



# ***PROCESADORES DEL LENGUAJE***

*Compilador en Java con JFlex y CUP*

**Práctica de laboratorio, 11 de diciembre de 2016**

Creación, a partir de los módulos integrados en el software Eclipse JFlex y CUP, de un compilador que admita archivos con la definición de sentencias de alto nivel de un programa.

Miguel Ángel García Romeral  
miguel.romeral@gmail.com – 09064317V

## ÍNDICE

1. DESCRIPCIÓN DE DISEÑO .....	2
2. DISEÑO DEL ANALIZADOR LÉXICO.....	4
EXPRESIONES REGULARES DEFINIDAS POR EL PROGRAMADOR.....	4
OTRAS EXPRESIONES REGULARES.....	7
3. DISEÑO DEL ANALIZADOR SINTÁCTICO .....	8
GRAMÁTICA: EXPLICACIÓN DE CADA PRODUCCIÓN .....	8
GRAMMOPHONE: OBTENCIÓN DEL AUTÓMATA Y LA TABLA DE ANÁLISIS ....	12
ATRIBUTOS Y MÉTODOS DEL ANALIZADOR SINTÁCTICO.....	12
4. DISEÑO DEL ANALIZADOR SEMÁNTICO.....	14
5. DISEÑO DE LA TABLA DE SÍMBOLOS .....	17
6. DISEÑO DEL AST .....	19
7. CASOS DE PRUEBA.....	21

## 1. DESCRIPCIÓN DE DISEÑO

El presente documento detalla la metodología seguida y las soluciones correspondientes a la realización de un compilador para un lenguaje definido por el programador e implementado en el lenguaje de alto nivel Java en el entorno de desarrollo integrado Eclipse, habiendo integrado anteriormente dos módulos, uno que se encargue de definir las sentencias de las diferentes expresiones regulares definidas por el programador y otro que analice la salida del analizador léxico y las trata según la sintaxis de su gramática. Estos módulos son JFlex y CUP respectivamente.

Además de la creación de los dos archivos correspondientes a los dos módulos, también se han definido más clases para el correcto funcionamiento de nuestro compilador, como puede ser la tabla de símbolos. Ésta está compuesta por una estructura dinámica que almacena los símbolos en función de su nombre. Por su parte, los símbolos también son creados con una clase definida por el programador en Java, donde los atributos referidos al nombre del símbolo, al valor o si ha sido inicializado componen dicha clase. También existe una clase manejadora que se encarga de iniciar el compilador recogiendo los archivos que se le pasen como parámetro y mostrando por pantalla el resultado de los diferentes análisis a los que es sometido. Sin embargo, todos los detalles referidos a la estructura de las diferentes clases serán detallados más ampliamente en las siguientes secciones del presente documento.

La memoria consta de siete partes si tenemos en cuenta esta introducción, que es la primera de todas. La segunda parte tiene el fin de explicar cómo se ha diseñado el analizador léxico, cuáles son los tokens que envía al analizador sintáctico, los autómatas de las expresiones regulares definidas, las acciones que realiza al encontrarse con un determinado token o al encontrarse un error. La tercera parte está destinada a la definición de la gramática implementada y detalles de ésta, tales como la posible ambigüedad existente y su solución o la recursividad en las producciones. La cuarta parte detalla la implementación del analizador sintáctico y las medidas que se llevan a cabo para cumplir con la semántica durante el análisis. En la quinta parte se matiza la clase creada para la tabla de símbolos que ya habíamos introducido anteriormente. En la sexta parte se incluye una descripción sobre cómo se ha llevado a cabo la generación de un árbol de sintaxis abstracto y los resultados que imprime por pantalla. En la séptima y última parte se muestra el funcionamiento del compilador con los resultados del análisis de diez casos de prueba diferentes.

El sistema ha sido dotado de una función adicional que permite analizar una lista de programas que se le pasen por argumento a la ejecución del compilador, evitando tener que ejecutarlo cada vez que se desee analizar un programa de una determinada lista.

Es importante mencionar los diferentes métodos que tiene la clase manejadora del compilador para su correcto funcionamiento.

- **public static void** main(String argv[]): método que inicia la ejecución del programa. La lista de programas que se desean analizar se encuentra en argv.
- **public static** void mostrarFallo(String path, int linea, int columna) : imprime por pantalla la línea del archivo donde se encuentra según la ruta path especificada y apunta según la columna al carácter que ha propiciado el error.
- **public static** ArrayList<String> sacaIDEs(String nodo) : obtiene los identificadores que se encuentran en el nodo según el valor que se le pasa

por argumento. Para obtener los identificadores compara que se haya introducido la cadena "ide(" para el árbol de sintaxis abstracta.

- **public static** ArrayList<String> tipoIDES(TablaSimbolos *ts*, ArrayList<String> *ides*) : indica el tipo de identificador de la lista de "ides" que se le pasa por argumentos, pero para ello debe poder ver la tabla de símbolos en la que pertenecen los identificadores que se quieren analizar.
- **public static boolean** finWhile(ArrayList<String> *vals*, String *nodo*) : comprueba que las variables que crean la condición del while (las cuales, se encuentran en la lista que se le pasa al método) hayan sido en algún momento usadas por una asignación en el bloque de sentencias del bucle, si no se ha asignado ninguna de las variables en ningún momento, el analizador semántico determinará que el bucle no tiene fin.
- **public static boolean** isNumeric(String *str*) : comprueba que la cadena que se le pasa por argumento se trate de un número.

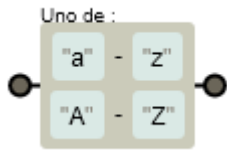
## 2. DISEÑO DEL ANALIZADOR LÉXICO

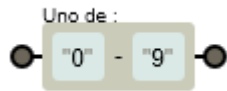
Comenzamos la sección del diseño del analizador léxico indicando los diferentes atributos que contiene la clase que JFlex genera en el fichero “.java” gracias al archivo “Scan.lex”.

En una lista dinámica de Java (ArrayList) almacenaremos los diferentes tokens en el orden en que son generados por el analizador en base a las expresiones regulares. Los tokens son cadenas de caracteres (String) que se mostrarán al final del análisis. También se guarda la ruta del archivo que se le pasa como parámetro. Por último, se le indicará el analizador sintáctico cuando éste sea instanciado. El motivo reside en que la lista de errores las mostrará la clase “Parser.java”, generada por el archivo “Parser.cup”, por lo que, si queremos añadir los errores léxicos a la lista de todos los errores encontrados, debemos pasarle el parser (o analizador sintáctico) para que allí se guarden. La primera opción implementada consistió en que al encontrar cualquier error léxico el compilador se parase y tan solo mostrase al usuario dicho error para que éste pudiera resolverlo y así continuar con la siguiente fase del compilador.


