Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Organización de Lenguajes y Compiladores 1

Sección B

Fecha: 06/03/2022

MANUAL TÉCNICO

Nombre:	Carné:
Gerson Rubén Quiroa del Cid	2020 00166

Índice

Requisitos del sistema 4
Para Windows 4
Mac OS 4
Linux 4
Introducción 5
Interfaz gráfica 6
Clase Inicio (Main) 6
Botón crear nuevo archivo y abrir un archivo existente 6
Botón guardar como y guardar 7
Botón analizar 8
Jlabel, combo box y botón siguiente 9
Clase Archivos 10
Función analizador léxico y sintáctico: 10
Archivos jflex y jcup 11
Jflex 12
Jcup 13
Paquete clases 13
Clase AFD 13
Función tablaTransciones 14
Función funciónEvaluar 14
Clase AFN 14
Función add 15
Clase AST 15
Función add 16
Función anulables y primeroUltimos 17
Función siguientes y recorridoSiguientes 17
Función transiciones y nuevaTransicion 18
Clase Conjunto 18
Clase Error 18
Clase ExpresionRegular 19
Clase Html 19
Clase RegExpEvaluar 20

Clase Siguiente 20
Clase Transicion 20

Requisitos del sistema

Para la instalación de la aplicación, su computadora y/o laptop debe cumplir como mínimo los siguientes requerimientos:

Para Windows

- Windows Vista SP2 (8u51 y superiores)
- Windows Server 2008 R2 SP1 (64 bits)
- Windows Server 2012 y 2012 R2 (64 bits)
- RAM: 128 MB
- Procesador: Mínimo Pentium 2 a 266 MHz
- Exploradores: Internet Explorer 9 y superior, Firefox

Mac OS

- Mac con Intel que ejecuta Mac OS X 10.8.3+, 10.9+
- Privilegios de administrador para la instalación
- Explorador de 64 bits
- Se requiere un explorador

Linux

- Oracle Linux 7.x (64 bits)2(8u20 y superiores)
- Red Hat Enterprise Linux 7.x (64 bits)2(8u20 y superiores)
- Suse Linux Enterprise Server 12.x (64 bits)2(8u31 y superiores)
- Ubuntu Linux 12.04 LTS, 13.x
- Ubuntu Linux 15.10 (8u65 y superiores)

IMPORTANTE: Independientemente del sistema operativo del usuario, es necesario que previamente instale Java en su computadora.

Editor utilizado: NetBeans IDE 8.2

Versión de Java utilizado: JDK "1.8.0_241"

Introducción

El manual técnico tiene como finalidad de explicar el funcionamiento de cada función utilizado en el presente programa, con el objetivo de que no haya confusión al leer el código del programa y cualquier persona con conocimiento básico de programación en Java pueda comprender lo que se hizo en cada línea de código. Por lo cual se recomienda que toda persona que quiera leer o modificar el código vea este manual para ahorraste tiempo y así sea más fácil su comprensión.

Interfaz gráfica

Estas clases son las que conforman la interfaz gráfica, solo se utilizó una clase y algunas ventanas emergente de confirmación o notificación.

Clase Inicio (Main)

Esta clase es la principal del programa donde inicialmente se ejecutará el menú con las diferentes opciones que puede elegir el usuario, dependiendo de la opción que se elija el usuario se ejecutarán las diferentes funciones detalladas más adelante.

```
/**
 *
 * @author Sammy Guergachi <sguergachi at gmail.com>
 */
public class Inicio extends javax.swing.JFrame {

    Archivos archivo = new Archivos();
    ArrayList<AFD> afds = new ArrayList();
    int siguiente = 0;

    /**
     * Creates new form Inicio
     */
    int contador = 1;

    public Inicio() {
        initComponents();
        this.setLocationRelativeTo(null);
    }
```

