Contenido

[Clase DescendantLL V0.1 1](#_Toc9187501)

[Clase DescendantLL V0.2 1](#_Toc9187502)

[Clase LL V0.1 1](#_Toc9187503)

[Clase LL V0.2 5](#_Toc9187504)

[Clase LL V0.3 8](#_Toc9187505)

[Class Node V0.1 12](#_Toc9187506)

[Class Paso V0.1 12](#_Toc9187507)

## Clase DescendantLL V0.1

De momento solo llama a la clase ll con el path de la gramática a convertir.

## Clase DescendantLL V0.2

Además construye el analizador descendente y procesa la cadena que se introduzca.

## Clase LL V0.1

**Atributos:**

**private HashMap<String, ArrayList<String>> grammarWithActions:** representa la gramática con acciones semánticas y con no terminales con índices. Los índices del mapa son los antecedentes y guarda todas las produccionesdicho antecedente.

**private HashMap<String, ArrayList<String>> gramar:** representa la gramática sin acciones semánticas y con no terminales sin índices. Los índices del mapa son los antecedentes y guarda todas las produccionesdicho antecedente.

**private String path:** ruta del archivo que guarda la gramatica

**private HashMap<String,ArrayList<String>> head:** guarda las cabeceras de los antecedentes. Los índices del mapa son los antecedentes.

**private HashMap<String,ArrayList<String>> headPrime:** guarda las cabeceras prima de los antecedentes. Los índices del mapa son los antecedentes.

**private HashMap<String,ArrayList<String>> nexts:** guarda los siguientes de los antecedentes. Los índices del mapa son los antecedentes.

**private HashMap<String,ArrayList<String>> directors:** guarda los directores de las producciones. Los índices del mapa son los antecedentes más la producción.

**private String axioma:** axioma de la gramática.

**private Integer[][] table:** tabla usada para poder reconocer la cadena de entrada.

**private ArrayList<String> terminals:** lista de terminales.

**private ArrayList<String> noTerminals:** lista de no terminales.

**Constructor:**

**public LL( String path):**

Recoge el path del archivo que contiene la gramática, inicializa los mapas y las listas, añade todos los antecedentes a la lista de terminales, crea una tabla con tantas filas como no terminales y tantas columnas como terminales, calcula las cabeceras, los siguientes y los directores.

**Métodos:**

**private void grammarWithoutActions():**

Crea una nueva gramática sin acciones y sin índices en los no terminales.

Por cada antecedente en la gramática, por cada producción, eliminamos las acciones y los índices de la producción y lo añadimos a una lista. Cuando no hay más producciones se guarda en un mapa la lista usando como índice el antecedente.

**private String removeActions(String production):**

Elimina las acciones semánticas y los índices de los no terminales de una producción.

Se divide la producción en símbolos y por cada símbolo si no empieza con “{”, si es un no terminal sin índice se añade a la nueva String que será la nueva producción, si no se añade una subcadena del símbolo desde el inicio has que aparecen dígitos. Devuelve la nueva producción.

Para saber si un elemento es un digito:

<http://lineadecodigo.com/java/%C2%BFes-ese-caracter-un-numero/>

**private Integer getNumberIndex(String symbol):**

Dado un símbolo en forma de String devuelve la posición donde comienzan los dígitos.

Se pasa el símbolo a un array de caracteres y por cada uno si es un digito devuelve la posición. Si se terminan los símbolos y no hay dígitos devuelve el tamaño de la cadena.

**private void readFile():**

Lee línea a línea el archivo con la gramática y la guarda.

Por cada línea la divide en producciones y antecedentes, si el contador es 0 se asigna el antecedente al axioma, se divide la String con todas las producciones en otras String con cada producción, se crea una lista y se guarda en el mapa con el antecedente como índice, se añaden todas las producciones a la lista y se suma uno al contador.

Para leer archivos con lambda:

<https://stackoverflow.com/questions/979932/read-unicode-text-files-with-java>

**public void directors():**

Calcula los directores de todas las producciones.

