Linnæus University
School of Computer Science, Physics and Mathematics

Degree Project

Automatic Java Code Generator for Regular Expression and Finite Automata

Author: Suejb Memeti
Date: 2012 - 05 - 31
Subject: Computer Science
Level: Master
Course code: 5DV00E

Abstract

Generalizing the common pattern of implementing Regular Expressions (RE) by converting them into a finite automaton that can be programmed, is the main idea of the Regular Expression Java Code Generator. Writing code that writes code has been introduced in Lisp, but taking this into a higher level by introducing the possibility to execute, and modify the written code is the main aim of this project. A "compiler" for regular expressions is useful to turn the expressions we write into executable code.

Programming and resolving problems within automata theory is a complex process, which is time consuming and still the results may not be reliable (because of the great possibility to make mistakes while doing the required traversing steps). This project is an introduction of a tool that automatically generates Java Source Code for a given RE or Finite Automata, which significantly reduces complexity and increases productivity in a very short time. This is also a learning process tool, that can be used as a starting point to better understand the automata theory.

Regular Expression Code Generator tool provides support for auto generating Java code for programming and executing (deterministic) finite automatons. It takes as input a regular expression and converts it into an equivalent nondeterministic and deterministic finite automaton, then automatically generates Java Source Code with the implementation of the DFA for the given RE.

Keywords: Finite Automata, Regular Expression, Automatic Programming, Code Generator

Table of Content

List of Figures

List of tables

Abbreviations

RegExCodeGen - Regular Expression Code Generator
FA – Finite Automata
FSA – Finite State Automaton
NFA – Non-Deterministic Finite Automata
DFA – Deterministic Finite Automata
ANTLR – ANother Tool for Language Recognition
RE, RegEx, REGEX – Regular Expression
GREP - Global Regular Expression Parser
HTML - Hypertext Markup Language
EBNF - Extended Backus-Naur Form
AST – Abstract Syntax Tree
BOR - Boolean Or
EXPRLIST - Expression List
CHAR – Character
NRINTERVAL - Number interval
CHARINTERVAL - Character interval
QMARK - Question mark (?))
XML – Extensible Markup Language
DOM – Document Object Model
UML – Unified Modeling Language

1. Introduction

According to Lawson [19] the theory of Finite Automata is a mathematical theory of algorithms that is important in computer science. In the 20th century the algorithms were classified into two approaches. The first classification was according to their running time, known as complexity theory and the second classification was according to the types of memory used in implementation, known as language theory, whereas the simplest algorithms are the ones that can be implemented by finite automatons. Finite automata were first introduced by Stephen Kleene in the 1950s, who found some important applications of them in computer science as in the design of computer circuits and in the lexical analyzers in compilers.

Nowadays the old dream of Computer Science for creating machines that can program themselves has shown a remarkable progress of becoming real.

Implementing finite automata for regular expression is a process that takes too much time, depending on the regular expression complexity. This project introduces an automatic way of implementing finite automata int Java Code.

1.1 Problem and Motivation

Implementing a deterministic finite automaton (DFA) in Java is a long process that may take plenty of time, and the possibility to make mistakes is too high, especially for large and complex problems (regular expressions). To implement an automaton, first a correct RE should be defined that describes that automaton, then this RE should be translated into a non-deterministic finite automaton (NFA). Usually additional translation between NFA and DFA is required, since NFA cannot be implemented because in NFA there exist some choice of transitions that leads to accepting state, while in DFA exactly one possible transition may lead to accepting state, it means that only after this translation has been done the automaton is ready to be implemented. There exist complex problems that may take days to solve manually. It may happen that after huge amount of work spent on traversing RE to DFA we realize that the mistake was made. In this case all the steps have to be repeated all over again, that makes the process even more time consuming.

While solving different problems with finite automata, you realize that it is an iterative process, and there may be an automatic way to implement the deterministic finite automata in a Java Code which represents the same given regular language (regular expression) input.

1.2 Goals

In order to create a Java Code Generator for Regular Expressions, the goals pursued by this project have been described:

- Creating an ANTLR grammar to define regular expressions
- Turning regular expression into non deterministic finite automaton
- Translating NFA into an equivalent deterministic finite automaton
- Generating Java Source Code that equivalently represent the deterministic finite automaton

The goal of this project is also to give a comprehensive view related to automatic programming and to increase the programmers productivity by providing source code in an automatic way.

In the summary it is mentioned that all the examples (including the one in Appendix B) used in this report have been generated using the Regular Expression Code Generator tool, which we are introducing in this project. Also in the Appendix A there is the link from where the Regular Expression Code Generator can be found and downloaded, which means the software is ready for use and by this fact it can be realized that all the forehead goals have been fulfilled.

1.3 Thesis outline

The project is notionally divided into two parts: Section 2 Theoretical description (Regular Expressions - a brief introduction to regular expressions, and Finite Automata), 3 Implementation and Experiments (Converting regular expressions into Finite Automata, and Automatic Programming - generating Java Source Code).

2. Theory

This chapter is a short introduction to Regular Expressions and Finite Automata. Also there are going to be given some examples that describes the use of RE and FA in the real world, as well as how these FA can be represented.

2.1 Regular Expressions

Regular Expressions are invented by Steven Kleene who showed that regular expressions describe the same language that finite automata described [15].