### EXPRESIONES REGULARES DEFINIDAS POR EL PROGRAMADOR

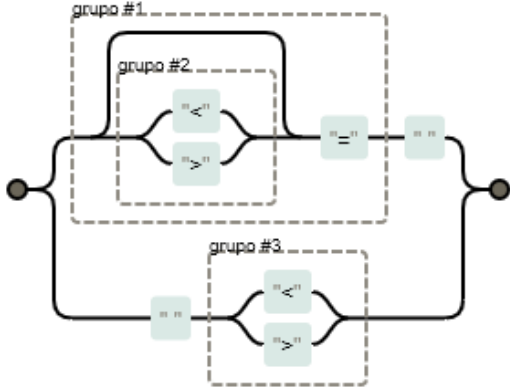
A continuación, listamos las diferentes expresiones regulares, sus autómatas finitos deterministas (generados con la herramienta REGEXPER) y las acciones del analizador cuando reconoce la expresión:

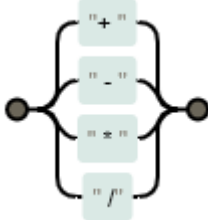
<b>Expresión Regular</b>	<b>Acción</b>
Letra = [a-zA-Z]	El analizador no realiza ninguna acción con esta expresión regular, pero sirve para crear los diferentes tokens que pasará al analizador sintáctico.
<b>Autómata Finito Determinista</b>	
	

<b>Expresión Regular</b>	<b>Acción</b>
Numero = [0-9]	El analizador no realiza ninguna acción con esta expresión regular, pero sirve para crear los diferentes tokens que pasará al analizador sintáctico.
<b>Autómata Finito Determinista</b>	
	

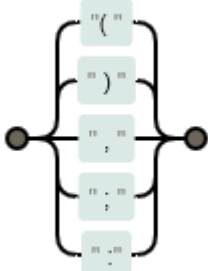
<b>Expresión Regular</b>	<b>Acción</b>
Asignacion = :=	Añade un nuevo token de valor “asign” y retornará el símbolo correspondiente a la
<b>Autómata Finito Determinista</b>	

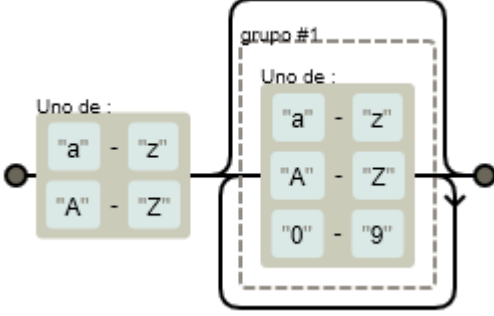
	asignación en la línea y carácter en que se ha leído con el valor “:=”
---	--

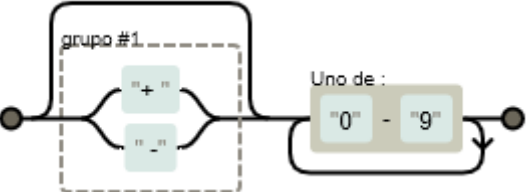
<i>Expresión Regular</i>	<i>Acción</i>
Comparacion = ((< >)?=)   (< >)	Añade un nuevo token de valor “comp” y retornará el símbolo correspondiente a la comparación en la línea y carácter en que se ha leído. A diferencia de la anterior expresión regular, el valor del símbolo que retorna dependerá, en función del método <code>operadorComparacion (String texto)</code> que devuelve una cadena de caracteres con la comparación en tipo texto. Por ejemplo, si leyera “<=” el valor del símbolo será “menorigual”
<i>Autómata Finito Determinista</i>	
	

<i>Expresión Regular</i>	<i>Acción</i>
OperadorMatematico = \+   -   \*   /	Retorna un símbolo cuyo valor depende del tipo de operador en la línea y carácter en que se han analizado. El valor, al igual que en la anterior expresión regular, depende del método <code>operadorMatematico (String texto)</code> que retorna el número del símbolo. Además, este método añade a la lista de tokens el valor según el operador. Por ejemplo, si se escribiera “+” el valor es “mas”.
<i>Autómata Finito Determinista</i>	
	

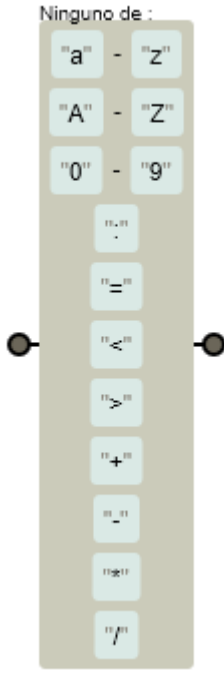
<i>Expresión Regular</i>	<i>Acción</i>
Puntuacion = \(   \)   ,   ;   :	Añade un nuevo token cuyo valor dependerá del tipo de puntuación y retornará el símbolo correspondiente al
<i>Autómata Finito Determinista</i>	

	<p>operador de puntuación en la línea y carácter en que se ha leído.</p>
---	--

<b>Expresión Regular</b>	<b>Acción</b>
<p>IDE = {Letra} ({Letra} {Numero})*</p>	<p>Debido a que los tokens correspondientes a las diferentes palabras reservadas coincide en estructura con la de un identificador que se le pase al usuario, se ha optado por tratar en un principio a las palabras reservadas e identificadores de la misma manera, sin embargo, el número de símbolo que retorna será el del token de la palabra reservada o, si se comprueba que el identificador no corresponde con ninguna palabra reservada, el de un identificador nuevo con el valor leído. Esto se realiza mediante una función que implementa un switch que asigna el número del símbolo según el tipo de identificador leído.</p>
<p><b>Autómata Finito Determinista</b></p> 	

<b>Expresión Regular</b>	<b>Acción</b>
<p>INTEGER = (\+   \-)?[0-9]+</p>	<p>Añade un token a la lista de tokens con el valor del número entero leído y retorna un símbolo de tipo entero.</p>
<p><b>Autómata Finito Determinista</b></p> 	

<b>Expresión Regular</b>	<b>Acción</b>
<p>NoReconocido = [^AsignacionComparacionOperadorMatem aticoPuntuacionINTEGERIDE]</p>	<p>Cuando el analizador léxico se encuentre con un carácter que no corresponda a ninguna de las reglas anteriores entonces llamará a una función que avisará al usuario de que el</p>
<p><b>Autómata Finito Determinista</b></p>	

	<p>carácter leído no es válido y lo añadirá a la lista de errores del analizador sintáctico para que éste los muestre al finalizar el análisis.</p>
---	---

### ***OTRAS EXPRESIONES REGULARES***

Las siguientes expresiones regulares han sido extraídas del manual de JFlex. Aunque no venga en el enunciado de la práctica, se ha visto oportuno su inclusión durante el proceso de desarrollo.

```
Comentario = {ComentarioTradicional} | {ComentarioFinLinea} |
{ComentarioDocumentacion}
```

```
ComentarioTradicional = "/"* [^*] ~"*/" | "/"* "*" + "/"
```

```
ComentarioFinLinea = "//" [^\r\n]* (\r|\n|\r\n)?
```

```
ComentarioDocumentacion = "/*" {ContenidoComentario} "*" + "/"
```

```
ContenidoComentario = ( [^*] | \*+ [^/*] )*
```



### 3. DISEÑO DEL ANALIZADOR SINTÁCTICO

La gramática definida en el analizador sintáctico es adjunta con el presente documento en el formato adecuado para ser introducida por la herramienta Grammophone. Sin embargo, aquí se muestra con anotaciones para comprender el porqué de esta definición.

