MANUAL
TÉCNICO

Damián Ignacio Peña Afre
202110568
Proyecto 1 - OLC1

CONTENIDO

DESCRIPCIÓN DEL PROYECTO

Sistema de análisis léxico y sintáctico que utiliza el Método del Árbol y el Método de Thompson para validar lexemas en expresiones regulares. Este sistema cuenta con una funcionalidad principal, un intérprete de expresiones regulares permitidas, que analiza un archivo de expresiones regulares para definir el patrón utilizado en el sistema y validar los lexemas.

DESCRIPCIÓN DE LA SOLUCIÓN

Como lenguaje principal se utilizó el lenguaje *Java*. Para la solución de análisis léxico y sintáctico en Java, se puede utilizar la herramienta JFlex para el análisis léxico y CUP para el análisis sintáctico. Estas herramientas permiten la generación automática del código Java para el análisis, simplificando así el proceso de desarrollo.

Una vez que se han generado los archivos de análisis léxico y sintáctico, se puede construir un árbol sintáctico abstracto (AST) utilizando los distintos recorridos (preorden, inorden, postorden) para la generación del Método del Árbol y Thompson. Estos métodos permiten la construcción de un *DFA* (autómata finito determinista) y *NDFA* (autómata finito no determinista) respectivamente. Cabe recalcar que esto se realiza a partir de expresiones regulares definidas en notación polaca.

Una vez que se ha generado el DFA utilizando el Método del Árbol, se puede utilizar para analizar la entrada y verificar si cumple con el patrón definido por la expresión regular.

En conclusión, para la solución de análisis léxico y sintáctico en Java, se puede utilizar JFlex y CUP para generar el código necesario. Luego, se pueden utilizar los distintos recorridos para construir un AST y generar el DFA utilizando el Método del Árbol o el Método de Thompson. Finalmente, se puede utilizar el DFA generado para analizar la entrada y verificar si cumple con el patrón definido por la expresión regular.