According to Aho et al. [1] Regular Expressions has come into common use for describing all the regular languages that can be build from some operators applied to some characters of input alphabet. Regular expressions are build in a recursive way out of smaller regular expressions. Each regular expression represents a language which is also defined in a recursive way from the languages represented by its subexpressions. As Ullman [15] described, regular expressions are a notion to describe patterns, such as patterns in text. Each regular expression is convertible into an automaton and each automaton can be converted into a regular expression, that proves that the language accepted by finite automatons as well as by regular expressions is exactly the same.

The basic operators of regular expressions are union (a|b ⇒ **a** or **b**), concatenation (ab ⇒ **a** followed by **b**), and closures (a* ⇒ zero or more **a,** a+ ⇒ one or more **a**). RE are built up by applying these operators to operands. Example:

(ac?|b)|[1..9]+|[A..Z]+*

For RE the value of each expression is a pattern that consist of a set of strings called regular language. Each regular expression **r** over Σ defines a regular language:

$$L(r) = \{x \in \Sigma^* \mid x \text{ matches } r\}$$

According to [15] union is expressed by "+" and is read as "or", but since during this project we are using extended regular expressions like Positive Closure (**r+** is equal to **rr*** where **r** stands for RE, * and + is Kleene Star closure, respectively Positive Closure operators), we will refer to the convention used in UNIX six by expressing the union with a vertical bar "|". From the example above we have **ac?|b**, it means that we can choose **ac?** or **b**.

Concatenation is operation made by taking the first string and concatenating with the second string. Lets say that we have two languages A and B, and the concatenation is AB, that means that the string of language A is merged with the string of language B. The concatenation in regular expression has no additional symbol, it is done simply by adding the second language next to the first one. In the forehead example concatenation occurs in **ac?** where **a** is concatenated with **c?**.

Aho et.al [1] described that there are two types of closures: Kleene closure and Positive closure. Kleene closure is represented by "*" after the character or group of characters, denoted as r* (r stands for regular expression and L(r) is the language denoted by r), and it means that the "r" expression can occur zero or more times (Example: $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup \ldots$). The Kleene star is formed by concatenating zero or more strings from L in any order. When it occurs zero times it is defined to be empty $\{\lambda\}$. The positive closure in some way is a form of the Kleene closure except that it should occur at least one time, it is denoted as r+, that is equal to rr*.

The parenthesis are used to group subexpressions, (r), and often regular expressions contains unnecessary pairs of parenthesis, which can be skipped for example: r = ((r)).

There are also some extended regular expression operators, while only three of them are being used during this project: the question mark (?), positive closure (+) and the interval of characters and numbers. An example of question mark operator is **r?** and it is read as "zero or one **r**", while the interval example r{m,n} is read as "occurrences of r between m and n". During this project the interval is represented among characters or numbers, for example: [a..z] respectively [0..9].

According to Ullman [15] the RE's definition basis consist of three parts:

1. The constants ε and 0 are regular expressions, denoting the languages {e} and 0, respectively. That is $L(\varepsilon) = \{\varepsilon\}$, and $L(0) = 0$.

2. If Ḷ is any symbol, then **a** is a regular expression. This expression denotes the language {Ḷ}. That is, L(a) = {Ḷ}. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that a refers to Ḷ should be obvious.

3. A variable, usually capitalized and italic such as *L,* is a variable, representing any language.

As Ullman [15][1] described, if **r** and **s** are two regular expressions then:

- r|s is a regular expression denoting the language $L(r) \cup L(s)$
- r s is a regular expression representing the language L(r)L(s)
- r* is a regular expression representing the language (L(r))* where L(r) and L(s) are languages represented by regular expression **r** respectively **s**.

The operator priority or precedence is as follows: the Kleene star has the highest precedence, then concatenation, then union. Examples:

- L(ab) = {ab} - represents the concatenation of the language consisting of one string which is **a** and the language consisting of one string **b**, and the result is **ab**.

- L(ab | c) = {ab, c} - represent the union of the language containing only the string **ab** and the language containing only string **c**.

- L(a(b|c)) = {ab, ac} - in this case we use parenthesis because of the operator precedence (concatenation takes priority over union).

- L(a*) = {e, a, aa, aaa, ...}

The symbols or characters defined in regular expression as a whole represents the alphabet of that regular expression, and is denoted with sigma (Σ) symbol. So if **a** and **b** are symbols in regular expression then the alphabet of **r** is $\Sigma(r)=\{a,b\}$

2.1.1 The use of Regular Expressions

Regular Expressions matches the pattern of strings in a very efficient manner, whether these strings are simple or complicated. In this section the question "Why regular expressions are really useful in the real world" is answered by solving some real world problems that programmers are facing every day.

Many systems that in someway describe patterns use regular expressions. Usually they are invisible because are embedded in the code of the company, but sometimes they are visible, like the UNIX commands. There are many UNIX commands that have in someway the notation of extended regular expression for an input. An appropriate example is the Global Regular Expression and Print (GREP) command. According to Ullman [15] it is not a coincidence that regular expressions figure into the UNIX commands. Before Ken Thompson did UNIX, he worked on a system for processing regular expressions by converting them only as far as a NFA and simulating the NFA code.