Los no terminales están escritos en mayúsculas, los terminales en minúsculas. El axioma es el no terminal INICIO. Se ha considerado incluir la producción que CUP proporciona de “error” cuando sucede un error de sintaxis.

#### **GRAMÁTICA: EXPLICACIÓN DE CADA PRODUCCIÓN**

- INICIO -> PROGRAM.

El programa está compuesto por los diferentes tokens del programa (la definición de éste) y después el bloque de definición de variables y la definición del programa.

- PROGRAM -> tk\_program tk\_identificador tk\_is  
BLOQUEDEFINICIONVARIABLE DEFINICIONPROGRAMA.

El bloque de definición de variables lo componen diferentes definiciones de variables o ningunas, por lo que se ha creado un no terminal auxiliar (DVA = Definición Variable Auxiliar). Además, la definición de este no terminal servirá, como los demás no terminales incluidos a lo largo de la gramática, para evitar la recursividad por la izquierda y generarla por la derecha.

- BLOQUEDEFINICIONVARIABLES -> DVA.

Así, el bloque de definición de variables puede ser ninguna definición, una sola definición, o las que sean necesarias, puesto que la gramática lo permite.

- DVA -> DEFINICIONVARIABLE DVA.
- DVA -> .

La definición de la variable es creada con el token “var”, la lista de variables que se quieren definir separadas por comas, dos puntos, el tipo de dato que son las variables y finalmente un punto y coma.

- DEFINICIONVARIABLE -> tk\_var LISTAVARIABLES tk\_dospuntos  
DECLARACIONTIPO tk\_puntoycoma.

Además, se incluyen las producciones de error que proporciona CUP para detectar los errores en la sintaxis y poder recuperarse de éstos.

- DEFINICIONVARIABLE -> error.

Una vez que se definirán variables, al menos un identificador será definido, por lo que, en este caso, al contrario que con DVA, la producción será recursiva por la derecha, pero habiendo añadido primero un identificador al menos.

- LISTAVARIABLES -> tk\_identificador LVA.
- LISTAVARIABLES -> error.

Ahora, la lista de variables puede ser varias variables separadas por comas, ser vacía, lo que quiere decir que no se definen más que una sola variable, o una lista errónea en cuanto a la sintaxis.

- LVA -> tk\_coma tk\_identificador LVA.
  - LVA -> .
  - LVA -> error.

El programa está definido por sentencias entre los tokens “begin” y “end”.

- DEFINICIONPROGRAMA -> tk\_begin BLOQUEDESENTENCIAS tk\_end.

La declaración del tipo solo podrá ser de tipo entero “integer” o de tipo “boolean” o haberse producido un error.

- DECLARACIONTIPO -> tk\_integer.
- DECLARACIONTIPO -> tk\_boolean.
- DECLARACIONTIPO -> error.

El bloque de sentencias, al igual que con DVA, puede estar vacío o componerse de una o más sentencias.

- BLOQUEDESENTENCIAS -> BDSA.
- BDSA -> DEFINICIONSENTENCIA BDSA.
  - BDSA -> .

Existen diferentes tipos de sentencias en nuestro programa. La primera es un skip, que en el código objeto que el compilador generaría si se le dotara de Back End su acción sería salir de un bucle.

- DEFINICIONSENTENCIA -> tk\_skip tk\_puntoycoma.

La segunda sentencia se trata de la lectura de un identificador.

- DEFINICIONSENTENCIA -> tk\_read tk\_identificador  
tk\_puntoycoma.

La tercera sentencia consta de la escritura en un identificador.

- DEFINICIONSENTENCIA -> tk\_write tk\_identificador  
tk\_puntoycoma.

La sentencia también podría tratarse de una sentencia condicional, la cual trataremos más adelante.

- DEFINICIONSENTENCIA -> DEFINICIONIF.

Otra sentencia sería un bucle while, en el que está compuesto por expresiones booleanas que determinan si hay que realizar las sentencias del bucle y además qué sentencias en concreto. El bucle acaba con la palabra “end” seguido de “while” y punto y coma.

- DEFINICIONSENTENCIA -> tk\_while LISTAEXPRESIONES tk\_do  
BLOQUEDESENTENCIAS tk\_end tk\_while tk\_puntoycoma.

El último tipo de sentencia es una asignación a un identificador. La asignación se escribe por criterios de diseño como “:=”, seguido de una lista de asignaciones que se explicarán más adelante.

- `DEFINICIONSENTENCIA -> tk_identificador tk_asignacion  
AVERQUEASIGNAMOS tk_puntoycoma.`

La definición del if, al igual que el while, consta de expresiones booleanas, el bloque de sentencias condicional y un hipotético bloque de sentencias para la condición “sino”, que se trata a continuación.

- `DEFINICIONIF -> tk_if LISTAEXPRESIONES tk_then  
BLOQUEDESENTENCIAS DEFINICIONELSE tk_end tk_if  
tk_puntoycoma.`

La definición del else puede no aparecer en todos los condicionales, por lo que, si se quiere añadir, deberá incluirse el token “else” seguido de las sentencias que se quieren añadir. Si, por el contrario, no se quiere añadir un else, esta producción será vacía.

- `DEFINICIONELSE -> tk_else BLOQUEDESENTENCIAS.`
- `DEFINICIONELSE -> .`

La lista de expresiones booleanas la componen, al menos, una expresión booleana. De nuevo, EAA representa un no terminal auxiliar para conseguir la recursividad por la derecha.

- `LISTAEXPRESIONES -> EXPRESIONBOOLEANA EAA.`

Si existen más de una expresión, será necesario añadir a la lista la condición para que se cumpla, que podrá ser AND u OR.

- `EAA -> SIGUEEXPRESION EXPRESIONBOOLEANA EAA.`
- `EAA -> .`

Para seguir la expresión, como se indicó anteriormente, se añaden los tokens AND u OR.

- `SIGUEEXPRESION -> tk_and.`
- `SIGUEEXPRESION -> tk_or.`

Existen diferentes tipos de expresiones booleanas. La primera consiste en una lista de expresiones booleanas delimitadas por paréntesis.

- `EXPRESIONBOOLEANA -> tk_parenizq LISTAEXPRESIONES  
tk_parender.`

También puede tratarse de una comparación, la cual describimos a continuación de

- `EXPRESIONBOOLEANA -> COMPARACION.`

La expresión booleana puede estar negada, por lo que está compuesta por el token “not” y entre paréntesis la nueva lista de expresiones booleanas.

- `EXPRESIONBOOLEANA -> tk_not tk_parenizq LISTAEXPRESIONES  
tk_parender.`

También puede ocurrir un error en la expresión.

- EXPRESIONBOOLEANA -> error.

La comparación puede ser un identificador, seguido de un operador de comparación y un entero o viceversa.