Arquitectura del compilador

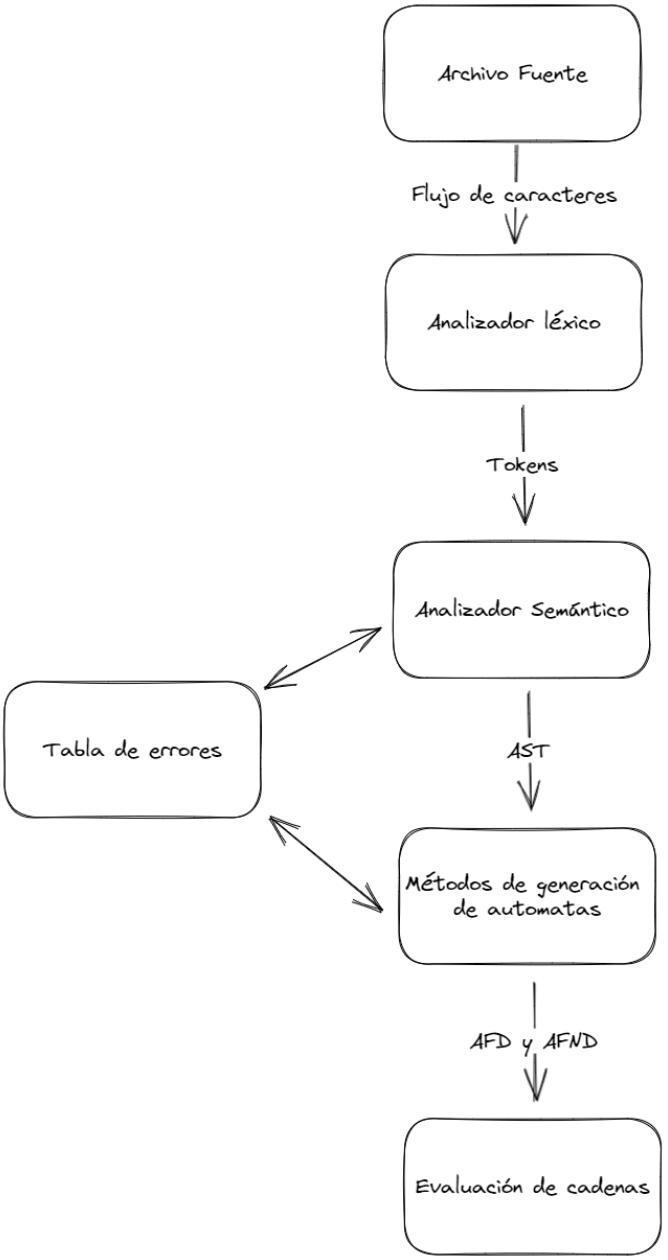
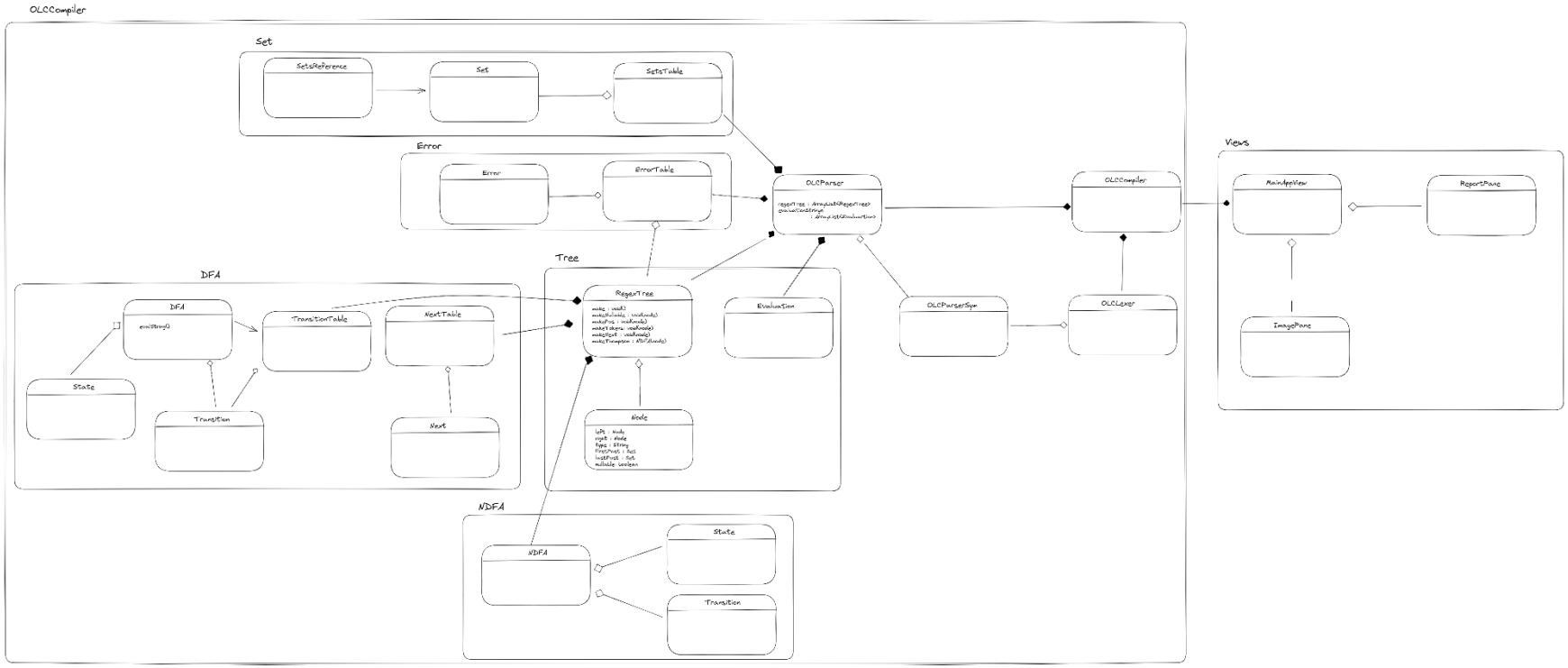


Diagrama de clases



ANALISIS LÉXICO

El análisis léxico es el primer paso en la construcción de un compilador. Su objetivo es escanear el código fuente y dividirlo en elementos de lenguaje conocidos, llamados tokens. Estos tokens son unidades atómicas de lenguaje que se utilizan en la construcción de estructuras de sintaxis más grandes.

