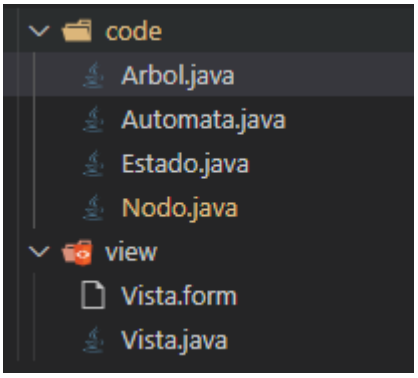
Documento de especificaciones segunda práctica de teoría de lenguajes 2022-1

JUAN ESTEBAN SALAS FLÓREZ
CRISTIAN ALEXANDER CASTAÑO MONTOYA

En este documento se plasman las **principales especificaciones** que fueron tenidas en cuenta para el desarrollo de la práctica número dos del curso teoría de lenguajes, el principal objetivo de esta práctica es convertir una expresión regular en un Automata Finito Determinístico por medio de un programa de computador y poder verificar una cadena y verificar que la ER sea correcta.

Para la elaboración se utilizaron 5 archivos .java, uno de ellos para la vista (vista.java) y los otros 4 con la lógica.



El **Nodo.java** es empleado conjuntamente con el Arbol.java. Se emplea para ayudar a construir el árbol, guarda cuál es el nodo izquierdo y derecho, además guarda un nombre que sería el carácter de la expresión (en caso de que sea necesario) o lambda.

```java
public class Nodo {
    Nodo hijo_izquierdo = null;
    Nodo hijo_derecho = null;
    char nombre;
    int posicion;
    Set<Integer> primera_posicion, ultima_posicion;
    boolean anulable = true;
    int n_hijos = 0;
    int indice = 0;

    public Nodo(char nombre){
        this.nombre = nombre;
        this.primera_posicion = new HashSet<>();
        this.ultima_posicion = new HashSet<>();
    }
```

Nota: Los demás métodos de esta clase son sencillos y se usan para darle mejor usabilidad a la clase nodo.

El **Arbol.java** es el encargado de construir la expresión regular, para poderla convertir en un AFD, con ayuda de la clase nodo. Este funciona cómo una lista ligada, crea un nodo, este se liga a otro y así sucesivamente.

```
public class Arbol {
    String expresion;
    Nodo raiz;
    Set<Integer> siguiente_posicion[];
    Hashtable<Character, Set<Integer>> pos = null;
    int n_caracteres = 0;
    int indice = 0;

    public Arbol(String expresion){
        this.expresion = "(" + expresion + ")#";
        this.raiz = new Nodo(nombre: '.');
        this.pos = new Hashtable<>();
        this.indice = 0;
        this.iniciar_posicion();
    }
```

Uno de los métodos importantes es el de **normalizar_expresion()**, si el usuario ingresa “ab”, la expresión queda a.b, cómo debe ser.

Para hacer uso de la expresión, la leemos en prefijo (cómo el usuario la ingresa) pero la pasamos posfijo, esto lo hacemos con el método **transformar()**
normalizar para que quede bien escrita

Una vez en posfijo, se emplea el método **arbol_de_posfijo(String posfijo)** el cual va recorriendo cada símbolo de la expresión, y de acuerdo al símbolo, si no es un operador, sino un carácter, genero un nodo y lo apilo en la pila, si es un símbolo operador desapilo y dependiendo de cuál es, hago la conexión de nodos respectiva.

```java
// Creación del arbol a partir de la expresion en posfijo
private void arbol_de_posfijo(String posfijo){
    Stack<Nodo> pila = new Stack<>();
    Nodo actual, hijo_derecho, hijo_izquierdo;
    int cnt = 1;
    for(int i = 0; i < posfijo.length(); i++){
        char simbolo = posfijo.charAt(i);
        if(!es_operador(simbolo)){
            actual = new Nodo(simbolo);
            if(simbolo != '&') actual.posicion = cnt++;
            pila.push(actual);
        }else{
            actual = new Nodo(simbolo);
            if(simbolo == '*' || simbolo == '+' || simbolo == '?'){
                hijo_izquierdo = pila.pop();
                actual.hijo_izquierdo = hijo_izquierdo;
            }else{
                hijo_derecho = pila.pop();
                hijo_izquierdo = pila.pop();
                actual.hijo_derecho = hijo_derecho;
                actual.hijo_izquierdo = hijo_izquierdo;
            }
            pila.push(actual);
        }
    }
    raiz = pila.pop();
}
```

