Segundo Proyecto: Lenguajes Libres de Contexto

Algoritmo CYK

Matemáticas Computacionales

Verónica Nataly Hernández Ochoa A01631314

Explicación YouTube: https://youtu.be/KA7Wtip7VPU

GitHub: https://github.com/nataly-8h/Analizador

Prof. Luis Ricardo Peña Llamas

Domingo 01 de noviembre del 2020

Índice

Descripción

Este proyecto consiste en la implementación de un analizador sintáctico utilizando el algoritmo CYK. Se recibirá una cadena y una gramática no necesariamente en la forma normal de Chomsky y el analizador determinará si la cadena es aceptada por la gramática o no. De ser aceptada utilizará el algoritmo CYK para crear un árbol de derivación.

He decidido implementar el proyecto utilizando el lenguaje java ya que es con la que estoy más familiarizada y además me basé en el paradigma de la programación orientada a objetos.

Utilicé las siguientes 4 clases para toda la implementación las serán descritas más adelante con mayor detalle:

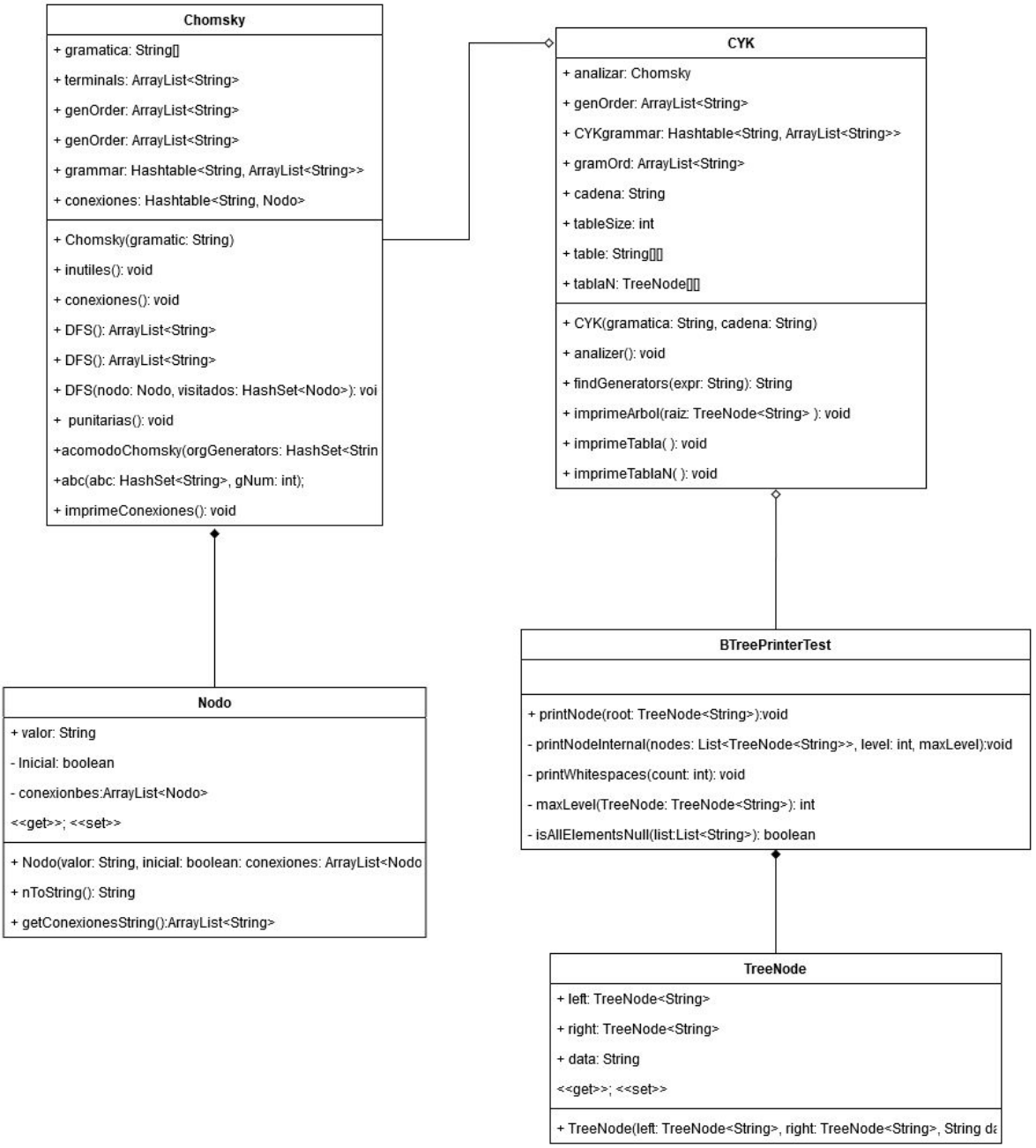
- CKY
- Chomsky
- Nodo
- BTreePrinterTest
- TreeNode