El analizador léxico utiliza expresiones regulares para identificar patrones en el código fuente y generar los tokens correspondientes. También se definen algunos estados específicos para el análisis de elementos de conjunto y expresiones regulares.

Los tokens reconocidos son retornados al analizador sintáctico (parser) que posteriormente construirá un árbol sintáctico a partir de los tokens recibidos.

Hablando propiamente de las tecnologías utilizadas, en JFlex únicamente es necesario configurar las expresiones regulares que serán los patrones de tokens para posteriormente indicarle el tipo de símbolo o token que generará. Particularmente para integrarlo con el CUP es necesario utilizar una clase que es generada al realizar el parser.

Expresiones regulares

```
LineTerminator = \r|\n|\r\n

InputCharacter = [^\r\n]

WhiteSpace     = {LineTerminator} | [ \t\f]


Comment = {MultiLineComment} | {SingleLineComment}

MultiLineComment = "<!" [^*] ~"!>" | "<!" "!"+ ">"
```

```
SingleLineComment = "//" {InputCharacter}* {LineTerminator}?


Identifier = [a-zA-Z_][a-zA-Z0-9_]*

Digit = [0-9]

Lowercase = [a-z]

Uppercase = [A-Z]

Ascii = [\x20-\x2F\x3A-\x40\x5B-\x60\x7B-\x7E]

Word = \w+
```

Estados

Para "ayudar" al *lexer* a darle cierta prioridad a la generación de un determinado tipo de tokens se utilizaron los siguientes estados, a los que se va saltando dependiendo de los tokens que se reconocen. Esto no limita la detección de otros tokens, simplemente se coloca más arriba dentro del archivo de especificación para que tenga cierta precedencia en la generación.

```
%state STRING

%state SET

%state SET_ELEMENT

%state SET_OPERATOR

%state REGEX

%state REGEX_TEST

%state REGEX_EXPRESSION

%state REGEX_STRING

%state SET_REFERENCE
```

Tokens reconocidos:

Se presenta un resumen de los tokens reconocidos de manera general, en cualquier estado.

```
{Comment} { /* ignore */ }

"{" { return symbol(OLCParserSym.LBRACE, yytext()); }
"}" { return symbol(OLCParserSym.RBRACE, yytext()); }

":" { return symbol(OLCParserSym.COLON, yytext()); }

";" { return symbol(OLCParserSym.SEMICOLON, yytext()); }

"-" { return symbol(OLCParserSym.ARROW_TAIL, yytext()); }
">" { return symbol(OLCParserSym.ARROW_HEAD, yytext()); }

"->" { return symbol(OLCParserSym.ARROW, yytext()); }

"%%" { return symbol(OLCParserSym.SCOPE_BREAK, yytext()); }

"~" { return symbol(OLCParserSym.TILDE, yytext()); }

"," { return symbol(OLCParserSym.COMMA, yytext()); }

{Digit} { return symbol(OLCParserSym.DIGIT,
Integer.valueOf(yytext())); }
```

```
{Lowercase} { return symbol(OLCParserSym.LOWERCASE, yytext()); }

{Uppercase} { return symbol(OLCParserSym.UPPERCASE, yytext()); }

{Ascii} { return symbol(OLCParserSym.ASCII, yytext()); }

"." { return symbol(OLCParserSym.AND, yytext()); }
"|" { return symbol(OLCParserSym.OR, yytext()); }
"*" { return symbol(OLCParserSym.KLEENE, yytext()); }
"+" { return symbol(OLCParserSym.PLUS, yytext()); }
"?" { return symbol(OLCParserSym.QUESTION, yytext()); }

{Word} { return symbol(OLCParserSym.WORD, yytext()); }

\\n { return symbol(OLCParserSym.ESCAPED_LINEBREAK, "\n"); }
\\' { return symbol(OLCParserSym.ESCAPED_SINGLE_QUOTE, "\'"); }
\\\" { return symbol(OLCParserSym.ESCAPED_DOUBLE_QUOTE, "\""); }

[^] { return symbol(OLCParserSym.LEXICAL_ERROR, yytext()); }
```

