Algoritmo de Decisión CYK

Angelo De Nadai 2023

INDICE

INTRODUCCIÓN

El objetivo principal de este proyecto ha sido implementar el "Algoritmo de Decisión CYK" para determinar si una palabra pertenece a un lenguaje generado por una gramática independiente del contexto

DECISIONES DE DISEÑO

- En este proyecto se emplean tres estructuras de datos.
    - Los elementos no terminales y terminales se guardan en un ArrayList de caracteres para aprovechar la versatilidad de los arreglos de tamaño variable y mantener el control sobre la posición de los datos. Si se desea, también sería posible implementar esto con LinkedList.
    - Las producciones se almacenan en un HashMap, donde la clave es un Character y el valor es un ArrayList de cadenas. Esto permite guardar el elemento no terminal como clave y las combinaciones de elementos no terminales o el elemento terminal único como cadenas en la lista de arreglos.

-Otros elementos presentes en la clase son:

- El startSymbol, representado como un Character único, ya que debe estar representado por un único elemento no terminal.
- La tabla donde se guarda el proceso está representada por una matriz bidimensional de Strings. La primera dimensión representa las filas y la segunda las columnas y los Strings se interpretan como Characters no terminales concatenados entre sí para evitar manejar el tamaño fijo de una matriz y hacer que el código sea más legible que implementar una matriz bidimensional de ArrayLists.

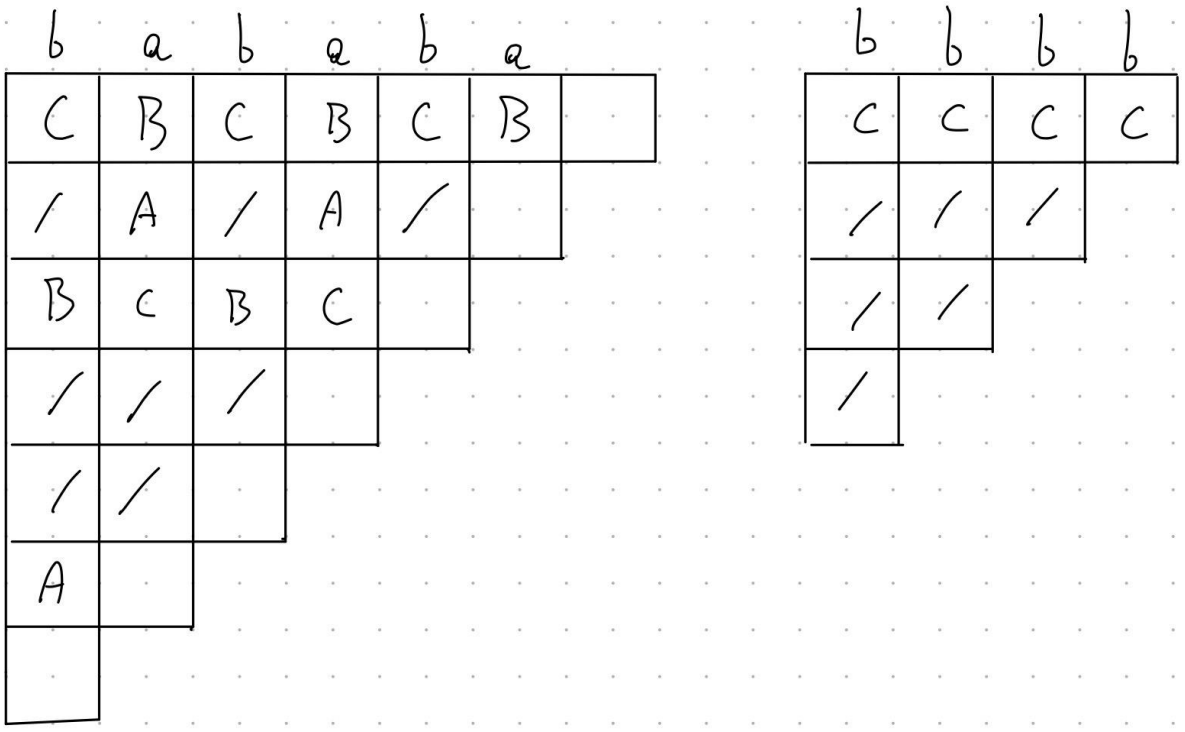
El acceso a todos los atributos es privado, solo son modificados por métodos de clase.

