ExpAnalyzer

Manual Técnico

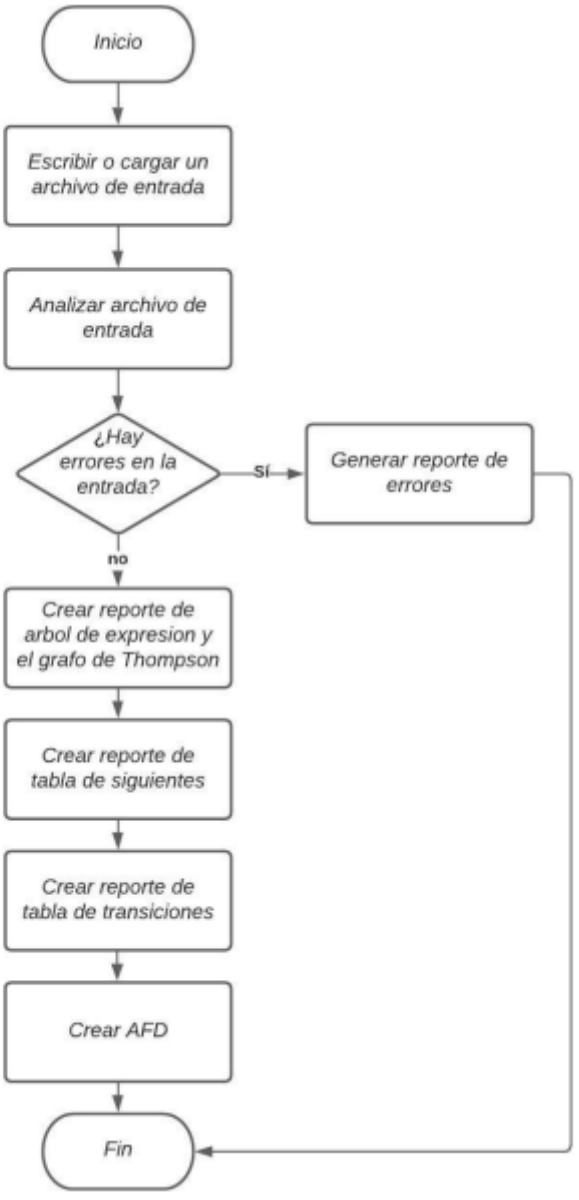
ExpAnalyzer

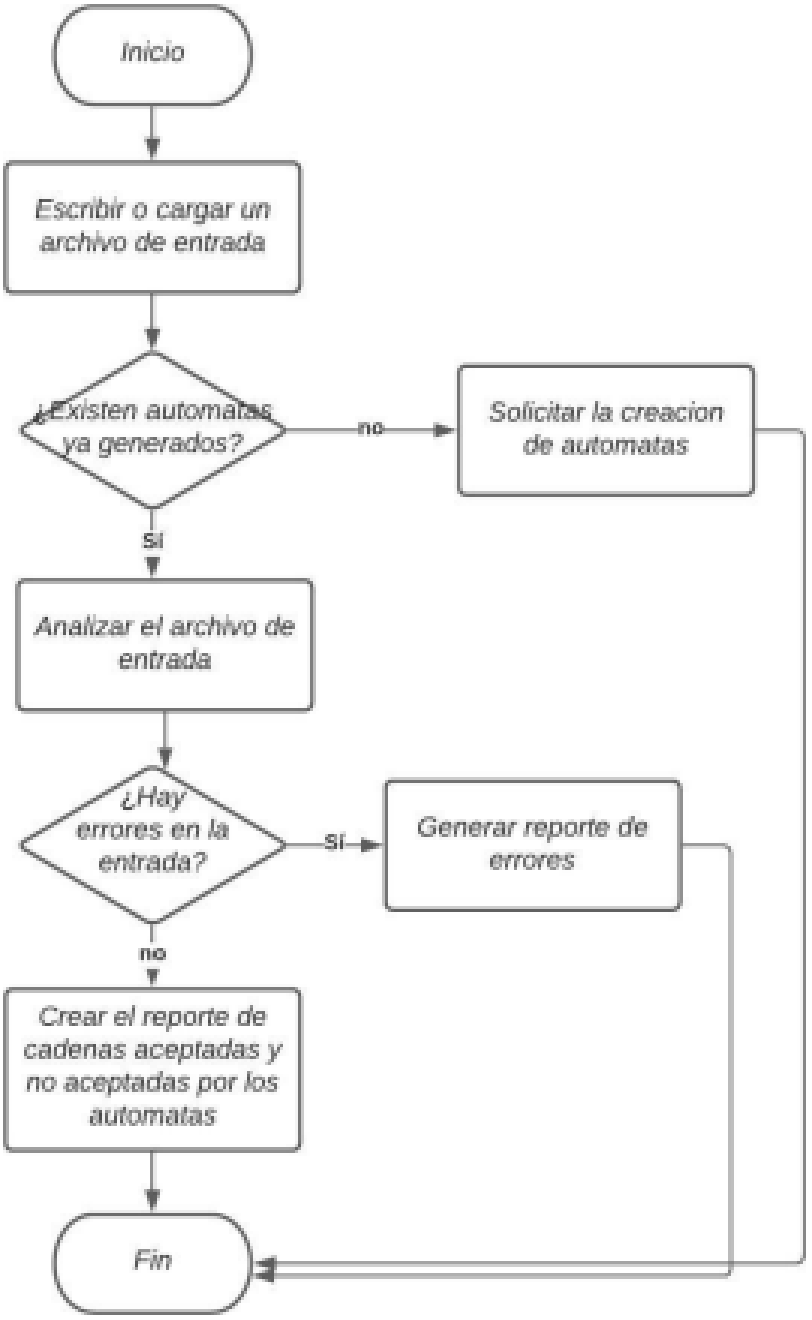
Lenguaje Utilizado: Java 17
Sistema Operativo: Windows 7 en adelante