Nota: Los demás métodos de esta clase, son para sencillos, cómo de ver la siguiente posición de un nodo, calculas la posición de cada nodo en el árbol…

El **Estado.java** hace parte del autómata y sirve para guardar los estados que va a tener la AFD, la variable de estado_final sirve, entre muchas cosas, para saber el estado de aceptación.

```java
package expresiones.code;

import java.util.HashSet;
import java.util.Set;

public class Estado {
    int nombre;
    Set<Integer> conjunto_posicion = null;
    boolean marcado = false;
    boolean estado_final = false;

    public Estado(int nombre){
        this.nombre = nombre;
        this.conjunto_posicion = new HashSet<>();
    }
```

Nota: Los demás métodos de esta clase son sencillos y se usan para darle mejor usabilidad a la clase Estado.java.

La clase **Automata.java** utiliza el árbol que construimos anteriormente, lo va recorriendo y va construyendo él automata.

```java
public class Automata {
    public ArrayList<Estado> estados = null;
    public Set<Character> alfabeto = null;
    public Hashtable<Character,Integer> transiciones[] = null;
    public Set<Integer> raiz = null, siguiente_posicion[] = null;
    public Hashtable<Character, Set<Integer>> pos = null;
    public int n_caracteres = 0;
    public int n_estados = 0;

    public Automata(Arbol arbol, String expresion){
        this.pos = arbol.pos;
        this.raiz = arbol.raiz.primera_posicion;
        this.siguiente_posicion = arbol.siguiente_posicion;
        this.n_caracteres = arbol.n_caracteres;
        this.transiciones = new Hashtable[this.n_caracteres + 1];
        for(int i = 0; i <= this.n_caracteres; i++) transiciones[i] = new Hashtable<>();
        getAlfabeto(expresion);
    }
```

```java
// Creacion de un automata a partir de un arbol
public void crear_automata(){
    estados = new ArrayList<>();
    Estado inicial = new Estado(++n_estados);
    inicial.conjunto_posicion = (raiz);
    estados.add(inicial);
    while(true){
        Estado actual = null;
        boolean finalizado = true;
        for(Estado estado : estados){
            if(!estado.marcado){
                actual = estado;
                finalizado = false;
            }
        }
        if(finalizado) break;
        actual.marcado = (true);
        actual.asignar_estado_final(n_caracteres);
        for(Character simbolo : alfabeto){
            Set<Integer> u = new HashSet<>();
            for(Integer posicion : actual.conjunto_posicion){
                if(pos.get(simbolo).contains(posicion)){
                    u.addAll(siguiente_posicion[posicion]);
                }
            }
            if(!u.isEmpty()){
                boolean existe = false;
                Estado estado_vist = null;
                for(Estado estado : estados){
```

El método **DFS** es el encargado de revisar la cadena que ingresa el usuario, es un método recursivo, en el que recorre la cadena, de acuerdo a las transiciones que tiene.