ANALISIS SINTÁCTICO

El análisis sintáctico es una de las principales herramientas de la gramática que se utiliza para analizar la estructura de las oraciones. El objetivo del análisis sintáctico es identificar y describir las diferentes partes de la oración, así como las relaciones que existen entre ellas.

CUP también permite la definición de acciones semánticas que se ejecutan durante el análisis sintáctico. Estas acciones se pueden utilizar para construir los árboles de análisis sintáctico o para almacenar información útil para la siguiente fase del proceso de compilación.

En una gramática para un lenguaje de programación, los símbolos no terminales y terminales se utilizan para definir las reglas sintácticas del lenguaje. Los símbolos no terminales representan elementos gramaticales que pueden ser desglosados en términos más pequeños hasta llegar a los símbolos terminales. Los símbolos terminales representan elementos sintácticos individuales del lenguaje que no pueden ser descompuestos en términos más pequeños. Por ejemplo, en una gramática para un lenguaje de programación, los símbolos terminales pueden ser palabras clave, operadores aritméticos, operadores lógicos, nombres de variables, números, etc.

Símbolos terminales

SET_DECLARATION	IDENTIFIER	LBRACE
RBRACE	COLON	SEMICOLON
ARROW	SCOPE_BREAK	STRING_LITERAL
TILDE	COMMA	LOWERCASE
UPPERCASE	ASCII	AND
OR	KLEENE	PLUS
QUESTION	WORD	ESCAPED_LINEBREAK
ESCAPED_SINGLE_QUOTE	ESCAPED_DOUBLE_QUOTE	LEXICAL_ERROR
ARROW_TAIL	ARROW_HEAD	DIGIT
Integer		

Símbolos no terminales

program	scopes	decl_scope
decl	decls	set_decl
regex_decl	regex_expr	regex_exprs
stmt_scope	eval_stmt	eval_stmts
eval_stmts	escaped_sequence	set
compr_set	extend_set	set_element
set_elements	set_refence	regex_terminal
regex_term		

Gramática[1]

Para la solución es implementada una gramática libre del contexto con recursión para las múltiples declaraciones, ya sea de conjuntos o expresiones regulares.

Es oportuno mencionar que al definirse en notación polaca las expresiones regulares no es necesario definir una precedencia de operadores para las expresiones regulares.

[1] Archivo de especificación gramatical en los anexos

MANEJO DE ERRORES

Durante las etapas de parsing y generación de autómatas es posible añadir errores a la tabla de errores.

Primeramente, en el analisis léxico son generados tokens *ERROR* que el parser interpreta como errores léxicos.

Por otro lado, si no llegara a satisfacerse una regla gramatical se añadiría como un error sintáctico.

Por último, si mientras la generación se llegara a producir alguna excepción en la definición de algun elemento, en su referencia o en su tipo son añadidos como errores en tiempo de ejecución a la tabla de errores.

MÉTODO DEL ARBOL

El método del árbol para la generación de DFAs es un método algorítmico que se utiliza para construir autómatas finitos deterministas (DFA, por sus siglas en inglés) a partir de expresiones regulares.

El proceso comienza con la construcción de un árbol sintáctico para la expresión regular dada, donde cada nodo interno del árbol representa una operación de concatenación, unión o clausura de Kleene, y las hojas representan los símbolos del alfabeto.

