Reynaldo Vega A01114523

Analizador sintáctico, algoritmo CYK y Expresiones Aritméticas

SOFTWARE DESIGN DOCUMENT

21 de octubre de 2020

Contenido

Introducción

Propósito

El propósito, del siguiente documento, es explicar la funcionalidad, diseño e implementación que se hizo para lograr la finalidad del proyecto de la segunda unidad de la materia de matemáticas computacionales. El documento abarcará todos aquellos requisitos que se pidieron en el proyecto número dos y tres de la materia, el cual consiste en la implementación de un analizador sintáctico, que utiliza el algoritmo CYK para verificar si una cadena pertenece o no a una Gramática Libre de Contexto (al igual que expresiones aritméticas); de igual manera, se abarcarán temas relacionados con la Forma Normal de Chomsky, ya que el algoritmo CYK solo se puede aplicar una vez habiendo convertido una GLC a esta respectiva forma.

Gramáticas libres de contexto (GLC)

Las Gramáticas Libres de Contexto se pueden definir formalmente como una cuádrupla de la siguiente manera:

$$G = (V, T, P, S)$$

V- es un conjunto finito de símbolos no terminales

T- es un conjunto finito de símbolos terminales

P- es un conjunto finito de producciones, donde su forma es de la siguiente manera: $V \rightarrow (T \cup V) *$

S- es el símbolo inicia de la GLC y $S \in V$

Forma Normal de Chomsky (FNCh)

Una gramática está en la Forma Normal de Chomsky si todas sus producciones son de alguna de las siguientes formas:

$$A \rightarrow BC$$

$$A \rightarrow \propto$$

Donde A, B y C forman parte del conjunto de símbolos no terminales y $\propto$ pertenece al conjunto de los símbolos terminales

Algoritmo CYK

El algoritmo de Cocke Younger Kasami, también conocido como CYK, es un algoritmo de programación dinámica, el cual nos permite definir si una cadena es aceptada o no por una Gramática Libre de Contexto; además, si es aceptada, el mismo algoritmo nos permite obtener una posible solución de árbol de derivación que lleva al producto de la cadena que se quiere verificar.

Tabla de CYK

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	V, VP	Det.	N	P	Det	N
she	eats	a	fish	with	a	fork

Diseño e implementación

Entrada/Input

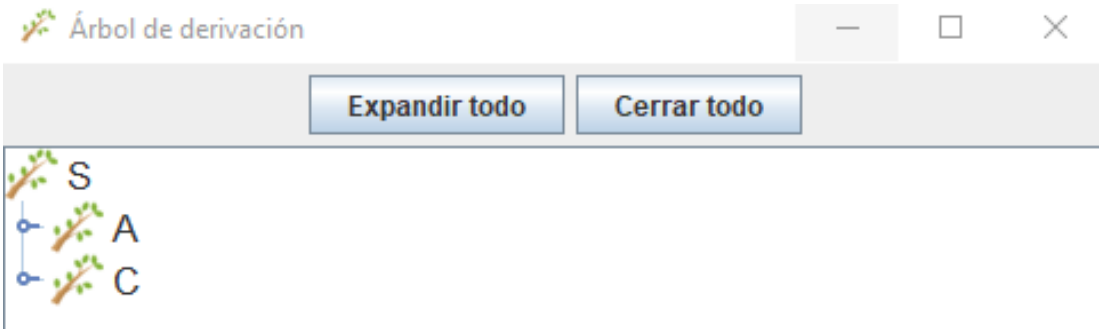
```
1 aabbab
2 S,A,B,C,D
3 a,b
4 S->AB,SS,AC,BD,BA
5 A->a
6 B->b
7 C->SB
8 D->SA
```

La entrada se compone por 4 componentes:

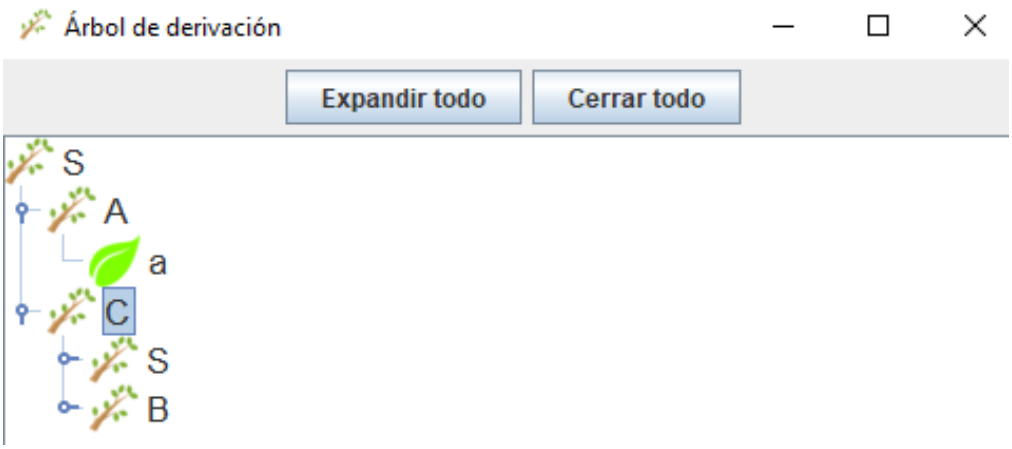
1. Cadena para probar
2. Conjunto de símbolos no terminales (separados por coma y solo pueden ser mayúsculas)
3. Conjunto de símbolos terminales (separados por coma)
4. Producciones de la manera V->AS,BC (donde la producciones deben estar separadas por coma)

Nota: las producciones se tienen que poner en el orden en el que se definió el conjunto de símbolos no terminales; además el primer símbolo definido es el símbolo inicial de la GLC. Las épsilon transiciones se representan con un “$”.

Output/GUI



Si se acepta la cadena por la GLC se desplegará este GUI con el árbol de derivación. En este ejemplo “S” es nuestro nodo raíz donde su hijo izquierdo es “A” y su hijo derecho es “C”.



El árbol se puede expandir si se realiza click en cada una de sus ramas, las cuales están representadas por una imagen de “rama”, si se encuentra una hoja, significa que es un nodo hoja y es el carácter que pertenece a la cadena, es decir, que, una vez expandido el árbol, la cadena terminal saldrá de forma vertical.

El árbol puede expandirse completamente sin necesidad de hacer click por cada rama hasta encontrar las hojas, para ello es solo necesario hacer click sobre el botón “Expandir todo” y si se quiere cerrar completamente se realiza click sobre “Cerrar todo”. En el ejemplo mostrado, la GLC aceptó la cadena, se muestra el árbol y la cadena aceptada es “aabbab”

Algoritmos

A continuación, se explicará detalladamente los algoritmos que se utilizaron para la implementación del proyecto del segundo parcial:

1. Eliminar símbolos no terminales que no generan productos terminales

```
public void eliminarProduccionesQueNoGeneranTerminales() {
    boolean prueba = false;
    LinkedHashSet<Character> N1 = new LinkedHashSet<Character>();
    LinkedHashSet<Character> N1USigma = new LinkedHashSet<Character>();
    N1USigma.addAll(this.simbolosTerminales);
    N1USigma.add('0');

    // Algoritmo para obtener N1
    while (true) {
        LinkedHashSet<Character> tmp = (LinkedHashSet<Character>) N1.clone();

        // Recorrer todas las producciones
        for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {

            // Recorrer lista de producciones
            for (int j = 0; j < me.getValue().size(); j++) {
                // Verificar si la producción pertenece a {N1 U Sigma}* donde Sigma es el
                // alfabeto de entrada (Símbolos no terminales inicialmente)
                for (int k = 0; k < me.getValue().get(j).length(); k++) {
                    if (!N1USigma.contains(me.getValue().get(j).charAt(k))) {
                        prueba = false;
                        break;
                    } else {
                        prueba = true;
                    }
                }

                if (prueba) {
                    N1.add(me.getKey());
                }
            }

        }
        if (tmp.containsAll(N1)) {
            break;
        } else {
            N1USigma.addAll(N1);
        }
    }
```

