Procesadores del Lenguaje – Trabajo Final

Desarrollo de un traductor de gramáticas en forma de Backus-Naur a forma normal de Chomsky

Memoria Técnica



Logotipo

Descripción generada automáticamente

Realizado por:

Álvaro Esteban Muñoz

**Índice de contenido**

[Introducción 4](#_Toc75271535)

[Análisis Léxico 4](#_Toc75271536)

[Análisis Sintáctico 6](#_Toc75271537)

[Análisis Semántico 7](#_Toc75271538)

[Árbol de Sintaxis Abstracta propuesto 7](#_Toc75271539)

[Transformación de la gramática en una gramática atribuida 10](#_Toc75271540)

[Gestión de errores 11](#_Toc75271541)

[Algoritmo de traducción de la gramática 13](#_Toc75271542)

[Primer paso de la traducción 14](#_Toc75271543)

[Segundo paso de la traducción 15](#_Toc75271544)

[Tercer paso de la traducción 16](#_Toc75271545)

[Cuarto paso de la traducción 17](#_Toc75271546)

[Funciones auxiliares empleadas en el algoritmo de traducción 17](#_Toc75271547)

[Pruebas sobre la aplicación 20](#_Toc75271548)

**Índice de ilustraciones**

[Ilustración 1: Implementación Léxica (1) 5](#_Toc75270704)

[Ilustración 2: Implementación Léxica (2) 5](#_Toc75270705)

[Ilustración 3: Implementación Sintáctica 6](#_Toc75270706)

[Ilustración 4: Implementación de Gramatica.java 7](#_Toc75270707)

[Ilustración 5: Implementación de Definición.java 8](#_Toc75270708)

[Ilustración 6: Implementación de Regla.java 8](#_Toc75270709)

[Ilustración 7: Implementación de Symbol.java 9](#_Toc75270710)

[Ilustración 8: Implementación de TSymbol.java 9](#_Toc75270711)

[Ilustración 9: Implementación de NTSymbol.java 9](#_Toc75270712)

[Ilustración 10: Implementación Semántica (1) 10](#_Toc75270713)

[Ilustración 11: Implementación Semántica (2) 10](#_Toc75270714)

[Ilustración 12: Método skipTo para la gestión de errores 11](#_Toc75270715)

[Ilustración 13: Ejemplo de tokens de sincronización 11](#_Toc75270716)

[Ilustración 14: Método para generar la traducción 13](#_Toc75270717)

[Ilustración 15: Primera parte del primer paso de traducción 14](#_Toc75270718)

[Ilustración 16: Segunda parte del primer paso de traducción 14](#_Toc75270719)

[Ilustración 17: Segundo paso de traducción 15](#_Toc75270720)

[Ilustración 18: Tercer paso de traducción 16](#_Toc75270721)

[Ilustración 19: Cuarto paso de traducción 17](#_Toc75270722)

[Ilustración 20: Método borraSimb() 17](#_Toc75270723)

[Ilustración 21: Método sustituyeSimb() - 1 18](#_Toc75270724)

[Ilustración 22: Método sustituyeSimb() - 2 18](#_Toc75270725)

[Ilustración 23: Método creaSimbolo() 18](#_Toc75270726)

[Ilustración 24: Método getNumTerminales() 19](#_Toc75270727)

[Ilustración 25: Ejemplo fichero de entrada 20](#_Toc75270728)

[Ilustración 26: Salida del traductor (Explorador de archivos) 20](#_Toc75270729)

[Ilustración 27: Salida del traductor 21](#_Toc75270730)

[Ilustración 28: Ejemplo de error en la entrada del traductor 21](#_Toc75270731)

[Ilustración 29: Salida de errores del traductor (Explorador de archivos) 22](#_Toc75270732)

[Ilustración 30: Salida de errores del traductor 22](#_Toc75270733)

# Introducción

El objetivo de nuestra aplicación es la generación de una gramática descrita en Forma Normal de Chomsky (CNF o Chomsky Normal Form) a partir de su descripción en forma de Backus-Naur (BNF o Backus-Naur Form). En otras palabras, estamos ante un traductor que toma como entrada la descripción de una gramática en BNF y la traduce a su descripción en CNF.