RE and Automata are very powerful to search efficiently for a set of words in repositories, where the repositories can be too large such as the Web. Regular Expressions technology has been found useful in the description of a defined class of patterns in text. With a little effort patterns can be described at a high level using regular expressions, and the modification of that description is very quick. Let us explore an example that detects e-mail addresses that arises in many Web pages, which can be useful to create an e-mail list that can be used for newsletters or something similar. To create such expression that recognizes email addresses we might use something like this:

```
[A..z]([A..z]||[0..9]|.|-)*@([A..z]||[0..9]|-)+.[a..z]+
```

This example will recognize email addresses of this form: email@name.com. The question is what happens if there is an email of this form: email@name.co.uk? It will not be recognized with the actual expression, but we can handle these cases by modifying the expression into one like this:

```
[A..z]([A..z]||[0..9] |.|-)*@([A..z]||[0..9]|-)+(.[a..z]+)+
```

RE can be used for validation, for example a telephone number pattern or maybe a credit card pattern. They find usage in identifying, removing or replacing specific text in a document. Extracting a substring from an input string, or extracting specific section from an HTML page can be done using regular expressions.

RE are useful in lexical analyzers as well. Lexical analyzers breaks the input of a program into its basic units called tokens. For example every identifier in a program is a token. Java identifiers, which are upper or lower case letters that are followed by any sequence of letters or digits may be expressed as

[A..z]([A..z]|[0..9])*

When an instance of the provided RE is recognized, then you can provide a piece of code as an action to be executed. For example, the code for when an integer is found might simply return that integer.

A number of lexical analyzer generators takes RE as input to describe the tokens and as a return statement produces a single finite automaton that recognizes any token.

RE can match just about anything, as Larry Wall, Tom Christiansen and Jon Orwant (the authors of Programming Perl) wrote: “If you take ‘text’ in the widest possible sense, perhaps 90% of what you do is 90% text processing.”

2.2 Finite Automata

According to Aho & Ullman [2] a finite automaton is a graph-based way of specifying patterns. You can think of automata either as a graph or as a table. The Finite Automata (FA) stores finite amount of information (finiteness of memory), which is great because we can do things that in general cannot be done with programs. For the FA programs you know what that program does, or if there is a shorter program or another one that does the same thing as the given automata. Finite automata so called finite automaton is built of a (finite) collection of states, and each state has name and represents what is remembered about its history. The process of changing the current state of an automaton in response to input characters, is called a transition.

Finite automata together with regular expressions are very important in text searching algorithms. They are also essential for the compiler to break the input into tokens, identifiers, key words like **if**, **for**, **while** and so on. FA and Regular Expressions (RE) are important for describing patterns. A pattern is a set of objects with some recognizable property, informally they are sequences of symbols or event of some sort.

Jeffrey Ullman during his lectures in Automata Course in Stanford University [14] gave a very good example of using finite automata for scoring a tennis game. The winner in a tennis game is the one who scores first 4 points and must win by at least two points difference. In the graph in Figure 2.1 you can see that the states of the automaton represent the number of points won by each player. Let us consider player A playing against player B, and every time one of player scores a point the automaton changes the state until it reaches the accepted state that means the end of the game. The game starts at state 0, and depending which player wins the first point the current state changes to state 15:0 if player A has scored, respectively to state 0:15 if player B has scored. Then again depending which player scores the second point there can be three possible states: 30:0 in case player A scores both points, 15:15 when each scores by one point and 0:30 player B scores both points. The interesting state is 15:15, because when the automaton reaches that state it does not know how it came till there, it may have been the sequence AB (player A scored first point and player B scored the second) or BA (vice versa), that is the good thing, FA remembers only what must be remembered. If we are in state 30:0 and player A scores again, we reach state A that indicates an accepted state, which in this case means that player A won the game, and the automaton has no more moves. If both players scored 3 points each, the current state would be 40:40, in this case the game is tied, and it does not remember the sequence of wins nor the sequence of losses of points, nor even how many points have been played. To win the game they must win two points sequentially when the game is in 40:40 state, because if player A wins one point the current state is 1:0, and in this case player A must score one more point to win the game otherwise with a point scored by player B the game again comes back to 40:40 state.