Se puede resumir en los siguientes pasos:

- Cálculo de anulables
- Cálculo de primera posición
- Cálculo de última posición
- Tabla de siguientes
- Tabla de transiciones

Para conseguir toda esta secuencia, son llamados métodos recursivos que recorren el AST teniendo en cuenta resultados de las hojas, o lo que es lo mismo, recórrelo en postorden. Se presentan los métodos propios dentro del árbol que define la expresión regular:

Método inicializador de la generación

```
public void make() {

        this.nextTable = new NextTable(this.name);
        this.tokens = new HashMap<Integer, Object>();
        this.terminals = new ArrayList<String>();

        makeAnullable(this.rootNode);
        makePos(this.rootNode);
        makeTokens(this.rootNode);
        makeNext(this.rootNode);

        this.transitionTable = new TransitionTable(this.nextTable,
this.rootNode.firstPos, this.nextTable.getAcceptanceNode(),
this.tokens, this.terminals, this.name);
        this.dfa = new DFA(this.transitionTable.transitions,
this.transitionTable.states, this.name);

        this.ndfa = this.makeThompson(this.rootNode.left);
        this.ndfa.finalState.isAcceptace = true;
        this.ndfa.name = this.name;

        this.generateReports();
    }
```

Método para calcular la anulabilidad

```
private void makeAnullable(Node node) {
        if (node != null) {

            // POSTORDER RECURSIVE CALLS
            if (node.left != null) {
                makeAnullable(node.left);
            }

            if (node.right != null) {
                makeAnullable(node.right);
            }

            // ANULABLE CONDITIONS
            if (node.type.equals(NodeType.NODE_I) ||
node.type.equals(NodeType.NODE_ACCEPT)) {
                node.nullable = false;
            } else if (node.type.equals(NodeType.NODE_OR)) {
                node.nullable = node.left.nullable ||
node.right.nullable;
            } else if (node.type.equals(NodeType.NODE_AND)) {
                node.nullable = node.left.nullable &&
node.right.nullable;
            } else if (node.type.equals(NodeType.NODE_KLEENE)) {
                node.nullable = true;
            } else if (node.type.equals(NodeType.NODE_PLUS)) {
                node.nullable = node.left.nullable;
            } else if (node.type.equals(NodeType.NODE_OPTIONAL)) {
                node.nullable = true;
            } else {
                throw new RuntimeException("Node type not found for
makeAnulable");
            }
        }
    }
```

Método para calcular la primera y última posición

```
private void makePos(Node node) {
        if (node != null) {

            // POSTORDER RECURSIVE CALLS
            if (node.left != null) {
                makePos(node.left);
            }

            if (node.right != null) {
                makePos(node.right);
            }

            // POS CONDITIONS
            if (node.type.equals(NodeType.NODE_I)) {
                node.firstPos.add(node.number);
                node.lastPos.add(node.number);
            } else if (node.type.equals(NodeType.NODE_OR)) {
                node.firstPos.addAll(node.left.firstPos);
                node.firstPos.addAll(node.right.firstPos);
                node.lastPos.addAll(node.left.lastPos);
                node.lastPos.addAll(node.right.lastPos);
            } else if (node.type.equals(NodeType.NODE_AND)) {
                if (node.left.nullable) {
                    node.firstPos.addAll(node.left.firstPos);
                    node.firstPos.addAll(node.right.firstPos);
                } else {
                    node.firstPos.addAll(node.left.firstPos);
                }
                if (node.right.nullable) {
                    node.lastPos.addAll(node.left.lastPos);
                    node.lastPos.addAll(node.right.lastPos);
                } else {
                    node.lastPos.addAll(node.right.lastPos);
                }
            } else if (node.type.equals(NodeType.NODE_KLEENE) ||
node.type.equals(NodeType.NODE_PLUS) ||
node.type.equals(NodeType.NODE_OPTIONAL)) {
                node.firstPos.addAll(node.left.firstPos);
                node.lastPos.addAll(node.left.lastPos);
            }else if (node.type.equals(NodeType.NODE_ACCEPT)) {
                node.firstPos.add(node.number);
                node.lastPos.add(node.number);
            }else {
                throw new RuntimeException("Node type not found for
POS");
            }
        }
    }
```