Entrada

La entrada serán dos Strings, uno que represente la cadena a analizar y otro representará la gramática. Ejemplo:

```
public static void main(String[] args) {
    String cadenaA =  "aabbab";
    CYK prueba = new CYK("S→AB|SS|AC|BD|BA A→a B→b C→SB D→SA", cadenaA);
}
```

Diagrama de clases



Descripción de clases

Las dos clases principales son CYK y Chomsky, son las que realizan todas o la mayoría de las tareas pedidas para este proyecto. La clase Nodo, TreeNode y BTreeTest son clases complementarias.

CYK

En esta clase sucede toda la implementación del algoritmo CYK, esta clase es donde se recibe la entrada mencionada anteriormente. Esta clase manda llamar tanto a la clase Chomsky como a la de BTreeTest. Dentro de la función analizer() es donde se implementa el algoritmo CYK de la siguiente manera:

Primero se ponen las orillas del arbol

```
for (int i = 0; i < table.length; i++) { //poner orillas del arbol
    this.table[i][i] =
findGenerators(Character.toString(this.cadena.charAt(i)));
    this.tablaN[i][i] = new TreeNode<>(null, null, this.table[i][i]); //
    this.tablaN[i][i].left = new TreeNode<>(null, null,
Character.toString(this.cadena.charAt(i)));
    //imprimeTablaN();
```

después de esta parte la tabla table queda de la siguiente manera:

Sucede lo mismo con la tabla N, que es donde se guardan los nodos que se utilizarán para el árbol de derivación.

Después se implementa todo el algoritmo CYK, donde se utiliza la siguientes variables principalmente:

- i : el apuntador de la casilla actual en i
- j : apuntador de la casilla actual en j
- bi : “busca i”, apuntador hacia la casilla que se va a buscar verticalmente
- bj : “busca j”, apuntador a la casilla que se va a buscar horizontalmente
- startj : esta indica hacia donde regresará el apuntador j para comenzar la nueva diagonal

```
//CYK
        int i = 0;
        int j = 1;

        int bi = 1; // buscar en i
        int bj = 0; // buscar en j

        int startj = 1; //donde comienza la diagonal
```

```
        while(startj < this.tableSize) {
            ArrayList<String> find = new ArrayList<String>();
            String found = "";
            TreeNode nodoi = null;
            TreeNode nodoj = null;
            while(bj < j) { //busqueda de generadores en la tabla
                String searchi = this.table[i][bj];
                String searchj = this.table[bi][j];
                String encuentraG = findGenerators(searchi + searchj);
                if(!find.contains(encuentraG)) {
                    find.add(encuentraG);
                    if(!encuentraG.equals("Ø")) {
                        nodoi = this.tablaN[i][bj];
                        nodoj = this.tablaN[bi][j];

                        this.tablaN[i][j] = new TreeNode<String>(nodoi,
nodoj, encuentraG);
                    }
                }
                bj++;
                bi++;
                //imprimeTablaN();
            }
            bj = i;
            bi = i +1;


            if(find.size() == 1) {
                found = found + find.get(0);
            }else {
                for (int k = 0; k < find.size(); k++) {
                    if(!find.get(k).equals("Ø")) {
                        found = found + find.get(k);
                    }
                }
            }

            this.table[i][j] = found;
            found = "";
            find.clear();
```

```
            if(j == this.table.length -1) {//cuando llega al final de la
diagonal vuelve a start diagonal
                startj ++;
                i = 0;
                j = startj;

                bi = 1;
                bj = 0;

            }else {
                i++;
                j++;
            }
        }
```

Cuando se llena la tabla, se evalúa si la cadena es aceptada por la gramatica dependiendo del valor que esté en la casilla de la esquina superior derecha al terminal el algoritmo CYK

```
 if(this.table[0][this.tableSize -1].equals("S")) {
            System.out.println("La cadena " + this.cadena + " pertenece a la
gramatica");
            imprimeArbol(this.tablaN[0][this.tableSize -1]);
        }else {
            System.out.println("La cadena " + this.cadena + " NO pertenece a
la gramatica");
        }for (int i = 0; i < table.length; i++) { //poner orillas del arbol
    this.table[i][i] =
findGenerators(Character.toString(this.cadena.charAt(i)));
    this.tablaN[i][i] = new TreeNode<>(null, null, this.table[i][i]); //
    this.tablaN[i][i].left = new TreeNode<>(null, null,
Character.toString(this.cadena.charAt(i)));
    //imprimeTablaN();
```

De pertenecer a la cadena, se manda a llamar a la clase BTreeTest para dibujar el arbol de derivación.