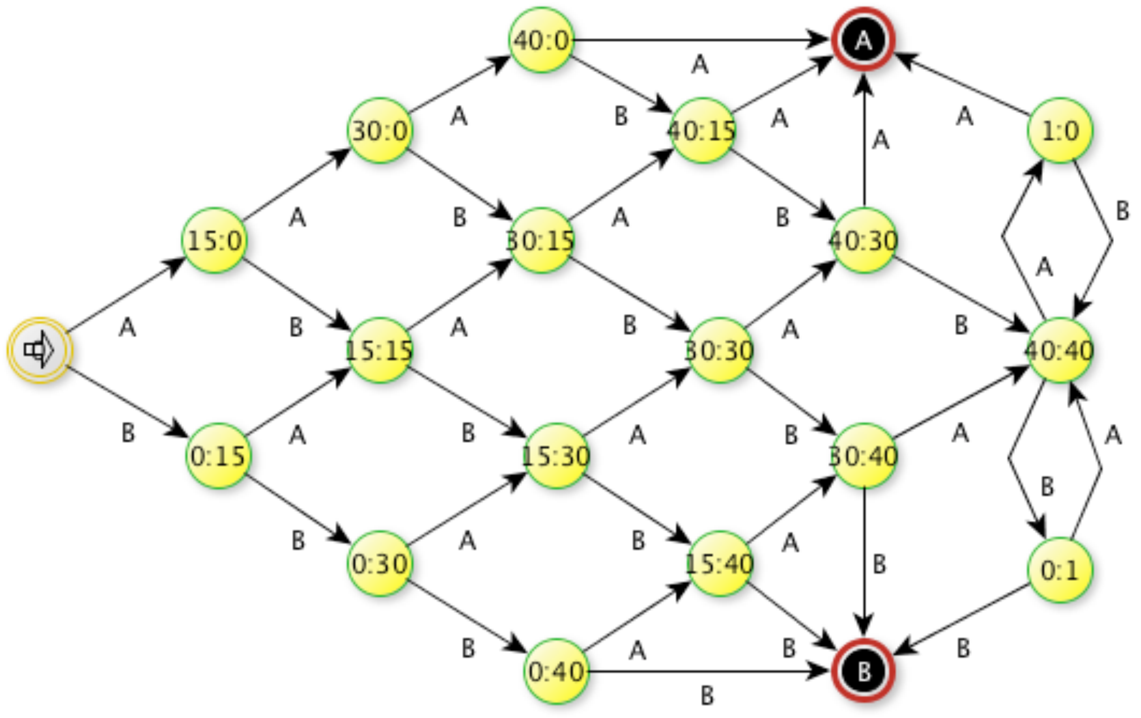


Figure 2.1 Tennis Score Game represented by DFA

The job of the automaton is to process the input string, and for each token we follow the transition from the current state to reach the next state. Formally, for an input string we say that it is accepted if and only if the last input symbol token has been processed and the current state has reached one of the possible accepted states, otherwise the string is rejected.

The language of an automaton is simply a set of strings that is called alphabet, in our example it is A and B, and the language for this automaton is denoted as L of A where L stands for language and A stands for automaton.

$$L(A) = \{A,B\}$$

The finite automata can be deterministic (Section 2.2.1) and non-deterministic (Section 2.2.2). Patterns can be described by Regular expressions as well as by finite automata. Regular expressions can easily be translated into a non-deterministic finite automata. Non-deterministic finite automata can be converted into an equivalent deterministic finite automaton using the subset construction, that will recognize the same patterns. In conclusion we say that a regular expression can be converted to automata and vice versa. The deterministic finite automaton is convertible into a program that recognizes its patterns.

Finite automata are recognizers, they simply return **true** or **false** about each possible input. Both DFA and NFA are capable of recognizing the same language.

To visualize and represent finite automata, we can use transition graphs or the table representation. They are two representations of the same idea. While using graphs each vertex or the node of the graph represents a state in finite automata and each edge represents a transition in automata. The detailed discussion can be found in the following sections 3.1 and 3.2.

2.2.1 Deterministic Finite Automata

According to Ullman [2], a deterministic Finite Automaton or DFA is defined by the quintuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q stands for a finite set of states,

Σ (Sigma) stands for the set of symbols, so called alphabet

$\delta: Q \times \Sigma \rightarrow Q$ (Delta) stands for a total function called transition function

q_0 stands for initial state synonymously start state, and $q_0 \in Q$

F is a set of final states synonymously accepted states where each state of F is element of Q as well ($F \subseteq Q$ (F is subset of Q))

Strings or input strings are sequence of symbols chosen from alphabet Σ. For example **abb** is a string of alphabet Σ={a,b}, also it can be Σ={a,b,c} but it just happens not to have any **c**'s.

Σ* stands for the set of all string over the alphabet Σ and the special string called empty string represented by epsilon. The language of a DFA is a subset of Σ* for some alphabet Σ.

With other words the language that DFA defines is the set of strings that may take the start state to a final state.

Languages that have an equivalent deterministic finite automaton are called regular languages, which also can be described by regular expressions or non-deterministic finite automata. While most of the languages are regular languages, there are also many of them that are not. To describe these kind of languages we need more powerful mechanism such as context-free grammar (this is not relevant with this project).

The transition function describes how the automaton moves from one state to another in response to inputs. We can say that this function is the one that makes the automaton works. It takes two arguments, a state **q** and an input symbol **a**, and it returns the states where the automaton moves when it is in the state **q** and next input symbol **a**. For example in the Figure 3.1from state 15:15 on input symbol A automaton make a transition to state 30:15. This function has a value for each state and symbol, but what happens when we reach a state that does not have any out transition (ex: state A and B in the tennis score game)? To fix up the situations where we do not want to continue in such situations we introduce the dead state synonymously trap state, which is a non-accepted state and it has transitions to itself on every input symbol. That means that once you get to a dead state, there is no possible way to leave it.

A DFA operates in the following manner: when program starts the current state is assumed to be the initial state q0, on each input symbol or character the current state is supposed to move on another state (including itself). When the string reach the most right symbol (last one) the string is accepted only if the current state is one of the accepted states, otherwise the string is rejected.

The abbreviation DFA for deterministic finite automata is called deterministic because on each state and input character a unique transition is defined.

The deterministic finite automaton is the only way so far to execute an automaton, that means that only a DFA can be implemented. No one yet has invented a computer that works with nondeterminism.

2.2.2 Nondeterministic Finite Automata

By nondeterminism we mean the choice of moves for an automaton. According to Aho et al. [1], NFA do not have any restriction on the label of their edges, it means that a symbol or character can appear in several edges out of the same state, including the empty symbol (λ), it means that NFA rather than allowing a unique move, it allows a set of possible moves in each situation. The ability of being at several states at once, or the definition of the transition function which range is a set of possible states makes NFA more complicated than DFA.

A nondeterministic Finite Automaton is defined by the quintuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q, Σ, q0, F are defined as for DFA except that the range of δ is not a single element of Q as in DFA but it is a subset of it, and mathematically it is expressed as:

$$\delta: Q \times (\Sigma \times \{\lambda\}) \rightarrow 2^Q$$

Likely the DFA, a NFA has one start state, where the input processing begins. The transition function in a non deterministic finite automaton returns a set of states, rather than a single state as the DFA. For example if the current state is q_1 and input symbol is **a** and $\delta(q_1,a) = \{q_0, q_2\}$, then the next state of the NFA can be either q_0 or q_2. An accepted string of NFA is one that leads the automaton from the start state to any of the accepted states, it differs from DFA because NFA is allowed to guess which way to go, and it does not matter how many wrong or useless guesses it makes, the NFA gets credit for the right guesses. The possibility of guessing of the NFA is one of the three main differences between NFA and DFA.