- COMPARACION -> tk\_identificador tk\_comparacion tk\_numeroentero.
- COMPARACION -> tk\_numeroentero tk\_comparacion tk\_identificador.

Además, la comparación puede realizarse sobre un identificador. Sin embargo, y esta es tarea del analizador semántico, el identificador que se le pasa a la comparación debe ser de tipo booleano. Más adelante en el documento se explicará cómo se ha resuelto el problema.

- COMPARACION -> tk\_identificador.

Por último, la comparación puede ser indefinida, con un true.

- COMPARACION -> true.

Lo que podremos asignar es “true” o “false” si se trata de un tipo booleano (de nuevo será tarea del semántico).

- AVERQUEASIGNAMOS -> tk\_true.
- AVERQUEASIGNAMOS -> tk\_false.

La asignación puede ser un identificador o un número entero y después una lista de asignaciones que explicamos a continuación.

- AVERQUEASIGNAMOS -> IDEOENTERO LISTADEASIGNACIONES.

La asignación podrá ser una expresión booleana negada o simplemente una lista de expresiones.

- AVERQUEASIGNAMOS -> tk\_not tk\_parenizq LISTAEXPRESIONES tk\_parender.
- AVERQUEASIGNAMOS -> tk\_parenizq LISTAEXPRESIONES tk\_parender.

La lista de asignaciones la compone en primer lugar un operador matemático, puesto que antes existe un entero (ya sea número o identificador de tipo entero) seguido de nuevo por un identificador o un entero además de más asignaciones que puedan definirse.

- LISTADEASIGNACIONES -> OPERADORMATES IDEOENTERO LISTADEASIGNACIONES.

Por ello, la lista de asignaciones también puede ser vacía, para el caso en que la asignación tan solo tenga un entero o un identificador o para que la lista de asignaciones termine en algún momento. De nuevo, la gramática es recursiva por la derecha.

- LISTADEASIGNACIONES -> .

Además, pueden existir errores en la lista de asignaciones.

- LISTADEASIGNACIONES -> error.

El operador matemático se compone de los tokens correspondientes a la suma, la resta, la multiplicación y la división.

- OPERADORMATES -> tk\_suma.
- OPERADORMATES -> tk\_resta.
- OPERADORMATES -> tk\_multiplicacion.
- OPERADORMATES -> tk\_division.

Por último, finaliza la gramática con la posibilidad de que en la lista de asignaciones se le asigne un identificador de tipo entero o un número entero.

- IDEOENTERO -> tk\_identificador.
- IDEOENTERO -> tk\_numeroentero.

### **GRAMMOPHONE: OBTENCIÓN DEL AUTÓMATA Y LA TABLA DE ANÁLISIS**

Tras definir la gramática, la analizaremos con la herramienta Grammophone, obteniendo de manera automática el análisis para la gramática. En análisis ha resultado satisfactorio, puesto que está libre de ambigüedades y la gramática pertenece a LALR(1).

Debido al enorme tamaño del autómata y de la tabla de análisis sintáctico, se proporciona un fichero de texto en el que está definida la gramática y que se puede copiar y pegar en la herramienta para observar los resultados.

Además de las diferentes producciones gramaticales, el análisis sintáctico está dotado de diferentes métodos que se explican en esta sección.

### **ATRIBUTOS Y MÉTODOS DEL ANALIZADOR SINTÁCTICO**

En la parte definida por el programador en `action code` se encuentran los siguientes métodos y atributos:

- `String tipo`: se guarda el valor del tipo que se ha asignado a una determinada lista de variables.
- `ArrayList<Simbolo> sinTipo`: Lista dinámica de símbolos pertenecientes a las variables leídas que aún no tienen tipo
- `private void nuevoIDE(String ide, int fila, int columna)`: añade la variable leída en la línea `fila` y carácter `columna`. Comprueba, además, que no está ya definida y que no se trata de una palabra reservada, en cuyo caso no añade la variable y reporta un mensaje de error.
- `private void ponerTipo(String tipo)`: pone el tipo de dato a las variables que se acaban de declarar y aún no han sido añadidas a la tabla de símbolos. Tras ponerle el tipo se añadirán a la tabla de símbolo. Las variables de una lista de definición de variables no son añadidas a la tabla de símbolos hasta que no se lee el tipo y he aquí donde reside el motivo de la implementación de dicho método.
- `private void newProduccion(String p)`: guarda en una lista dinámica la producción gramática que acaba de visitar el analizador sintáctico.

En el `parser code` están los siguientes métodos y atributos:

- `public TablaSimbolos ts`: estructura definida para la tabla de símbolos del Front End del compilador.

- `private String path`: variable que almacena la dirección donde se encuentra el archivo que se está analizando.
- `public ArrayList<String> errores`: lista dinámica que almacena los errores que se encuentran en todo el compilador.
- `public ArrayList<String> producciones`: lista de producciones por las que la gramática está pasando.
- `public void report_error(String message, Object info)`: Añade el error que se le pasa por argumento en la lista de errores. añadiéndole la información del segundo objeto si no es nulo.
- `public void newError(String m)`: añade el error junto al número de éste en la lista.
- `public void mostrarErrores()`: muestra la lista de los diferentes errores encontrados durante el análisis del programa.
- `public void limpiar()`: vacía los atributos del analizador sintáctico para que pueda ser analizado otro programa inmediatamente después.
- `public void ponerPath(String path)`: añade la dirección del archivo que se está analizando.
- `public void syntax_error(Symbol s)`: redefinición del método que proporciona por defecto CUP. El mensaje lo mostramos en español e indicamos que es un error de tipo sintáctico.
- `public void report_fatal_error(String message, Object info)`: reporta un error irrecuperable y muestra la lista de errores del programa.
- `public void imprimirProducciones()`: imprime por pantalla la lista de todas las producciones que la gramática ha visitado en su orden correspondiente.
- `public void finalAnálisis()`: muestra los resultados del análisis sintáctico, la lista de producciones, la tabla de símbolos y los errores si existiese alguno.
- `public String fallo(int linea, int columna)`: llama a la clase manejadora para que muestre por pantalla el lugar exacto en el archivo donde se encuentra el error de la línea y columna pasadas como argumentos.

Respecto a los terminales debe decirse que todos ellos, a excepción de los tokens pertenecientes a los números enteros (tratados como `integer`) y los identificadores (tratados como símbolo, estructura de datos creada por el programador cuya explicación vendrá más adelante), todos los demás terminales son de tipo `String`, que es el valor por defecto de CUP para los terminales que no tengan un tipo definido.

A diferencia de los terminales, los no terminales son todos ellos, a excepción del axioma, de la estructura nodo, la cual hablaremos en detalle en la sección de la memoria referente al árbol de sintaxis.

Existe también una breve lista de precedencias, todas ellas por la derecha, en la que, por orden de menor a mayor importancia, se encuentra el token referente al operador binario “or”, el del operador “and”, el “not” o negación, las comparaciones, la suma respecto de la resta y, por último, la multiplicación respecto a la división.