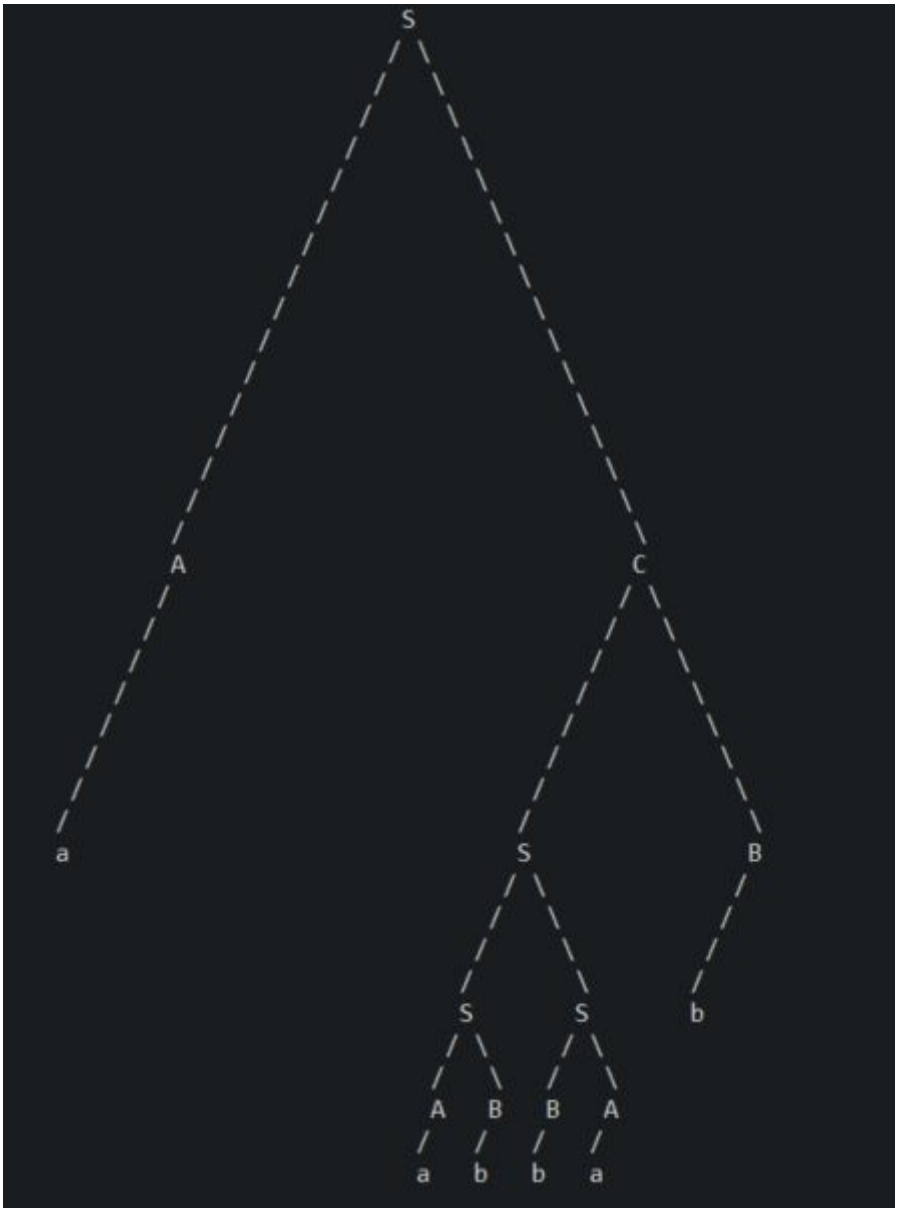
Ejemplo:

Para la gramatica S→AB|SS|AC|BD|BA A→a B→b C→SB D→SA

Si se ingresa la cadena aabbab la tabla utilizada en el algoritmo CYK quedaría de la siguiente manera:

```
[A, Ø, Ø, S, D, S]
[ , A, S, C, S, C]
[ ,  , B, Ø, Ø, Ø]
[ ,  ,  , B, S, C]
[ ,  ,  ,  , A, S]
[ ,  ,  ,  ,  , B]
```

La cadena es aceptada y el árbol de derivación luce de la siguiente manera:



Chomsky

Esta clase es la encargada de modificar la gramática recibida a la Forma Normal de Chomsky. Esta clase es capaz de recibir un String que representa la gramática, de la siguiente manera: “S→a|aS|A A→ab” todos los generadores deben ser representados con mayúsculas y tener el símbolo → después. Además debe de haber un espacio entre cada

generador. En el caso de que exista un generador X que no está definido, se descartará como un generador inútil en el momento de procesar los datos, ya que no va a pertenecer a los generadores, ejemplo: “S→a|X|SS”