DESCRIPCIÓN DE TESTS REALIZADOS

Gramatica 1, resultado del método **algorithmStateToString**:

```
G=({a,b},{A,B,C},A,P)
P={
A::=BC
B::=a|CA
C::=AB|b
}
```

Desarrollo gráfico del algoritmo:



El algoritmo de verificación de palabras comienza con el método "**isDerived**"
Al inicio se realizan las comprobaciones de consistencia de los datos, luego se crea la matriz delegando la tarea al método **createMatrix**(), finalmente se toma el resultado que estará presente en la última celda de la tabla y se verifica que el “startSymbol " es contenido, de lo contrario, el método devuelve falso

```
public boolean isDerived(String word) throws CYKAlgorithmException {
    if (productions.isEmpty() || terminals.isEmpty() || startSymbol == null) //If there are no productions or terminals or startSymbol is equal to null
    {
        throw new CYKAlgorithmException();
    }

    for (char c : word.toCharArray()) {
        if (!terminals.contains(o: c)) //If the string doesn't contain only terminals( of the grammar)
        {
            throw new CYKAlgorithmException();
        }
    }
    createMatrix(word);

    //I take the string at the top of the triangle
    String result = table[table[0].length - 1][0];

    if (result.contains(s: Character.toString(c: startSymbol))) {
        return true;
    } else {
        return false;
    }
}
```

El método "**createMatrix**" se divide en tres partes:

- Inicialización de la tabla.
- Relleno de la primera línea de la tabla mediante el cálculo de los caracteres individuales de la palabra.
- Llenado de las celdas restantes de la tabla mediante un algoritmo que sigue los siguientes pasos:
    - Extracción de la columna de cadenas "sobre" la celda a calcular utilizando el método "**getColumn**".
    - Extracción de las celdas de la diagonal mediante el método "**getDiagonal**".

.

```
/**
 * create and fill the matrix
 *
 * @param word
 */
private void createMatrix(String word) {
    int length = word.length();
    table = new String[length][length];
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < length; j++) {
            table[i][j] = "";
        }
    }

    //fill the first row
    for (int i = 0; i < length; i++) {
        for (Map.Entry<Character, List<String>> entry : productions.entrySet()) //scroll through productions in search of the letter
        {
            if (entry.getValue().contains(o: String.valueOf(c: word.charAt(index: i)))) {
                table[0][i] += "" + entry.getKey();
            }
        }
    }

    //start the algorithm
    for (int i = 1; i < length; i++) {
        for (int j = 0; j < length - i; j++) {
            table[i][j] = getCombinationsResult(array1: getColumn(i, j), array2: getDiagonal(i, j));
        }
    }
}
```

Como se mencionó anteriormente, los métodos "**getDiagonal**" y "**getColumn**" se encargan de extraer las celdas relevantes necesarias para calcular la celda actual de la tabla. Estas celdas extraídas posteriormente se utilizan en el método "getCombinationsResult":

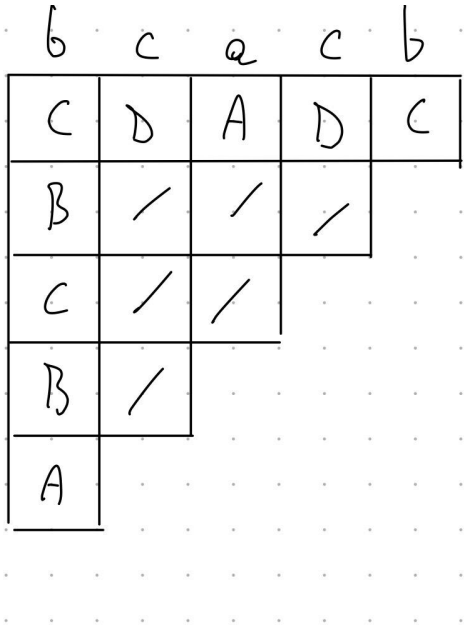
```
/**
 * select the column of elements to compare in the algorithm
 *
 * @param i current row
 * @param j current column
 * @return string array with the elements to compare in the algorithm
 */
private String[] getDiagonal(int i, int j) {
    String[] array = new String[i];
    int k = 0;
    while (i != 0) {
        i--;
        j++;
        array[k] = table[i][j];
        k++;
    }
    return array;
}

/**
 * select the row of elements to compare in the algorithm
 *
 * @param i current row
 * @param j current column
 * @return string array with the elements to compare in the algorithm
 */
private String[] getColumn(int i, int j) {
    String[] array = new String[i];
    int k = 0;
    while (k != i) {
        array[k] = table[k][j];
        k++;
    }
    return array;
}
```

