



E.A.P. INGENIERÍA DE SISTEMAS

“Analizador Sintáctico JAB”

Curso: Lenguajes y Compiladores

Integrantes:

Castillo Cornejo, Jeffrey Bryan 18200058

Porras Muñoz, Bruno Franchesco 18200176

Quispe Vega, Anthony Yair 18200179

Estrada Vargas, Erick Alexander 18200021

CICLO:

2020-II

**ANALISIS SINTACTICO**

**1. DEFICION**

Un **analizador sintáctico** (o *parser*) es un programa informático que analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal. El término proviene del latín *pars*, que significa parte (del discurso).

**2. GRAMATICA FORMAL**

Una **gramática formal** es una estructura matemática con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje formal o natural.

Los componentes de esta estructura son 4:

* Conjunto de reglas (P)
* Conjunto de caracteres terminales (VT)
* Conjunto de caracteres no terminales (VN)
* Axioma de la gramática (S)

A continuación, presentaremos la gramática de nuestro lenguaje JAB.

**G = {VT, VN, P, S}**

**VN = {** S, A, B, AMBITO, CLASE\_BASE, CUERPO, FUNCION\_PRINCIPAL, INSTRUCCIONES, C, D, CONSTRUCTOR, PARAMETROS, E, VECTOR, TIPO\_DATO, F, SP, SV, SF, MAS\_VARIABLES, G, H, MAS\_PARAMETROS, I, J, INSTRUCCIONES, K, L, SENT\_ASIG, SENT\_CF, SENT\_LECT, SENT\_ESCR, SENT\_INV, CREAR\_OBJETOS, PARAMETROS2, MAS\_PARAMETROS2, M, N, SENT\_ASIG2, EXP1, SENT\_IF, SENT\_WHILE, SENT\_FOR, INSTR\_COND, SENT\_SINO, OPER\_RELAC, EXPR\_CAD, Q, P, X, Z, TERMINSTR\_COND2, OPER\_LOGICO, OPER\_LOGICO1, OPER\_ARITMETICO, CONSTANTE,IRT1**}**

**VT = {** CLASE, Id, PUBLICO, PRIVADO, extiende, PRINCIPAL, (, ), {, }, ENTERO, REAL, CADENA, CARACTER, LOGICO, RETORNAR, VACIO, OBJETO, @, ,, ;, :, verdad, falso, Cad, Car, Cn, SI, SINO, MIENTRAS, PARA, LEER, ESCRIBIR,CONSTANTE, \_, +, -, \*, /, %, !, >, <, =, <=, <=, &&, || , [ , ] **}**

**P = {**

<S> ⇒ <A> <B>

<B> ⇒ <A> <B> | λ

<A> ⇒ CLASE <AMBITO> Id <CLASE\_BASE> { <CUERPO> <FUNCION\_PRINCIPAL> }

<AMBITO> ⇒ PUBLICO | PRIVADO | λ

<CLASE\_BASE> ⇒ extiende Id | λ

<FUNCION\_PRINCIPAL> ⇒ PRINCIPAL ( ) { <INSTRUCCIONES> } | λ

<CUERPO> ⇒ <C> <D> <CONSTRUCTOR>| λ ***// <C>-Variables globales o funciones***

<D> ⇒ <C> <D> | λ

<C> ⇒ <E>

<CONSTRUCTOR> ⇒ PUBLICO Id (<PARAMETROS>) {<INSTRUCCIONES>}

<E> ⇒ <TIPO\_DATO> Id <F> | <SP> | <CONSTANTE>

<F> ⇒ <SV> | <SF> | <VECTOR> ***// <SV>-más variables // <SF>-para funciones***

<VECTOR> ⇒ [ Cn ];

<CONSTANTE> ⇒ CONSTANTE Id : Cn ;