El traductor se va a implementar usando el lenguaje de programación Java y haciendo uso de un metacompilador llamado JavaCC. El metacompilador nos ayudará a resolver el análisis léxico, el análisis sintáctico y nos ayudará con gran parte del análisis semántico.

# Análisis Léxico

El metacompilador JavaCC hace muy sencilla la implementación del análisis léxico ya que solo tendremos que plasmar las expresiones regulares usadas para describir cada token en un fichero “.jj“ y JavaCC se encargará de crear todas las clases relacionadas.

|  |  |
| --- | --- |
| Especificación | Expresión regular |
| blanco | ( " " | "\r" | "\n" | "\t" ) |
| comentario | "/\*" ( ("\*")\* ~["\*", "/"] | "/" )\* ("\*")+ "/" |
| NOTERMINAL | ["\_","a"-"z","A"-"z"]  ( ["\_","a"-"z","A"-"Z","0"-"9"] )\* |
| TERMIINAL | "<" ["\_","a"-"z","A"-"z"]  ( ["\_","a"-"z","A"-"Z","0"-"9"] )\* ">" |
| EQ | "::=" |
| BAR | "|" |
| SEMICOLON | ";" |

La parte en del fichero “BNF2CNF\_Parser.jj” en la que se describe la especificación léxica se divide dependiendo de la acción a tomar por el AFN generado por el metacompilador, usando la cláusula “SKIP” el AFN omitirá la cadena y con la cláusula TOKEN el AFN generará un token.

* Cadenas para omitir:

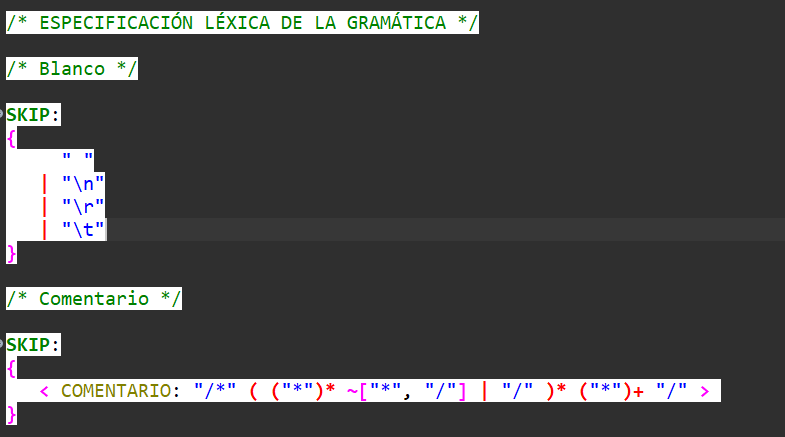


Ilustración 1: Implementación Léxica (1)

* Generadores de token:

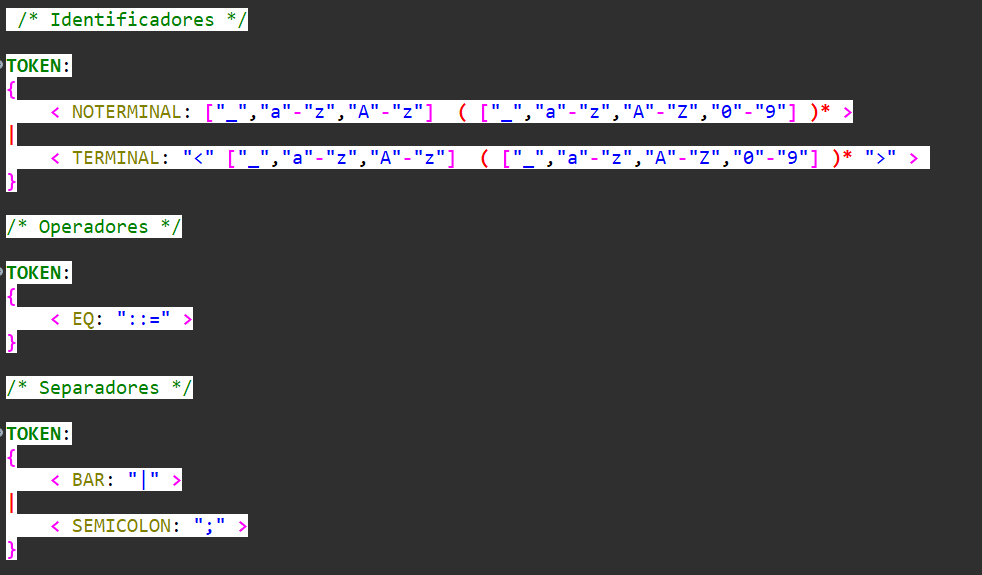


Ilustración 2: Implementación Léxica (2)

# Análisis Sintáctico

La implementación del analizador sintáctico es tan sencilla como la del analizador léxico, de hecho, se realiza dentro del mismo fichero “.jj” que el analizador léxico. La parte del fichero “BNF2CNF\_Parser.jj” dedicada al analizador sintáctico está colocada justo a continuación del analizador léxico para que sea fácil de leer.

La especificación sintáctica de nuestro lenguaje a ser reconocido es la siguiente:

* Gramatica ::= (  Definicion  )\*
* Definicion  ::=    **NOTERMINAL**   **EQ**  ListaReglas  **SEMICOLON**
* ListaReglas ::=  Regla  ( **BAR**  Regla )\*
* Regla ::=  ( **NOTERMINAL**  |  **TERMINAL** )\*

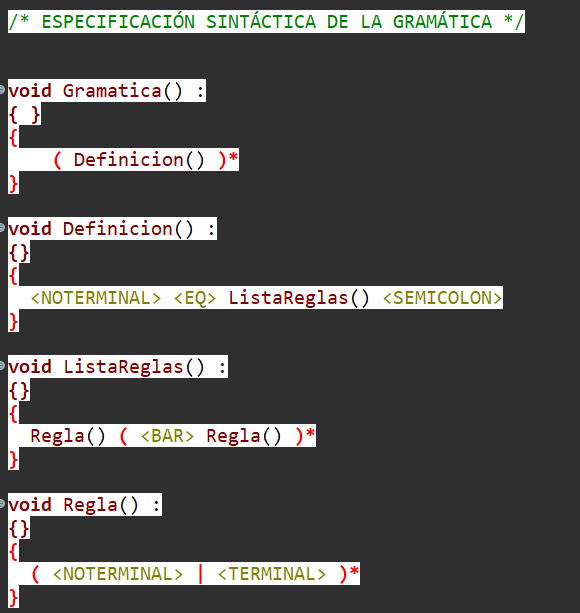


Ilustración 3: Implementación Sintáctica

# Análisis Semántico

Para la implementación del análisis semántico se complican un poco las cosas, puesto que debemos pensar en una forma de estructurar la información que vamos a almacenar en la memoria, aquella información sobre el lenguaje procesado que luego usaremos para generar el código de salida.

## Árbol de Sintaxis Abstracta propuesto

La información que mantendremos en memoria se estructurará según las siguientes clases:

* **Gramática**: Contendrá una estructura *ArrayList* de objetos de la clase *Definición*
* **Definición**: Constatará de dos partes, un objeto de la clase *NTSymbol* y una estructura *ArrayList* de objetos de la clase *Regla*.
* **Regla**: Contendrá una estructura *ArrayList* de objetos de tipo *Symbol*.
* **Symbol**: Interfaz que implementarán tanto los *NTSymbol* como los *TSymbol*, de esta forma podremos abarcarlos en el mismo *ArrayList* independientemente del tipo de su instancia.
* **NTSymbol**: Contiene una *String* que guardará el nombre del símbolo, en otras palabras, el lexema usado para representar a dicho símbolo no terminal.
* **TSymbol**: Es exactamente igual que un *NTSymbol*, creamos una clase diferente para poder diferenciar el tipo de símbolo que estamos leyendo.