## 4. DISEÑO DEL ANALIZADOR SEMÁNTICO

Para obtener un buen análisis semántico se han incluido diferentes operaciones a lo largo de la gramática sintáctica definida para conseguir detectar algunas sentencias que han superado los análisis léxico y sintáctico pero que sin embargo no tienen sentido alguno en la implementación del programa.

En esta sección indicaremos la producción y las operaciones que se realizan en ésta. Aunque se hayan dado por obvio, es importante mencionar que las comprobaciones semánticas son escritas en el archivo CUP que generará el código en java. Esto se consigue asignando tipos de datos a los terminales y no terminales como se indicó al final del punto tres del presente documento, y dando nombre a los terminales y no terminales que se encuentran en una producción. También se pueden asignar, además de a la parte izquierda de la producción, un valor a la parte derecha. Esto se consigue con la variable que proporciona CUP llamada RESULT. Como hemos definido que todos los no terminales (o casi todos, a excepción del axioma) son nodos, entonces todos los resultados que asignemos deberán ser del mismo tipo.

Comenzaremos viendo las comprobaciones semánticas más altas de la gramática.

<b><i>Producción gramatical</i></b>
PROGRAM ::= tk_program tk_identificador tk_is BLOQUEDEFINICIONVARIABLE DEFINICIONPROGRAMA
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>La variable que indica el nombre del programa no haya sido definida en el bloque de definición de variables.</li> <li>Las variables definidas en el bloque de definición de variables hayan sido utilizadas al menos una vez en la definición del programa.</li> </ul>

<b><i>Producción gramatical</i></b>
DEFINICIONVARIABLE ::= tk_var LISTAVARIABLES tk_dospuntos DECLARACIONTIPO tk_puntoycoma
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>Añadir el tipo de variable a las diferentes variables definidas en la lista mediante el método <code>ponerTipo(String tipo)</code>.</li> </ul>

<b><i>Producción gramatical</i></b>
LISTAVARIABLES ::= tk_identificador LVA
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>Añadir la variable del token a la lista de variables que aún no tienen tipo mediante <code>nuevoIDE(String nombre, int linea, int columna)</code>.</li> </ul>

<b><i>Producción gramatical</i></b>
LVA ::= tk_coma tk_identificador LVA
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>Añadir la variable del token a la lista de variables que aún no tienen tipo mediante <code>nuevoIDE(String nombre, int linea, int columna)</code>.</li> </ul>

<b><i>Producción gramatical</i></b>
-------------------------------------

DECLARACIONTIPO ::= tk_integer
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Indica que el tipo de variable es entero.</li> </ul>

<b>Producción gramatical</b>
DECLARACIONTIPO ::= tk_boolean
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Indica que el tipo de variable es booleano.</li> </ul>

<b>Producción gramatical</b>
DEFINICIONSENTENCIA ::= tk_read tk_identificador tk_puntoycoma
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Comprobar que la variable que se quiere leer ha sido antes declarada.</li> <li>Comprobar que la variable que se quiere leer es de tipo entero.</li> </ul>

<b>Producción gramatical</b>
DEFINICIONSENTENCIA ::= tk_write tk_identificador tk_puntoycoma
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Comprobar que la variable que se quiere escribir ha sido antes declarada.</li> <li>Comprobar que la variable que se quiere escribir haya sido anteriormente inicializada (esto se obtiene al asignar algo a dicha variable).</li> </ul>

<b>Producción gramatical</b>
DEFINICIONSENTENCIA ::= tk_while LISTAEXPRESIONES tk_do BLOQUEDESENTENCIAS tk_end tk_while tk_puntoycoma
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Verificar que el while tiene fin, esto se consigue mediante un método del manejador.</li> </ul>

<b>Producción gramatical</b>
DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion AVERQUEASIGNAMOS tk_puntoycoma
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Comprobar que la variable a la que se quiere asignar un determinado valor haya sido declarada.</li> <li>Comprobar la concordancia existente entre los tipos de la parte izquierda de la asignación y la parte derecha.</li> </ul>

<b>Producción gramatical</b>
COMPARACION ::= tk_numeroentero tk_comparacion tk_identificador
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Comprobar que la variable que se quiere comparar haya sido declarada.</li> <li>Determinar si la comparación se realiza sobre un identificador de tipo entero.</li> </ul>

<b>Producción gramatical</b>
COMPARACION ::= tk_identificador tk_comparacion tk_numeroentero
<b>Comprobaciones semánticas</b>
<ul style="list-style-type: none"> <li>Comprobar que la variable que se quiere comparar haya sido declarada.</li> </ul>



- Determinar si la comparación se realiza sobre un identificador de tipo entero.

<b><i>Producción gramatical</i></b>
COMPARACION ::= tl_identificador
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>• Comprobar que la variable que se quiere comparar haya sido declarada.</li> <li>• Determinar si la variable que se quiere comparar sea de tipo booleano.</li> </ul>

<b><i>Producción gramatical</i></b>
LISTADEASIGNACIONES ::= OPERADORMATES IDEOENTERO LISTADEASIGNACIONES
<b><i>Comprobaciones semánticas</i></b>
<ul style="list-style-type: none"> <li>• Comprobar que, si se le pasa un identificador, haya sido declarado.</li> <li>• Determinar si en la operación se está realizando una división entre cero.</li> </ul>

## 5. DISEÑO DE LA TABLA DE SÍMBOLOS

Para realizar la tabla de símbolos hubo que pensar en la manera de almacenar las diferentes entradas de ésta. Una cadena de caracteres sería una opción que se quedaría muy corta, puesto que muchas de las operaciones que se desean realizar con una cadena no podrían verse solucionadas. Finalmente, se llegó a la conclusión de la necesidad de crear un nuevo tipo de dato que no proporcionase la API de Java, fue entonces cuando se creó la clase “Simbolo.java”.

Los atributos de esta clase son el nombre del símbolo, el tipo, su valor, la fila y columna en la que fueron declarados (o línea y carácter), un indicador booleano que indica si ha sido inicializada y una lista dinámica que indica dónde ha sido usada a lo largo del programa.

Es importante matizar que el valor será indefinido para los tipos “pseudo”, cero para los enteros y falso para los booleanos, ya que, debido a diferentes complicaciones, los intentos de obtener el valor de una asignación o comparación han resultado fallidos.

Esta clase cuenta con un método para mostrar la línea y carácter donde fue declarada (gran utilidad a la hora de mostrar los errores) y un método para mostrar toda la información que se almacena sobre el símbolo. Este método es utilizado por la tabla de símbolos cuando quiere imprimir los símbolos ordenadamente y formateados.

Una vez que tenemos la estructura de datos para los símbolos, debemos almacenarlos según los analizadores lo vayan necesitando. Para ello, se ha creado una clase que tiene como atributo un HashMap cuya llave es una cadena de caracteres correspondiente al nombre del símbolo y el objeto será el propio símbolo. Además, también contiene la lista de palabras reservadas que, al instanciar la tabla de símbolos, se añadirán todas las palabras reservadas a la tabla de símbolos.