El algoritmo consiste en tener un conjunto llamado N1, el cual acumulará símbolos generadores que llevan a un producto de terminales. Para ello, N1 estará vacío, es decir, $N1 = \{\emptyset\}$, posteriormente se recorren todas las producciones de la gramática y por cada iteración se realiza $(N1 \cup \sum\ )$, donde Sigma pertenece al alfabeto de entrada de la GLC, una vez realizado este paso, se buscará si alguna de las producciones pertenece al conjunto $(N1 \cup \sum\ )$*, si alguno pertenece se agrega el símbolo no terminal al conjunto N1. El algoritmo para hasta no encontrar más símbolos generadores nuevos, lo que da como resultado un conjunto de símbolos que si llevan a cadenas terminales; por lo tanto, se obtiene el conjunto disjunto con respecto al conjunto de símbolos no terminales, para buscar cuales no generan y eliminarlos.

2. Eliminar símbolos no alcanzables

```
public void eliminarProduccionesNoAlcanzables() {
    // Grafo de dependencia
    LinkedHashMap<Character, ArrayList<String>> grafo = new LinkedHashMap<Character, ArrayList<String>>();

    // Generacion de grafo
    for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {
        grafo.put(me.getKey(), new ArrayList<String>());
        for (int j = 0; j < me.getValue().size(); j++) {
            for (int k = 0; k < me.getValue().get(j).length(); k++) {
                if (this.simbolosNoTerminales.contains(me.getValue().get(j).charAt(k))) {
                    if (!grafo.get(me.getKey()).contains(Character.toString(me.getValue().get(j).charAt(k)))) {
                        grafo.get(me.getKey()).add(Character.toString(me.getValue().get(j).charAt(k)));
                    }
                }
            }
        }
    }

    // BFS (Buscar nodos que son alcanzables)
    LinkedList<Character> queue = new LinkedList<Character>();
    LinkedHashSet<Character> visited = new LinkedHashSet<Character>();
    queue.add(this.simbolosNoTerminales.get(0));

    while (!queue.isEmpty()) {
        Character current = queue.poll();
        visited.add(current);
        ArrayList<String> vecinos = grafo.get(current);
        for (String m : vecinos) {
            if (!visited.contains(m.charAt(0))) {
                visited.add(m.charAt(0));
                queue.add(m.charAt(0));
            }
        }
    }
```

El algoritmo es un generador de grafos de dependencia, el cual recorre toda la Gramática Libre de Contexto para generarlo. Se recorre toda la GLC y se almacena en un hash set de listas de adyacencia, ya que esta es una manera de representar un grafo; después de generar el grafo, se realiza un Breadth-First-Search, para buscar todos aquellos símbolos no terminales que son alcanzados partiendo desde el símbolo inicial de la Gramática; una vez obtenido el conjunto de símbolo alcanzables, se obtiene el conjunto disjunto de los alcanzables con respecto al conjunto de símbolos no terminales, obteniendo por consecuencia todos aquellos que son inalcanzables y luego de esta manera poder eliminarlos.

3. Cerradura de producciones

```
public void generarCerraduraDeProducciones() {
    while (true) {
        LinkedHashMap<Character, ArrayList<String>> tmp = clone(this.producciones);

        // Checar si tiene Epsilon Producciones
        for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {
            for (int j = 0; j < me.getValue().size(); j++) {
                for (int k = 0; k < me.getValue().get(j).length(); k++) {
                    if (this.simbolosNoTerminales.contains(me.getValue().get(j).charAt(k))) {
                        if (this.checarSiTieneEpsilonProduccion(me.getValue().get(j).charAt(k))) {
                            String word = this.producciones.get(me.getKey()).get(j);
                            word = word.substring(0, k) + "" + word.substring(k + 1);
                            if (word.equals("")) {
                                word = "@";
                            }
                            if (!me.getValue().contains(word)) {
                                this.producciones.get(me.getKey()).add(word);
                            }
                        }
                    }
                }
            }
        }

        // Checar si tiene producciones unitarias
        for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {
            for (int j = 0; j < me.getValue().size(); j++) {
                if (me.getValue().get(j).length() == 1
                        && this.simbolosNoTerminales.contains(me.getValue().get(j).charAt(0))) {
                    if (this.checarSiTieneProduccionUnitaria(me.getValue().get(j))) {
                        if (!me.getValue().contains(this.simboloUnitario)) {
                            this.producciones.get(me.getKey()).add(this.simboloUnitario);
                        }
                    }
                }
            }
        }

        if (tmp.equals(this.producciones)) {
            break;
        }
    }
```