NFA are able to make so called spontaneous transition on epsilon synonymously empty. The empty transition defined as λ means that the NFA can make a transition without consumption of an input character, which is the second thing that makes NFA differ from DFA. The epsilon transition in not an input symbol, that results that is not a member of the input alphabet. The third difference is that in NFA the set of $\delta(q_i,a)$ may be empty, it means that there is no transition defined for the specified situation.

In some way a DFA is an NFA that does not have any non-determinism. For example for a deterministic automaton A with the transition function δA, a NFA can be created with exactly the same states and inputs as the DFA. The only thing that differs is that the transition function of NFA returns a set of states rather than a single state, but that set will be exactly the one state that the transition function of DFA returns. This proves that any language accepted by a DFA can be accepted by some NFA.

There is also an automaton DFA that accepts exactly the same language as the NFA automaton, and the proof is the subset construction algorithm described in details in section 7.

NFA can also be represented by transition graphs, the difference is that in NFA there are some vertices labeled with λ that represents the empty transition, and in NFA we can have several edges with the same label originated from one node. The accepted string can have one or more sequence of possible moves.

While representing the NFA using transition tables the difference is that there is a column for the λ transition, and for each state of the automaton we set all the possible next reached states on empty transition.

2.3 Finite State Machines Representation

In this chapter we introduce and compare the two most convenient ways of representing Finite State Machines: Transition Diagrams and Transition Tables.

2.3.1 Transition Diagrams - Graph Representing State Machines

The behavior of a program that we got from the deterministic automaton can be represented using directed graphs. Each node of the graph represents a state of the automaton, and the state can be start state, a regular state or the final state or accepting state. Conventionally the states are represented as follow: *start state or initial state* is an ellipse shape with an in-degree arc pointing from nowhere, a *normal state* is represented as ellipse shape but with at least one predecessor pointing from another state, and the *final state* is represented as a double circles. T h e r e i s another state which is a combination of initial state and accepted one, that means that it is initial and at the same time accepted state (Figure 2.2). When the accepted states is reached it means that we have found the desired pattern. The arcs or edges so called transitions of the graph are labeled by the set of characters. If **a** is a character which labels the transition from state q0 to state q1, then δ(q0,a) is q1 (if the current state is 0 and the next character is **a** we say we make a transition on **a** to state 1).

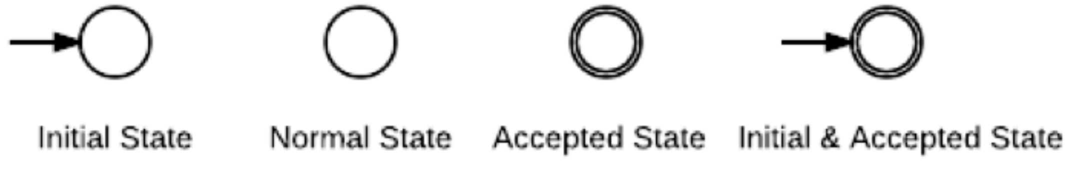


Figure 2.2 Conventional representation of automaton states

During this project the following convention for Finite Automaton States will be used:

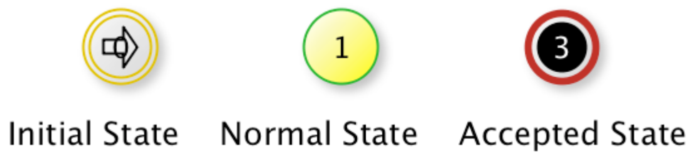


Figure 2.3 Regular Expression Code Generator states representation convention

The labels on the vertices of the graph represents the names of the states, while the labels on the edges are the current values of the input symbol.

The automaton's behavior is simple, it accepts sequence of characters so called input sequence, it begins from the initial state of the graph, and start to read from the first character of the input sequence. Depending on that character a transition can be made to another state or probably to the same state. An example of how an automaton looks like is shown in Figure 2.4.

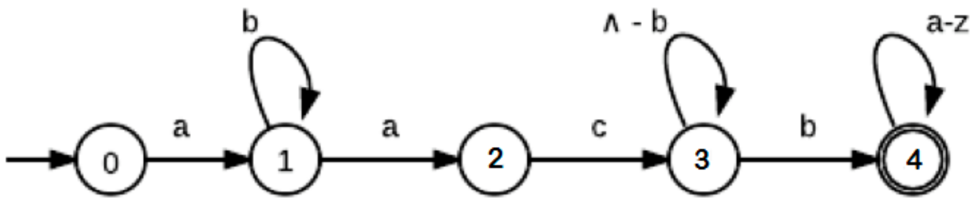


Figure 2.4 Finite Automaton Example

2.3.2 Transition table - Table representation of State Machines

While graphs are most convenient for visualizing finite automata, there are other ways of automaton representation which are also useful. For example the automaton in Figure 2.4 can be represented as a table. The column header represents the input symbols, while the row header corresponds to states of the automaton. In the column header are shown all possible input characters, that is the alphabet, while in the row header are shown all the states of the automaton.