Método para calcular siguientes

```
private void makeNext(Node node) {
        if (node != null) {

            // NEXT CONDITIONS, ONLY FOR AND, PLUS AND KLEENE
            if (node.type.equals(NodeType.NODE_AND) && (node.left !=
null) && (node.right != null)) {
                for (Integer i : node.left.lastPos) {
                    this.nextTable.addNext(i, this.tokens.get(i),
new HashSet<>(node.right.firstPos));
                }
            }

            if ((node.type.equals(NodeType.NODE_KLEENE) ||
node.type.equals(NodeType.NODE_PLUS)) && (node.left != null)) {
                for (Integer i : node.left.lastPos) {
                    this.nextTable.addNext(i, this.tokens.get(i),
node.left.firstPos);
                }
            }

            if (node.type.equals(NodeType.NODE_ACCEPT)) {
                this.nextTable.addNext(node.number,
this.tokens.get(node.number), null);
            }

            // ? POSTORDER RECURSIVE CALLS
            if (node.left != null) {
                makeNext(node.left);
            }

            if (node.right != null) {
                makeNext(node.right);
            }

        }
    }
```

Generación de la tabla de transiciones

Este método es propio de su clase y toma calculos posteriores dentro del árbol así como el estado inicial que es obtenido de la raiz.

Primero determina si debe crear un nuevo estado a partir del conjunto de siguientes introducido, luego procede a encontrar las transiciones y los estados relacionados con el terminal que define el nodo numerado.

```
private State makeNode(Set<Integer> nextSet){

        State state = this.evalCreateNewState(nextSet);

        if (state == null) {
            return null;
        }


        // FIND ASSOC TRANSITIONS WITH TOKENS

        //    token : nextSet
        Map<Object, Set<Integer>> assocTransitions = new HashMap<>();
        for (Integer next: nextSet) {
            if (next.equals(this.acceptanceNode)) {
                continue;
            }

            Object t = this.tokens.get(next);
            if(assocTransitions.containsKey(t)) {
                assocTransitions.get(t).addAll(new
HashSet<Integer>(this.nextTable.getNext(next).next));
            }else{
                assocTransitions.put(t, new
HashSet<Integer>(this.nextTable.getNext(next).next));
            }
        }

        // CREATE TRANSITIONS
        for (Map.Entry<Object, Set<Integer>> entry:
assocTransitions.entrySet()) {
            State nextState =
this.evalCreateNewState(entry.getValue());
            if (nextState != null) {
                this.transitions.add(new Transition(state, nextState,
entry.getKey()));
            }
```

```
        }

        // RECURSIVE CALL

        state.marked = true;

        for (State notMakedStates : this.states) {
            if (!notMakedStates.marked) {
                this.makeNode(notMakedStates.nextSet);
                break;
            }
        }

        return state;
    }
```

```
private State evalCreateNewState(Set<Integer> nextSet){

        if (nextSet == null) {
            return null;
        }

        for (State state: this.states) {
            if (state.nextSet.equals(nextSet)) {
                return state;
            }
        }
        this.statesCounter++;
        State newState =  new State(this.statesCounter, nextSet);
        newState.setAcceptace(nextSet.contains(this.acceptanceNode));
        this.states.add(newState);
        return newState;
    }
```

Es pasada entonces la lista de estados y sus respectivas transiciones al AFD para que este pueda reconocer cadenas.