En este algoritmo, se recorren todas las producciones de la gramática, para buscar si esta contiene épsilon transiciones o unitarias. Para ello, por cada símbolo no terminal encontrado en una producción, se buscará si está lleva a una épsilon, en caso de encontrarse una, se agregará la producción resultante de la transición a la producción por donde se empezó a buscar; posteriormente, se buscan las producciones unitarias, donde solo se buscan producciones que se conforman de un solo símbolo no terminal, que llevan a una producción de un solo símbolo terminal, en caso de encontrarse uno, se agrega la producción de transición a la producción por donde se empezó a buscar; cabe destacar que, las épsilon transiciones se buscan a partir de producciones que contienen uno o más símbolos, mientras que las unitarias solo son por un símbolo no terminal. Una vez habiendo añadido las nuevas producciones a la gramática, se vuelve a recorrer, ya que se pueden generar más épsilon producciones por consecuencia de las nuevas añadidas, es decir, el algoritmo para hasta no encontrar más producciones, algo parecido al primer algoritmo del programa (N1). Una vez terminado el algoritmo, se borran todas aquellas producciones que se generaron y se aplica nuevamente el algoritmo de símbolos no alcanzables, ya que por eliminar símbolos se generan nuevos.

4. Convertir GLC a FNCh

```
public void convertirChomsky() {
    LinkedHashSet<Character> terminalesVisitados = new LinkedHashSet<Character>();

    // Recorremos y buscamos posibles simbolos terminales para reemplazar
    for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {
        for (int j = 0; j < me.getValue().size(); j++) {
            if(me.getValue().get(j).length() > 1) {
                for (int k = 0; k < me.getValue().get(j).length(); k++) {
                    if (this.simbolosTerminales.contains(me.getValue().get(j).charAt(k))) {
                        terminalesVisitados.add(me.getValue().get(j).charAt(k));
                    }
                }
            }
        }
    }

    // Añadimos nuevas producciones y vemos por cuales simbolos reemplazaremos los
    // simbolos terminales
    LinkedHashMap<Character, Character> reemplazador = new LinkedHashMap<Character, Character>();
    this.simbolosChomsky.removeAll(this.simbolosNoTerminales);

    for (Character ch : terminalesVisitados) {
        this.producciones.put(this.simbolosChomsky.get(0),     *
                new ArrayList<String>(Arrays.asList(Character.toString(ch))));
        reemplazador.put(ch, this.simbolosChomsky.get(0));
        this.simbolosChomsky.remove(0);
    }

    // Reemplazamos simbolos terminales por nuevos símbolos no terminales
    for (Entry<Character, ArrayList<String>> me : this.producciones.entrySet()) {
        for (int j = 0; j < me.getValue().size(); j++) {
            if (me.getValue().get(j).length() > 1) {
                for (int k = 0; k < me.getValue().get(j).length(); k++) {
                    if (reemplazador.containsKey(me.getValue().get(j).charAt(k))) {
                        String word = this.producciones.get(me.getKey()).get(j);
                        word = word.substring(0, k) + reemplazador.get(me.getValue().get(j).charAt(k))
                                + word.substring(k + 1);
                        this.producciones.get(me.getKey()).set(j, word);
                    }
                }
            }
        }
    }

   // Generamos más producciones para que las producciones actuales solo sean
   // concatenaciones de 2
   for (int i = 0; i < this.simbolosNoTerminales.size(); i++) {
       for (int j = 0; j < this.producciones.get(this.simbolosNoTerminales.get(i)).size(); j++) {
           while (this.producciones.get(this.simbolosNoTerminales.get(i)).get(j).length() > 2) {
               String word = this.producciones.get(this.simbolosNoTerminales.get(i)).get(j);
               word = word.substring(1, 3);
               this.producciones.put(this.simbolosChomsky.get(0), new ArrayList<String>(Arrays.asList(word)));

               word = this.producciones.get(this.simbolosNoTerminales.get(i)).get(j);
               word = word.substring(0, 1) + this.simbolosChomsky.get(0) + word.substring(3);
               this.producciones.get(this.simbolosNoTerminales.get(i)).set(j, word);
               this.simbolosChomsky.remove(0);
           }
       }
   }
}
```

