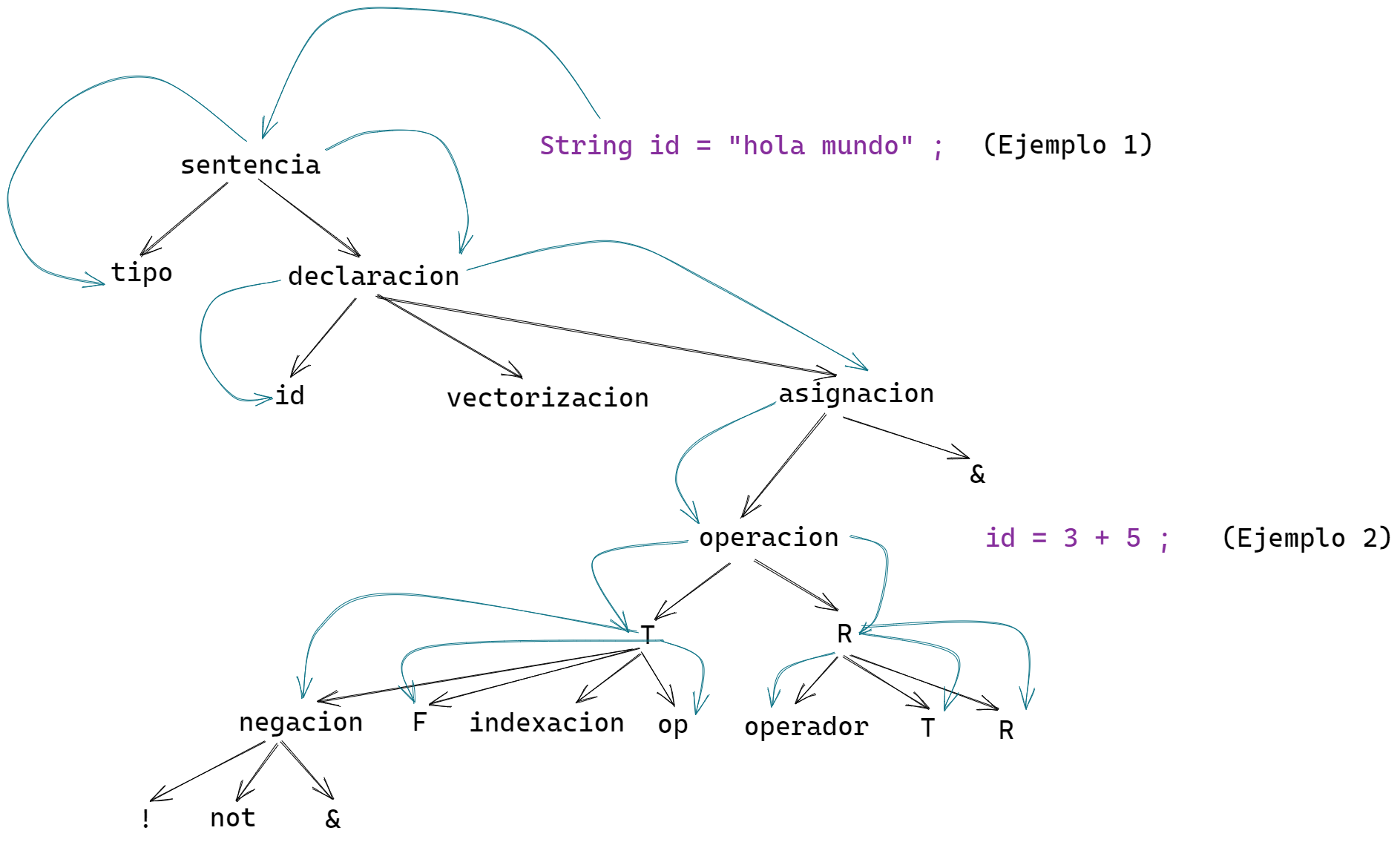
#### Algoritmo de Analisis semántico

La parte del análisis semántico consiste en un método que está al final del método analizar de la clase Parser del ADP. Para realizar este proceso nos centramos en algunas producciones en particulares. Las producciones son estas:

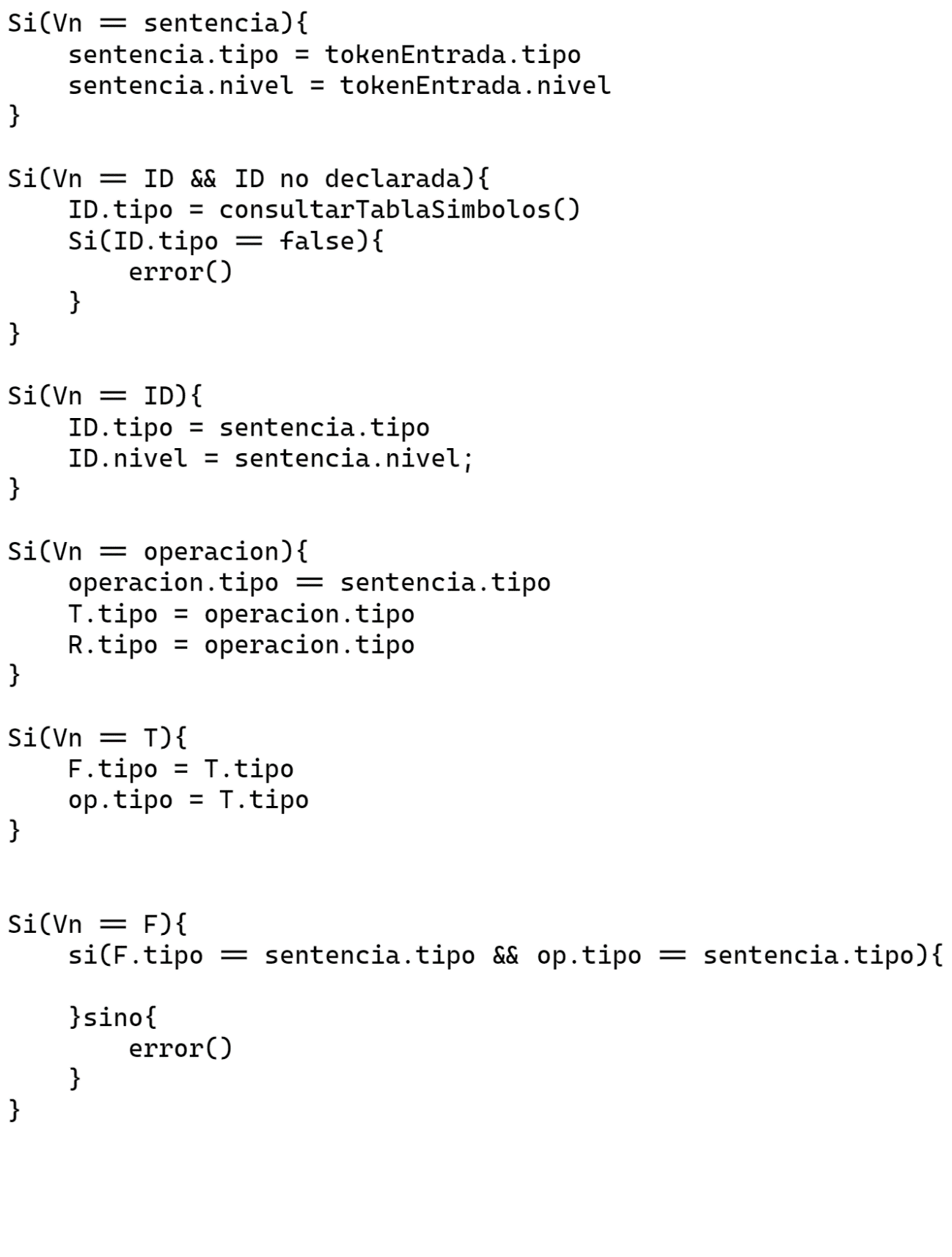


En estas producciones es que radican todas las operaciones de asignación, sumas, restas, llamadas a funciones, etc. Es por esto que cada No terminal este asociado a un Atributo con el mismo nombre. El

Estas producciones generan el siguiente Arbol decorado.



Las flechas azules nos indican como viaja la información del Token hacia los atributos. La información se obtiene en primera instancia del Token de entrada y va viajando hacia los nodos hijos. Si se tratara de una declaración de variable la información vendría desde el nodo Sentencia, mientras que si solo se utilizara una operación con un id el árbol partiría desde el nodo Operación.