Dos de los atributos más importantes son la HashTable grammar, así como el ArrayList genOrder. genOrder solo la utilicé para referirme a los generadores por orden, ya que más facil la comprensión del orden en el que se analizan o cambian. La HashTable grammar es la que guarda cada generador como key y dentro guarda un Arraylist que contiene cada expresión de dicho generador. Si se hace un print grammar se ve algo así:

```
{A=[a], S=[AB, SS, AC, BD, BA], D=[SA], C=[SB], B=[b]}
```

Como se muestra en la imagen, al crear un objeto Chomsky, se mandan a llamar las funciones en el orden de pasos para transformación a FNCh vistos en clase:

```
//Pasos para acomodo Chomsky:
inutiles();
conexiones();
punitarias();
acomodoChomsky(orgGenerators);
```

Considero que una de las partes más importantes de la transformación es la de la eliminación de generaciones inútiles, la cual fue implementada de la siguiente manera

Primera parte: se establece el alfabeto en el arreglo Ns, que funciona como ∑

```
ArrayList<String> N1 = new ArrayList<String>(); // N1 = ø
ArrayList<String> Ns = new ArrayList<String>(); // Σ

for (int i = 0; i < this.terminals.size(); i++) {  // agregar los valores de
Σ
    Ns.add(this.terminals.get(i));
}
```

Segunda parte, en un ciclo while se van agregando las generaciones que contienen lo que está dentro de Ns
El while se detiene cuando no hay cambios en el tamaño de Ns.

```
 int prevSize = 0;
int currentSize = Ns.size();

while(currentSize != prevSize) {
    for(int i = 0; i < this.genOrder.size(); i++) { //cada generador
        if(!Ns.contains(this.genOrder.get(i))) {
            for (int j = 0; j <
this.grammar.get(this.genOrder.get(i)).size(); j++) { //j para ir a cada
```

```
elemento de la lista
                String[] expre =
this.grammar.get(this.genOrder.get(i)).get(j).split(""); // hacer lista de
cada elemento de una expresion generada
                String finexpresion = "";
                for (int k = 0; k < expre.length; k++) {
                    if(Ns.contains(expre[k])) {
                        finexpresion = finexpresion + expre[k];

if(finexpresion.equals(this.grammar.get(this.genOrder.get(i)).get(j))) {
//Si al terminar la expresion es igual a toda la expresion, agregar el
generador
                            if(!N1.contains(this.genOrder.get(i)) &&
!Ns.contains(this.genOrder.get(i))) {
                                N1.add(this.genOrder.get(i));
                            }
                        }else {
                            if(k == expre.length -1) {
                                finexpresion = "";
                            }
                        }
                    }else {
                    }
                }
            }
        }else {

        }
    }

    prevSize = Ns.size();
    for (int i = 0; i < N1.size(); i++) {
        if(!Ns.contains(N1.get(i))) {
            Ns.add(N1.get(i));
        }
    }
    currentSize = Ns.size();
}
```

posteriormente se utiliza la clase Nodo para generar un grafo y verificar los generadores que son alcanzables o no por el nodo inicial.

En la clase conexiones() se crea todo el grafo que se utilizará para determinar los nodos alcanzables por el nodo inicial. Se guarda cada nodo en una HashTable y se hace una búsqueda depth first search DFS() que regresa un ArrayList con cada nodo alcanzable por el nodo inicial.

Nodo

Contiene los siguientes atributos:

- `String valor` - funciona para obtener el valor del nodo
- `private boolean Inicial` - determina si el nodo es el nodo inicial o no
- `private ArrayList<Nodo> conexiones` - una lista de todos los nodos a los que está conectado

BTreeTest

La función de esta clase es de imprimir el árbol de derivación de una forma que se pueda comprender visualmente. Esta clase fue creada por el usuario **michal.kreuzmann** en Stack Overflow, no desarrollaré en su implementación ya que no cambié casi nada.

Las únicas modificaciones que se hicieron en esta clase fue que el nodo sea tipo TreeNode<String> en vez de Node<T>.
Referir a la siguiente página para ver código original:
https://stackoverflow.com/questions/4965335/how-to-print-binary-tree-diagram/8948691

TreeNode

Esta clase es la que crea los nodos TreeNode<String> que se utilizaron para imprimir el árbol de derivación. Cada nodo contiene 3 atributos:

- `TreeNode<String> left` - el nodo que se encuentra a la izquierda
- `TreeNode<String> right` - el nodo que se encuentra a la derecha
- `String data` - el valor que se mostrará en el árbol