Por cada antecedente, por cada producción calcula el director.

**public void director(String production,String antecedent):**

Calcula los directores de una producción.

Separa la producción en símbolos,

Se inicializa la lista donde se van a guardar los directores.

Si el primer carácter del primer símbolo está en mayúscula:

Si la cabecera contiene landa se insertan los símbolos de la cabecera prima en la lista y los directores de la producción a partir del símbolo si no están ya en la lista.

Si no, inserta los símbolos de la cabecera si no están ya en la lista.

Si el primer símbolo es landa se insertan los siguientes del antecedente de la producción

Si no se añade el símbolo si no está ya.

Por último, se añaden los directores al mapa con el antecedente más la producción como índice.

Para distinguir entre mayúscula y miniscula:

<https://brainly.lat/tarea/7075675>

**public ArrayList<String> directorAux(String production,String antecedent):**

Calcula los directores de una producción que no se vayan a guardar en el mapa de directores. Es igual al anterior solo que en vez de guardar el resultado en un mapa lo devuelve.

**public void heads():**

Calcula las cabeceras de todos los antecedentes.

Por cada antecedente calcula su cabecera.

**public void head(String antecedent):**

Calcula la cabecera de un antecedente.

Si la cabecera no se ha calculado ya obtiene las producciones del antecedente e inicializa la lista donde va a guardar los símbolos de las cabeceras.

Por cada producción:

Se obtiene la primera letra, Si está en mayúscula:

Divide la producción en símbolos y si ya existe la cabecera prima del antecedente le añade los símbolos a la lista, si no calcula la cabecera prima y se los añade a la lista.

Si no divide la producción en símbolos y añade el primero a la lista

Añade la lista al mapa de cabeceras con el antecedente como índice.

**public void headPrime(String antecedent):**

Calcula la cabecera prima del antecedente.

Funciona igual que **head(String antecedent)**  pero no añade lambda a la lista.

**public void nexts():**

Calcula los siguientes de todos los antecedentes.

Por cada antecedente calcula los siguientes.

**private void ifNoRepeatInsert(ArrayList<String> oldElements,ArrayList<String> newElements ):**

Asigna los valores de la seguanda lista a la primera si esta no los contiene ya.

Por todos los nuevos elementos si la primera lista no los contiene los añade.

**public void next(String antecedente):**

Calcula los siguientes del antecedente.

Se inicializa la lista que va a contener los siguientes, si el antecedente es el axioma se añade “$” a la lista.

Se busca todas las producciones que contengan el antecedente, por cada producción:

Se divide la producción en símbolos, si a la derecha del antecedente en la producción no hay ningún símbolo y el antecedente de la producción no es igual al antecedente para el cual está calculando los siguientes:

Si se ha calculado el siguiente del antecedente de la producción se añaden a la lista los siguientes del antecedente de la producción.

Si no se calcula los siguientes del antecedente de la producción y se añaden si no están ya.

Si el elemento después del antecedente en la producción está en mayúscula:

Se calcula la posición del siguiente símbolo del antecedente en la producción, si la cabecera prima de dicho símbolo se ha calculado ya se insertan los símbolos en la lista si no están ya.

Si no se calcula y se añaden a la lista si no están ya.

Si el siguiente contiene lambda en sus producciones y el siguiente del siguiente existe:

Si el antecedente de la producción no es igual al antecedente para el cual estamos calculando los siguientes y los siguientes del antecedente de la producción se han calculado se añaden a la lista si no están ya.

Si no se calculan y se añaden si no existen ya.

Si el siguiente contiene landa:

Si el siguiente del siguiente empieza con mayúscula calcula los siguientes de este y los añade a la lista sin no están ya.

Si no añade el siguiente del siguiente a la lista si no está ya.

Si no se añade el siguiente a la lista si no está ya.