El algoritmo, primero empieza buscando producciones de dos o más símbolos que contengan símbolos terminales, ya que estos son los que se pueden reemplazar por un símbolo no terminal; una vez obtenido el conjunto de los símbolos por reemplazarse, se generan las nuevas producciones y posteriormente se recorre toda la GLC, para reemplazar los símbolos terminales por sus nuevas producciones. De esta manera se obtiene una gramática con producciones de generadores y terminales, pero no mezcladas. Después, se recorre la GLC nuevamente, para buscar si existen producciones de 3 o más símbolos generadores, si se encuentra alguna, se generan nuevas producciones conteniendo dos símbolos generadores de la producción que se está manipulando y se sustituyen por las nuevas producciones, generando una FNCh.

5. Algoritmo CYK

```
public boolean aplicarAlgoritmo() {
    // Llenar primera linea
    for (int i = 0; i < this.word.length(); i++) {
        for (Entry<Character, LinkedHashSet<String>> me : this.p.entrySet()) {
            if (me.getValue().contains(Character.toString(this.word.charAt(i)))) {
                this.matriz.get(0).get(i).add(me.getKey());
            }
        }
    }

    // Algoritmo
    for (int i = 1; i < this.word.length(); i++) {
        for (int j = 0; j < this.matriz.get(i).size(); j++) {
            int w = j;
            int z = i;
            for (int k = 0; k < i; k++) {
                LinkedHashSet<Character> tmp = this.checarSiExisteProduccion(this.matriz.get(k).get(j),
                        this.matriz.get(--z).get(++w));
                this.matriz.get(i).get(j).addAll(tmp);
            }
        }
    }
    return this.matriz.get(word.length() - 1).get(0).contains(this.simbolosNoTerminales.get(0));
}
```

El algoritmo consiste en una matriz, la cual su primera fila será llenada por aquellos símbolos no terminales que generan los símbolos terminales de la cadena. Una vez rellenada la primera fila, se empezará a recorrer por casillas las matriz partiendo con dos índices, uno yendo de abajo hacia arriba con respecto a la casilla, y uno en diagonal de arriba hacia abajo; por cada iteración, se harán combinaciones con los símbolos no terminales encontrados en las casillas, si se encuentra algún símbolo no terminal que contenga una de las combinaciones, se agrega el símbolo no terminal en la casilla por donde partieron los índices, hasta rellenar la matriz completamente; cabe destacar que este es un algoritmo de programación dinámica y solo si la última casilla contiene el símbolo inicial, significa que la cadena si es aceptada por la GLC.

6. Generar recorrido del árbol

```
public void generarRecorridoArbol() {
    this.nodoActual.add(this.simbolosNoTerminales.get(0));

    LinkedList<Integer> i = new LinkedList<Integer>();
    LinkedList<Integer> j = new LinkedList<Integer>();

    i.add(word.length() - 1);
    j.add(0);

    while (!(i.isEmpty() && j.isEmpty())) {
        int indexI = i.poll();
        int indexJ = j.poll();
        int z = indexI;
        int w = indexJ;
        Character n = this.nodoActual.poll();
        this.recorrido.add(new ArrayList<Character>(Arrays.asList(n)));

        if (indexI == 0) {
            this.recorrido.get(this.indexArray++).add(this.word.charAt(indexJ));
        } else {
            for (int k = 0; k < indexI; k++) {
                if (this.checarCorrespondencia(this.matriz.get(k).get(indexJ), this.matriz.get(--z).get(++w), n)) {
                    i.add(k);
                    j.add(indexJ);
                    i.add(z);
                    j.add(w);
                    break;
                }
            }
        }
    }
    this.construirArbol();
}
```