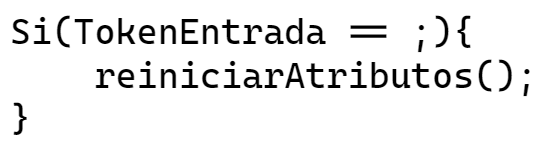
Las reglas que hemos utilizado son las siguientes:

Este proceso se realizará repetitivamente. Y nos servirá a realizar el análisis semántico para la asignación de tipos, es decir para que existe congruencia entre estos. Debemos precisar que existen más reglas; sin embargo estas son las más importantes, también es necesario decir que gran parte del trabajo lo realizar el método consultarTablaSimbolos, ya que es método recupera tipos de los identficadores y es el de una función recupera el número de variables y sus tipos y lo almacena en un Atributo auxiliar.

La implementación en código corresponde a la siguiente.



También debemos aclarar la ultima regla que es la regla finalizadora.



En nuestra gramática el terminal “;” sirve para finalizar una instrucción con lo cual para que en el proceso de análisis de las siguientes instrucciones no exista un error debido a que hay información que persiste en los atributos de instrucciones pasadas se realiza un proceso de redeclaración de variables, este proceso solo se realizará cuando se lea un “;”.

# Clases de apoyo

#### Clase HashTable

La clase HashTable corresponde a la implementación de una tabla hash o de dispersión que maneja un array de una dimensión especifica y asigna los elementos a través de una función hash y los recupera con esta misma función.



Si no se aprecia, se puede consultar el código directamente. Los métodos implementados son hash, isInLIsta, AddDato, searchDato, imprimirDatos, imprimirListaEnlazada.

Utilizamos métodos de listas porque esta Clase soluciona el problema de las colisiones a través de encadenamiento, es decir, creando listas enlazadas.

Tambíen encontramos clases internas como la Clase Nodo y la Clase Dato. La clase Nodo nos permite crear las listas enlazadas y la clase Dato es el valor que contiene el Nodo.