Botón crear nuevo archivo y abrir un archivo existente

Dependiendo de que botón se precios se abrirá una ventana para que el usuario escriba la ruta donde quiere que se cree el nuevo archivo, o una ventana donde podrá elegir el archivo que quiera abrir.

```java
private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {
    // Crear un nuevo archivo
    Rutas ruta = new Rutas("");
    ruta.setVisible(true);
    jTextArea1.setText("");
}

private void jButton2MouseClicked(java.awt.event.MouseEvent evt) {
    // Abrir un archivo ya creado
    JFileChooser fc = new JFileChooser();
    FileNameExtensionFilter filtro = new FileNameExtensionFilter("*.exp", "exp");
    fc.setFileFilter(filtro);
    int seleccion = fc.showOpenDialog(this);
    if (seleccion == JFileChooser.APPROVE_OPTION) {
        File fichero = fc.getSelectedFile();
        Archivos.rutaG = fichero.getAbsolutePath();

        try (FileReader fr = new FileReader(fichero)) {
            String cadena = "";
            int valor = fr.read();
            while (valor != -1) {
                cadena = cadena + (char) valor;
                valor = fr.read();
            }
            this.jTextArea1.setText(cadena);
        } catch (IOException e) {

        }
```

Botón guardar como y guardar

Estos botones como su nombres lo indican, el primero se desplegará una ventana donde el usuario ingresará una ruta donde quiera que el archivo se guarde y el segundo botón para simplemente guardar en el mismo archivo ya creado.

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    // guardar archivo actual
    try {
        FileWriter fileWriter;
        fileWriter = new FileWriter(Archivos.rutaG);
        fileWriter.write(jTextAreal.getText());
        fileWriter.close();
        JOptionPane.showMessageDialog(null, "Se ha guardado con éxito");
    } catch (IOException ex) {
        Logger.getLogger(Inicio.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    // guardar como
    Rutas ruta = new Rutas(jTextAreal.getText());
    ruta.setVisible(true);
}
```

Botón analizar

En cuanto a la interfaz, este botón es el más importante ya que en él se realizan todas las operaciones y llamadas a métodos para la creación de todo el programa, como lo son el analizador léxico y sintáctico, así como también la creación del árbol, autómata finito no determinista y el autómata finito determinista.

```
String resultado = "";
try {
    // analizador lexico
    resultado = archivo.analizadorLexico((String) jTextAreal.getText());
} catch (IOException ex) {
    Logger.getLogger(Inicio.class.getName()).log(Level.SEVERE, null, ex);
}
// analizador sintáctico
resultado += archivo.analizadorSintactico((String) jTextAreal.getText());

// lógica de las evaluaciones de las expresiones regulares
if (resultado.contains("Análisis realizado correctamente.")) {
    AST arbol;
    resultado += "\nAUTOMATAS\n";
    for (ExpresionRegular exp : archivo.expresionesRegulares) {
        // primer reporte (arbol sintactico)
        arbol = new AST(exp.nombreVariable);
        arbol.regex(exp.expresionRegular);
        arbol.anulables(arbol.raiz);
        arbol.id = 1;
        arbol.primerosUltimos(arbol.raiz);
        arbol.graphviz();
        // segundo reporte (tabla de siguientes)
        arbol.siguientes();
        // tercer reporte (tabla de transiciones)
        arbol.transiciones();
        // automata AFD
        AFD afd = new AFD(exp.nombreVariable);
```

Jlabel, combo box y botón siguiente

Estos 3 componentes se utilizaron para el manejo de las gráficas previamente creadas mediante el botón de analizar. Con el combo box podemos movernos por las diferentes carpetas, con el botón siguiente podemos pasar por cada gráfica creada y en el label se mostrará cada gráfica.

```
void mostrarImagen(String ruta) {
    ImageIcon imagen = new ImageIcon(ruta);
    ImageIcon icono = new ImageIcon(
            imagen.getImage().getScaledInstance(this.imagenLabel.getWidth(),
                    this.imagenLabel.getHeight(),
                    Image.SCALE_SMOOTH)
    );
    this.imagenLabel.setIcon(icono);
    this.repaint();
}
```

Clase Archivos

Esta clase es la encargada de guardar diferentes archivos como lo son archivos json, así como también los métodos para guardar un nuevo archivo y guarda como.