<TIPO\_DATO> ⇒ ENTERO | REAL | CADENA | CARACTER | LOGICO

<SV> ⇒ <MAS\_VARIABLES> **;**

<MAS\_VARIABLES> ⇒ <G> <H> | λ

<H> ⇒ <G> <H> | λ

<G> ⇒ **,** Id <SV1>

<SF> ⇒ ( <PARAMETROS> ) { <INSTRUCCIONES> RETORNAR Id ; }

<SP> ⇒ VACIO Id ( <PARAMETROS> ) { <INSTRUCCIONES> }

<PARAMETROS> ⇒ <TIPO\_DATO> Id <MAS\_PARAMETROS> | λ

<MAS\_PARAMETROS> ⇒ <I> <J> | λ

<J> ⇒ <I> <J> | λ

<I> ⇒ **,** <TIPO\_DATO> Id

<INSTRUCCIONES> ⇒ <K> <L> | λ

<L> ⇒ <K> <L> | λ

<K> ⇒ <TIPO\_DATO> Id <SV1> <SV> | <SENT\_ASIG> | <SENT\_CF> | <SENT\_LECT> | <SENT\_ESCR> | <SENT\_INV> | <CREAR\_OBJETOS> ;

<CREAR\_OBJETOS> ⇒ OBJETO Id Id (<PARAMETROS>)

<SENT\_INV> ⇒ ; Id\_clase @ Id\_objeto (<PARAMETROS2>)

<PARAMETROS2> ⇒ Id <MAS\_PARAMETROS2> | λ

<MAS\_PARAMETROS2> ⇒<M><N> | λ

<N> ⇒ <M> <N> | λ

<M> ⇒, Id

<SV1> ⇒ : <SENT\_ASIG2> | λ

<SENT\_ASIG> ⇒ Id : <SENT\_ASIG2> ;

<SENT\_ASIG2> ⇒ <EXP1> | verdad | falso | <SENT\_LECT> | Cad | Car

<SENT\_CF> ⇒ <SENT\_IF> | <SENT\_WHILE> | <SENT\_FOR>

<SENT\_IF> ⇒ SI ( <INSTR\_COND> ) { <INSTRUCCIONES> } <SENT\_SINO>

<SENT\_SINO> ⇒ SINO { <INSTRUCCIONES> } | λ

<SENT\_WHILE> ⇒ MIENTRAS ( <INSTR\_COND> ) { <INSTRUCCIONES> }

<SENT\_FOR> ⇒ PARA (ENTERO Id : Cn ; Id <OPER\_RELAC> Cn ; Id + Cn) { <INSTRUCCIONES>}

<SENT\_LECT> ⇒ LEER ( ) ;

<SENT\_ESCR> ⇒ ESCRIBIR ( <EXPR\_CAD> <Q> ) ;

<EXPR\_CAD> ⇒ Cad | Id

<Q> ⇒ <P> <Q> | λ

<P> ⇒ \_ <EXPR\_CAD> ***//para concatenar***

<EXP1> ⇒ <X> <Z> | ( <EXP1> ) <Z>

<X> ⇒ Cn | Id

<Z> ⇒ <OPER\_ARITMETICO> <EXP1> | λ

<INSTR\_COND> ⇒ <IRT1> <INSTR\_COND2> | ( <INSTR\_COND> ) <INSTR\_COND2>

<INSTR\_COND2> ⇒ <OPER\_LOGICO> <INSTR\_COND> | λ

<IRT1> ⇒ Id <OPER\_RELAC> Id | <OPER\_LOGICO1> ( Id <OPER\_RELAC> Id )

<OPER\_RELAC> ⇒ > | < | >= | <= | <> | =

<OPER\_LOGICO> ⇒ && | ||

<OPER\_LOGICO1> ⇒ !

<OPER\_ARITMETICO> ⇒ + | - | \* | / | % **}**

**OBSERVACIÓN**

**Cad**: Cadena. Ejm "Hola, mundo" \_ "concatenación"

**Car**: Caracter. Ejm 's'

**Cn**: Número. Ejm 1,2.35,-3,-4.36,...

**Id**: Identificador

**3. IMPLEMENTACION DEL ANALIZADOR SINTACTICO**

**3.1 Metodología a usar**

Nuestro compilador realizara un método de análisis TOP-DOWN, específicamente el Análisis Descendente Predictivo LL(1).

