UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
INE5421 - Linguagens Formais e Compiladores

Trabalho Prático I

Relatório

Anna Victoria Oikawa
Matteus Legat

Software utilizado

OpenJDK:

```
openjdk version "1.8.0_144"
OpenJDK Runtime Environment (build 1.8.0_144-b01)
OpenJDK 64-Bit Server VM (build 25.144-b01, mixed mode)
```

Eclipse IDE:

```
Eclipse IDE for Java Developers
Version: Oxygen.1a Release (4.7.1a)
Build id: 20171005-1200
```

JUnit:

```
JUnit5 5.0.0v20170910-2246
```

WindowBuilder:

```
Eclipse WindowBuilder 1.9.1.201710100405
```

ASCIITable:

```
btc-ascii-table-1.0
https://code.google.com/archive/p/java-ascii-table/
```

Implementação

Linguagem Regular

Uma vez que uma gramática regular, um autômato finito e uma expressão regular representam uma linguagem regular, a abordagem seguida na implementação foi a criação de uma **classe base** *RegularLanguage*. Logo, as classes *FiniteAutomata*, *RegularGrammar* e *RegularExpression* **extendem da classe base RegularLanguage.**

```
public class RegularExpression extends RegularLanguage {

public class RegularGrammar extends RegularLanguage {

public class FiniteAutomata extends RegularLanguage {
```

```
/**
 * Representation of a Regular Language
 *
 */
public abstract class RegularLanguage {

    // RL can be represented by a Regular Expression, a Regular Grammar or a Finite Automata
    public enum InputType {RE, RG, FA, UNDEFINED};

    // Operations allowed in a RL
    public enum Operation {UNION, CONCATENATION, INTERSECTION, DIFFERENCE};

    protected String input; // The input entered by the user
    private String id; // an unique ID for the RL
    private InputType type = InputType.UNDEFINED; // the type that the RL is represented
```

Os principais atributos da classe são:

- **InputType**: os tipos permitidos que representam uma linguagem regular.
- **Operation**: as operações que podem ser realizadas.
- **input**: a entrada inserida pelo usuário.
- **id**: um identificador único para a linguagem.
- **type**: o tipo (ER, GR, AF)

A classe contém métodos abstratos para que suas subclasses implemente-os.

```java
/**
 * Return the String representation of the RL
 * in the format according to its type
 * @return the RL String representation
 */
public abstract String getDefinition();

/**
 * Return a Regular Grammar object
 * @return the RG object
 */
public abstract RegularGrammar getRG();
/**
 * Return a Regular Expression Object
 * @return the RE object
 */
public abstract RegularExpression getRE();
/**
 * The Finite Automata Object
 * @return the automata object
 */
public abstract FiniteAutomata getFA();
```

Assim, **getDefinition()** por exemplo, retornará a representação da linguagem regular de acordo com cada tipo (ER, AF ou GR).

A seguir, será discutido com mais detalhes a implementação de cada uma das subclasses.

Autômato Finito

O autômato finito é implementado pela classe FiniteAutomata do pacote RegularLanguages. Todos os seus atributos são imutáveis, portanto as suas instâncias também são imutáveis.

```java
/**
 * Immutable representation of a Finite Automata
 * Instance must be created using a FiniteAutomata.FABuilder instance
 *
 */
public class FiniteAutomata extends RegularLanguage {

    private final UUID uuid;
    private final State initial;
    private final SortedSet<State> states;
    private final SortedSet<State> finals;
    private final SortedSet<Character> alphabet;
    private final Map<TransitionInput, SortedSet<State>> transitions;
```

A classe possui 6 atributos principais que definem o autômato:

- **uuid:** identificador único utilizado em operações entre autômatos.
- **states:** representa o conjunto de estados do autômato.
- **initial:** representa o estado inicial do autômato.
- **finals:** representa o conjunto de estados finais do autômato.
- **alphabet:** representa o conjunto dos símbolos aceitos pelo autômato.
- **transitions:** representa o mapa de transições do autômato, mapeando de uma entrada (par de estado e símbolo) para um conjunto de estados.

Como as instâncias de autômatos são imutáveis, para a construção de um novo autômato foi utilizado o padrão Builder. Assim, o construtor da classe é privado e para criar uma nova instância, devemos utilizar um construtor de autômato finito, representado pela classe interna FABuilder.

A classe FABuilder possui métodos que permitem a criação de novos estados, importação de estados de outros autômatos, criação de novas transições, e definição de estados como finais ou iniciais.

Exemplo do uso de um construtor para se obter um autômato:

```
FABuilder builder = new FABuilder();  // instancia novo construtor
State[] q = new State[5];  // armazena os estados do novo automato
for (int i = 0; i < 5; i++) {
    q[i] = builder.newState();  // cria novo estado
}

FiniteAutomata fa = builder.setInitial(q[0])    // define q[0] como inicial
        .setFinal(q[4])                         // define q[4] como final
        .addTransition(q[0], 'a', q[1])         // transição de q[0] por 'a' leva para q[1]
        .addTransition(q[0], 'b', q[0])         // transição de q[0] por 'b' leva para q[0]
        .addTransition(q[1], 'b', q[2])         // transição de q[1] por 'b' leva para q[2]
        .addTransition(q[2], 'c', q[3])         // transição de q[2] por 'c' leva para q[3]
        .addTransition(q[4], 'd', q[4])         // transição de q[4] por 'a' leva para q[4]
        .build();                               // termina construção do novo AF
```

As operações realizadas com autômatos finitos são executadas pela classe estática FAOperator, do pacote RegularLanguages.Operators.