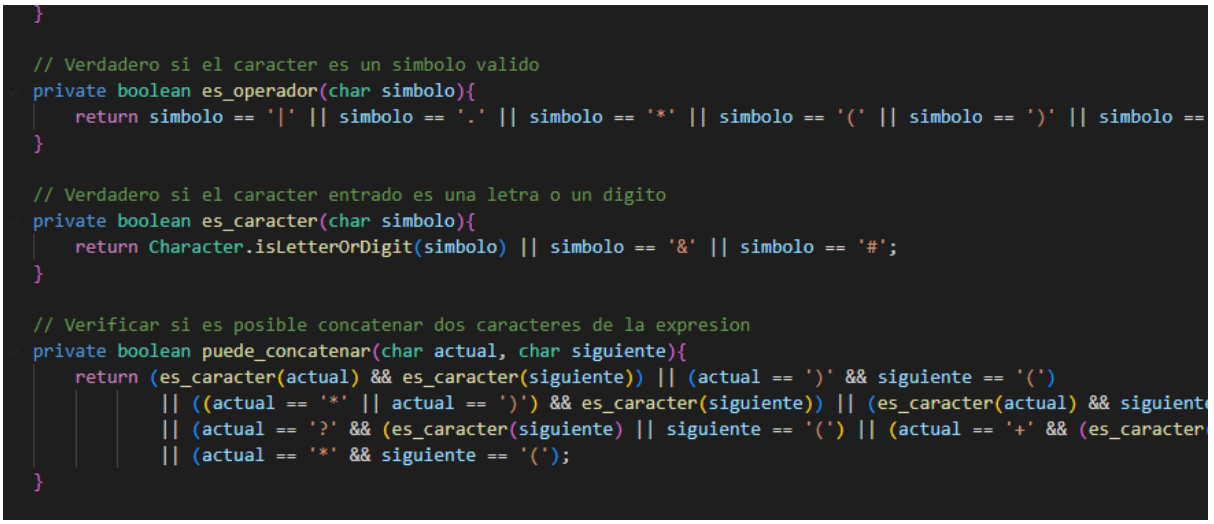
```java
// Retorna falso si la cadena ingresada no es aceptada por el automata
private boolean DFS(String cadena, int i, int estado){
    if(i == cadena.length() && estados.get(estado - 1).estado_final) return true;
    else if(i < cadena.length()){
        char simbolo = cadena.charAt(i);
        if(!transiciones[estado].isEmpty() && transiciones[estado].get(simbolo) != null){
            return DFS(cadena, i + 1, transiciones[estado].get(simbolo));
        }
    }
    return false;
}
```

VERIFICANDO LA ER:

El metodo **verificar_expression()** verifica la expresion ingresada, que no tenga *,+ seguidos o signos que no deban de ir.

```java
private void verificar_expression(){
    for(int i = 0; i < this.expresion.length(); i++){
        char c = this.expresion.charAt(i);
        if(c == '*'){
            if(i == 1){
                JOptionPane.showMessageDialog(parentComponent: null, message: "No puede comenzar con un *");
                return;
            }
            if(this.expresion.charAt(i + 1) == '*'){
                JOptionPane.showMessageDialog(parentComponent: null, "Error en la expresion regular, no pueden ha
                return;
            }
        }
        if(c == '+'){
            if(i == 1){
                JOptionPane.showMessageDialog(parentComponent: null, message: "No puede comenzar con un +");
                return;
            }
            if(this.expresion.charAt(i + 1) == '+'){
                JOptionPane.showMessageDialog(parentComponent: null, "Error en la expresion regular, no pueden ha
                return;
            }
        }
        if(c == '.'){
            if(i == 1){
                JOptionPane.showMessageDialog(parentComponent: null, message: "No puede comenzar con un .");
```

Los siguientes metodos ayuda a verificar la expresion regular.



```
}

// Verdadero si el caracter es un simbolo valido
private boolean es_operador(char simbolo){
    return simbolo == '|' || simbolo == '.' || simbolo == '*' || simbolo == '(' || simbolo == ')' || simbolo ==
}

// Verdadero si el caracter entrado es una letra o un digito
private boolean es_caracter(char simbolo){
    return Character.isLetterOrDigit(simbolo) || simbolo == '&' || simbolo == '#';
}

// Verificar si es posible concatenar dos caracteres de la expresion
private boolean puede_concatenar(char actual, char siguiente){
    return (es_caracter(actual) && es_caracter(siguiente)) || (actual == ')' && siguiente == '(')
            || ((actual == '*' || actual == ')') && es_caracter(siguiente)) || (es_caracter(actual) && siguient
            || (actual == '?' && (es_caracter(siguiente) || siguiente == '(') || (actual == '+' && (es_caracter
            || (actual == '*' && siguiente == '(';
}
```

Contacto:

En caso de que este archivo no resuelva sus dudas, puede contactarse al correo:

- cristian.castano1@udea.edu.co
- juan.salasf@udea.edu.co