La tabla de símbolos contiene diferentes métodos los cuales pasaremos a describir a continuación:

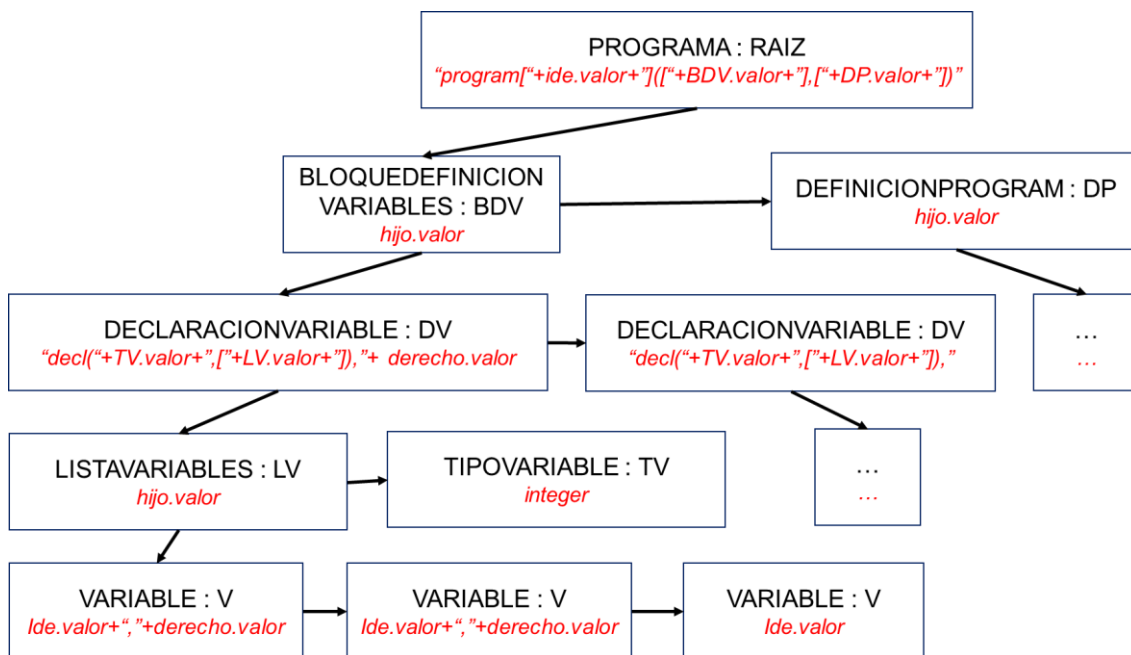
- **public void** insertar(Simbolo s): añade el símbolo a la tabla de símbolos con su nombre.
- **public** String tipo(String nombre): devuelve el tipo del símbolo cuyo nombre se le pasa como argumento.
- **public** String toString(): muestra en una lista los símbolos de la tabla. Primero se muestran los símbolos de tipo reservado y por último las de las variables que se han declarado. Además, en las variables se mostrarán la línea en la han sido definidas o si están inicializadas o no. Ambos tipos de símbolos mostrarán el nombre y una lista en la que se indica dónde se han usado, además de su tipo.
- **public** ArrayList<String> sinUsar(String nodo): retorna una lista dinámica de las diferentes variables que se encuentran en la cadena de caracteres que se le está pasando por argumento.
- **public void** usada(String res, int fila, int columna): añade la línea y carácter del símbolo que se le pasa por argumento (se le indica el nombre) a la lista de lugares donde se ha usado el símbolo.
- **public** ArrayList<Simbolo> getIdentificadores(): devuelve todos los símbolos de todos los identificadores que se encuentran en la tabla de símbolos, a excepción de los símbolos cuyo tipo sea reservado.

- **public** Simbolo getSimbolo(String nombre) : devuelve el símbolo correspondiente al nombre que se le pasa como argumento.
- **public** boolean esta(String nombre) : indica si en la tabla de símbolos se encuentra un símbolo cuyo nombre coincide con el nombre que se le pasa como argumento.

## 6. DISEÑO DEL AST

Para la implementación del árbol de sintaxis abstracta, se ha implementado una clase de tipo árbol que almacenará un nodo, que será la raíz. Por ello, se necesita una estructura de datos para los nodos, esto es una clase nueva. La clase de los nodos tendrá atributos para el valor del nodo, otro para el padre, de tipo Nodo también, y para su hijo y sus hermanos, tanto derecho como izquierdo. Los nodos solo podrán tener un padre (a excepción de la raíz, que su padre será nulo) y un solo hijo. Los nodos que son hijos de otro nodo son los hermanos izquierdos de los nodos que están en el mismo nivel.

Un ejemplo es la declaración de variables. El programa tiene como hijo al bloque de definición de variables y éste tiene como hermano a la definición del programa, por lo que la definición del programa no tiene como padre al nodo del programa, pero su hermano izquierdo sí lo tiene, por lo que automáticamente es hijo del nodo padre. Ahora, el bloque de definición de variables tiene un hijo si existe al menos una definición de variable. Si existiesen más declaraciones de variables entonces serían hermanos por la parte derecha del primer nodo de variables creado, que tiene a su vez como padre al nodo correspondiente al bloque de definición de variables. La definición de la variable tiene un hijo que es la lista de variables, y éste tiene un hermano que es el tipo de variable que son los identificadores de la lista de variables. Por último, la lista de variables tiene un hijo por cada variable que se está definiendo. Se puede observar gráficamente la explicación con más claridad:



En negro se indica el nombre del nodo, en rojo el valor de dicho nodo. Se puede observar cómo se relacionan los diferentes nodos. En el ejemplo se hace una abstracción sobre las sentencias, pero la filosofía es la misma, puesto que la misma estructura se crea por el otro lado del árbol.

Gracias a la variable RESULT que proporciona CUP e, indicando todos los no terminales como Nodos, se puede generar el árbol de sintaxis desde las hojas, de manera ascendente, puesto que la gramática, aunque vaya analizando sintácticamente de manera descendente, el código que primero ejecutará será el de las hojas, cuando la

derivación de la producción llegue a su fin. De esta manera, para imprimir por pantalla el AST tan solo nos vale con mostrar el valor del nodo raíz. La clase `Arbol.java` está dotada de los métodos necesarios para emparentar ya sea entre hermanos o entre padres e hijos, los diferentes nodos.