Por último, guarda la lista en el mapa de siguientes usando como índice el antecedente.

**public ArrayList<String> productionsWith(String noTerminal):**

Busca todas las producciones donde aparece el no terminal dado.

Por cada antecedente, por cada producción, si la producción contiene el no terminal la añaden a la lista que se va a devolver. Por último, se devuelve la lista con las producciones.

**private boolean containLambda(String symbol):**

comprueba si alguna de las producciones del símbolo dado es lambda.

Se obtiene las producciones del símbolo y por cada una si es lambda se devuelve true, si ninguna es landa se devuelve false.

**private boolean containLambda(ArrayList<String> symbols):**

Comprueba si la lista contiene lambda.

Por cada símbolo si uno es igual a lambda devuelve true, si ninguno es lambda devuelve false.

**private String getAntecedent(String production):**

Busca el antecedente de la producción dada.

Por cada antecedente si una de sus producciones contiene la producción que se ha dado devuelve el antecedente, sino devuelve null.

**private void groupTerminals():**

Busaca y almacena todos los terminales.

Por cada antecedente, por cada producción divide la producción en símbolos y por cada símbolo si no empieza por mayúscula se añade a la lista de terminales.

## Clase LL V0.2

**Atributos:**

Sustituye **private Integer[][] table** por **private HashMap<String, HashMap<String, Integer>> table** para permitir un acceso mas sencillo a los elementos.

**private HashMap<Integer, String>numRules:** guarda una regla según su posición en la gramática.

**private HashMap<String, Integer>indexRules:** guarda la posición en la gramática según la regla.

**private String entryChain:** representa la cadena de entrada a procesar.

**private HashMap<String, Double> values:** guarda el valor de las variables usando el nombre de la variable como índice.

**private Document doc:** documento donde se va a escribir el XML.

**private Integer ruleCount=1:** contador de reglas para poder añadirle el id a cada regla.

**private ArrayList<String> antecedentes:** lista de los antecedentes de la gramática.

**Constructor:**

Se inicializan la lista de antecedentes, la tabla, los mapas de posición de reglas y el mapa de valores.

**Métodos:**

**public void build():**

Construye el analizador descendente. Primero calcula las cabeceras, los siguientes de los antecesores y los directores. Después indexa las reglas según su posición y puebla la tabla que se va a utilizar.

**public void proccess(String entryChain):**

Dada una cadena de entrada la procesa y escribe el xml correspondiente.

Crea el documento xml y le añade como primera etiqueta raíz, a esta etiqueta le añade la etiqueta espec con un atributo con valor "Especificación del XML". Luego escribe la parte de la gramática y de la cadena sobre la etiqueta espec.

Cuando termina de escribir la parte de la cadena y la gramática empieza a procesar la cadena de entrada.

Inicializa la pila y le añade el símbolo de dólar y el axioma, luego inicializa una pila con el símbolo dólar y la cadena de entrada. Mientras la pila no esté vacía:

Si el primer elemento de la pila de la cadena es un digito desapilo el elemento y lo guardo en el mapa de valores con el índice “"num.vlex"”. Por último, añade a la pila de la cadena “num” que es el terminal correspondiente a los números.

Si la cabeza de la pila coincide con la de la pila de la cadena se sacan ambas de las pilas respectivas.

Si no:

Si la cabeza de la pila no empieza con “{” (no es una acción semántica):

Si la cabecera de la pila es lambda se elimina de la pila.

Si no:

Si no es un antecedente se saca de la gramática y se añade el mismo elemento sin la parte numérica.

Si lo es se obtiene el índice de la regla de la tabla y después la regla a través del índice. Por ultimo se añaden los símbolos de la regla a la pila.

Si es acción semántica, se desapila y se divide en nombre de variable donde se va a guardar y valor.

Si es una igualdad (asignación de un valor a una variable) se calcula la posición de la variable donde empiezan los dígitos si los tiene:

Si la posición es menor que la anchura de la variable donde se guardaría el valor se elimina la parte numérica.

se calcula la posición del valor donde empiezan los dígitos si los tiene y si es menor que el tamaño del valor se elimina la parte numérica.