*\*Todas las clases tienen un método toString() que hemos implementado para poder depurar el programa de forma más intuitiva.*



Ilustración 4: Implementación de Gramatica.java

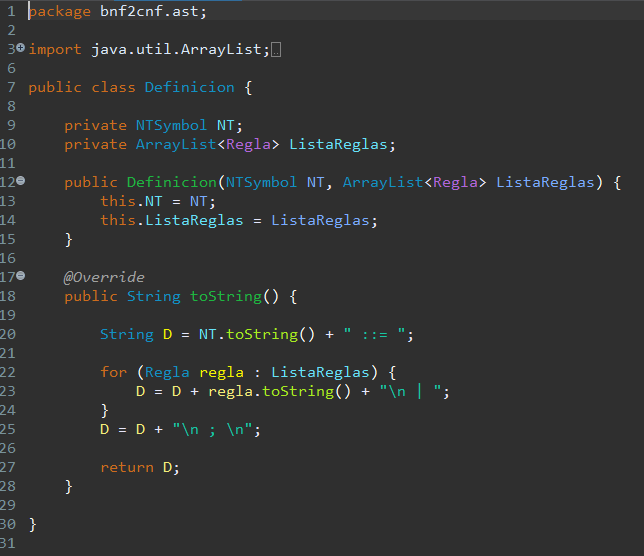


Ilustración 5: Implementación de Definición.java

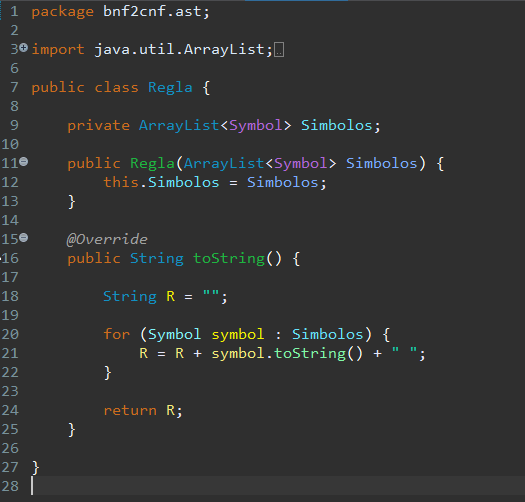


Ilustración 6: Implementación de Regla.java

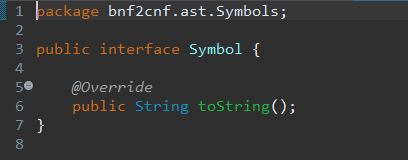


Ilustración 7: Implementación de Symbol.java

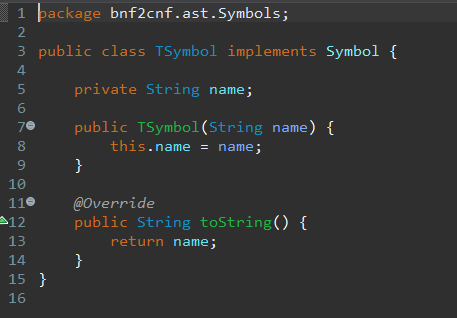


Ilustración 8: Implementación de TSymbol.java

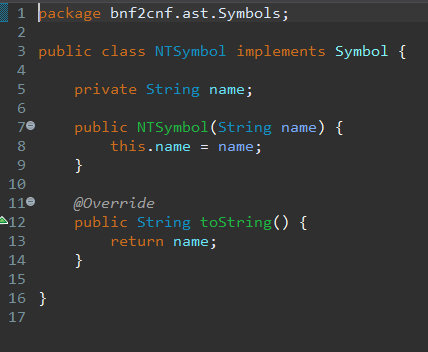


Ilustración 9: Implementación de NTSymbol.java

## Transformación de la gramática en una gramática atribuida

Una vez tenemos creada la estructura que tendrá el árbol de sintaxis abstracta, tendremos que convertir nuestra gramática en una gramática atribuida, de forma que pueda almacenar la información en las clases que hemos implementado.

Para transformar nuestra gramática en una gramática atribuida tendremos que desarrollar el **ETDS** (esquema de traducción dirigido por la sintaxis) en nuestro fichero de javaCC.