## 7. CASOS DE PRUEBA

Para comprobar que el compilador funciona correctamente, desarrollamos diez casos de prueba diferentes, de los cuales tres de ellos serán programas bien definidos y los otros siete con errores. Este apartado incluye la salida de un archivo “.prog” correctamente y el listado de errores de los otros siete. Los otros dos casos de prueba correctos que no se detallarán aquí son adjuntados con el presente documento en un fichero “.txt”. Los ficheros correctos son mediaEntera.txt, sumaPrimerosCinco.txt y factorial.txt

La salida para el programa del fichero mediaEntera.txt (tras haber cambiado su tipo a “.prog” y pasárselo al analizador) es el siguiente (diferentes aspectos de la salida han sido modificados para que se ajustara al siguiente cuadro y facilitar la legibilidad):

```
***** Análisis del archivo "file.prog" *****
-----
Análisis léxico completado.

[program, ide(mediaentera), is, var, ide(numero), coma, ide(total),
dos_puntos, integer, punto_coma, var, ide(count), coma, ide(media),
coma, ide(hipot), dos_puntos, integer, punto_coma, var, ide(real),
coma, ide(acabar), dos_puntos, boolean, punto_coma, begin, ide(count),
asign, num(0), punto_coma, ide(total), asign, num(0), punto_coma,
while, not, paren_izq, ide(acabar), paren_der, do, read, ide(numero),
punto_coma, if, ide(numero), comp, num(-1), then, ide(acabar), asign,
true, punto_coma, else, ide(count), asign, ide(count), mas, num(1),
punto_coma, ide(total), asign, ide(total), mas, ide(numero),
punto_coma, end, if, punto_coma, end, while, punto_coma, ide(media),
asign, ide(total), entre, ide(count), punto_coma, write, ide(media),
punto_coma, ide(hipot), asign, ide(media), por, ide(count),
punto_coma, if, paren_izq, ide(media), comp, ide(hipot), paren_der,
then, ide(real), asign, true, punto_coma, else, ide(real), asign,
true, punto_coma, end, if, punto_coma, write, ide(real), punto_coma,
end]

-----
Análisis Sintáctico completado

AST resultado del analisis.
program[mediaEntera]([decl(integer,[numero,total]),decl(integer,
[count,media,hipot]),decl(boolean,[real,acabar])],[asign(count,0),
asign(total,0),while(not(ide(acabar))],[read(ide(numero)),if(exp(
igual,ide(numero),num(-1)),
[asign(acabar,true)],else[asign(count,exp(ide(count),mas,num(1))),
asign(total,exp(ide(total),mas,ide(numero)))]), skip),
asign(media,exp(ide(total),entre,ide(count))),write(ide(media)),
asign(hipot,exp(ide(media),por,ide(count))),if(exp(igual,ide(media),
ide(hipot))],[asign(real,true)],else[asign(real,true)]),write(
ide(real))])

-----
Orden de las producciones gramaticales:

Regla #1 --> LVA ::= .
Regla #2 --> LVA ::= tk_coma tk_identificador LVA
Regla #3 --> LISTAVARIABLES ::= tk_identificador LVA
Regla #4 --> DECLARACIONTIPO ::= tk_integer
```

```

Regla #5 --> DEFINICIONVARIABLE ::= tk_var LISTAVARIABLES tk_dospuntos
DECLARACIONTIPO tk_puntoycoma
Regla #6 --> LVA ::= .
Regla #7 --> LVA ::= tk_coma tk_identificador LVA
Regla #8 --> LVA ::= tk_coma tk_identificador LVA
Regla #9 --> LISTAVARIABLES ::= tk_identificador LVA
Regla #10 --> DECLARACIONTIPO ::= tk_integer
Regla #11 --> DEFINICIONVARIABLE ::= tk_var LISTAVARIABLES
tk_dospuntos DECLARACIONTIPO tk_puntoycoma
Regla #12 --> LVA ::= .
Regla #13 --> LVA ::= tk_coma tk_identificador LVA
Regla #14 --> LISTAVARIABLES ::= tk_identificador LVA
Regla #15 --> DECLARACIONTIPO ::= tk_boolean
Regla #16 --> DEFINICIONVARIABLE ::= tk_var LISTAVARIABLES
tk_dospuntos DECLARACIONTIPO tk_puntoycoma
Regla #17 --> DVA ::= .
Regla #18 --> DVA ::= DEFINICIONVARIABLE DVA
Regla #19 --> DVA ::= DEFINICIONVARIABLE DVA
Regla #20 --> DVA ::= DEFINICIONVARIABLE DVA
Regla #21 --> BLOQUEDEFINICIONVARIABLES ::= DVA
Regla #22 --> IDEOENTERO ::= tk_numeroentero
Regla #23 --> LISTADEASIGNACIONES ::= .
Regla #24 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #25 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #26 --> IDEOENTERO ::= tk_numeroentero
Regla #27 --> LISTADEASIGNACIONES ::= .
Regla #28 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #29 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #30 --> COMPARACION ::= tk_identificador
Regla #31 --> EXPRESIONBOOLEANA ::= COMPARACION
Regla #32 --> EAA ::= .
Regla #33 --> LISTAEXPRESIONES ::= EXPRESIONBOOLEANA EAA
Regla #34 --> EXPRESIONBOOLEANA ::= tk_not tk_parenizq
LISTAEXPRESIONES tk_parender
Regla #35 --> EAA ::= .
Regla #36 --> LISTAEXPRESIONES ::= EXPRESIONBOOLEANA EAA
Regla #37 --> DEFINICIONSENTENCIA ::= tk_read tk_identificador
tk_puntoycoma
Regla #38 --> COMPARACION ::= tk_identificador tk_comparacion
tk_numeroentero
Regla #39 --> EXPRESIONBOOLEANA ::= COMPARACION
Regla #40 --> EAA ::= .
Regla #41 --> LISTAEXPRESIONES ::= EXPRESIONBOOLEANA EAA
Regla #42 --> AVERQUEASIGNAMOS ::= tk_true
Regla #43 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #44 --> BDSA ::= .
Regla #45 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #46 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #47 --> IDEOENTERO ::= tk_identificador
Regla #48 --> OPERADORMATES ::= tk_suma
Regla #49 --> IDEOENTERO ::= tk_numeroentero
Regla #50 --> LISTADEASIGNACIONES ::= .
Regla #51 --> LISTADEASIGNACIONES ::= OPERADORMATES IDEOENTERO
LISTADEASIGNACIONES
Regla #52 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #53 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #54 --> IDEOENTERO ::= tk_identificador