Esta classe implementa as seguintes operações:

determinize(FA):

```
/**
 * Determinizes NDFA to DFA
 * @return Determinized FA version
 */
public static FiniteAutomata determinize(FiniteAutomata fa) {
```

Método implementado conforme o algoritmo visto em aula para a transformação de um AFND em um AFD. Recebe um AF como entrada e retorna um novo AFD equivalente.

removeStates(FA, conjunto<Estado>):

```
/**
 * Remove states from AF
 * Obs. initial state is never removed
 * @param states Set of states to be removed
 * @return copy of this AF without states received
 * @throws InvalidStateException
 */
public static FiniteAutomata removeStates(FiniteAutomata fa, Set<State> rmStates)
```

Método que remove um conjunto de estados de um autômato finito. Utilizado na remoção de estados mortos e inalcançáveis.

getUnreachableStates(FA):

```
/**
 * Get unreachable states
 * @return Set of unreachable states
 */
public static Set<State> getUnreachableStates(FiniteAutomata fa) {
```

Retorna conjunto de estados inalcançáveis conforme algoritmo visto em aula.

getDeadStates(FA):

```
/**
 * Get dead states
 * @return Set of dead states
 */
public static Set<State> getDeadStates(FiniteAutomata fa) {
```

Retorna conjunto de estados mortos conforme algoritmo visto em aula.

minimize(FA):

```
/**
 * Minimizes FA
 * @return Minimized FA version
 */
public static FiniteAutomata minimize(FiniteAutomata fa) {
```

Retorna um novo AFD mínimo equivalente ao AF recebido. Primeiro o AF é determinizado, então são removidos os estados mortos e inalcançáveis, e logo após são criadas as classes de equivalência conforme algoritmo visto em aula. Para cada classe de equivalência, um estado no novo AF é criado junto com as transições de acordo com o AF original.

union(FA1, FA2):

```
/**
 * Get FA equivalent to the union of another two
 * @param fa1 FA1
 * @param fa2 FA2
 * @return New FA = FA1 union FA2
 */
public static FiniteAutomata union(FiniteAutomata fa1, FiniteAutomata fa2) {
```

Retorna um novo AF equivalente à união dos AFs recebidos.

Para isto, é criado um novo estado inicial (que também é final se algum dos AFs aceita &) e todos os estados dos AFs recebidos são copiados para o novo. Os estados que eram finais continuam sendo finais, as transições são todas copiadas, e as transições que partiam de um estado que era inicial são copiadas novamente, desta vez partindo do novo estado inicial.

complement(FA, alfabeto):

```
/**
 * Get FA equivalent to the complement of another
 * @param fa FA
 * @param alphabet New alphabet symbols
 * @return New FA = complement(FA)
 */
public static FiniteAutomata complement(FiniteAutomata fa, Set<Character> alphabet) {
```

Retorna um novo AF equivalente ao complemento do AF recebido, considerando também os novos símbolos recebidos como parte de seu alfabeto.

Para isto, o AF é determinizado, então é criado um novo estado morto, todas as transições indefinidas passam a levar para o novo estado morto, os estados que eram finais deixam de ser, e os que não eram passam a ser.

intersection(FA1, FA2):

```
/**
 * Get FA equivalent to the intersection of another two
 * @param fa1 FA1
 * @param fa2 FA2
 * @return New FA = FA1 intersection FA2
 */
public static FiniteAutomata intersection(FiniteAutomata fa1, FiniteAutomata fa2)
```

Retorna um novo AF equivalente à interseção dos AFs recebidos. Implementado utilizando os métodos de complemento e união, pois de acordo com as propriedades básicas, $L1 \cap L2 = \overline{\overline{L1} \cup \overline{L2}}$

intersectionSteps(FA1, FA2):

```
/**
 * Get one FA for each step of the intersection of another two
 * @param fa1 FA1
 * @param fa2 FA2
 * @return Map of FAs =
 * {"C1" : complement(FA1), "C2": complement(FA2), "U": union("C1", "C2"), "I": intersection("U")}
 */
public static Map<String, FiniteAutomata> intersectionSteps(FiniteAutomata fa1, FiniteAutomata fa2)
```

Retorna um grupo de AFs relacionados a cada passo do cálculo da interseção:

- $\overline{L1}$
- $\overline{L2}$
- $\overline{L1} \cup \overline{L2}$
- $L1 \cap L2$

isSubset(FA1, FA2):

```
/**
 * Check if the language accepted by FA1 is a subset of the language accepted by FA2
 * @param fa1 FA1
 * @param fa2 FA2
 * @return whether L(FA1) is a subset of L(FA2)
 */
public static boolean isSubset(FiniteAutomata fa1, FiniteAutomata fa2) {
    return isEmptyLanguage(intersection(fa1, complement(fa2, fa1.getAlphabet())));
}
```

verifica se a linguagem de FA1 está contida em FA2, fazendo uso da característica $L1 \subseteq L2 \Leftrightarrow L1 \cap \overline{L2} = \phi$

isEquivalent(FA1, FA2):

verifica se a linguagem de FA1 é equivalente à de FA2, fazendo uso da característica $L1 = L2 \Leftrightarrow L1 \subseteq L2 \wedge L2 \subseteq L1$

isInfiniteLanguage(FA):

```java
/**
 * Check if the language accepted by an FA is infinite
 * @param fa FA to be verified
 * @return boolean whether it is infinite
 */
public static boolean isInfiniteLanguage(FiniteAutomata fa) {
    FiniteAutomata faMin = minimize(fa);
    return isCyclic(faMin, faMin.getInitial(), new HashSet<State>());
}
```