Resumen

ExpAnalyzer es un analizador de texto tomando como base una serie de expresiones regulares. Utilizando el lenguaje exp, el usuario puede definir una serie de expresiones regulares, con o sin el uso de conjuntos, y verificar si una palabra pertenece o no al lenguaje definido en la expresión.

Flujo general del programa




Inicio
Escribir o cargar un archivo de entrada
¿Existen automatas ya generados?
no
Solicitar la creacion de automatas
si
Analizar el archivo de entrada
¿Hay errores en la entrada?
si
Generar reporte de errores
no
Crear el reporte de cadenas aceptadas y no aceptadas por los automatas
Fin

Explicación del flujo del programa:

En palabras simples, el programa se divide en tres partes principales: El analizador léxico de la entrada del archivo, el análisis de sintaxis del mismo y el manejo de la información de la interfaz.
El analizador léxico es creado gracias a la herramienta JFlex diseñada para este fin. En él, se describen cuales son los terminales que debe de reconocer el lenguaje y la estructura que deben poseer para lograrlo a través de expresiones regulares. Más allá de eso, no maneja la gran cosa. A lo mucho, tiene en su poder una lista para ir guardando los errores que se vayan encontrando en la lectura de caracteres.

```
digito =  [0-9]
letra = [A-Za-z]
simbolos = [!-$]|[&-)]|\/|-|[<->]|@|[\[-\`]
mullinea = \<\!([^"!>"]|[\r|\f|\s|\t|\n])*\!\>
comlinea = \/\/.*
strings = \"(\!|[\#-\»]|\s)*\"
flecha = -(\s)*>
id = {letra}({letra}|{letra}|{digito}|_)+
especial  =  \\n|\\\"|\\\'
espacio = [\r|\f|\s|\t|\n]

%%

"%"     { return new Symbol(sym.porcentaje, yyline, yycolumn, yytext()); }
"CONJ"     { return new Symbol(sym.conjunto, yyline, yycolumn, yytext());}
"."     { return new Symbol(sym.concatenacion, yyline, yycolumn, yytext());}
"|"     { return new Symbol(sym.or, yyline, yycolumn, yytext());}
"*"     { return new Symbol(sym.kleene, yyline, yycolumn, yytext());}
"+"     { return new Symbol(sym.positiva, yyline, yycolumn, yytext());}
"?"     { return new Symbol(sym.cerouno, yyline, yycolumn, yytext());}
"~"     { return new Symbol(sym.guion, yyline, yycolumn, yytext());}
"{"     { return new Symbol(sym.apertura, yyline, yycolumn, yytext());}
"}"     { return new Symbol(sym.cierre, yyline, yycolumn, yytext());}
","     { return new Symbol(sym.coma, yyline, yycolumn, yytext());}
":"     { return new Symbol(sym.dospuntos, yyline, yycolumn, yytext());}
";"     { return new Symbol(sym.puntocoma, yyline, yycolumn, yytext());}
{espacio}   { }
{mullinea}  { }
{comlinea}  { }
{strings}   { return new Symbol(sym.cadena, yyline, yycolumn, yytext());}
```