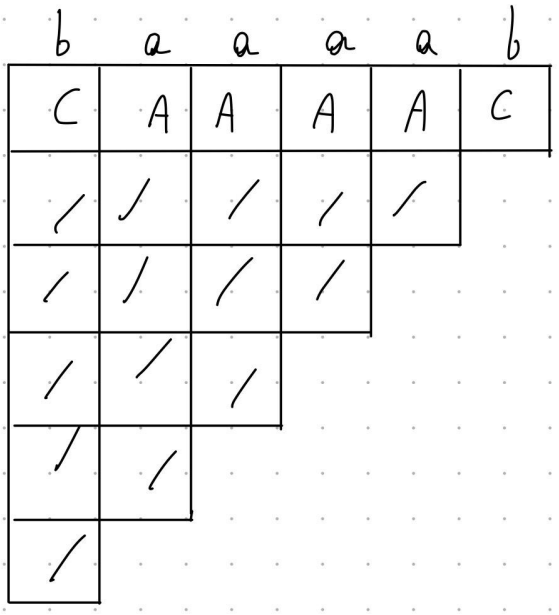
En este método se recorren dos arrays de cadenas obtenidos de los métodos anteriores. Se utiliza el índice "i" para recorrer todas las celdas de los dos arrays. Luego, los dos bucles "for" con los índices J y K apuntarán respectivamente a los caracteres de la primera cadena y la segunda cadena, generando todas las posibles combinaciones. Con el último bucle "for", se verifica la presencia de estas combinaciones en el HashMap "productions" y, en caso afirmativo, se agregará el valor clave a la cadena que se devolverá del método.

```
/**
 * calculates the combinations between the cells of the two arrays and
 * checks that they are valid
 *
 * @param array1
 * @param array2
 * @return Result string of the calculation with the cyk algorithm
 */
private String getCombinationsResult(String[] array1, String[] array2) {
    String result = "";

    for (int i = 0; i < array1.length; i++) {   // iterate through the two arrays using the same index i

        for (int j = 0; j < array1[i].length(); j++) {  //first string
            for (int k = 0; k < array2[i].length(); k++) {  //second string
                String s = "" + array1[i].charAt(index: j) + array2[i].charAt(index: k);
                if (isStringContained(searchValue:s)) {
                    for (Map.Entry<Character, List<String>> entry : productions.entrySet()) {
                        if (entry.getValue().contains(o: s) && !isCharacterPresent(str:result, ch: entry.getKey())) {
                            result += "" + entry.getKey();
                        }
                    }
                }
            }
        }
    }
    return result;
}
```

Gramatica 2, resultado del método algorithmStateToString:

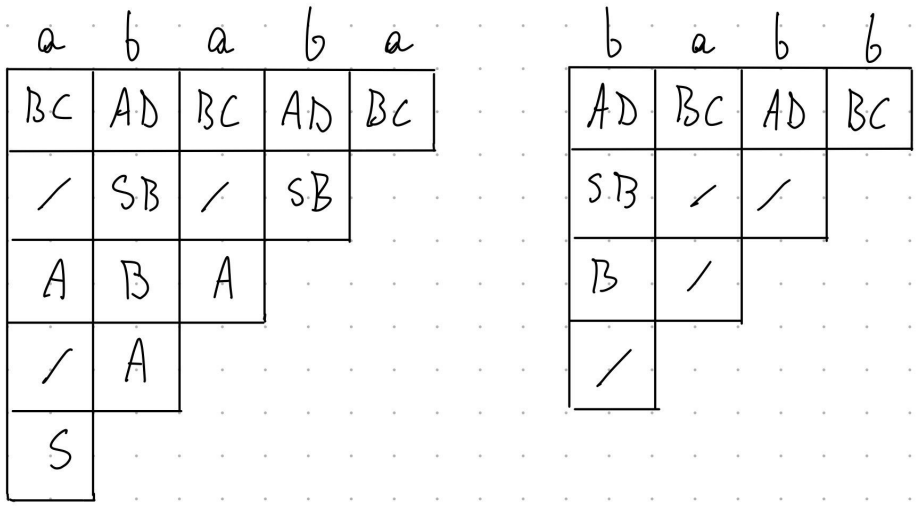
```
G=({a,c,b},{A,B,C,D},A,P)
P={
A::=BC|a
B::=CD
C::=b|BA
D::=c
}
```



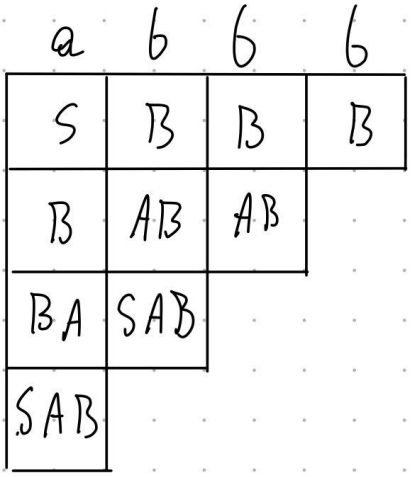


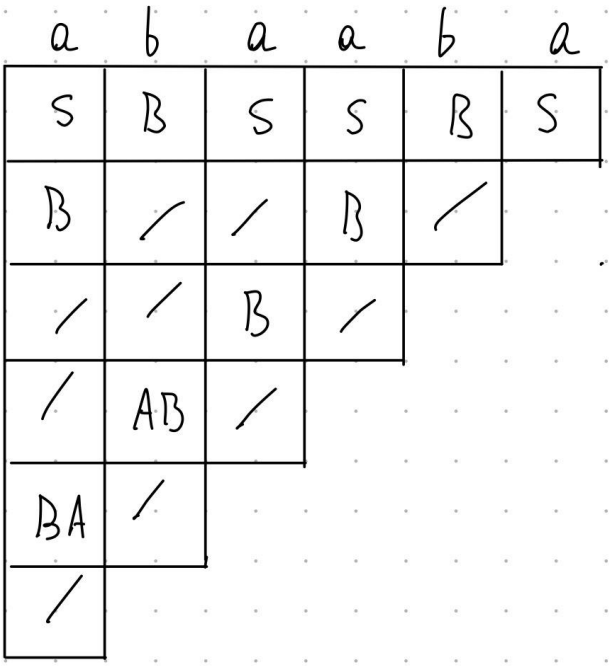
Gramatica 3, resultado del método algorithmStateToString:

```
G=({a,b},{S,C,A,B,D},S,P)
P={
A::=BS|b
B::=DC|SA|a
S::=AB
C::=a
D::=b
}
'ababa' IsDerived: true
BC AD BC AD BC
   SB    SB
A  B  A
   A
S
```