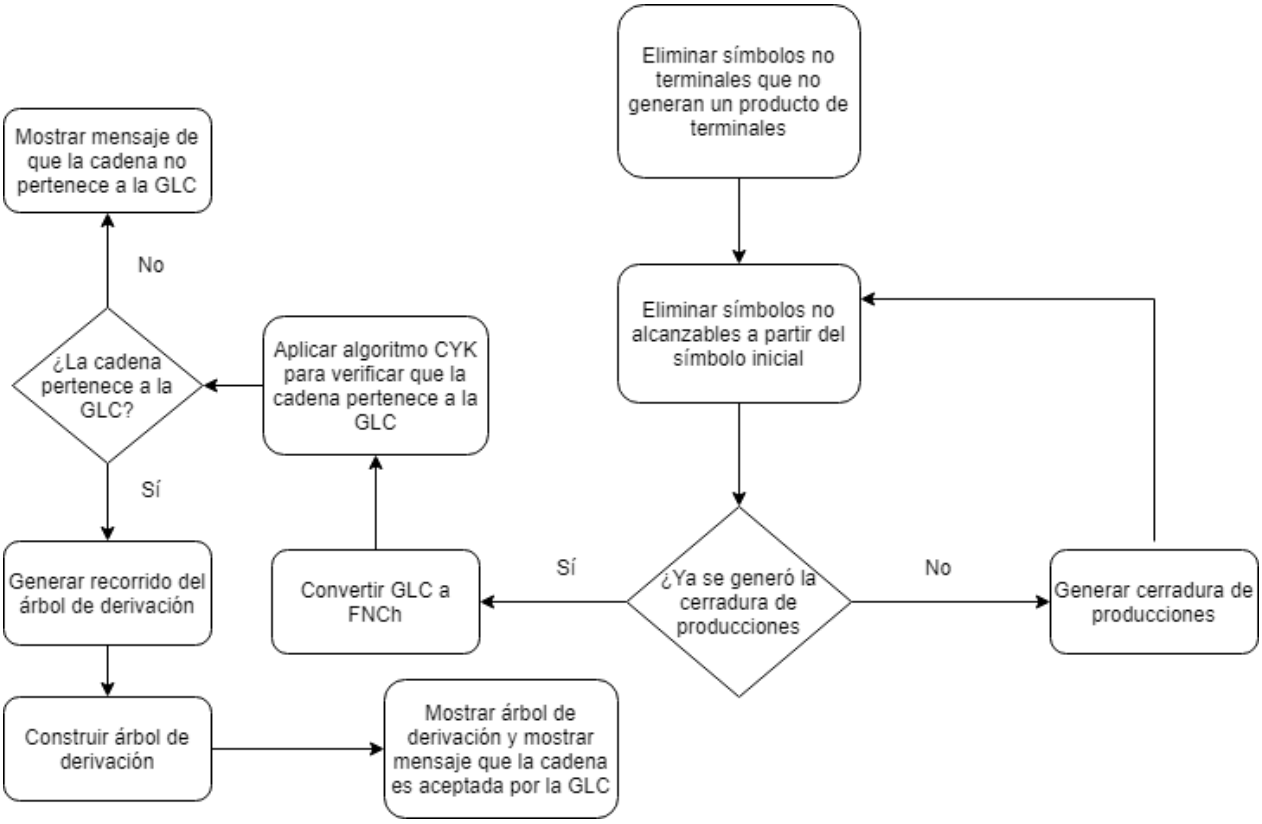
Una vez rellenada la matriz y habiendo verificado que la cadena es aceptada por la GLC, se recorre la matriz al revés, haciendo un back tracking con queues y verificando cuáles combinaciones corresponden a las casillas, de esta manera, se genera un recorrido por niveles, el cual sirve para la generación del árbol de derivación. Después de haberse generado el recorrido se manda a llamar el método que construye el árbol de derviación.