**3.2 Primeros y Siguientes**

****

*Se adjunta el Excel conteniendo los Primeros y Siguientes*

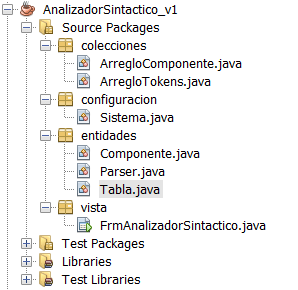
**3.3 Generación del TAS**

****

*Se adjunta el Excel conteniendo el TAS desarrollado*

**4. DISEÑO Y ARQUITECTURA**

**4.1 Especificación de la Arquitectura**

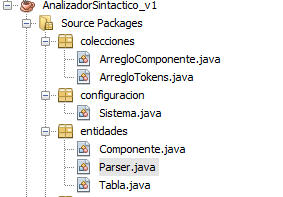


Como ya vimos en la primera parte del proyecto para el Analizador Léxico trabajamos con las clases ***ArregloTokens***, en la cual almacenábamos las palabras reservadas, símbolos y operadores de nuestro lenguaje,***ArregoComponente****,* en la cual se transformaba la lista de tokens a una matriz de tokens y la clase ***Componente****,* en la cual se describía los atributos que debían contener los tokens.

Ahora en parte sintáctica trabajaremos con las clases ***Sistema***, la cual tendrá el número de filas, columnas de la tabla TAS, así como también el número máximo de caracteres que contendrá la pila. La clase ***Parser,*** la cual se encagará de realizar todo el análisis sintáctico del código ingresado en el programa. Por último, la clase ***Tabla***, en la cual encontremos el TAS y será en donde ingresaremos las reglas de producciones de la gramática trabajada por nuestro lenguaje JAB, con las cuales se realizará un correcto análisis sintáctico del código fuente.

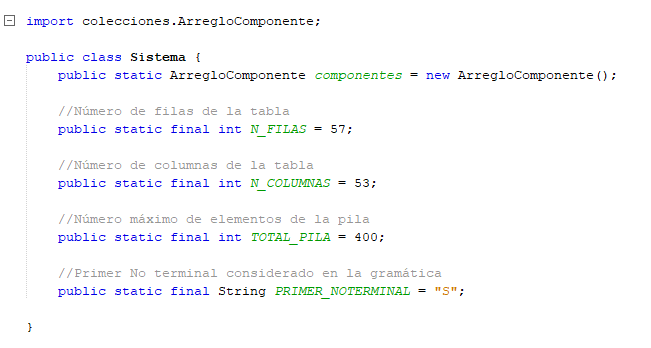
**4.2 Interfaz Principal**

**4. 3 Especificación de clases principales**

****

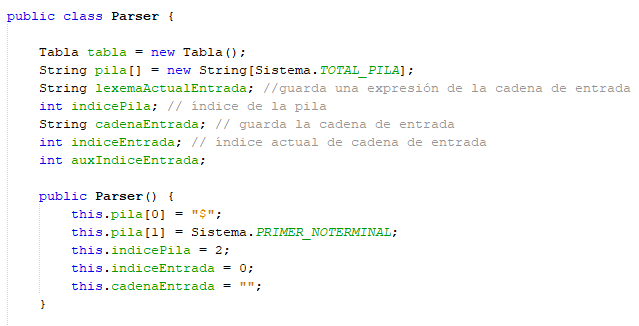
Clases

* **Configuración**
  + Sistema.java: determinamos los parámetros necesarios para el sintáctico como el número de filas, columnas, los elementos por pila, y el primer no terminal de la gramática.

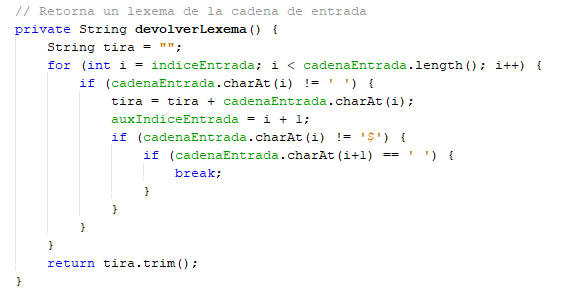


De acuerdo a la matriz, se genera una cantidad de filas y columnas de 57 y 53 campos respectivamente. Adicional a eso tomamos un valor necesario de 400 para los elementos admisibles en la pila, y declaramos el primer no terminal de la gramática “S”.