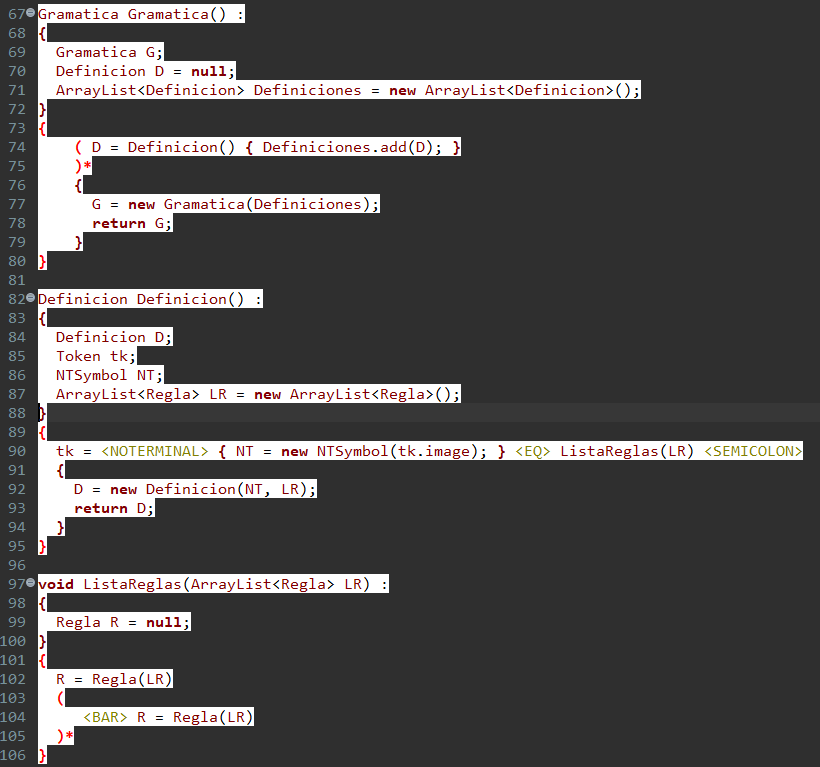


Ilustración 10: Implementación Semántica (1)

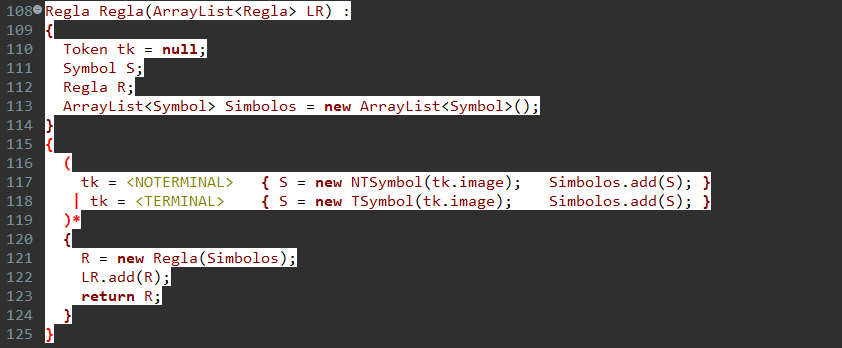


Ilustración 11: Implementación Semántica (2)

# Gestión de errores

En JavaCC la gestión de errores léxicos se hace automáticamente así que no tendremos que preocuparnos por esa parte, simplemente definiremos la forma de recuperarnos de errores sintácticos y semánticos.

Para recuperarnos de los errores hemos creado la función skipTo() que nos permite saltarnos los tokens de entrada hasta llegar a un token de sincronismo.