Si la acción contiene algún signo de operación aritmética se calcula y se guarda en el mapa de valores con el nombre de la variable como índice.

Si no se busca en el mapa de valores el valor de la variable que se va a usar como valor y se guarda en el mapa de valores con el nombre de la variable como índice.

Por ultimo se guarda el documento en el archivo xml.

**readFile():**

Además, guarda los antecedentes sin repetirlos.

**public void poblationTable():**

Puebla la tabla que se va a usar para el análisis sintáctico.

Por cada regla de los directores se divide en antecedente y consecuente.

Por cada director de la regla si el antecedente ya tiene directores asociados en la tabla se añade a estos con la posición de la regla en la gramática como valor y el director como índice, si no, se añade a un mapa nuevo y se añade el mapa a la tabla con el antecedente como índice.

**private void indexingRules():**

Rellena los de mapas de posiciones de las reglas en la gramática.

Por cada antecedente , por cada regla guarda en un mapa el consecuente de la regla con su posición como índice y en otro la posición con la regla como índice.

**public Double calculateValue(String operation):**

Calcula la operación aritmética que represente la acción semántica dada.

Dependiendo del símbolo aritmético realiza una operación u otra y devuelve el resultado de la operación.

**private void writeTraductor(Element espec):**

Escribe en el documento del xml la parte de la gramática y la cadena.

Se crea la etiqueta “traductor” y se añade a la etiqueta espec. Se crea la etiqueta “tipo” y se añade a traductor, después se añade el texto “Descendente” a tipo. Por cada antecedente , por cada producción se añade la regla. Cuando se han añadido todas las reglas se crea la etiqueta “cadena” y se añade a espec, por ultimo se añade el texto de la cadena de entrada a cadena.

**private void addRule(String antecedent ,String production,Element traductor):**

Dado un antecedente , una producción y la etiqueta a la que añadir la regla , añade una regla al xml.

Forma el id de la regla añadiendo a “R” el contador de las reglas. Crea la etiqueta “regla” y la añade a la etiqueta traductor, después le añade un atributo “id” con el id y recoge las acciones semánticas de la regla. Por cada acción semántica:

crea una etiqueta “accionSemantica”, la añade a la etiqueta de regla, obtiene la posición de la acción semántica a la regla para crear un atributo ”pos” con el valor obtenido. Añade como texto la acción semántica y si la acción esta en el medio de la regla se añade una etiqueta “intermedio” con el texto sí.

Añade los símbolos de la producción. Y suma uno al contador de reglas.

**private ArrayList<String> actions(String production):**

Dada una producción devuelve todas las acciones en ella.

Separa la producción en símbolos y por cada símbolo si es una acción la añade a la lista que va a devolver.

**private Integer getPos(String action, String production):**

Dada una acción y una producción devuelve la posición detrás de la que se encuentra la acción.

Divide la producción en símbolos y mientras no sean igual a la acción si el símbolo no es una acción se suma uno a la posición y se pasa al siguiente símbolo.

**private void addSymbols(String production, Element regla,String antecedente):**

Dada una producción, un antecedente y la etiqueta regla escribe los símbolos de la producción en el xml.

Separa la producción en símbolos, por cada símbolo:

Si es el primero crea la etiqueta “simbolo” , la añade a regla , crea la etiqueta “valor” y la añade a símbolo. Si es la primera producción de la regla le añade a valor el antecedente si no añade “|”. Por último, crea la etiqueta “terminal”, la añade a símbolo y le añade el texto “false”.

Si el símbolo no es una acción crea la etiqueta “simbolo” , la añade a regla , crea la etiqueta “valor” y la añade a símbolo. Le añade a valor el símbolo. Crea la etiqueta “terminal”, la añade a símbolo y le añade el texto “false” si la primera letra es mayúscula y “true” si no.