Verifica se a linguagem gerada por um AF é infinita. Para isto, é realizada uma busca por ciclos no AFD mínimo equivalente.

isCyclic(FA):

```java
/**
 * Recursive function to check if a FA has cycles
 * Called to check a language finiteness
 * @param fa FA to be verified
 * @param currentState state being verified
 * @param visited States visited on the current path
 * @return boolean whether it is infinite
 */
private static boolean isCyclic(FiniteAutomata fa, State currentState, Set<State> visited)
```

Função recursiva que realiza busca em profundidade a partir do inicial, marcando estados como visitados e retornando verdadeiro quando um ciclo é encontrado (alcança estado já visitado).

getTransitionsTable(FA):

```java
/**
 * Get FA formatted transitions table
 * @param fa FA to be formatted
 * @return Transitions table formatted as String
 */
public static String getTransitionsTable(FiniteAutomata fa) {
```

Função que retorna a tabela de transições no formato de String, formatada de maneira legível com o uso da biblioteca ASCIITable.

isDeterministic(FA):

```java
/**
 * Check if FA is deterministic
 * @return boolean whether it is deterministic
 */
public static boolean isDeterministic(FiniteAutomata fa) {
    // Check if any transition output has size greater than 1
    return fa.getTransitions().values().stream().noneMatch(ts -> ts.size() > 1);
}
```

Método que verifica se um AF é determinístico, verificando se nenhuma transição leva para mais de um estado.

FAtoRG(FA):

```java
/**
 * Convert FA to RG
 * @param fa FA
 * @return regular grammar equivalent to FA
 */
public static RegularGrammar FAtoRG(FiniteAutomata fa) {
```

Método que retorna uma gramática regular equivalente ao autômato recebido. A nova gramática é construída com base no algoritmo visto em aula.

RGtoFA(RG):

```java
/**
 * Convert RG to FA
 * @param rg Regular Grammar
 * @return Equivalent Finite Automata
 */
public static FiniteAutomata RGtoFA(RegularGrammar rg) {
```

Método que retorna um autômato finito equivalente à gramática regular recebida. O novo autômato é construído com base no algoritmo visto em aula.

Gramática Regular

```java
/**
 * Representation of a Regular Grammar
 * Eg.: S -> aA | a
 *
 */
public class RegularGrammar extends RegularLanguage {

    private HashSet<Character> vn;  // non terminal symbols
    private HashSet<Character> vt;  // terminal symbols
    private HashMap<Character, HashSet<String>> productions;    // production rules S -> aA | a
    private char s; // initial S
    private static Scanner prodScan;
```

A classe de Gramática Regular possui quatro atributos principais que definem formalmente uma gramática:

- **vn:** representa os símbolos não terminais que pertencem a gramática G.
- **vt:** representa os símbolos terminais que pertencem a G.
- **productions:** regras de produção de G. Contém as produções para cada símbolo terminal *vn*.
- **s:** representa o símbolo inicial da gramática G.

Para a construção de um objeto desta classe, é necessário que o usuário entre com as regras de produção de uma gramática regular da forma: **S -> aA | a**. Após a inserção da entrada, a função **isValidRG(String inp)** é invocada, realizando todas as verificações necessárias para validar a string de entrada.

```java
/**
 * Verify if a given input is a valid RG
 * @param inp the input entered by the user
 * @return a Regular Grammar if valid, null if invalid
 */
public static RegularGrammar isValidRG(String inp) {
```

Assim, uma análise léxica é realizada para checar se todos os símbolos utilizados são permitidos no contexto de uma gramática regular. São exemplos de símbolos válidos:

- Letras e dígitos.
- | - > &

Se todos os símbolos são válidos, é verificado se os formatos das produções (*aS* | *a*) estão de acordo com aqueles permitidos para uma GR. Isso é feito pelo método **validateProductions:**

```
/**
 * Verify if every non terminal symbol vN
 * is valid in the context of a Regular Grammar
 * @param nt array with every non terminal of the grammar
 * @param rg the RegularGrammar object
 * @return null it is not valid, or a RegularGrammar object if valid
 */
private static RegularGrammar validateProductions(String[] nt, RegularGrammar rg) {
```

O método verifica se cada símbolo não terminal (S -> …) está no formato válido para uma GR. Se sim, todas as produções de S serão validadas pelo método **validateProduction:**

```
/**
 * Verify if every production of a vN is valid
 * @param vn the non terminal symbol the productions belong to
 * @param productions the productions of a given non terminal
 * @param prodList the set of all the productions from the grammar
 * @param rg the Regular Grammar object
 * @return true if all productions are valid
 */
private static boolean validateProduction(Character vn, String productions,
        HashSet<String> prodList, RegularGrammar rg) {
```

Assim, para cada vN que pertence a G, cada uma de suas produções será analisada para garantir que está de acordo com o formato de uma Gramática Regular. São feitas verificações como:

- A -> aA | A ∈ vN ∧ a ∈ vt ∧ |A| = 1
- Produções da forma A -> Aa, A -> aa, A -> aAa, A -> B não são permitidas.
- Outras produções proibidas para uma GR.