```java
// ============== guardar json ==============
public void crearJson(int numero, String cadena) {
    FileWriter fichero = null;
    PrintWriter pw = null;

    try {
        fichero = new FileWriter("REPORTES/SALIDAS_202000166/" + numero + ".json");
        pw = new PrintWriter(fichero);
        pw.write("[\n");
        pw.write(cadena);
        pw.write("]");
        pw.close();
        fichero.close();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    } finally {
        if (pw != null) {
            pw.close();
        }
    }
}
```

Función analizador léxico y sintáctico:

Estas funciones son las más importantes en esta clase, ya que por acá pasa el analizador léxico y sintáctico que generamos mediante jflex y jcup.

En la función del analizador léxico, se lee cada token y se valida si es un token perteneciente al lenguaje o en su defecto un token desconocido, que generará un error léxico.

Por otra parte, en la función del analizador sintáctico, se analiza el archivo de entrada y éste devuelve un string que dirá si el contenido ha sido aceptado o por su defecto, que el contenido contiene un error de tipo sintáctico.

```java
public String analizadorLexico(String expr) throws IOException {
    // === Variables y objetos para recuperar la info necesaria
    String flag = "", variable = "", regex = "";
    Conjunto conjunto = new Conjunto();
    ExpresionRegular expresionRegular = new ExpresionRegular();
    RegExpEvaluar regExpEvaluar = new RegExpEvaluar();
    // ========================================================
    int linea = 1, columna = 0;
    Lexer lexer = new Lexer(new StringReader(expr));
    String resultado = "";
    while(true) {
        Tokens token = lexer.yylex();
        if (token == null) {
            return resultado;
        }
        switch(token) {
            case SaltoLinea:
                columna = 0;
                linea++;
                break;
            case LlaveAbre:
                regex += lexer.lexeme;
                columna += lexer.yylength();
                break;
```

Archivos jflex y jcup

Estos archivos y clases se crearon con ayuda de las librerías ya mencionadas, dentro de estos dos archivos creamos la lógica de nuestros analizadores tanto léxico como sintáctico. Una vez escrito dichos archivos, en una clase main, compilamos los archivos y éstos crearán las clases necesarias que serán nuestros analizadores.

Jflex

```
package analizadores;
import static analizadores.Tokens.*;
%%
%class Lexer
%type Tokens
ALFANUMERICO = ([A-Za-z_])[A-Za-z0-9_]+
NUMERO = [0-9]+([.][0-9]+)?
IGNORAR = [ ,\t,\r]+
COMENTARIOS = "<!"(.|\n)*"!>"|"//".*
STRING = \"[^\"]*\"
RANGO = ."~".|.(",".)+
%{
    public String lexeme;
%}
%%
"{" {lexeme = yytext(); return LlaveAbre;}
"}" {lexeme = yytext(); return LlaveCierre;}
"CONJ" {lexeme = yytext(); return Conjunto;}
"%%" {lexeme = yytext(); return Separador;}
":" {lexeme = yytext(); return DosPuntos;}
"->" {lexeme = yytext(); return Asignador;}
";" {lexeme = yytext(); return PuntoComa;}
"\n" {lexeme = yytext(); return SaltoLinea;}
"." {lexeme = yytext(); return Punto;}
"*" {lexeme = yytext(); return Asterisco;}
"\"" {lexeme = yytext(); return Comilla;}
```

Jcup

```
package analizadores;

import java_cup.runtime.Symbol;

parser code
{:
    private Symbol s;

    public void syntax_error(Symbol s) {
        this.s = s;
    }

    public Symbol getS() {
        return this.s;
    }
:};

terminal LlaveAbre, LlaveCierre, Conjunto, Separador, DosPuntos, Asignador,
    Rango, PuntoComa, SaltoLinea, Punto, Asterisco, Comilla, Barra, Negativo,
    Identificador, Numero, Mas, Interrogacion, Coma, ComillaSimple,
    Cadena, ERROR;

non terminal INICIO, CUERPO, DECLARACIONES, EVALUACIONES, ER,
    CONJUNTO, EXPRESION_REGULAR, SEPARADOR, EXPRESION_EVALUAR;

start with INICIO;
```

Paquete clases

Clase AFD

Esta clase es la que nos ayuda a pasar de la tabla de transiciones previamente hecha en la clase AST, hacia una estructura de tipo AFD, para luego poder recorrer el autómata con las cadenas a evaluar y poder saber si la cadena es aceptada o no.