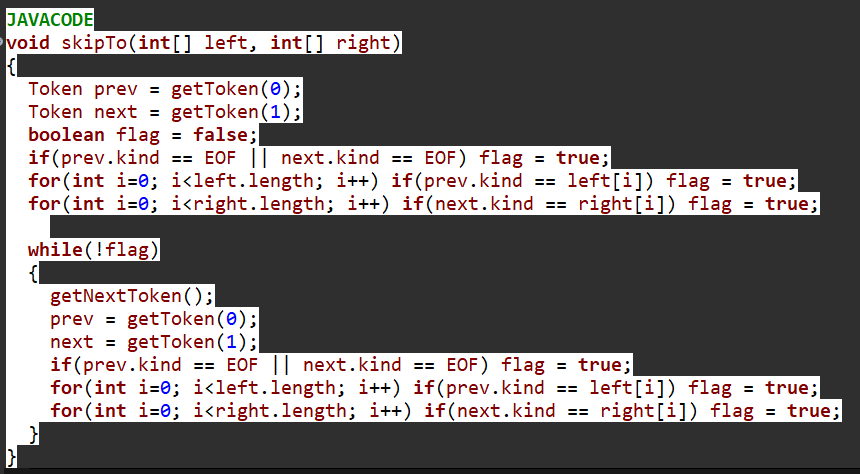


Ilustración 12: Método skipTo para la gestión de errores

Una vez creado este método, lo usaremos para recuperarnos de los errores, tendremos que definir los tokens de sincronismo a izquierda y derecha al comienzo de cada clase.

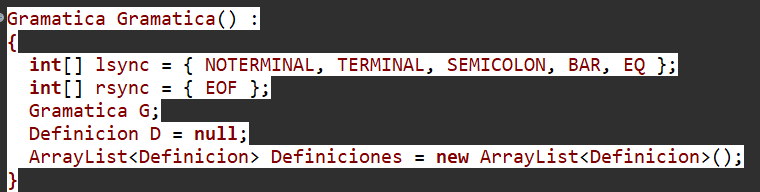


Ilustración 13: Ejemplo de tokens de sincronización

Los tokens de sincronismo a izquierda (lsync) y derecha (rsync) quedarán de la siguiente forma:

- Gramática:

Izquierda 🡪 <NOTERMINAL>, <TERMINAL>, <SEMICOLON>, <BAR>, <EQ>

Derecha 🡪 <EOF>

- Definición:

Izquierda 🡪 <SEMICOLON>

Derecha 🡪 [ ]

- ListaReglas:

Izquierda 🡪 [ ]

Derecha 🡪 <SEMICOLON>

- Regla:

Izquierda 🡪 [ ]

Derecha 🡪 <BAR>, <SEMICOLON>

# Algoritmo de traducción de la gramática

Para la implementación del algoritmo de traducción hemos creado una clase nueva *“ChomskyGenerato.java”* que se encargará de recibir el árbol de sintaxis abstracta de la gramática analizada y la traducirá al nuevo árbol de sintaxis abstracta de la gramática en forma normal de Chomsky. El algoritmo de traducción es algo largo, pero se puede dividir en cuatro pasos que hemos implementado de la siguiente forma:

Texto

Descripción generada automáticamente

Ilustración : Método para generar la traducción

## Primer paso de la traducción

El primer paso consiste en eliminar las reglas lambda, hemos dividido este trabajo en dos partes, primero buscamos aquellas definiciones que contengan alguna regla lambda y almacenamos el símbolo NT de la definición en un **ArrayList**. Tras esto, recorremos el **ArrayList** de los símbolos de las definiciones que han sido modificadas y si encontramos una regla que derive en dicho símbolo, crearemos una copia de esa regla, pero eliminando el símbolo que se encuentra en el array de la regla copiada, así tendremos una regla que deriva en el símbolo y otra que no.

Texto

Descripción generada automáticamente

Ilustración : Primera parte del primer paso de traducción

Texto

Descripción generada automáticamente

Ilustración : Segunda parte del primer paso de traducción

## Segundo paso de la traducción

En el segundo paso debíamos aislar los símbolos terminales a definiciones que deriven directamente en ellos, para lograr esto, buscábamos aquellas reglas dentro de cada definición que contuvieran símbolos terminales y seguidamente los sustituíamos por un símbolo no terminal nuevo o por uno ya existente si fuera posible.

Texto

Descripción generada automáticamente

Ilustración : Segundo paso de traducción

## Tercer paso de la traducción

En el tercer paso de la traducción debíamos expandir las reglas que tuvieran un único símbolo no terminal en la parte derecha. Conseguir esto se resume en buscar en cada definición aquellas reglas con un único símbolo terminal a la derecha y añadir las reglas en las que deriva ese símbolo terminal a la definición que estamos comprobando en ese momento.