Portanto, a principal função desta classe é verificar a entrada digitada pelo usuário e garantir que de fato é uma Gramática Regular válida. Em caso positivo, será criado com sucesso um objeto que representa uma GR. Em caso negativo, o usuário será informado que o formato inserido não é válido e será necessário entrar com outra gramática.

Expressão Regular

Os principais atributos que compõem a classe de Expressão regular são:

```
/**
 * Representation of a Regular Expression
 * Eg.: a*(b?c|d)*
 */
public class RegularExpression extends RegularLanguage {
    private HashSet<Character> vt;  // terminal symbols
    private String regex; // the regular expression entered by the user
    private String formatted_regex; // an equivalent simplified representation of the regex
```

- **vt**: o conjunto de símbolos terminais que pertencem a expressão regular inserida.
- **regex**: a expressão regular digitada pelo usuário.
- **formatted_regex**: expressão regular equivalente a de entrada, mas em um formato simplificado (a** ≡ a*). Usada apenas para manipular operações.

Assim como na classe de Gramática Regular, é realizado uma verificação de entrada digitada pelo usuário no método **isValidRE(String inp):**

```
/**
 *
 * Verify if a given input is a valid RE
 * @param inp the input entered by the user
 * @return a RegularExpression object if valid, null if invalid
 */
public static RegularExpression isValidRE(String inp) {
```

Ou seja, uma análise léxica é realizada para garantir que os símbolos inseridos são permitidos no contexto de uma ER. Isso é verificado na função **lexicalValidation(String inp, RegularExpression re)**, que garante que os símbolos inseridos são permitidos, como é mostrado na figura a seguir:

```java
/**
 * Validate if symbols in the RE are lexically valid
 * @param str the input entered by the user
 * @return true if lexically valid, or false otherwise
 */
public static boolean lexicalValidation(String inp, RegularExpression re) {
    if (!inp.matches("^[a-z0-9\\(\\)\\?\\*|&\\+]*")) { // Verify invalid symbols
        return false;
    }
    return true;
}
```

Uma análise sintática também é feita uma vez que é preciso garantir que a estrutura e a combinação dos símbolos são válidas (**|*** por exemplo, não é válido).

A verificação sintática é feita pela função **syntaticAnalysis**:

```java
/**
 * Verify if the combination of symbols in the
 * regular expression is syntactically valid
 * @param inp the RE entered by the user
 * @param re the RE object
 * @return true if syntactically correct
 */
public static boolean syntaticAnalysis(String inp, RegularExpression re) {
```

Assim, se a expressão regular digitada pelo usuário for válida após a realização de todas as análises, então obtém-se um objeto que representa uma Expressão Regular.

Di Simone

A classe DiSimone é responsável por realizar todas as operações para obter um autômato finito a partir de uma expressão regular, de acordo com o algoritmo visto em aula.

Os principais atributos da classe são:

- **root**: a raiz da árvore da ER.
- **regex**: a expressão regular que a árvore representa
- **postOrderRegex**: uma representação da ER em *notação polonesa reversa* para auxílio da criação da árvore.

- **nTerminals**: quantidade de terminais que a ER contém, usado para a numeração dos nodos folhas.
- **automata**: o autômato equivalente à ER, obtido a partir da árvore de expressão.

```java
/**
 * Class that represents the DiSimone method to
 * transform a regular expression into a
 * finite automata by constructing the expression
 * threaded tree
 */
public class DiSimone {

    private Node root; // root of the tree
    private String regex; // regex that it represents
    private String postOrderRegex; // regex in postfix notation
    private int nTerminals = 0; // quantity of terminals
    private FiniteAutomata automata; // the equivalent automata
```

A árvore é composta por nodos que podem conter um símbolo terminal (letra minúscula ou dígito) ou um operador (*** | ? + ( )**). Logo, é necessário uma classe Node para essa representação. Ela é composta por:

- **left**: o filho da esquerda, caso tenha.
- **right**: o filho da direita (no caso de operadores binários), ou a costura (no caso de nodos com costura, dito pelo boolean isThreaded).
- **data**: o dado do nodo (terminal ou operador).
- **isThreaded**: boolean que informa se o nodo possui costura.
- **nodeNumber**: numeração do nodo folha, caso seja um.

```java
/**
 * Class that represents a node in the three
 */
public class Node {
    public Node left; // left node
    public Node right; // right node
    public char data; // node data
    boolean isThreaded; // If right pointer is a normal right pointer or a pointer to inorder successor.
    public int nodeNumber; // number of terminal leaf node
```

Assim, o processo de obtenção da árvore é feito da seguinte forma:

Dado uma expressão regular inserida pelo usuário, será realizado todas as verificações pela classe RegularExpression, garantindo que a ER é de fato válida.