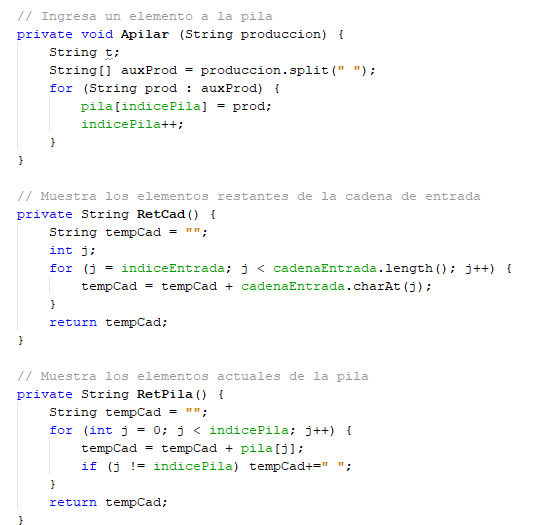
* **Entidades** 
  + Parser.java: es la clase donde se genera el código encargado de hacer el análisis sintáctico (funciones).



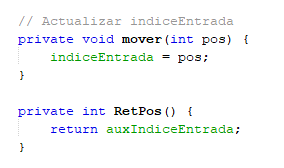
Generamos los atributos necesarios de la Pila para poder iniciar el programa



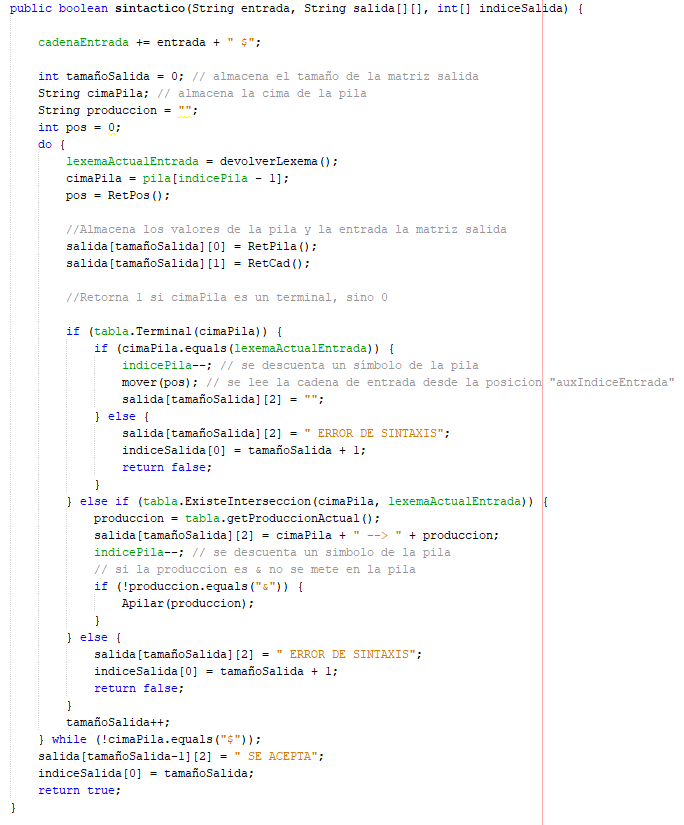
En esta función generamos un for para poder analizar la cadena de entrada y retornar un lexema de dicha cadena.



Los siguientes métodos nos ayudaran a mostrar los elementos necesarios de la cadena de entrada.



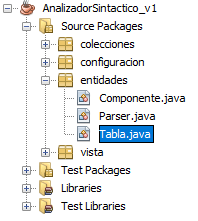
Las 2 funciones siguientes ayudaran a tener actualizada el índice de entrada con ayuda de un auxiliar.