The entries in the table define the next state or the values of the transition function applied to the state that is the row and the symbol that is the column, it means that from corresponding state with the corresponding input character it moves to the next state which is the state inside the cell (row x column).

Σ	a	b	c	∧-b	a-z
0	1				
1	2	1			
2			3		

Σ	a	b	c	∧-b	a-z
3		4		3	
4					4

Table 2.1 Transition table example

From the table example above we clearly know that from state 0 in an input **a** the current state changes to 1 (δ(q0,a) = q1), or from current state 1 with input symbol **a** the next state is state 2 (δ(q1,a) = q2).

While representing the NFA using transition table each entry is represented in curly braces that we use to form sets, in the DFA each entry is a single state, so we may therefore represent the state without the curly braces.

The advantage of using transition table is that we can easily find the transitions on a given state and input. The disadvantage is that it takes too much space when the input alphabet is large, since the most states do not have any move transition on most of the input symbols.

3. Implementation and Experiments

This chapter briefly describes the methods, techniques and algorithms used to implement the necessary steps to create a tool that automatically generates Java Code for a given Regular Expression or Finite Automata.

3.1 Creating of an ANTLR specified grammar for valid regular expressions

ANTLR (ANother Tool for Language Recognition) is a parser and translator generator tool that uses LL(*) (left to right parser and constructs a leftmost derivation of the sentence) parsing. As input ANTLR takes a grammar (formal language grammar) that specifies a language and as output it generates source code that recognizes all the strings for the given input grammar. The ANTLR input language is specified using the context-free grammars that is expressed using EBNF (Extended Backus-Naur Form)[7]. ANTLR can create lexers, parsers and AST's (Abstract Syntax Tree). The role of lexer is to quantify the meaningless stream of characters into discrete groups that have meaning when processed by the parser. The important task of the lexer is to look at the entire program being compiled, and breaking it up into tokens. Tokens are sequences of characters that logically belong together. The lexer generates errors for sequences of characters that cannot match the specific token type defined by the rules. Unlike lexer, parser consider the semantic of the sequences of the characters within the context of the whole program. The parser generates error for the sequences of tokens that cannot match to the specific syntactical arrangements allowed, as decreed by the grammar.

The AST is a special kind of tree that that represents the abstract syntactic structure of the code, we say abstract because it does not represent every detail from the code (real syntax). AST's contains additional informations implicitly that makes it easier to translate to a target language. Creating an abstract syntax tree is the most important part of a language translation process. AST can have an arbitrary number of subtrees, which by themselves are ASTs. [7]

The AST for the specified grammar is constructed by implementing the rewrite rules. While constructing the AST we should carry on for the operator precedence.

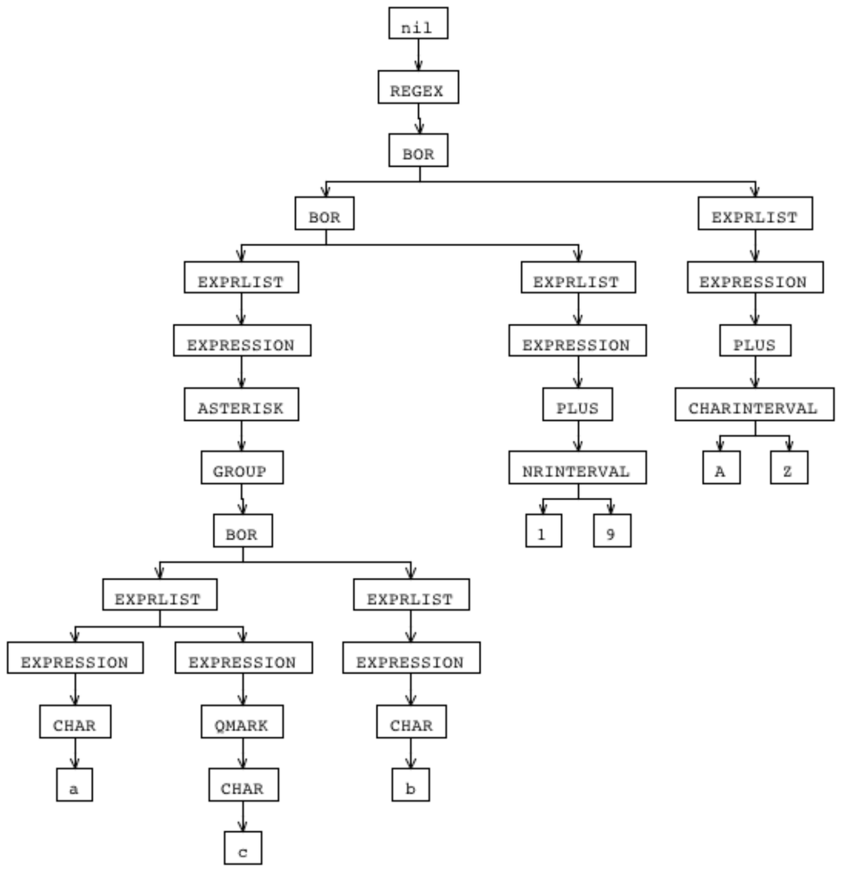


Figure 3.1 The Abstract Syntax Tree for "(ac?|b)|[1..9]+|[A..Z]+" regular expression*

The AST is constructed as following: The root node is called "REGEX" which stands for Regular Expression, based on the operator priority the following node of root can be "BOR" or "EXPRLIST" which stands for Boolean Or node, respectively Expression List. The expression list child nodes are always expression nodes, but we can have one or more of them.