7. Generar árbol de derivación

```
public void construirArbol() {
    // 1era iteración
    DefaultMutableTreeNode parent = new DefaultMutableTreeNode(this.simbolosNoTerminales.get(0));
    DefaultMutableTreeNode c1 = new DefaultMutableTreeNode(this.recorrido.get(0).get(1));
    DefaultMutableTreeNode c2 = new DefaultMutableTreeNode(this.recorrido.get(0).get(2));

    DefaultTreeModel modelo = new DefaultTreeModel(parent);
    modelo.insertNodeInto(c1, parent, 0);
    modelo.insertNodeInto(c2, parent, 1);

    JTree tree = new JTree(modelo);
    tree.collapseRow(0);
    final Font currentFont = tree.getFont();
    final Font bigFont = new Font(currentFont.getName(), currentFont.getStyle(), currentFont.getSize() + 5);
    tree.setFont(bigFont);
    DefaultTreeCellRenderer render = (DefaultTreeCellRenderer) tree.getCellRenderer();
    render.setLeafIcon(new ImageIcon("leaf.png"));
    render.setOpenIcon(new ImageIcon("branch.png"));
    render.setClosedIcon(new ImageIcon("branch.png"));

    LinkedList<DefaultMutableTreeNode> parents = new LinkedList<DefaultMutableTreeNode>();

    parents.add(c1);
    parents.add(c2);

    while (!parents.isEmpty()) {
        for (int j = 1; j < this.recorrido.get(0).size(); j++) {
            for (int k = 1; k < this.recorrido.size(); k++) {
                if (this.recorrido.get(0).get(j) == this.recorrido.get(k).get(0)) {
                    if (this.recorrido.get(k).size() == 2) {
                        DefaultMutableTreeNode child = new DefaultMutableTreeNode(this.recorrido.get(k).get(1));
                        modelo.insertNodeInto(child, parents.get(0), 0);
                        parents.poll();
                        this.recorrido.remove(k);
                        break;
                    } else {
                        DefaultMutableTreeNode child1 = new DefaultMutableTreeNode(this.recorrido.get(k).get(1));
                        DefaultMutableTreeNode child2 = new DefaultMutableTreeNode(this.recorrido.get(k).get(2));
                        modelo.insertNodeInto(child1, parents.get(0), 0);
                        modelo.insertNodeInto(child2, parents.get(0), 1);
                        parents.poll();
                        parents.add(child1);
                        parents.add(child2);
                        break;
                    }
                }
            }
        }
        this.recorrido.remove(0);
        if (this.recorrido.isEmpty()) {
            break;
        }
    }

    JFrame v = new JFrame("Arbol de derivación");
    JScrollPane scroll = new JScrollPane(tree);
    v.setSize(new Dimension(500, 500));
    v.setIconImage(new ImageIcon("branch.png").getImage());
    v.getContentPane().add(scroll);
    v.setLocationRelativeTo(null);
    v.setVisible(true);
    v.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}
```