```
public class AFD {

    // aquí estarán todos los estados
    ArrayList<Estado> estados;
    public String nombreRegex;

    public AFD(String nombreRegex) {
        this.nombreRegex = nombreRegex;
        this.estados = new ArrayList<>();
    }
```

Función tablaTransciones

Esta función recibe la tabla de transiciones proporcionada por la clase AST y esta crea una estructura a partir de la tabla.

```
public void tablaTransiciones(ArrayList<Transicion> tablaTransiciones) {
    // primero agregamos todos los estados al array
    int id = 0;
    for (Transicion transicion : tablaTransiciones) {
        estados.add(new Estado(id, transicion.estado, transicion.esAceptacion));
        id++;
    }
```

Función funciónEvaluar

Esta función es la que se encarga de recibir una cadena la cual queremos evaluar si es aceptada mediante los autómatas creados. Si es aceptada devuelve un true, de lo contrario regresa un false.

```
public boolean evaluar(RegExpEvaluar valor) {
    String estadoActual = "S0";
    // recorriendo el string
    for (char c : valor.cadena.toCharArray()) {
        valor.esAceptado = false;
        // recorriendo los estados
        for (Estado estado : this.estados) {
            if (estado.nombreEstado.equals(estadoActual)) {
```

Clase AFN

Esta clase es la encargada de crear el autómata finito no determinista con el método de Thompson. Este autómata se crea con ayuda del árbol, ya que se recorre el árbol y se construye a través de este.

```
public class AFN {
    String nombreRegex;
    public NodoAFN inicio;
    // estas estructuras nos ayudarán a armar el autómata
    public Stack<String> hojas;
    public Stack<NodoAFN[]> conjuntos;
    int contador = 1;

    public AFN(String nombreRegex) {
        this.nombreRegex = nombreRegex;
        this.inicio = new NodoAFN("S0", "");
        this.hojas = new Stack();
        this.conjuntos = new Stack();
    }
```

Función add

Esta función ayuda a crear el AFN, este método es extenso ya que para cada operación existe una forma estándar de hacerlo, por lo que para la concatenación (.), como para el or (|), como para las cerraduras, etc. Tiene su forma diferente de crearlos y por lo tanto se debe tener en cuenta cada posibilidad.

```
// aqui agregaremos los nodos del afn
public void add(Nodo nodo) {
    if(nodo.tipo.equals("hoja")) {
        // agregando a la pila las hojas
        this.hojas.push(nodo.dato);
    } else if(nodo.dato.equals("+")) {
        // si su hijo es una hoja lo enlazamos con la hoja
        if(nodo.izquierda.tipo.equals("hoja")) {
            // obtenemos la hoja que esta en el top
            String hojal = this.hojas.pop();
            // construimos el conjunto y lo añadimos a la cola de conjuntos
            NodoAFN nodo2 = new NodoAFN("S" + contador, "£");
            this.contador++;
            NodoAFN nodo3 = new NodoAFN("S" + contador, hojal);
            this.contador++;
            NodoAFN nodo4 = new NodoAFN("S" + contador, "£");
            this.contador++;
            // enlazando
            nodo3.estadoSiguiente1 = nodo2;
            nodo3.estadoSiguiente2 = nodo4;
            nodo2.estadoSiguiente1 = nodo3;
            NodoAFN[] aux = {nodo2, nodo4};
            this.conjuntos.push(aux);
        // sino lo enlazamos con un conjunto
```

Clase AST

Esta clase se encarga de crear el árbol binario que genera cada expresión regular ingresada en el archivo fuente. Esta clase es una de las más larga ya que dentro de ella están las funciones para crear el árbol, anulables, primeros y últimos, los siguientes y tabla de transiciones.

```java
public class AST {

    public Nodo raiz;
    public String nombreRegex;
    public int id;
    public Siguiente[] tablaSiguientes;
    public ArrayList<Transicion> tablaTransiciones;

    public AST(String nombreRegex) {
        this.nombreRegex = nombreRegex;
        // inicializando el arbol siempre con un . - # que es la terminacion de la cadena
        this.raiz = new Nodo(".", "binario", 1);
        this.raiz.derecha = new Nodo("#", "hoja", 2);
        this.id = 3;
    }
```

Función add

Esta función ayuda a crear el árbol binario mediante la expresión regular proporcionada por el usuario. Cabe resaltar que la expresión regular está dada en notación polaca, por lo que se creo de la misma manera el árbol. Para obtener la misma expresión se deberá de recorrer en pre-orden, si se quiere en notación normal se recorrerá en orden.