The expression can be a simple character, number, symbol, interval of characters or numbers, it can be also one of them with an asterisk, plus or question mark node before or it can be a group followed by an expression list or a boolean-or node. In the case the expression is followed by one of the operators like plus, asterisk or question mark, it means that that subtree can be repeated zero or more time in case of asterisk, one or more time in case of plus, and zero or one time in case of question mark.

The algorithm that will be used for traversing the tree while building the Non-Deterministic Finite Automata from the given regular expression will be left to right known as Depth First Search.

3.2 Converting Regular Expression into Non-Deterministic Finite Automata

It turns out that every Regular Expression has an equivalent NFA and vice versa. There are multiple ways to translate RE into equivalent NFA's but there are two main and most popular approaches. The first approach and the one that will be used during this project is the Thompson algorithm and the other one is McNaughton and Yamada's algorithm. In the next section briefly will been discussed these two algorithms and the algorithm we have used during this project.

3.2.1 Thompson's algorithm

Thompson algorithm was first described by Thompson in his CACM paper in 1968.
Thompson's algorithm parse the input string (RE) using the bottom-up method, and construct the equivalent NFA. The final NFA is built from partial NFA's, it means that the RE is divided in several subexpressions, in our case every regular expression is shown by a common tree, and every subexpression is a subtree in the main common tree. Based on the operator the subtree is constructed differently which results on a different partial NFA construction. For example the NFA for matching a single character look like:

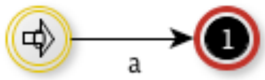


Figure 3.2 Automaton that represent a single character 'a' (a)

The concatenation is constructed by connecting the final arrow of first expression to the first node of second expression:

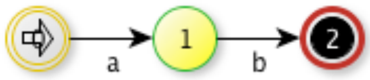


Figure 3.3 Automaton that represent the concatenation of two characters, 'a' and 'b' (ab)

The alternation of a|b is constructed by adding a new state with a choice of first expression and an other choice to second expression:

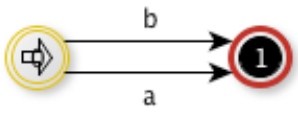


Figure 3.4 Automaton that represent the union of two characters, 'a' and 'b' (a|b)

The NFA for a? is constructed in the same way as a|b but instead of b we have ø closure:

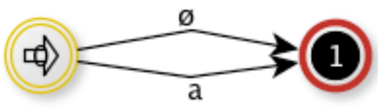


Figure 3.5 Automaton that represent the question mark operator over character 'a' (a?)

The loops as a* or a+ are almost similar, and “a+” can be written as “aa*”, so the NFA graph looks like:

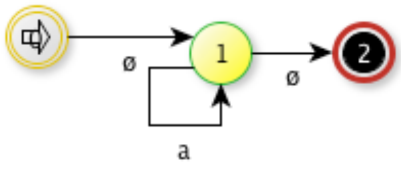


*Figure 3.6 Automat representing a**

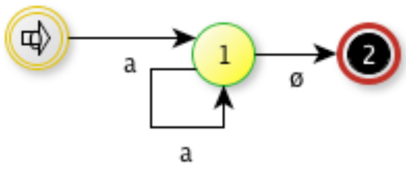


Figure 3.7 Automaton representing a+

So, for the given regular expression example in the ANTLR tree (given in Figure 3.1), (ac?|b)*|[1..9]+|[A..Z]+, the equivalent NFA look like:

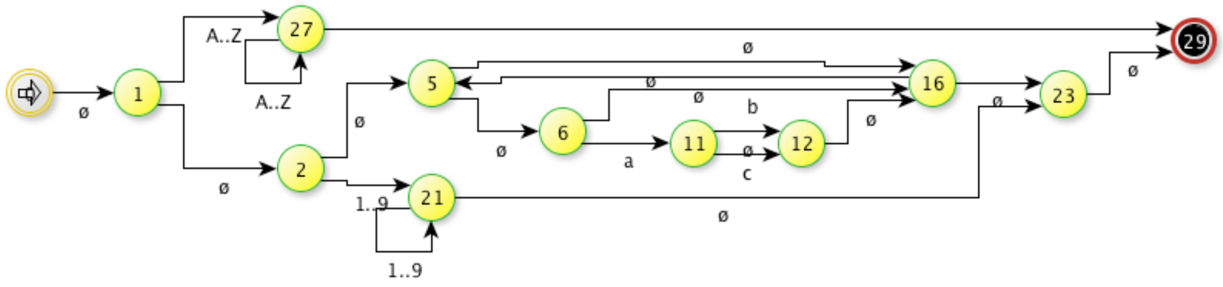


Figure 3.8 NFA Automaton representing (ac?|b)|[1..9]+|[A..Z]+ regular expression*

3.2.2 McNaughton and Yamada Algorithm

The idea of the McNaughton and Yamada algorithm is that it makes diagrams for sub-expressions in a recursive way and then puts them together. According to Storer [10] and Chang [9] the McNaughton and Yamada’s NFA has a distinct state for every character in RE except the initial state. We can say that McNaughton and Yamada’s automaton can also be viewed as a NFA transformed from Thompson’s NFA.