#### Clase Excel

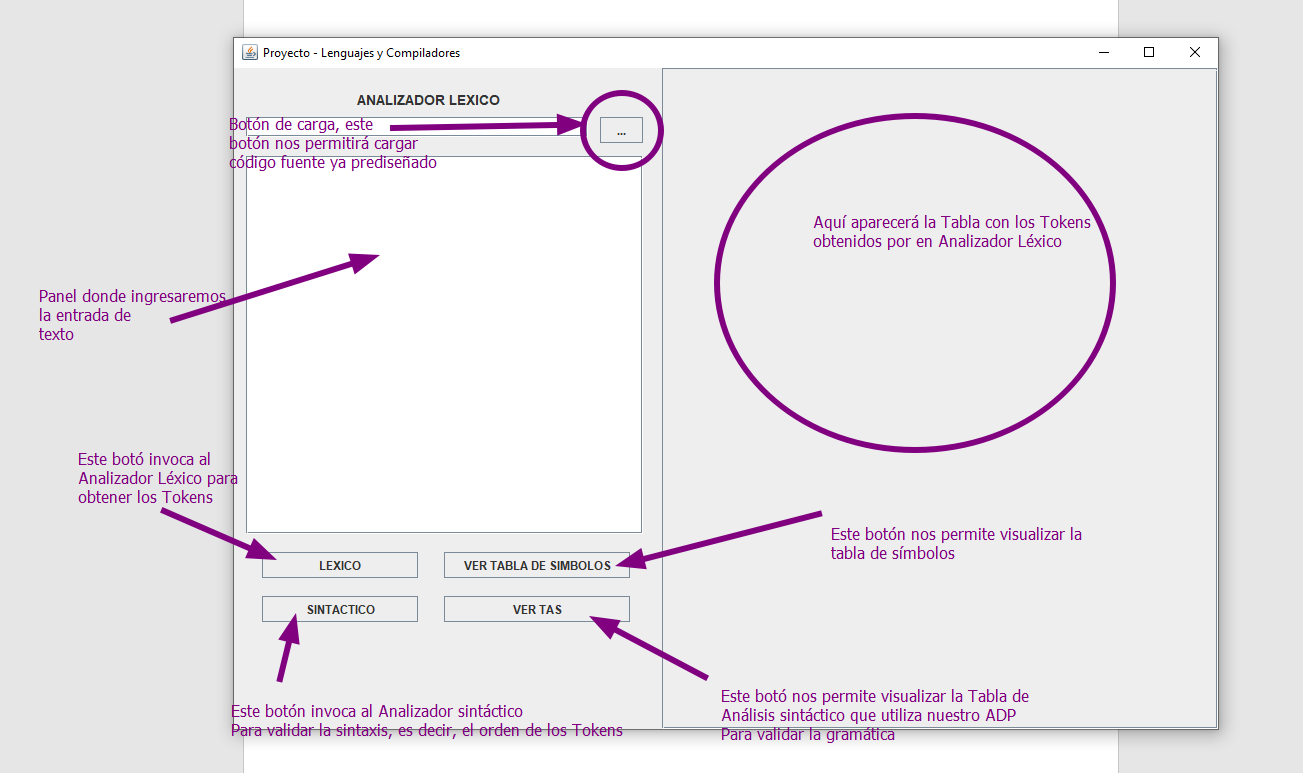


La clase Excel corresponde a la clase para exportar la tabla de producciones y para cargar el TAS

# Manejo del Software

### Clase Vista

La clase Vista hace referencia al Menú principal. Es la primera interfaz que visualizaremos cuando ejecutemos el programa. En esta interfaz deberemos ingresar un texto o podemos cargarlo a partir de un archivo de texto plano con el botón de carga. Luego debemos invocar al Analizador Léxico para obtener los tokens y por último debemos llamar al sintáctico para validar la sintaxis. Los componentes se explican en la siguiente imagen



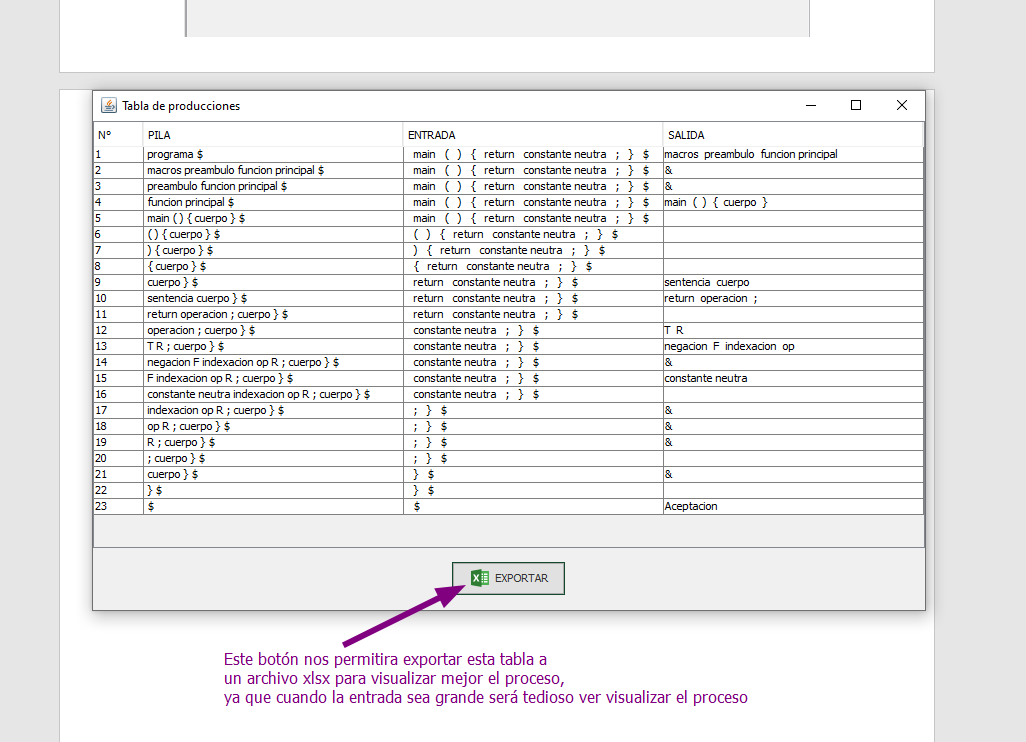
### Clase SymbolTable

La clase TablaSimbolos o SymbolTable es una clase para visualizar la tabla de símbolos. Aquí se muestra el resumen de todos los tokens leídos por nuestro Analizador Léxico. En esta clase no tenemos nada que explicar ya que es solo para visualizar, es decir no hay interacción con el usuario.