Se o resultado for positivo, então a ER inserida será convertida para uma notação polonesa reversa pela função **getPostOrder(String r)**:

```
/**
 * Transforms Regular Expression to
 * Reverse Polish Notation
 * @param r the regex
 * @return the regex converted to reverse polish notation
 */
private String getPostOrder(String r) {
```

Ou seja, para a ER (a.b)*, sua representação em notação reversa será ab.* e é visivelmente trivial para a criação da árvore. O nodo raiz se encontra mais à direita, seguido pelo seu filho e pelos nodos folhas a e b.

A função **precedence(char c)** informa a precedência de um dado operador, usado durante o processo de criação da árvore:

```
/**
 * Get the priority of an operator
 * @param c the operator
 * @return the operator priority
 */
public static int precedence(char c) {
    switch (c) {
    case '|':
        return 1;
    case '.':
        return 2;
    case '*':
    case '?':
    case '+':
        return 3;

    }
    return -1;

}
```

Assim, a criação da árvore propriamente dita será feita pelo método **createTree(char postfix[])** que realizará os passos para montar uma árvore binária a partir da notação reversa da expressão regular:

```
/**
 * Create a DiSimone Tree
 * @param postfix the regex in postfix order
 * @return the root of the tree
 */
public Node createTree(char postfix[]) {
```

Para a realização das costuras dos nodos, a abordagem utilizada foi a conversão da árvore binária comum para uma árvore binária com costura (**binary threaded tree**). Ou seja, onde cada nodo aponta para sua costura, caso ele possua uma.

Esse trabalho é realizado pelo método **makeThreadedTree(Node node)**, onde o nodo recebido como parâmetro é a raiz da árvore. Cria-se uma lista de nodos percorridos em ordem a partir da raiz, para que posteriormente essa lista seja percorrida realizando as costuras, que nada mais é do que uma referência (ponteiro) para a direita (nodo direita).

```
/**
 * Converts the tree to a threaded tree
 * @param node
 */
public void makeThreadedTree(Node node) {
```

Após a árvore ter todas as costuras feitas e os nodos folhas já numerados (isso é feito ao adicionar um nodo folha na árvore), as composições dos estados são realizadas. O método **createStatesComposition()** é o responsável por esse processo e retorna um **autômato finito** que *representa a expressão regular* inserida pelo usuário.

```
/**
 * Create all the states composition
 * @return the finite automata given by the regular expression
 * @throws InvalidStateException
 */
public FiniteAutomata createStatesComposition() throws InvalidStateException {
```

A forma que esse processo é realizado é como se segue:

O percorrimento inicial parte da raiz da árvore de acordo com as rotinas descer e subir para cada operador, realizadas pelos seguintes métodos:

downPath(Node node): retorna o conjunto de nodos atingidos a partir do nodo *node*, de acordo com as rotinas *descer* de cada operador atingido no percurso.

```
/**
 * Traverse the tree according to the down routines
 * for each operator + | ? * .
 * @param  node the node from which the traversal begins
 * @return the set of nodes reached by the current node
 */
public Set<Node> downPath(Node node) {
```

upPath(Node node): retorna o conjunto de nodos atingidos a partir do nodo *node*, de acordo com as rotinas *subir* de cada operador atingido no percurso.

```
/**
 * Traverse the tree according to the up routines
 * for each operator + | ? * .
 * @param node the node from which the traversal begins
 * @return the set of nodes reached by the current node
 */
public Set<Node> upPath(Node node) {
```

É importante destacar que o operador **+** foi adicionado nas rotinas:

- **+ (descer):** só desce.
- **+ (subir):** sobe e desce.

Após a obtenção da composição inicial a partir da raiz, pode-se criar o estado inicial **q0** do autômato. Caso **lambda** (representado por **$**) seja atingido nesse percurso, q0 será final.

Depois, para cada um dos nodos atingidos, cria-se a composição para os símbolos dos mesmos. Se mais de um símbolo é atingido por nodos diferentes (ex.: 1a e 3a), o próximo estado será atingido pelo composição da **união** deles. O método que auxilia esse processo é **getStateQiComposition()**, que retorna o conjunto de estados atingidos pelas rotinas **subir** para cada nodo que contém o mesmo símbolo**:**

```
/**
 * Return the union of compositions for a given symbol
 * @param unionSymbolsComposition a mapping for a given symbol to a set of nodes it reaches
 * @param qiComposition the composition of the state qi
 * @return the union of compositions for every symbol in state qi
 */
public HashMap<Character, Set<Node>> getStateQiComposition
        (HashMap<Character, Set<Node>> unionSymbolsComposition, Set<Node> qiComposition) {
```

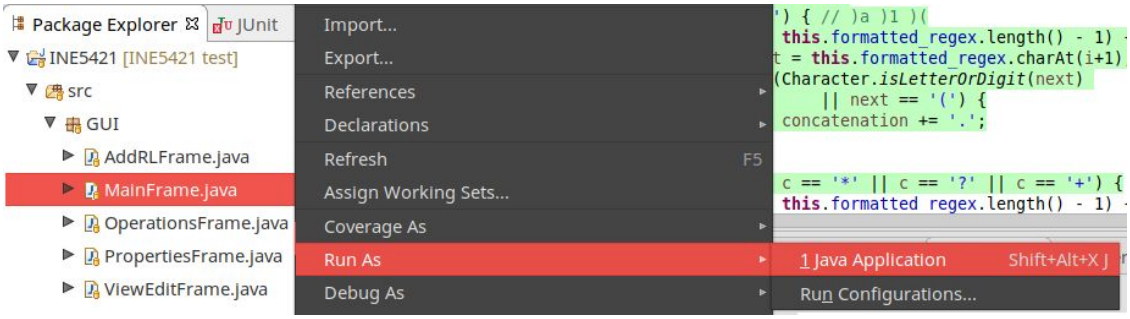
Por fim, se a composição obtida é uma composição nova, ou seja, não foi atingida antes, cria-se um novo estado do autômato e suas respectivas transições. Caso lambda pertença à essa composição, esse estado será final.