El analizador de sintaxis, es sin duda, la parte que acarrea la mayor parte de las funciones del programa Está diseñado gracias a la herramienta CUP y se aprovecha del recorrido que realiza de la estructura del archivo para ir extrayendo la información y agrupando en objetos que se analizan más adelante. El lenguaje exp se divide en tres partes: la declaración de conjuntos, la declaración de las

expresiones regulares y por último, la declaración de las palabras a validar con los autómatas generados.

Para el manejo de conjuntos, el programa se encarga de readaptar la estructura con la que vienen en la entrada para que pueda adaptarse al código de graphviz. Aparte, se comienza a tomar en cuenta su uso en los autómatas, para lo cual, se crea una lista de Strings donde estará almacenado cada uno de los caracteres que conforman el conjunto.

```
/*--------------- Manejo de Conjuntos ----------------------*/

CONJUNTO ::= conjunto dospuntos id:a flecha NOT:b puntocoma
            {:
                ArrayList<String> ncar = new ArrayList<>();
                for(int i = 0; i < car.size(); i++){
                    ncar.add(car.get(i));
                }
                Conjunto nuevo = new Conjunto(a, ncar, b);
                con.add(nuevo);
                car.clear();
            :}
            | error
            {:
                Errores crear = new Errores("Error sintaxix", s.value.toString(), s.right+1, s.left+1);
                a.add(crear);
            :}
;

NOT ::= letra:a guion letra:b
        {:
            int min = a.toCharArray()[0];
            int max = b.toCharArray()[0];
            if(min > max){
                int ntem = min;
                min = max;
                max = ntem;
            }
            for(int i=min;i<=max;i++){
                car.add(Character.toString(i));
```

```
  |SYM:a guion SYM:b
  {:
      int min = a.toCharArray()[0];
      int max = b.toCharArray()[0];
      if(min > max){
          int ntem = min;
          min = max;
          max = ntem;
      }
      for(int i=min;i<=max;i++){
          car.add(Character.toString(i));
      }
      String temp = "[" + a + "-" + b +"]";
      RESULT = temp;
  :}

  |GRUPO:a
  {:
      String temp = "(" + a + ")";
      Collections.reverse(car);
      RESULT = temp;
  :}
  | error
 {:
      Errores crear = new Errores("Error sintaxix", s.value.toString(), s.right+1, s.left+1);
      a.add(crear);
  :}
```

El manejo de las expresiones se divide en dos partes: una va orientada al diseño de los autómatas con Thompson, y la otra para el manejo del método del árbol. Para el método de Thompson, la idea principal es generar una estructura, con el uso de nodos, para recrear de forma abstracta las conexiones que conforman cada nodo de la figura. Ya que Cup analiza de forma recursiva, se explota esa función para ir añadiendo y creando las conexiones que puedan surgir entre las expresiones regulares. Al tener todo el objeto creado por completo, se almacena en un objeto de tipo Thompson, el cual será utilizado luego para crear el grafo de la figura.