Texto

Descripción generada automáticamente

Ilustración : Tercer paso de traducción

## Cuarto paso de la traducción

Para finalizar, teníamos que trocear las reglas que contuvieran más de dos símbolos a la derecha, así que recorríamos las definiciones buscando aquellas reglas con más de dos símbolos y si este era el caso, almacenábamos los símbolos sobrantes en un Array que eran sustituidos por un símbolo nuevo.

Texto

Descripción generada automáticamente

Ilustración : Cuarto paso de traducción

## Funciones auxiliares empleadas en el algoritmo de traducción

Para realizar ciertas funciones y no perdernos en la complejidad del código, hemos implementado unas funciones sobre las que hemos delegado algunas tareas importantes o repetitivas, a continuación, se puede observar una lista de dichas funciones y su implementación.

Texto

Descripción generada automáticamente

Ilustración 20: Método borraSimb()

Texto

Descripción generada automáticamente

Ilustración 21: Método sustituyeSimb() - 1

Texto

Descripción generada automáticamente

Ilustración 22: Método sustituyeSimb() - 2

Para el siguiente método hemos creado una variable nueva en la clase “**Regla**”, es una variable estática que nos permite llevar un mismo contador para todos los objetos de tipo de “**Regla**”, de esta forma cada vez que creemos un nuevo símbolo no terminal no tendremos problemas con que se repita el nombre de Aux1, Aux2, …

Esta variable se obtiene mediante el método estático *getContador()* y se actualiza mediante el método estático *setContador()*.

Texto

Descripción generada automáticamente

Ilustración 23: Método creaSimbolo()

Texto

Descripción generada automáticamente

Ilustración 24: Método getNumTerminales()

# Pruebas sobre la aplicación

Para comprobar el correcto funcionamiento de la aplicación hemos tomado el fichero de entrada proporcionado junto al enunciado de la práctica.

Texto

Descripción generada automáticamente

Ilustración : Ejemplo fichero de entrada

Si todo va bien, al ejecutar el traductor junto al archivo de entrada en el mismo directorio se generará una salida.

Captura de pantalla de un celular

Descripción generada automáticamente

Ilustración : Salida del traductor (Explorador de archivos)

El resultado es algo similar a lo siguiente:

Tabla

Descripción generada automáticamente

Ilustración : Salida del traductor

También nos interesa comprobar que la gestión de errores funciona sin ningún problema, modificaremos el fichero para comprobar que se genera una salida de errores.

Texto

Descripción generada automáticamente

Ilustración : Ejemplo de error en la entrada del traductor

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Ilustración : Salida de errores del traductor (Explorador de archivos)

El resultado obtenido es el fichero siguiente:

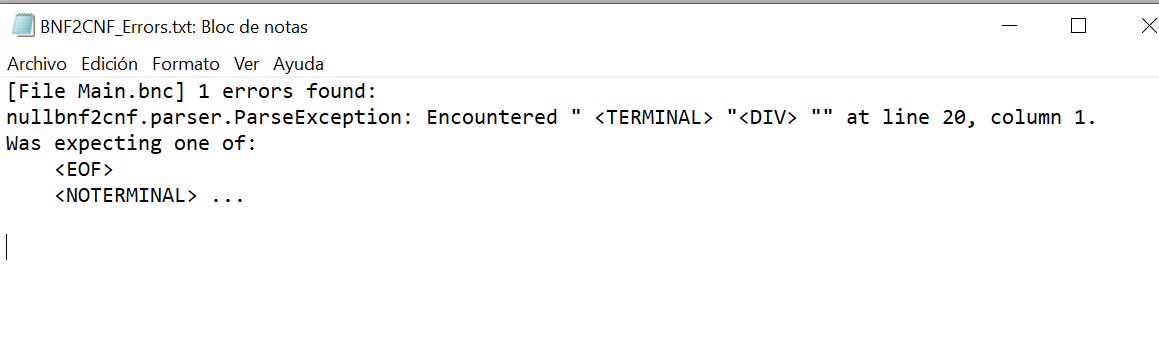


Ilustración : Salida de errores del traductor