Ao fim do método, será retornado o autômato criado:

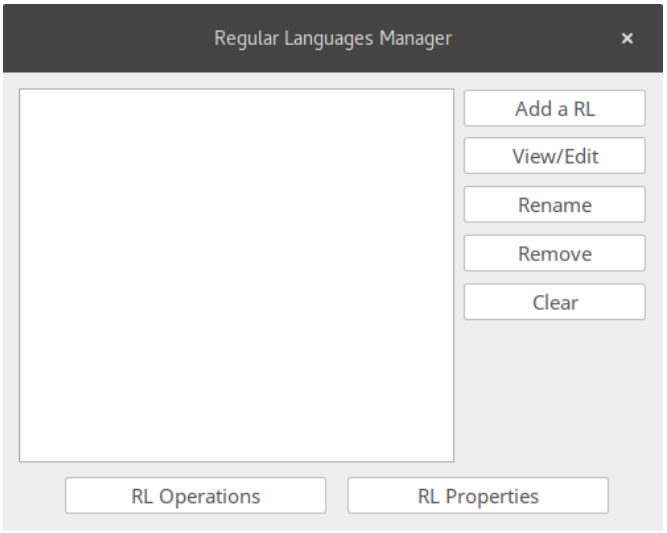
```
try {
    FiniteAutomata automata = fa.build();
    return automata; // return the automata
} catch (IncompleteAutomataException | InvalidBuilderException  e) {
    e.printStackTrace();
}
```

Utilização

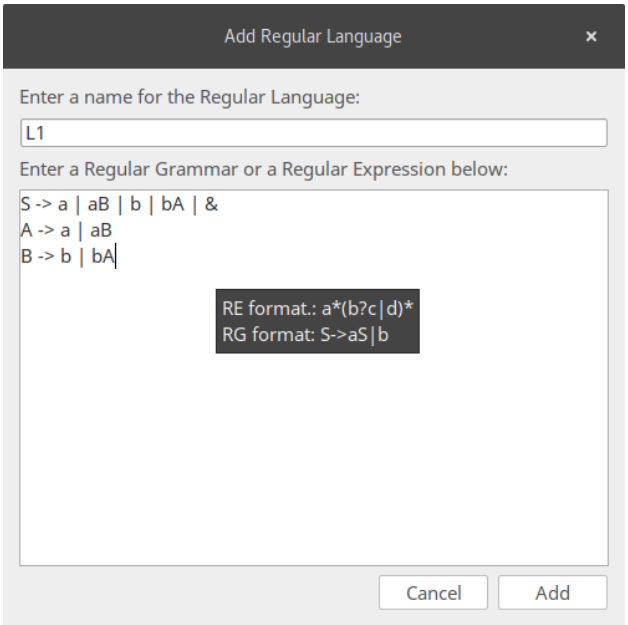
Para iniciar a interface do programa, o usuário deve executar o .jar fornecido ou então abrir o projeto no Eclipse e executar a classe MainFrame do pacote GUI (Clique direito na classe -> Run As -> Java Application).



Assim que o programa é iniciado, sua tela principal é exibida. Nesta tela, serão listadas as linguagens regulares adicionadas pelo usuário posteriormente, e também os botões que serão explicados em seguida.



Para adicionar uma nova linguagem regular, o usuário deve clicar no botão “Add a RL". Assim que ele for clicado, uma nova janela para a adição da linguagem será aberta. Nesta nova janela existem dois campos: um para inserir o nome da linguagem (que será utilizado para identificação nas listas) e outro para a inserção da linguagem.



A Linguagem deve ser inserida através de uma expressão regular ou de uma gramática regular que a represente. O alfabeto de entrada é composto por letras minúsculas ou dígitos.

Ex: `a?(ba)*b?` ou

```
S -> a|aB|b|bA|&
A -> a|aB
B -> b|bA
```

A gramática ou expressão devem estar de acordo com o formato visto em aula. A **concatenação** é implícita através da justaposição (*abcd*). Logo, pontos não são aceitos para concatenar os símbolos (*a.b.c.d é inválido*).

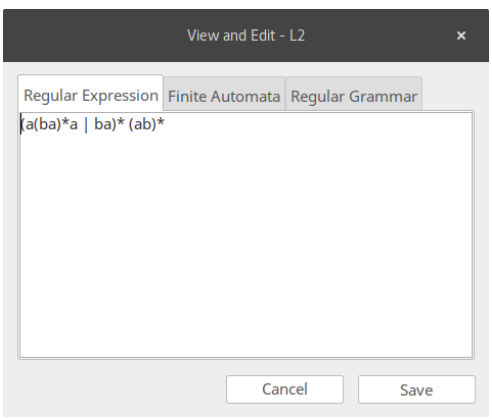
Após a entrada da linguagem, o usuário deve clicar no botão “Add” para salvá-la, ou então pode-se clicar em “Cancel” para voltar à tela principal sem salvá-la.