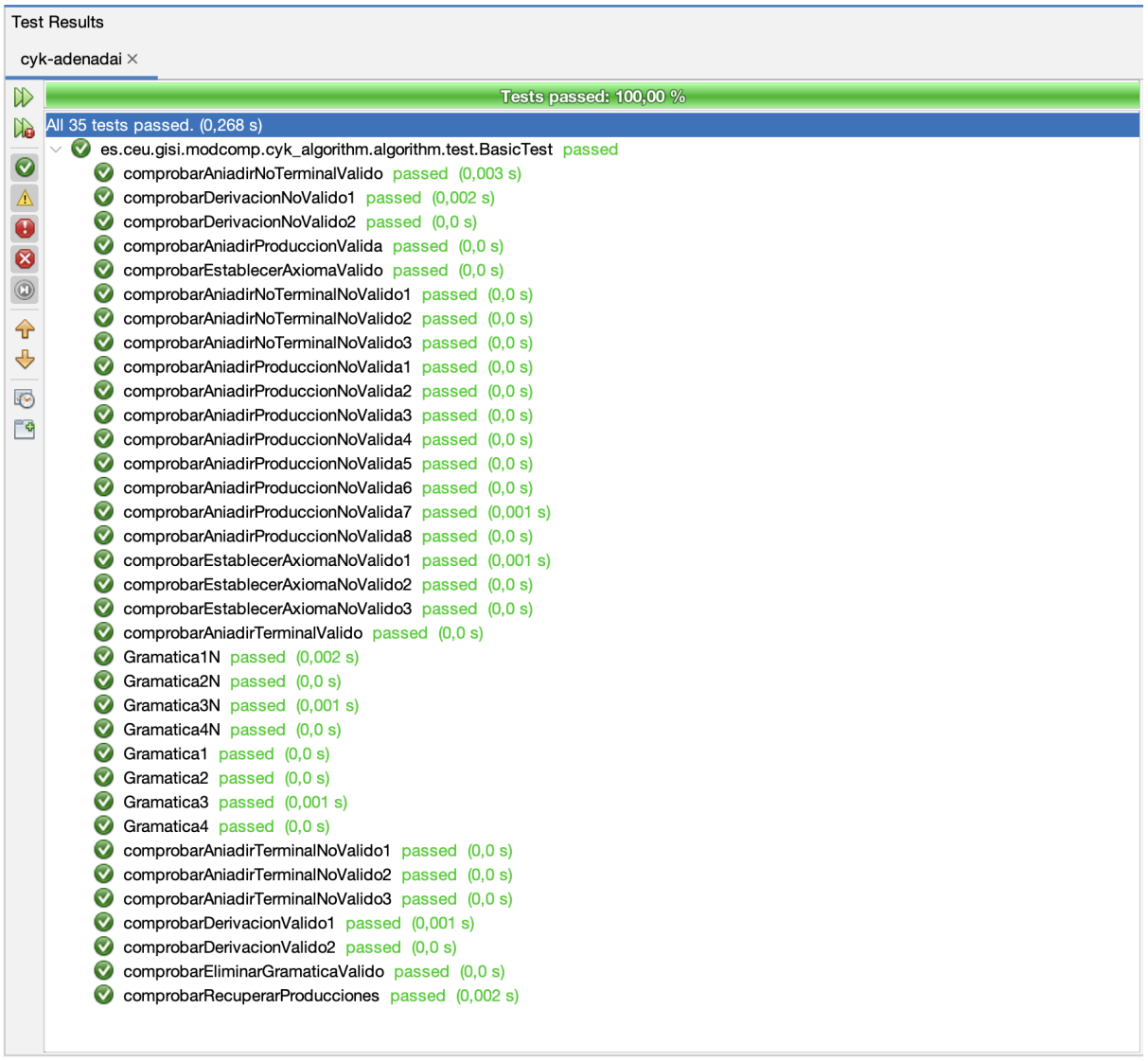
Gramatica 4, resultado del método algorithmStateToString:





TESTS DEFINIDOS

Las pruebas del proyecto se ejecutan correctamente:



CONCLUSIONES

Este proyecto me ha involucrado durante varias horas a lo largo de dos semanas, para un total de aproximadamente 20 horas. Sin duda, se podría mejorar el código implementando métodos más eficientes, pero solo mejoraría unos pocos puntos porcentuales. Estoy en general satisfecho con mi trabajo y entusiasmado de haber programado en un entorno muy controlado, donde mis implementaciones debían basarse en las que ya habían sido realizadas por otra persona, como ocurre en nuestro sector

Enlace del proyecto:https://github.com/Angelo-De-Nadai/cyk-adenadai