```

```

Regla #55 --> OPERADORMATES ::= tk_suma
Regla #56 --> IDEOENTERO ::= tk_identificador
Regla #57 --> LISTADEASIGNACIONES ::= .
Regla #58 --> LISTADEASIGNACIONES ::= OPERADORMATES IDEOENTERO
LISTADEASIGNACIONES
Regla #59 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #60 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #61 --> BDSA ::= .
Regla #62 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #63 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #64 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #65 --> DEFINICIONELSE ::= tk_else BLOQUEDESENTENCIAS
Regla #66 --> DEFINICIONIF ::= tk_if LISTAEXPRESIONES tk_then
BLOQUEDESENTENCIAS DEFINICIONELSE tk_end tk_if tk_puntoycoma
Regla #67 --> DEFINICIONSENTENCIA ::= DEFINICIONIF
Regla #68 --> BDSA ::= .
Regla #69 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #70 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #71 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #72 --> DEFINICIONSENTENCIA ::= tk_while LISTAEXPRESIONES tk_do
BLOQUEDESENTENCIAS tk_end tk_while tk_puntoycoma
Regla #73 --> IDEOENTERO ::= tk_identificador
Regla #74 --> OPERADORMATES ::= tk_division
Regla #75 --> IDEOENTERO ::= tk_identificador
Regla #76 --> LISTADEASIGNACIONES ::= .
Regla #77 --> LISTADEASIGNACIONES ::= OPERADORMATES IDEOENTERO
LISTADEASIGNACIONES
Regla #78 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #79 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #80 --> DEFINICIONSENTENCIA ::= tk_write tk_identificador
tk_puntoycoma
Regla #81 --> IDEOENTERO ::= tk_identificador
Regla #82 --> OPERADORMATES ::= tk_multiplicacion
Regla #83 --> IDEOENTERO ::= tk_identificador
Regla #84 --> LISTADEASIGNACIONES ::= .
Regla #85 --> LISTADEASIGNACIONES ::= OPERADORMATES IDEOENTERO
LISTADEASIGNACIONES
Regla #86 --> AVERQUEASIGNAMOS ::= IDEOENTERO LISTADEASIGNACIONES
Regla #87 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #88 --> COMPARACION ::= tk_identificador tk_comparacion
tk_identificador
Regla #89 --> EXPRESIONBOOLEANA ::= COMPARACION
Regla #90 --> EAA ::= .
Regla #91 --> LISTAEXPRESIONES ::= EXPRESIONBOOLEANA EAA
Regla #92 --> EXPRESIONBOOLEANA ::= tk_parenizq LISTAEXPRESIONES
tk_parender
Regla #93 --> EAA ::= .
Regla #94 --> LISTAEXPRESIONES ::= EXPRESIONBOOLEANA EAA
Regla #95 --> AVERQUEASIGNAMOS ::= tk_true
Regla #96 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #97 --> BDSA ::= .
Regla #98 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #99 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #100 --> AVERQUEASIGNAMOS ::= tk_true
Regla #101 --> DEFINICIONSENTENCIA ::= tk_identificador tk_asignacion
AVERQUEASIGNAMOS tk_puntoycoma
Regla #102 --> BDSA ::= .

```



```

Regla #103 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #104 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #105 --> DEFINICIONELSE ::= tk_else BLOQUEDESENTENCIAS
Regla #106 --> DEFINICIONIF ::= tk_if LISTAEXPRESIONES tk_then
BLOQUEDESENTENCIAS DEFINICIONELSE tk_end tk_if tk_puntoycoma
Regla #107 --> DEFINICIONSENTENCIA ::= DEFINICIONIF
Regla #108 --> DEFINICIONSENTENCIA ::= tk_write tk_identificador
tk_puntoycoma
Regla #109 --> BDSA ::= .
Regla #110 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #111 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #112 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #113 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #114 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #115 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #116 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #117 --> BDSA ::= DEFINICIONSENTENCIA BDSA
Regla #118 --> BLOQUEDESENTENCIAS ::= BDSA
Regla #119 --> DEFINICIONPROGRAMA ::= tk_begin BLOQUEDESENTENCIAS
tk_end
Regla #120 --> PROGRAM ::= tk_program tk_identificador tk_is
BLOQUEDEFINICIONVARIABLES DEFINICIONPROGRAMA

```

#### Tabla de símbolos

##### Identificadores reservados:

.....

Nombre Usado en lineas (Linea,Carácter)

.....

skip	[ ]
program	[ (1,1) ]
integer	[ (2,20), (3,26) ]
do	[ (9,19) ]
while	[ (9,1), (17,5) ]
not	[ (9,7) ]
else	[ (13,2), (27,1) ]
and	[ ]
end	[ (16,2), (17,1), (29,1), (32,0) ]
write	[ (20,1), (31,1) ]
if	[ (11,2), (16,6), (25,1), (29,5) ]
read	[ (10,2) ]
or	[ ]
var	[ (2,1), (3,1), (4,1) ]
false	[ ]
is	[ (1,21) ]
then	[ (11,17), (25,18) ]
boolean	[ (4,20) ]
true	[ (12,13), (26,10), (28,10) ]
begin	[ (5,0) ]

.....

Nombre	Tipo	Valor	Linea	Carac.	Inic.	Usada en lineas (Linea,Carácter)
numero	integer	0	2	5	No	[ (10,7), (11,5), (15,20) ]
media	integer	0	3	12	Sí	[ (18,1), (20,7), (22,10), (25,4) ]
total	integer	0	2	13	Sí	[ (7,1), (15,12), (15,3), (18,10) ]
acabar	boolean	false	4	11	Sí	[ (9,11), (12,3) ]
hipot	integer	0	3	19	Sí	[ (22,1), (25,12) ]
mediaEntera	pseudo	undefined	1	9	No	[ ]

count	integer	0	3	5	Sí	[(6,1), (14,12), (14,3), (18,18), (22,18)]
real	boolean	false	4	5	Sí	[(26,2), (28,2), (31,7)]

A continuación, se muestra los diferentes errores de los programas erróneos (de nuevo, cambiando su extensión a “prog”):

#### escrituraAntesDeLectura.txt

```
--> ERROR #1 - (SEMÁNTICO) int no se puede escribir si no ha sido
antes inicializada. Linea: 4, carácter: 7
    write int;
      ^
```

#### sinUsar.txt

```
--> ERROR #1 - (SEMÁNTICO) La variable segunda ha sido declarada
pero no usada.
    var primera, segunda : integer;
      ^
```

#### concordancia.txt

```
--> ERROR #1 - (SEMÁNTICO) bool es de tipo boolean, mientras que se
le está asignando un valor de tipo entero. Linea: 7, carácter: 1
    bool := entero;
      ^
```

#### sinFin.txt

```
--> ERROR #1 - (SEMÁNTICO) Es posible que el while definido en la
línea 5, carácter: 1 sea indefinido.
    while sinfin do
      ^
```

#### sinDeclarar.txt

```
--> ERROR #1 - (SEMÁNTICO) sindeclarar no fue declarada. Linea: 4,
carácter: 18
    declarada := not(sindeclarar);
                  ^
```

#### nosFaltaPuntoYComa.txt

```
--> ERROR #1 - (SINTÁCTICO) Error de sintaxis línea 6, columna 1
    k := 10;
      ^
--> ERROR #2 - (SINTÁCTICO) Problema en la sentencia de la línea 5,
columna 6
    j := 10
      ^
--> ERROR #3 - (SEMÁNTICO) La variable j ha sido declarada pero no
usada.
    var i, j, k : integer;
      ^
```

## LexemaNoReconocido.txt

```
--> ERROR #1 - (LÉXICO) lexema no reconocido en línea 4, columna 1
    $temp := 10;
    ^
--> ERROR #2 - (SEMÁNTICO) temp no fue declarada. Linea: 4,
carácter: 2
    $temp := 10;
    ^
--> ERROR #3 - (LÉXICO) lexema no reconocido en línea 5, columna 8
    if i > $temp then
        ^
--> ERROR #4 - (SEMÁNTICO) temp no fue declarada. Linea: 5,
carácter: 9
    if i > $temp then
        ^
```