MÉTODO DE THOMPSON

Aprovechando el método inicializador son llamadas las reglas para generar el AFND con el método de Thompson. De manera resumida este realiza pequeños AFND's que con capaces de concatenarse o unirse con otros autómatas dependiendo de la operación padre que los envuelva.

```
private NDFA makeThompson(Node node) {

        // POST-ORDER RECURSIVE CALLS

        NDFA mainNdfa= new NDFA();
        NDFA leftNdfa = null;
        NDFA rightNdfa = null;

        if (node.left != null) {
            leftNdfa = makeThompson(node.left);
        }

        if (node.right != null) {
            rightNdfa = makeThompson(node.right);
        }

        // THOMPSON CONDITIONS

        if (node.type.equals(NodeType.NODE_I)) {
            mainNdfa.nodei(this.tokens.get(node.number));
            return mainNdfa;
        }else if (node.type.equals(NodeType.NODE_AND)) {
            mainNdfa.concat(leftNdfa, rightNdfa);
            return mainNdfa;
        }else if(node.type.equals(NodeType.NODE_OR)) {
            mainNdfa.union(leftNdfa, rightNdfa);
            return mainNdfa;
        }else if(node.type.equals(NodeType.NODE_KLEENE)) {
            mainNdfa.kleene(leftNdfa);
            return mainNdfa;
        }else if(node.type.equals(NodeType.NODE_PLUS)) {
            mainNdfa.plus(leftNdfa);
            return mainNdfa;
        }else if(node.type.equals(NodeType.NODE_OPTIONAL)) {
            mainNdfa.optional(leftNdfa);
            return mainNdfa;
        }

        return null;
    }
```

EVALUACIÓN DE CADENAS

Para evaluar una cadena se parte del estado inicial del DFA y se recorren todos los caracteres de una cadena. Es posible que en vez de ser un carácter terminal se tenga que evaluar la referencia a un conjunto, por lo que son obtenidos los elementos del mismo y comparados.

Recorrido de caracteres

```
public boolean evalString (String evalStr){

        // Get the initial state
        State currentState = this.states.get(0);

        for (char evalChar: evalStr.toCharArray()) {

            // Get the next state with the current eval char
            currentState = this.getNextState(currentState,
evalChar);

            // If the next state is null, the string is not accepted
            if (currentState == null) return false;
        }

        // If the current state is an acceptance state (After
reading the whole string), the string is accepted
        if (currentState.isAcceptace) return true;

        return false;
    }
```

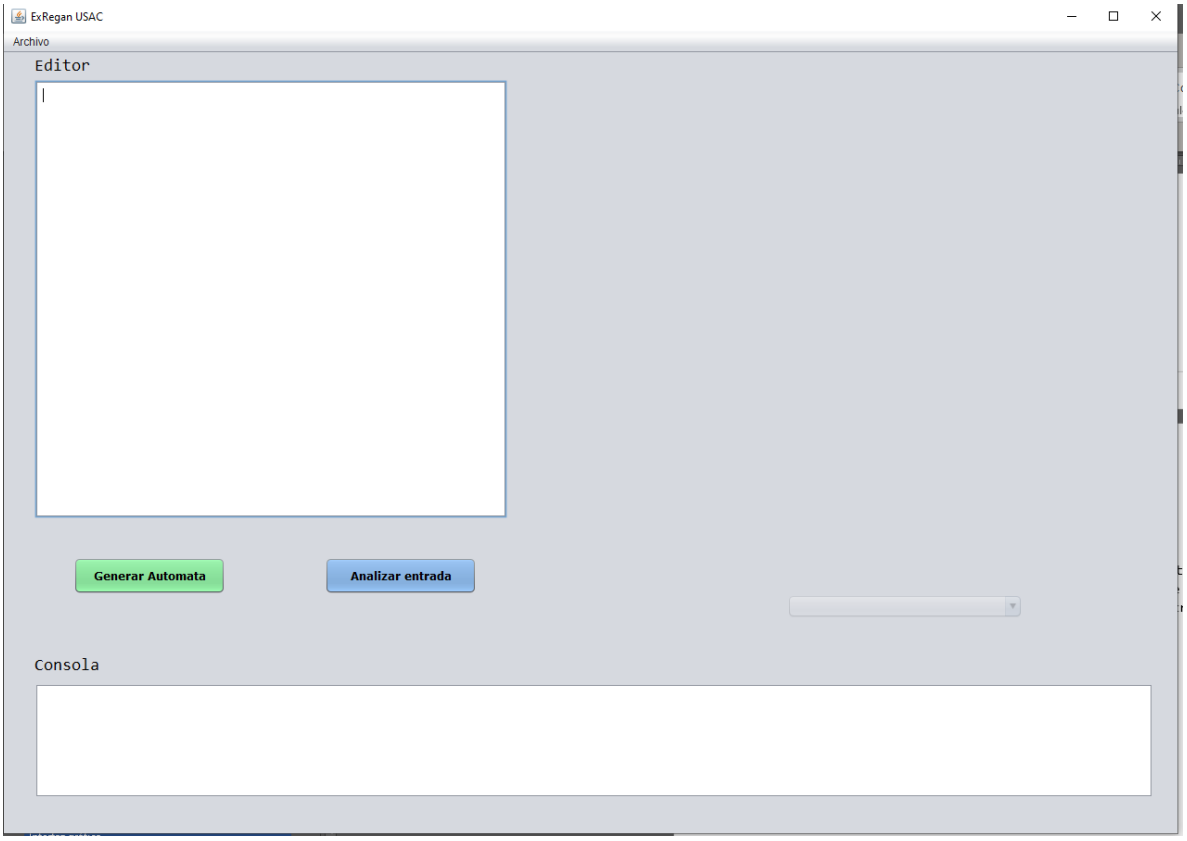
Puede llegar al caso de no encontrar alguna transición para dicho carácter, en tal caso la cadena no sería valida. También puede darse el caso de no haber llegado a un estado de aceptación si bien se han recorrido todos los caracteres, de igual forma la cadena sería tomada como invalida.