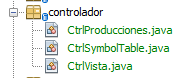
### Clase VistaProducciones

Es la interfaz invocada después de finalizar el análisis sintáctico, este se mostrará todo el proceso que ha realizado el programa para determinar si la entrada es válida o no. En la siguiente imagen se muestra la interfaz mostrando el proceso para validar cierta entrada.



## Carpeta controlador

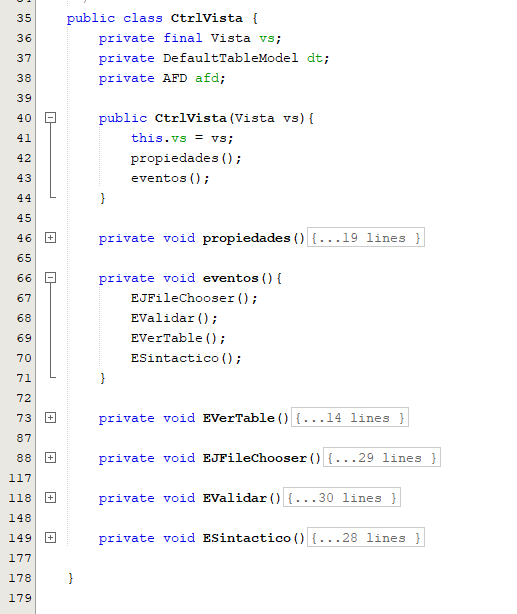
Esta carpeta contiene toda la lógica del programa, es decir las interacciones que deben existir entre las interfaces y las clases. Esta clase contiene tres clases controladoras.



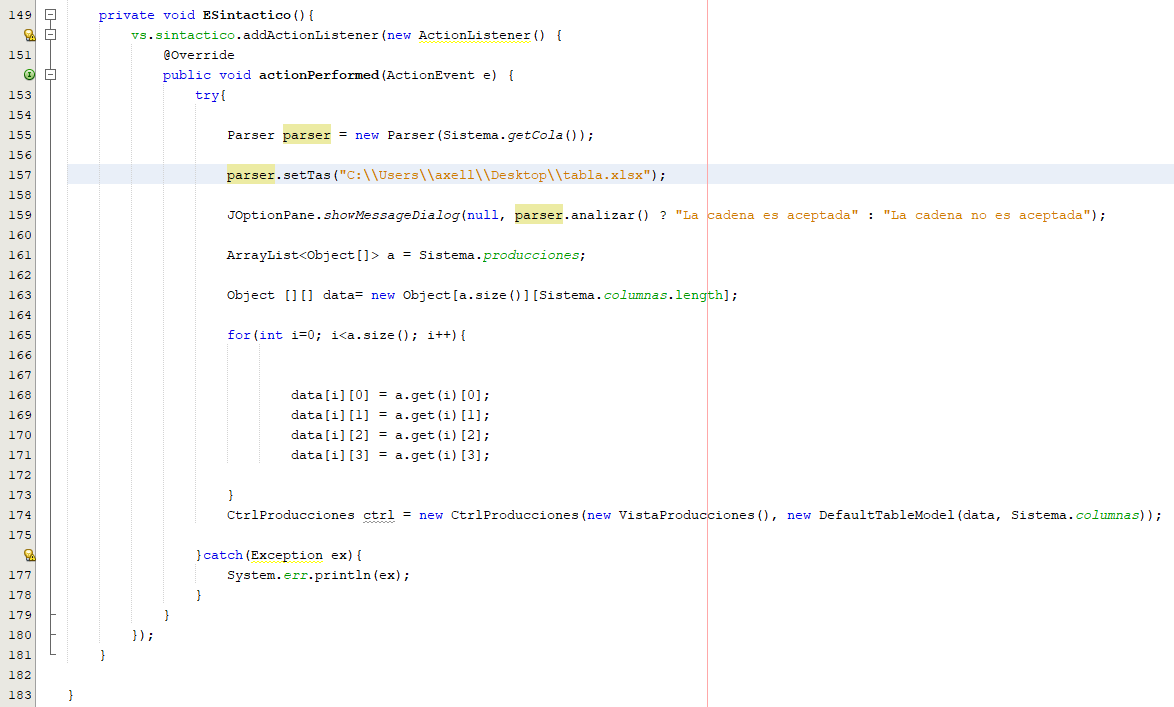
No tiene mucho mérito explicar esta carpeta ya que se trata del manejo de una serie de eventos, en tal caso solo explicaremos la clase con más contenido ya que esta explicación sirve para el resto, ya que todas hacen lo mismo.