Para la parte del árbol, el objetivo es similar. Se busca recrear las conexiones que puedan representar su estructura, en función de los diferentes operadores que vayan apareciendo en la lectura. Para el caso del árbol, la cosa no queda ahí ya que luego de crearlo, se debe de comenzar inmediatamente su análisis. Con la ayuda de los nodos, es posible ir arrastrando los valores de los siguientes, por cada regreso que se vaya dando. Eso hace que para cuando termine de analizar el árbol, la cabecera cargue con todo el resumen del análisis. Con ello, es posible crear la tabla de siguientes, la tabla de transiciones y el autómata en sí mismo. Como era de imaginarse, el autómata como tal funciona como una estructura de nodos. Usando la tabla de transiciones, se crea la clase estado la cual almacena no solo el nombre y los caracteres que reconoce, sino también, una lista de conexiones que tiene con otros de los estados y los caracteres que lo llevan hasta ellos.

```
/*--------------- Manejo de Expresiones ----------------------*/
EXPRESION ::= id:a flecha E:b puntocoma
        {:
           //Procedimiento del metodo Thompson
           NodoT Cabeceral = (NodoT)b.get(0);
           NodoT Salidal = (NodoT)b.get(1);
           ArrayList<NodoT> al = (ArrayList<NodoT>)b.get(2);
           Thompson nuevo = new Thompson(a, al, Cabeceral);
           tom.add(nuevo);

           //Procedimiento del metodo del arbol
           NodoA hijol = (NodoA)b.get(3);
           ArrayList<Siguiente> sl = (ArrayList<Siguiente>)b.get(4);
           ArrayList<NodoA> nl = (ArrayList<NodoA>)b.get(5);

               //Creacion del nodo aceptacion
           NodoA all = new NodoA();
           all.simbolo = "$";
           all.valor = "$";
           all.anulable = false;
           all.nombre = Integer.toString(contador);
           contador++;
           all.tipo = "Hijo";
           all.first.add(all.nombre);
           all.last.add(all.nombre);

           ArrayList<NodoA>listaNombre = new ArrayList<>();
           listaNombre.add(all);
```

```
//Mezcla de Siguientes
ArrayList<Siguiente> sis = new ArrayList<>();
for(int i = 0; i < s1.size(); i++){
    sis.add(s1.get(i));
}
for(int i = 0; i < s2.size(); i++){
    sis.add(s2.get(i));
}

//Calculo de Siguientes
ArrayList<String> ast = hijo2.first;
for(int i = 0; i < hijo1.last.size(); i++){
    String st = hijo1.last.get(i);
    for(int j = 0; j < sis.size(); j++){
        Siguiente ste = sis.get(j);
        if(ste.hoja == st){
            if(ast.size() == 0){
                for(int k = 0; k < ast.size(); k++){
                    ste.LS.add(ast.get(k));
                }
            }else{
                for(int k = 0; k < ast.size(); k++){
                    if(Arrays.asList(ste.LS).contains(ast.get(k)) == false){
                    ste.LS.add(ast.get(k));
                    }
                }
            }
            break;
        }
    }
}
```

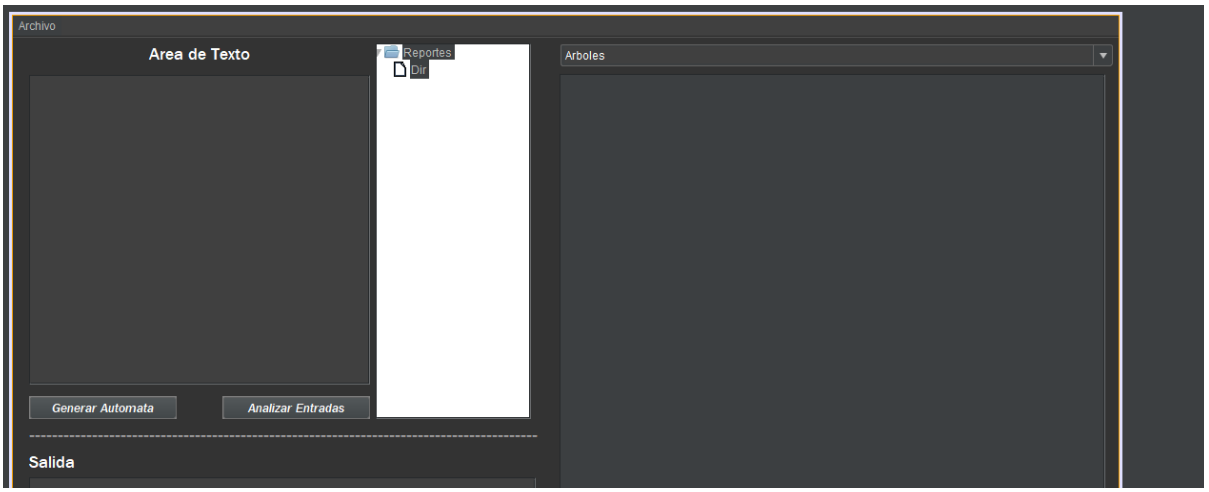
La lista de validaciones, al igual que la de los conjuntos es bastante simple. Al terminar de crear los autómatas, se van guardando en objetos y estos se van almacenando en una lista a la que posteriormente se puede acceder a través de la instancia. La sección de validaciones, solo se encarga de buscar el autómata por el id que indica la entrada y al encontrarlo, lo guarda en un objeto junto a la cadena que se va a analizar.

```
|id:a dospuntos cadena:b puntocoma
{:
    Automata temp = null;
    int ban = 0;
    for(int i = 0; i < auto.size(); i++){
        temp = auto.get(i);
        if(temp.ID.equals(a)){
            ban = 1;
            break;
        }
    }

    if(ban == 0){
        temp = null;
    }
    else{
       //Crear el objeto validar
        Validar vnuevo = new Validar(b, temp);
        val.add(vnuevo);
    }


:}
| error
{:
     Errores crear = new Errores("Error sintaxix", s.value.toString(), s.right+1, s.left+1);
     a.add(crear);
 :}
```