## Clase LL V0.3

**Atributos:**

**private Integer numNodos:** Cuenta el número de nodos.

**private ArrayList<Node> nodes:** Almacenalos nodos.

**private Integer nivel:** Guarda el nivel en cada paso del procesamiento.

**private Integer altura:** Guarda la altura del árbol.

**private Integer paso:** Cuenta los pasos.

**private ArrayList<Paso> steps:** Guarda los pasos.

**private HashMap<String, Integer> relNodes:** Mapa que guarda por el elemento el id de los nodos.

**private HashMap<String,String> ruleId:** Mapa que guarda por la producción de una regla el id de la regla.

**private HashMap<String, String> idsRules:** Mapa que guarda por el id las reglas

**private HashMap<String, Stack<String>> symbolRules:** Mapa que guarda para cada símbolo a la regla que pertenece.

**Constructor:**

Se inicializa los mapas relNodos, ruleId, symbolRules y idsRules.

Se inicializa las listas nodes y steps.

**Métodos:**

**proccess(String entryChain):**

Se inicia el nivel a 1 despues de añadir el axioma a la pila y se inicializa la String de la cadena leída.

Mientras la pila no esté vacía.

Si el pico de la pila es $ significa que un nivel del árbol ha terminado y se resta uno al nivel y quito $ de la pila.

Se obtiene la cadena pendiente en forma de String.

Se añade a la condición de si el primer elemento de la cadena es digito si la cadena se ha terminado de procesar.

Si la cedena no se ha terminado de procesar y el pico de esta es igual al de la pila.

Sacar el elemento de la pila.

Si el elemento es num añado el valor a la cadena leída, lo elimino de la cadena pendiente y creo un paso con el contador de pasos, el tipo igual a despDes, la cadena leída, la cadena pendiente, el elemento, el valor del paso (el valor de num en este caso) y el nodo anterior a este.

Si no añado el elemento a la cadena leída quitándolo de la cadena pendiente y creo un paso con el contador de pasos, el tipo igual a despDes, la cadena leída, la cadena pendiente, el elemento y el nodo anterior a este.

Creo un nodo con el contador de nodos, el elemento, el valor terminal igual a true y el nivel del nodo en el árbol.

Sumo uno al contador, añado el nodo a la lista de nodos y el paso a la lista de pasos.

Si no

Si el pico de la pila no es una acción

Si el pico es lambda

Saco lambda de la pila, creo un nodo con el contador de nodos, el elemento, el valor terminal igual a true y el nivel del nodo en el árbol, aumento el contador de nodos, añado el nodo a la lista de nodos, recojo el id de la regla de este paso y creo un paso con el contador de pasos, el tipo igual a derivación, la cadena leída, la cadena pendiente, el elemento, el nodo anterior, el id de la regla y el tamaño de la regla, añado el paso a la lista de pasos y aumento el contador de pasos.

Si no es lambda

Si no es un antecedente( por ejemplo, C1 y no C) Saca el símbolo de la pila, lo modifico a su versión sin números y lo añado a la pila. Se obtiene el índice de la regla de la tabla y después la regla a través del índice. Se añade $ para representar el fin del nivel, se añade el símbolo de la regla a la pila, se añade al mapa de nodos relacionados por símbolo el número de paso, y añado el id de la regla a la que pertenece el símbolo al mapa. Aumento un nivel calculo la altura máxima, creo un nodo con el contador de nodos, el símbolo, el valor de terminal a false y el nivel (actual menos uno), aumento el contador de nodos y añado el nodo a la lista. Recojo id de la regla a la que pertenece el símbolo, recojo el valor del paso y creo un paso con el contador de pasos, el tipo igual a derivación, la cadena leída, la cadena pendiente, el símbolo, el nodo anterior, el id de la regla y el tamaño de la regla, añado el paso a la lista de pasos y aumento el contador de pasos.