#### Clase CtrlVista

El controlador de la Clase Vista (CtrlVista)



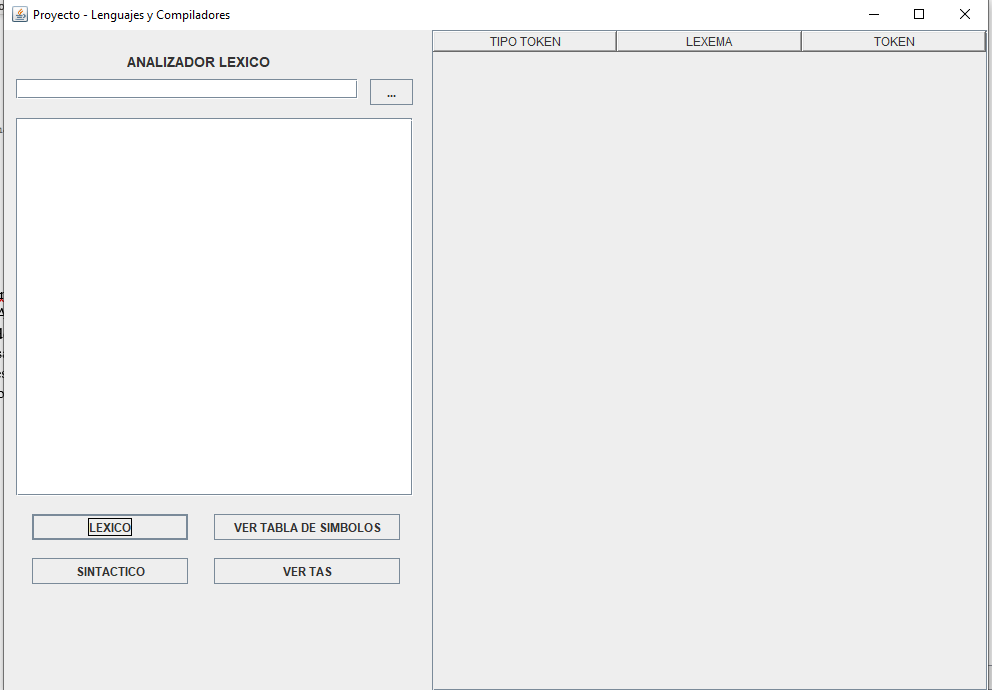
El controlador se instancia recibiendo una interfaz de tipo Vista, luego le asigna las propiedades y el manejo de eventos. Los eventos que asigna están implementados con métodos que empiezan con la letra E, por ejemplo, EVerTable, EValidar, etc. Estos eventos para este proyecto en particular son eventos de botón. Cada vez que presionemos el botón este método le dirá al programa que tarea hacer. Por ejemplo, El método ESintactico le dice al programa que hacer cuando se presiona el botón sintáctico, en la siguiente imagen se puede ver que el contenido de este método.