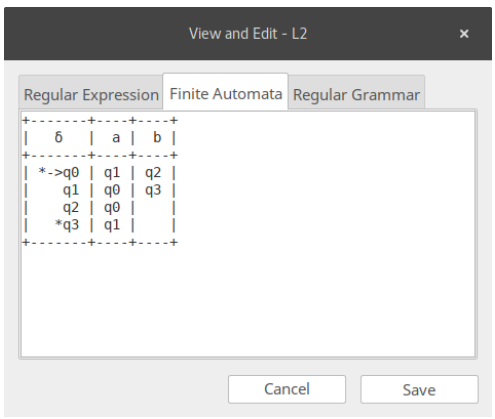
Para visualizar, editar, renomear ou excluir uma linguagem da lista, o usuário deve primeiramente selecionar a linguagem desejada na lista, com um clique simples, e então clicar no botão relacionado à ação desejada. O botão “Clear” remove todas as linguagens da lista após confirmação, enquanto que o botão “Remove” faz o mesmo somente com a linguagem selecionada. Já o botão “Rename” solicita a entrada de um novo nome para a linguagem selecionada.

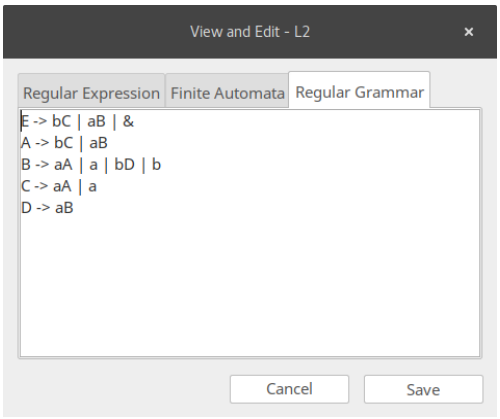
Ao clicar no botão “View/Edit”, será aberta uma nova janela com a linguagem selecionada. Na parte superior, existem três abas para selecionar a forma de visualização da linguagem: Expressão Regular, Autômato Finito ou Gramática Regular.

A aba de Expressão Regular estará disponível somente caso a linguagem tenha sido inserida como uma, pois a conversão (de AF ou GR) para uma expressão regular não faz parte do escopo do trabalho.

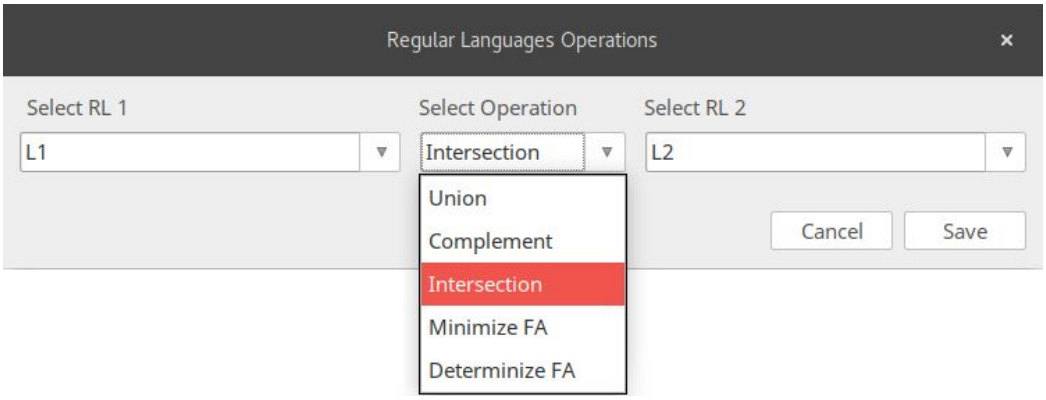
Caso o usuário queira, é possível editar a expressão ou a gramática, mas não o autômato. Para isto, basta editar o texto da representação desejada (ER ou GR) e então clicar em “Save”, substituindo a linguagem pela nova entrada. A linguagem salva será a que está visível na aba selecionada no momento em que a ação é executada.