Si lo es saca el símbolo de la pila, se obtiene el índice de la regla de la tabla y después la regla a través del índice. Se añade $ para representar el fin del nivel, se añade el símbolo de la regla a la pila, se añade al mapa de nodos relacionados por símbolo el número de paso, y añado el id de la regla a la que pertenece el símbolo al mapa. Aumento un nivel calculo la altura máxima, creo un nodo con el contador de nodos, el símbolo, el valor de terminal a false y el nivel (actual menos uno), aumento el contador de nodos y añado el nodo a la lista. Si el elemento es el axioma creo un paso con el contador de pasos, el tipo igual a primero, la cadena pendiente, el símbolo, el nodo anterior.Si no lo es recojo id de la regla a la que pertenece el símbolo, recojo el valor del paso y creo un paso con el contador de pasos, el tipo igual a derivación, la cadena leída, la cadena pendiente, el símbolo, el nodo anterior, el id de la regla y el tamaño de la regla, añado el paso a la lista de pasos y aumento el contador de pasos.

Si es una acción semántica recojo en una lista los paso que van a cambiar de valor y por cada paso añado al ultimo paso añadido a la lista de pasos el nodo que va a cambiar y el valor al que va a cambiar.

Escribo las partes de <árbol> y <contenido> al xml.

**public String removeOnlyActions(String production):**

Remueve solo las acciones semanticas de la producción. Por cada símbolo de la regla si no tiene “{” lo añade al resultado que va a devolver.

**addRule(String antecedent ,String production,Element traductor):**

añade al mapa de ruleId el id de la regla con la producción como clave y al mapa de IdsRules la regla con el id de la regla como clave.

**private void writeArbol(Element espec):**

Escribe la parte de <arbol> del xml.

Crea la etiqueta árbol y la añade al documento. Crea la etiqueta numNodos y la añade la etiqueta arbol con el número de nodos como texto. Crea la etiqueta altura y la añade la etiqueta arbol con la altura del árbol como texto. Por cada nodo en la lista de nodos:

Creo la etiqueta nodo y la añado a la etiqueta arbol con un atributo con el id del nodo. Creo la etiqueta elemento y la añado a la etiqueta nodo con el elemento del nodo como texto, creo la etiqueta nivel y la añado a la etiqueta nodo con el nivel del nodo como texto, creo la etiqueta terminal y la añado a la etiqueta nodo con true si es un terminal y false si no como texto.

**private String representChain(Stack<String> stackChain):**

Pasa la pila de la cadena a String. Por cada elemento si es num añade su valor si no añade el elemento.

**private ArrayList<Paso> findSteps(String changeValue, Paso pasoActual):**

Busca los pasos que hay que actualizar. Mientras el paso relacionado (paso padre) sea mayor que cero, si el valor del paso no es nulo y contiene el valor que ha cambiado se añade el paso a una lista y se cambierte el paso actual al paso padre.

**private void writeContenido(Element espec):**

Escribe la parte de <contenido> del xml.

Crea la etiqueta contenido y la añade al documento. Por cada paso de la lista pasos:

Crea la etiqueta paso y la añade a contenido con un atributo id. Crea la etiqueta tipo y la añade a contenido con el texto del tipo de paso, Si el paso tiene regla:

Crea la etiqueta nuevaRegla y la añade a paso con un atributo refRegla con el id de la regla y la regla como texto. Crea la etiqueta widthRegla y la añade a paso con el ancho de la regla como texto.

Crea la etiqueta cadena y la añade a paso. Crea la etiqueta leido y la añade a cadena con la cadena leído como texto. Crea la etiqueta pendiente y la añade a cadena con la cadena pendiente como texto. Crea la etiqueta elemento y la añade a paso con el elemento del paso como texto. Crea la etiqueta valor y la añade a paso con el valor del paso como texto. Si tiene nodos que cambien de valor por él:

Crea la etiqueta accionSemanticaEjecutada y la añade a paso. Por cada nodo que cambia de valor:

Crea la etiqueta nodo y la añade a accionSemanticaEjecutada. Crea la etiqueta refNodo y la añade a nodo con el id del node que cambia. Crea la etiqueta atributos y la añade a nodo con el valor al que cambia el nodo.