```java
public void add(Nodo nuevo, Nodo aux) {
    // si es binario (+|.) puede tener hijos en ambos lados
    if (aux.tipo.equals("binario")) {
        // si está vacío la izquierda se agrega el nodo, sino sigue recorriendo
        if (aux.izquierda == null) {
            nuevo.estaAsignado = true;
            aux.izquierda = nuevo;
        } else {
            add(nuevo, aux.izquierda);
        }
        // si está vacío la derecha se agrega el nodo, sino sigue recorriendo
        if (!nuevo.estaAsignado) {
            if (aux.derecha == null) {
                nuevo.estaAsignado = true;
                aux.derecha = nuevo;
            } else {
                add(nuevo, aux.derecha);
            }
        }
        // si es unario (+|*|?) solo tiene un hijo
```

Función anulables y primeroUltimos

Estas funciones como sus nombres lo indican, se utiliza para darle a cada nodo del árbol, sus anulables y primeros y últimos. Para poder ir en un orden ascendente, desde sus hojas hasta su raíz, se recorrió en post orden.

```
// ============= RECORRIDOS =============
public void anulables(Nodo aux) {
    if (aux != null) {
        this.anulables(aux.izquierda);
        this.anulables(aux.derecha);
        if (aux.tipo == "hoja") {
            // si es hoja: es NO anulable
            aux.esAnulable = false;
        } else if (aux.dato.equals("+")) {
            // si su hijo es anulable, es anulable, de lo contrario es NO anulable
            aux.esAnulable = aux.izquierda.esAnulable;
        } else if (aux.dato.equals("*") || aux.dato.equals("?")) {
            // si es * o ?: es anulable siempre
            aux.esAnulable = true;
        } else if (aux.dato.equals(".")) {
            // si el hijo izquierdo y derecho son anulables, es anulable de lo contrario es NO anulable
            aux.esAnulable = aux.izquierda.esAnulable && aux.derecha.esAnulable;
        } else if (aux.dato.equals("|")) {
            // si el hijo izquierdo o derecho es anulable, entonces es anulable sino es NO anulable
            aux.esAnulable = aux.izquierda.esAnulable || aux.derecha.esAnulable;
        }
    }
}
```

Función siguientes y recorridoSiguientes

Estas funciones son las encargadas de crear la tabla de siguientes del árbol generado anteriormente. Cuando se termine de crear la tabla de siguientes, se llama a una función que crea un html con la tabla e información necesaria.

```
public void siguientes() {
    // para saber cuantas hojas tiene el arbol, iremos a ver a el ider
    int cantidadHojas = Integer.parseInt(this.raiz.derecha.primeros);
    this.tablaSiguientes = new Siguiente[cantidadHojas];
    // primero pondremos las hojas con sus identificadores
    recorridoSiguientes(this.raiz);
    // reporte de siguientes
    Html html = new Html();
    html.reportSiguientes(this.nombreRegex, this.tablaSiguientes);
}

public void recorridoSiguientes(Nodo aux) {
    if (aux != null) {
```

Función transiciones y nuevaTransicion

Estas funciones crean la tabla de transiciones a partir de la tabla de siguientes. Si existe una nueva transición, se llama a la función nuevaTransicion sino se termina la recursividad y por último se llama a la función que creará el html con la tabla de transiciones.

```
public void transiciones() {
    // inicializamos el array
    this.tablaTransiciones = new ArrayList<>();
    // añadimos el estado inicial que son los primeros de la raiz
    String[] aux = this.raiz.primeros.split(",");
    ArrayList<Integer> estadoInicial = new ArrayList<>();
    for (String aux1 : aux) {
        estadoInicial.add(Integer.parseInt(aux1));
    }
    this.tablaTransiciones.add(new Transicion("S0", estadoInicial, this.tablaSiguientes));
    recorridoTransiciones(this.tablaTransiciones.get(0));
    // reporte de transiciones
    Html html = new Html();
    html.reportTransicion(this.nombreRegex, this.tablaTransiciones);
}
```

Clase Conjunto

Esta clase nos servirá para guardar información de los conjuntos que se definen en el archivo de entrada.