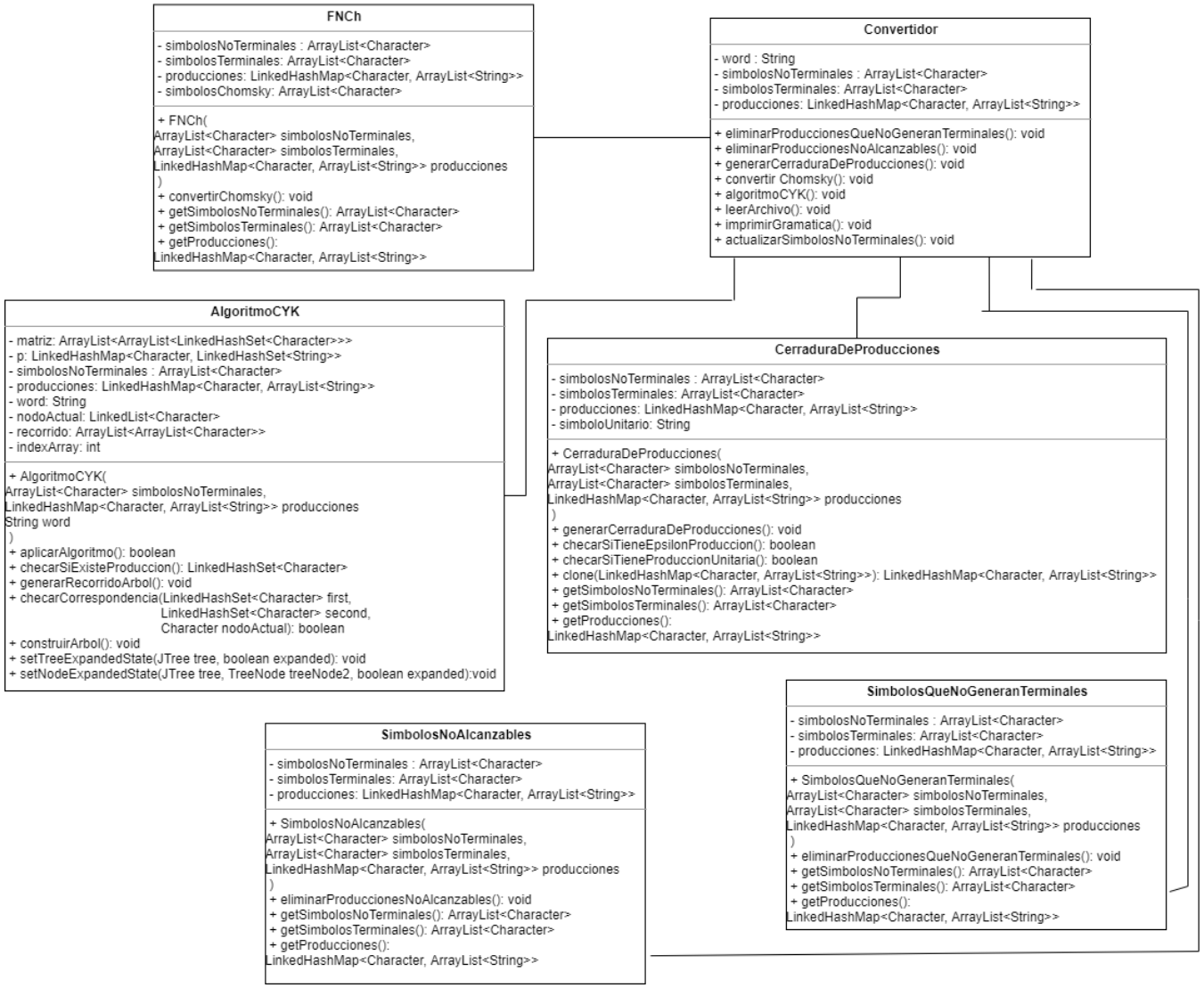
El agoritmo, recorre el recorrido del árbol por niveles, se generan nodos y se van a agregando por nivel a una estructura de datos llamado JTree, la cual permite crear árboles gráficamente con tal solo pasarse a una GUI. Más a detalle, el algoritmo recorre el arraylist recibiendo como primer índice el nodo padre y posteriormente sus nodos hijos, creando de esta manera los nodos, sus conexiones y finalmente añadiendolos al JTree.

Flujo del programa



1. Eliminar símbolos no terminales que no generen como producto una cadena de terminales.
2. Eliminar símbolos que no son alcanzables a partir del símbolo inicial de la GLC.
3. Generar cerradura de producciones, eliminar producciones épsilon y unitarias.
4. Eliminar nuevamente posibles símbolos no alcanzables por los símbolos eliminados por la cerradura de producciones.
5. Convertir GLC a FNCh
6. Aplicar algoritmo CYK para verificar que la cadena pertenece a la GLC.
    - Si se aceptó la cadena, generar recorrido del árbol de derivación, construirlo, mostrarlo en pantalla y mandar mensaje que se aceptó la cadena
    - Si no se aceptó la cadena, mostrar mensaje de que no se aceptó

Diagrama de clases



Las estructuras de las clases del programa están representadas en este diagrama; cabe destacar que todas las clases son manejadas por la clase convertidor, quien es el manejador de todas las clases, es decir, es la encargada de mandar a llamar y ejecutar los métodos que se encuentran dentro de ellas.

GLC de aritmética

$$S \rightarrow SS \mid S+S \mid S-S \mid S*S \mid (S) \mid -A \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$A \rightarrow AA \mid \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Con esta GLC se pueden representar y validar cadenas que pertenezcan al conjunto de expresiones aritméticas por los símbolos terminales “(”,”)”,”-”,”+”,”*”.