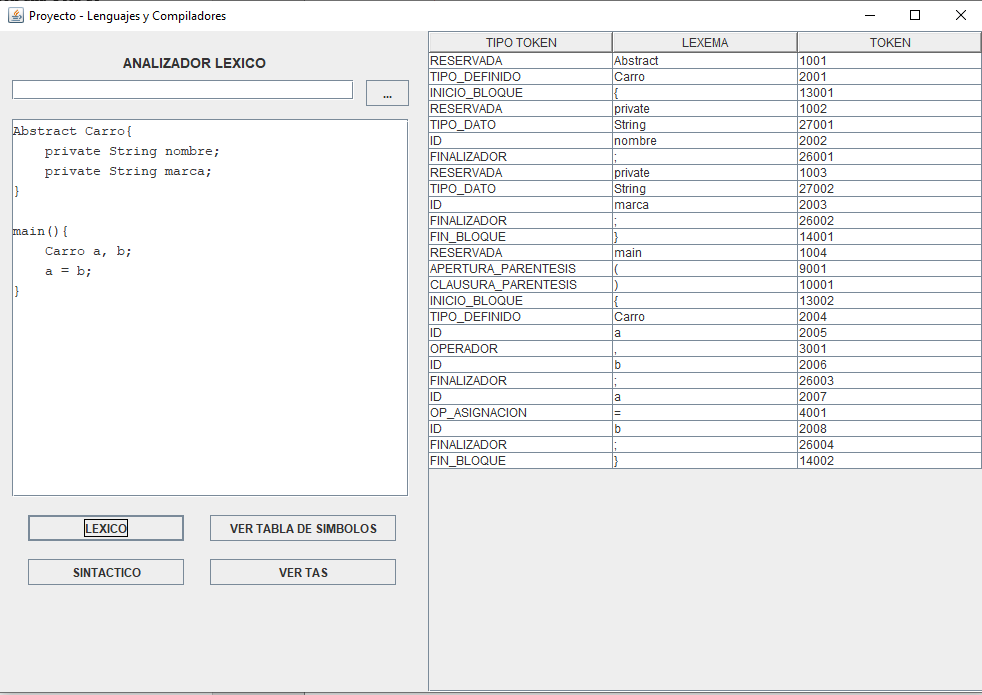
Como se observa, este método instancia un objeto de la Clase Parser con una Cola de Token que el Sistema le proporciona, luego este parser establece el TAS que utilizará y luego en la línea 158 envía un mensaje de acuerdo con el proceso de análisis que ha realizado los Tokens. Por último, llama a la siguiente interfaz pasando la tabla de producciones que ha obtenido en su proceso de análisis. Debemos que este proceso puede ser interrumpido por la fase semántica generando nuevos errores que no estaba previstos.

# Analisis de resultados

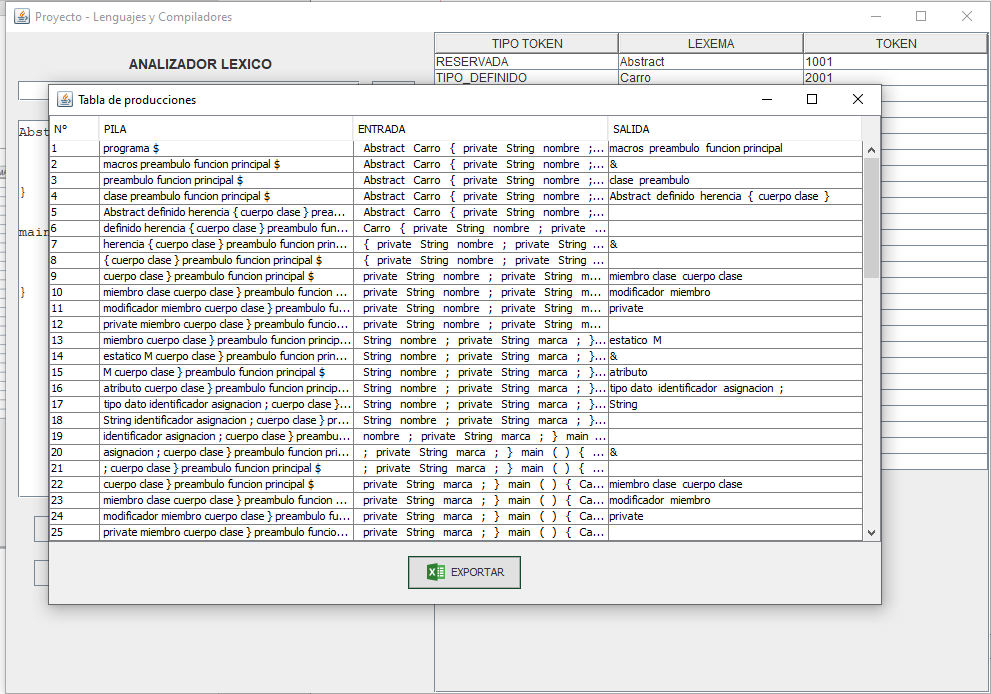
1. Ejecutamos el programa



1. Ingresamos la entrada y llamamos al anilzador léxico



1. Procedemos a la fase sintáctica



1. Análisis éxito de los resultados y fin de la ejecución