```
public class Conjunto {
    public String nombreVariable;
    public String []rango;

    public Conjunto(String nombreVariable, String []rango){
        this.nombreVariable = nombreVariable;
        this.rango = rango;
    }

    public Conjunto() {
        this.nombreVariable = "";
        this.rango = null;
    }
```

Clase Error

Esta clase es la encargada de guardar los errores que existan en el archivo de entrada tanto léxico como sintácticamente.

```java
public class Error {
    public String tipo;
    public String descripcion;
    public int linea;
    public int columna;

    public Error(String tipo, String descripcion, int linea, int columna) {
        this.tipo = tipo;
        this.descripcion = descripcion;
        this.linea = linea;
        this.columna = columna;
    }
```

Clase ExpresionRegular

Esta clase guarda las expresiones regulares que se definieron en el archivo de entrada.

```java
public class ExpresionRegular {
    public String nombreVariable;
    public String[] expresionRegular;

    public ExpresionRegular (String nombreVariable, String[] expresionRegular) {
        this.nombreVariable = nombreVariable;
        this.expresionRegular = expresionRegular;
    }
```

Clase Html

Esta clase se encarga de crear cada reporte que sea necesario en html. Los reportes que se crearán cuando se necesiten son los siguientes: reporte de errores (reportErrores()), reporte de siguientes (reportSiguientes()) y reporte de transiciones (reportTransicion()), a continuación se mostrará uno de ellos.

```java
public void reportTransicion(String nombreArchivo, ArrayList<Transicion> tablaTransiciones) {
    try {
        fichero = new FileWriter("REPORTES/TRANSICIONES_202000166/" + nombreArchivo + ".html");
        pw = new PrintWriter(fichero);

        pw.println("<!DOCTYPE html><!--Declarar el tipo de cumento -->\n"
                + "<html>\n"
                + "\n"
                + "<!--Encabezado-->\n"
                + "<head>\n"
                + "<meta charset=\"UTF-8\"><!--codififcaion de caracteres ñ y á-->\n"
                + "\n"
                + "\n"
                + "<meta name=\"name\" content=\"Reporte\"><!--nombre de la pagina-->\n"
                + "<meta name=\"description\" content=\"name\"><!--autor de la pagina-->\n"
                + "<meta name=\"keywods\" content=\"uno,dos,tres\"><!--Palabras claavez para, separadas por comas-->\n"
                + "<meta name=\"robots\" content=\"Index, Follow\"><!--Mejora la busqueda-->\n"
                + "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1\"><!--visibilidaad en diferentes p
```

Clase RegExpEvaluar

Esta clase guarda las cadenas a evaluar que se ingresaron en el archivo de entrada y la información importante como, por ejemplo, si es aceptada la cadena o no.

```
public class RegExpEvaluar {
    public String nombreVariable;
    public String cadena;
    public boolean esAceptado;
    public String matchCon;

    public RegExpEvaluar(String nombreVariable, String cadena, boolean esAceptado){
            this.nombreVariable = nombreVariable;
            this.cadena = cadena;
            this.esAceptado = esAceptado;
            this.matchCon = "";
    }
```

Clase Siguiente

Esta clase guarda información sobre los siguientes que nos proporcionan los árboles binarios.

```
public class Siguiente {

    String hoja;
    int identificador;
    ArrayList<Integer> siguientes;

    public Siguiente(String hoja, int identificador) {
        this.hoja = hoja;
        this.identificador = identificador;
        this.siguientes = null;
    }
}
```

Clase Transicion

Esta clase guarda información sobre cada transición, o más bien cada estado, dentro de esta clase se sabrá hacia que otro estado se irá dependiendo de la información que contenga.

```java
public class Transicion {

    boolean esAceptacion;
    String estado;
    ArrayList<Integer> contenido;
    ArrayList<Hoja> transiciones;

    public Transicion(String estado, ArrayList<Integer> contenido, Siguiente[] tablaSiguientes) {
        this.esAceptacion = false;
        this.estado = estado;
        this.contenido = contenido;
        initTransiciones(tablaSiguientes);
        System.out.println("");
    }
```