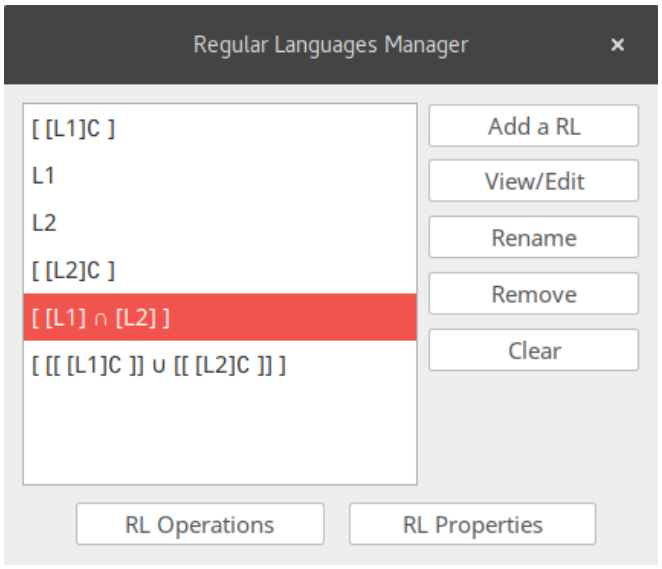




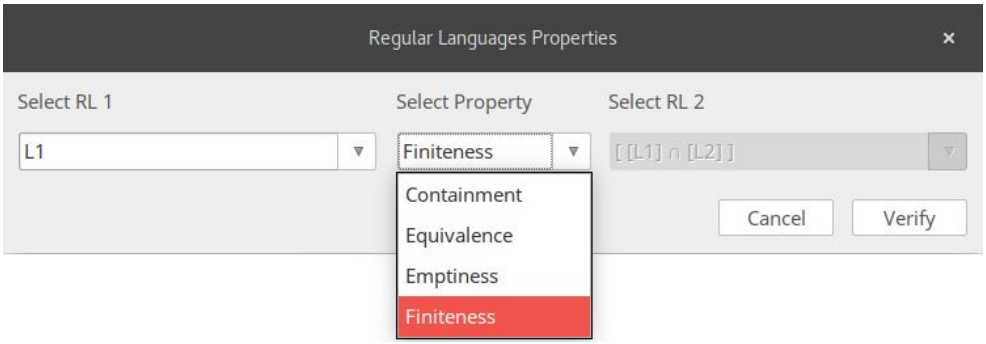
Para realizar operações com linguagens regulares, o usuário deve clicar no botão “RL Operations” da tela principal. Após clicá-lo, uma nova janela é exibida.



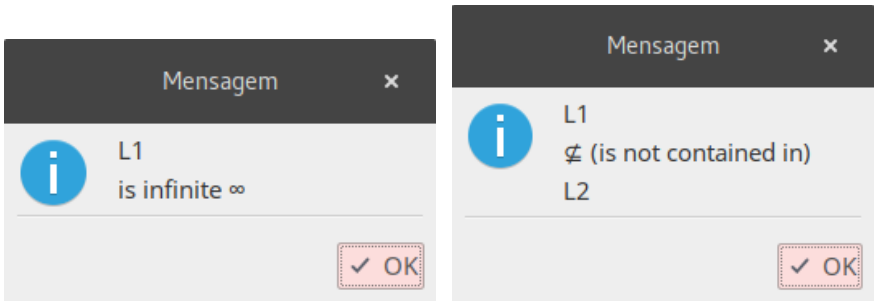
Nesta nova janela, o usuário deve selecionar uma operação e a(s) linguagem(ns) desejadas. Caso uma operação unária seja selecionada, a caixa de seleção da segunda linguagem é desabilitada. Ao clicar em “Save”, uma nova linguagem equivalente à operação selecionada será adicionada à lista da tela principal. No caso da interseção, as linguagens intermediárias (complementos e união) também serão adicionadas à lista.



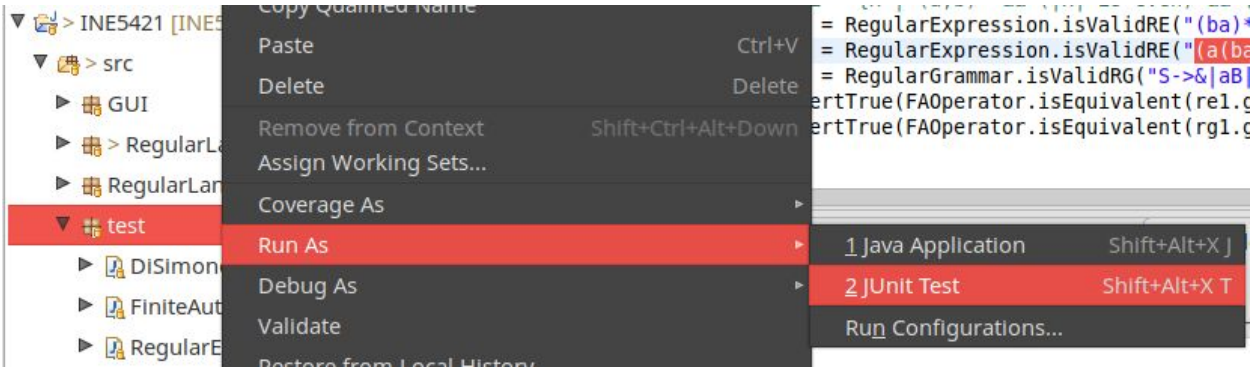
Para verificar propriedades de linguagens regulares, o usuário deve clicar no botão “RL Properties”. Após clicá-lo, uma nova janela é exibida.



Nesta nova janela, o usuário deve selecionar uma propriedade e a(s) linguagem(ns) desejadas. Caso uma propriedade unária seja selecionada, a caixa de seleção da segunda linguagem é desabilitada. Ao clicar em “Verify”, uma nova janela é mostrada informando se a propriedade selecionada é verdadeira na(s) linguagem(ns) selecionada(s).



Para rodar os testes unitários, o usuário deve abrir o projeto no Eclipse e executar o pacote “test“ (Clique direito no pacote -> Run As -> JUnit Test).



Após a execução dos testes, é possível ver os testes que foram executados, assim como se eles passaram, falharam ou se ocorreram erros.