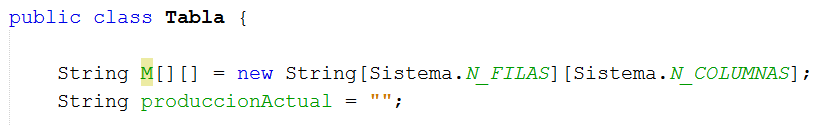
Este es el código principal del sintáctico donde inicialmente almacenaremos la información necesaria de la pila. Seguidamente almacenaremos los valores de la pila retornando 1 si la entrada es un terminal, caso contrario se devolverá 0. Luego se leerá la entrada de la pila símbolo por símbolo desde la posición del auxiliar de entrada. Finalmente se analizará las intersecciones y los vacíos a través de la tabla **ExisteIntersección.**

**4.4 Tabla reconoce todo**

Dentro del paquete “entidades”, la clase “Tabla.java”:



Dentro de esa clase, definimos los atributos de un objeto que representará nuestra tabla:

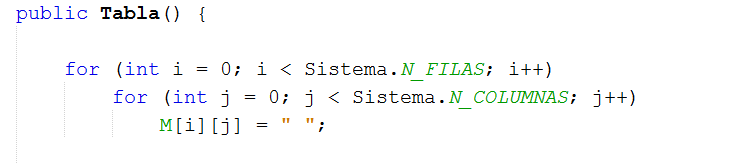


\*“M” es una matriz en la que colocaremos nuestro TAS, las dimensiones se reciben de la clase sistema en la que se establecen las dimensiones (cantidad de variables terminales y no terminales)

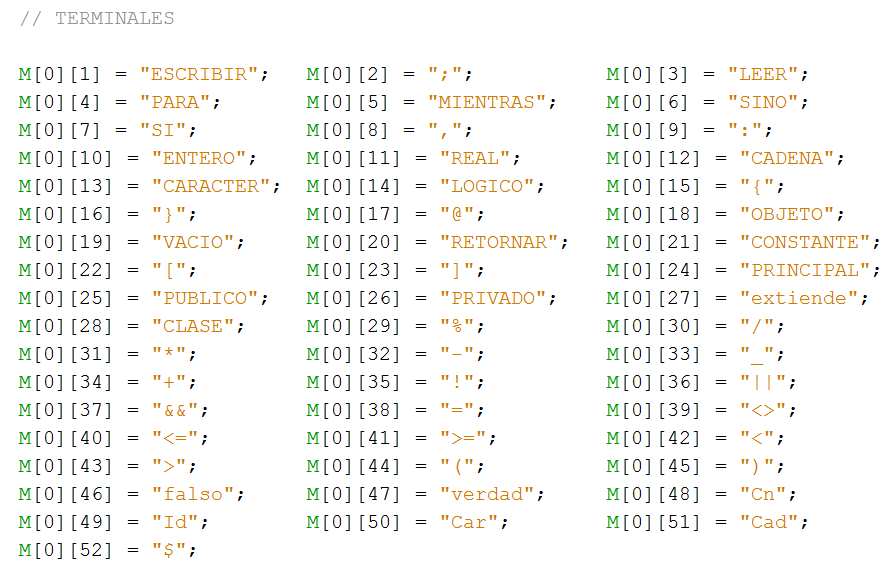
\*“produccionActual” se usará dentro de uno de los métodos que se mostrarán luego.

Para el constructor de nuestra matriz:

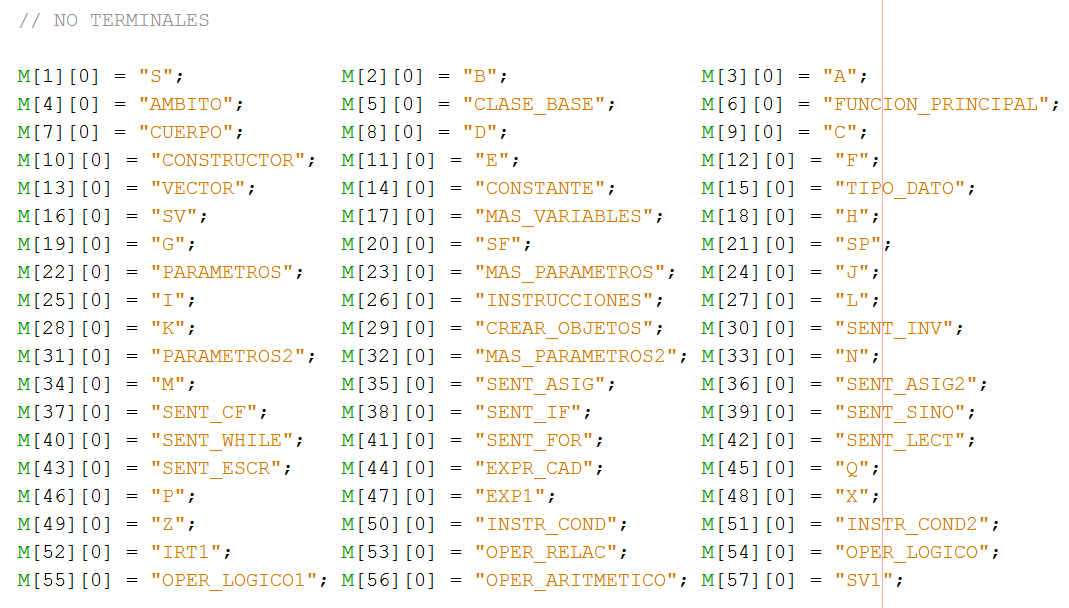
Primero, llenamos de espacios en blanco todos los espacios.



Luego, definimos nuestras variables TERMINALES en la matriz como las columnas:



Luego, definimos nuestras variables NO TERMINALES en la matriz como las filas:

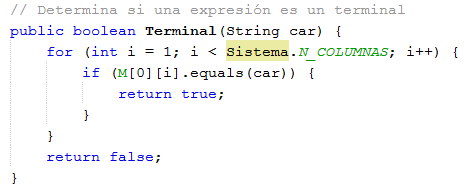


A continuación, se llena el resto de la matriz con las intersecciones, que se obtienen en base a los primeros y siguientes de nuestra gramática, las reglas de producción deberán ser ingresadas de forma inversa, ya que esas reglas serán usadas en una pila.

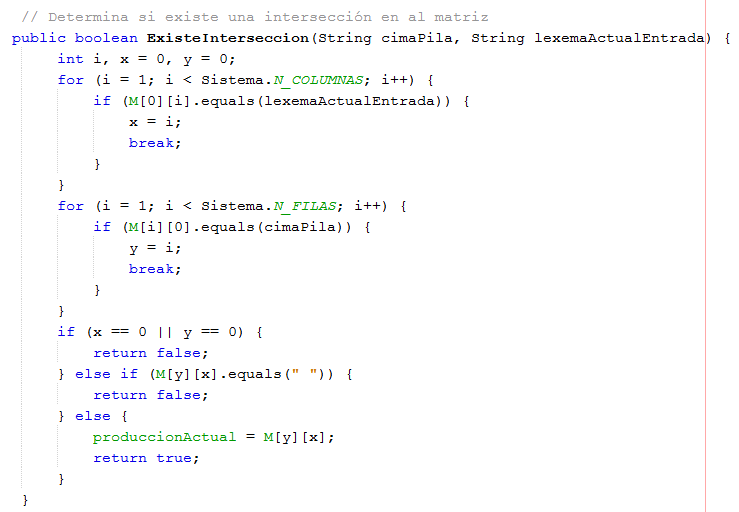
Ejemplo: Si se tiene como regla: **S -> i B** (considerando que “i” es una variable terminal y “B” es una variable no terminal), dentro de la matriz se deberá colocar en la intersección entre “S” e “i” a “**B i**” que es la regla escrita de forma inversa.



Como métodos de la clase, se tienen los siguientes:



Verifica si una expresión es un símbolo terminal, comparándolo con todos los terminales escritos en el primer paso



Verifica si existe una intersección en la matriz, comprobando que se cumpla la regla de producción correspondiente