La Clase Main es la interfaz de usuario. Su principal tarea, es ser la conexión entre los analizadores y la información de entrada.



El método actualizar tiene la tarea de actualizar los elementos de la interfaz con cada una de las salidas que se vayan creando. Para este caso, hace que el árbol de archivos se actualice y también monta las imágenes en las listas correspondientes.

```
//Diseño del arbol de archivos: https://www.youtube.com/watch?v=zWdA8lMEz3A
//El metodo actualizar se va a activar cada que se crea una nueva imagen
private void actualizar(){
    File fichero = new File("Reportes/");
    raiz = new DefaultMutableTreeNode(fichero.getName());
    modelo = new DefaultTreeModel(raiz);
    crear(fichero, raiz);
    Archivos.setModel(modelo);

    //Actualizar listas de imagenes
    arbol = new LinkedList<>();
    Collections.addAll(arbol, archivos("Reportes/ARBOLES_201909035/"));

    transicion = new LinkedList<>();
    Collections.addAll(transicion, archivos("Reportes/TRANSICIONES_201909035/"));

    siguiente = new LinkedList<>();
    Collections.addAll(siguiente, archivos("Reportes/SIGUIENTES_201909035/"));

    afd = new LinkedList<>();
    Collections.addAll(afd, archivos("Reportes/AFD_201909035/"));

    afnd = new LinkedList<>();
    Collections.addAll(afnd, archivos("Reportes/AFND_201909035/"));

}
```

El método crear es un auxiliar del método actualizar. Lo que hace es buscar y actualizar, de forma recursiva, el nodo que se coloca en el árbol de la interfaz. El método archivos solo busca entre las carpetas de salidas cada una de las coincidencias para html, json y png.

```
private void crear(File dir, DefaultMutableTreeNode nodo){
    File[] archivos =dir.listFiles(new FileFilter() {
        @Override
        public boolean accept(File pathname) {
            return pathname.getName().toLowerCase().endsWith(".png")
                    ||pathname.getName().toLowerCase().endsWith(".json")
                    || pathname.isDirectory();
        }
    }
    );
    if(archivos != null){
        int c = 0;
        for(File f:archivos){
            DefaultMutableTreeNode hijo = new DefaultMutableTreeNode(f.getName());
            modelo.insertNodeInto(hijo, nodo, c);
            c++;
            if(f.isDirectory()){
                crear(f, hijo);
            }
        }
    }
}
private File[] archivos(String ruta){
    File a = new File (ruta);
    File[] lista = a.listFiles(new FileFilter() {
        @Override
        public boolean accept(File pathname) {
            return pathname.getName().toLowerCase().endsWith(".png");
        }
    }
    );
```

El botón generar tiene la funcionalidad de generar los autómatas del programa por medio de una instancia de ambos tipos de analizadores, Si todo salió bien en la ejecución, se accede a las instancias y se extraen las listas con los autómatas, arboles, follows y Thompson. De ellos, se ejecuta su método de graficación correspondiente y se actualizan las listas de imágenes. En el mismo método, se crean también el html con los errores de la ejecución.