Obtención del nuevo estado

```
    public State getNextState(State state, char evalChar){
        for (Transition t: this.transitions) {
            if (t.prevState == state){
                if(t.token instanceof String &&
t.token.toString().equals(String.valueOf(evalChar))) return
t.nextState;
                if(t.token instanceof SetReference &&
((SetReference)
t.token).getSet().getElements().contains(String.valueOf(evalChar)))
return t.nextState;
            }
        }
        return null;
    }
```

INTERFAZ GRÁFICA

Para la interfaz gráfica se utilizó *Swing*. Se introdujo un área de texto para la colocación del código fuente y los botones correspondientes para disparar ciertas etapas dentro del compilador.



ANEXOS

Especificación gramatical

```
PROGRAM ::= <LBRACE> SCOPES <RBRACE>
SCOPES ::= DECL_SCOPE <SCOPE_BREAK> <SCOPE_BREAK> STMT_SCOPE
DECL_SCOPE ::= DECL DECLS
DECL ::= SET_DECL | REGEX_DECL
DECLS ::= DECL DECLS | ε


SET_DECL ::= <SET_DECLARATION> <COLON> <ID> <ARROW> SET <SEMICOLON>
SET ::= COMPR_SET | SET_ELEMENTS
COMPR_SET ::= SET_ELEMENT <TILDE> SET_ELEMENT
SET_ELEMENT ::= <ASCII> | <UPPERCASE> | <LOWERCASE> | <DIGIT>
EXTEND_SET ::= SET_ELEMENT SET_ELEMENTS
SET_ELEMENTS ::= <COMMA> SET_ELEMENT | ε

REGEX_DECL ::=  <ID> <ARROW> REGEX_EXPR
REGEX_EXPR ::= REGEX_TERM REGEX_EXPRS <SEMICOLON>
REGEX_EXPRS ::= REGEX_TERM REGEX_EXPRS | ε // <- !!!
REGEX_TERM ::= <AND> REGEX_TERM REGEX_TERM
               | <OR> REGEX_TERM REGEX_TERM
               | <KLEENE> REGEX_TERM
               | <PLUS> REGEX_TERM
               | <QUESTION> REGEX_TERM
               | REGEX_TERMINAL

REGEX_TERMINAL ::= <LETTER>
```

```
                    | <DIGIT>
                    | <ESCAPED>
                    | <STRING >
                    | SET_REFERENCE
SET_REFERENCE ::= <LBRACE> <ID> <RBRACE>

STMT_SCOPE ::= EVAL_STMT EVAL_STMTS
EVAL_STMT ::= <ID> <COLON> <STRING> <SEMICOLON>
EVAL_STMTS ::= EVAL_STMT | ε
```