## Clase LL V0.4

**Atributos:**

**private HashMap<String, Stack<Integer>> values:**

El atributo de values cambia a un mapa de pilas de enteros para poder almacenar los valores de no terminales con recursividad por la derecha (C ::=…C1).

**Métodos:**

**proccess(String entryChain):**

Antes de empezar a procesar la cadena inicializa un contador para poder saber cuando sacar un valor dado a C1 de la pila de valores correspondiente, cuando se procesa un símbolo el contador vuelve a cero y cuando procesa una acción pasa a 1. Cada vez que va a introducir algo nuevo a la pila de valores comprueba si en el mapa ya hay una pila y si no la hay la crea. Cuando se va a introducir una acción se le añade el no terminal que la ha producido. Cuando se procesan las acciones se recoge el no terminal que lo ha producido y lo elimina de la acción. Las acciones intermedias no las escribe. Si el ultimo elemento procesado era lambda, el contador estaba a 0 y existe el atributo .valor se elimina de la pila.

**private String getValues(String symbol):**

Devuelve los valores de los atributos del símbolo dado. Por cada pila del mapa si existen atributos del símbolo y la pila correspondiente no esta vacía se añade al resultado la cima de la pila.

**findSteps(String changeValue, Paso pasoActual, String noTerminalWithNumber):**

Se añade un parámetro nuevo(**noTerminalWithNumber**) para distinguir si se tiene que actualizar un paso con un no terminal recursivo(por ejemplo C1).Se añade una condición mas , si el elemento del paso es igual a **noTerminalWithNumber** entoncesse añade al resultado

## Clase LL V0.5

**Atributos:**

**private HashMap<String, Stack<Integer>> relNodes:**

relNodes se convierte en un mapa de pilas para poder mantener varios símbolos iguales sin perder con que nodo están relacionados.

**proccess(String entryChain, String pathResult):**

El nuevo parámetro contendrá el path de destino del xml. Al inicializar la pila y la pila de la cadena se añade primero $ para mostrar cuando no hay mas elementos. Se usa un booleano para que si el axioma se usa en otro momento que no sea al principio se entienda como un no terminal cualquiera. Se cambia el símbolo de fin de nivel de $ a #. Cada vez que se ejecuta un paso el nodo relacionado se saca de la pila correspondiente.

**public ArrayList<String> productionsWith(String noTerminal):**

Se hace un bucle más para comprobar símbolo a símbolo.

## Class Node V0.1

**Atributos:**

**private Integer id:** id del nodo.

**private String element:** elemento del nodo.

**private Boolean terminal:** true si es terminal false si no.

**private Integer nivel:** nivel del nodo en el árbol.

**Constructor:**

Recoge el id, el elemento, si es terminal y el nivel y lo guarda en los atributos.

## Class Paso V0.1

**private Integer id:** id del paso.

**private String tipo:** tipo de paso (despDes, derivacion, primero).

**private String leído:** cadena leído.

**private String pendiente:** cadenapendiente.

**private String elemento:** elemento del paso.

**private String valor:** valordelpaso.

**private String regla:** regla atribuida al paso.

**private Integer widthRule:** ancho de la regla.

**private Integer relNodo:** nodo anterior o padre de este.

**private ArrayList<Integer> changedNodes:** ids de los nodos cuyo valor cambia debido a este paso.

**private ArrayList<String> changes:** valores a los que cambian los nodos de la lista anterior.

**Constructor:**

Recoge el id, el tipo, la cadena leída, la cadena pendiente, el elemento, el valor, la regla, el ancho de la regla y el nodo relacionado. Además inicializa la lista de nodos y valores que van a cambiar.