The McNaughton and Yamada’s algorithm in the initial phase creates disconnected initial and accepted state:

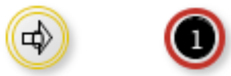


Figure 3.9 Automaton representing ∅ (Empty set)

Then, if the RE is a simple string like R=a or R=ø it connects the states by adding an edge with the corresponding label for the regular expression:

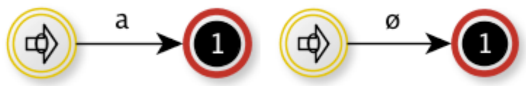


Figure 3.10 Automaton representing 'a' and ϵ (empty transition)

The union (boolean-or expression) using McNaughton and Yamada's algorithm is expressed as follow:

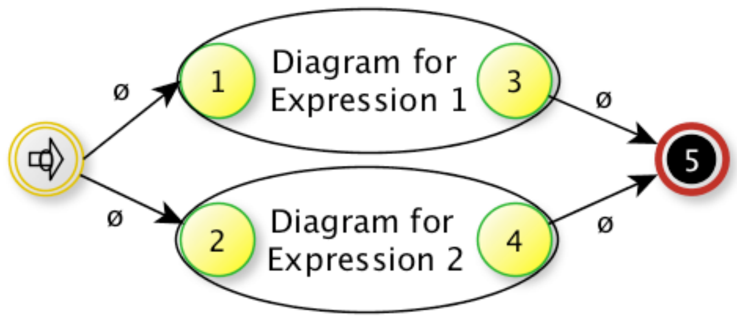


Figure 3.11 McNaughton and Yamada automaton representing union

From the example above we assume our expression as E1|E2. The NFA for this expression is formed by introducing a new start and a new final state. The old final states of both expressions are no longer accepted. Instead there are empty transitions from the new start state to each of the two old start states, and there are empty transitions from the old accepted states to the new accepted state as well. The path from the initial state to the accepted one must go either through expression 1 (E1) or through the expression 2 (E2) diagram, moreover it must have an empty transition at the start and at the end.

The string concatenation, or as during this project will be called expressionList is shown as:

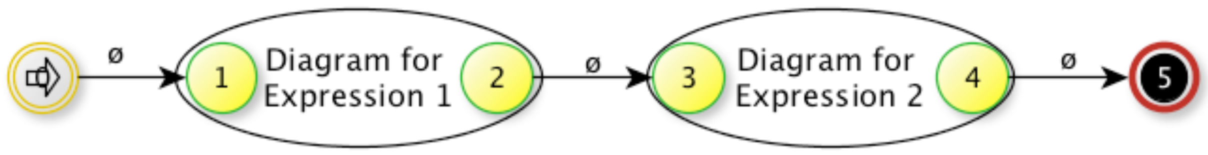


Figure 3.12 McNaughton and Yamada automaton representing concatenation

Again we suppose that there are two expressions E1E2. The concatenation is made by adding a new start and a new final state, here also the old final state is no longer final, but the old and the new final state are connected with an empty transition, and from the new initial state an empty arc transition is added to the old start state as well. While concatenating the automaton of E1 expression and the automaton of E2 expression, an empty transition is added from the final state of automaton of E1 to the start state of the automaton of E2. The path from the start state to the final state of the concatenated automaton of E1 and E2 must pass through E1 first then it takes the empty transition that connects the two automatons, and finally it pass through E2.

The Kleene Star E* is constructed as follows:

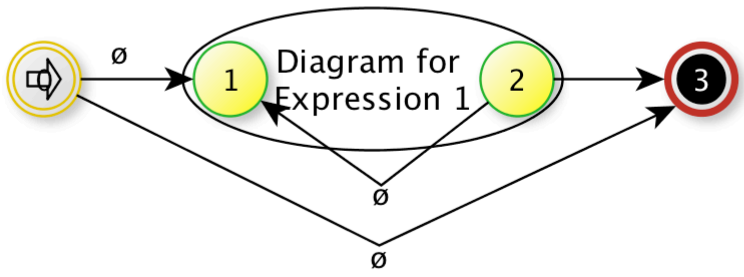


Figure 3.13 McNaughton and Yamada automaton representing Kleene star

If we assume that we have our expression E, and we want to construct the E* automaton. The construction of E* is made by adding new start and final states, and the old final state is no longer final. The start state is connected with the final state with an empty transition, because epsilon is part of E* automaton (in cases when we choose zero, from zero or more).
With empty transition we connect the new start state with the old one, and the old final state with the new one. To describe the cases when we choose more than one from zero or more, we must connect the old final state with the old start state with an empty transition pointing to the old start state. That means that if we got an E, we can take an other E by going back from the old final state to the old start state, and then concatenate the E with the new E, and the result will be EE, and when the desired pattern of E's is taken the automaton ends by going to the final state in an empty transition.

3.2.3 Combination of Thompson's and McNaughton and Yamada Algorithm

During this project, the translation to NFA from an RE is done by using a combination of both algorithms. The conversion is done as follows: the regular expression is converted into a Antlr Tree, then using a depth first search method we traverse the tree and start to build the NFA graph.

Each subtree is converted into a subgraph then it is concatenated (merged or joined) to the main graph. The program starts with the REGEX tree node, which child node can be an EXPRLIST (expressionList) or BOR (boolean or that represents the union). Considering the child node the specific subgraph is created and joined into the main graph. The ANTLR grammar construction for each Tree Node has been shown in Table 3.1.

```
goal     : expression EOF ->^(REGEX expression);
expression       : (e1=primaryExprList -> $e1) (OR e2=primaryExprList -> ^(BOR
$expression $e2))*;
primaryExprList         :primaryExpression+ -> ^(EXPRLIST primaryExpression+);
primaryExpression:    primaryExpr->^(EXPRESSION primaryExpr);
primaryExpr   :(characters ->characters) (AS -> ^(ASTERISK characters) | PL -> ^(PLUS
          characters) | QM -> ^(QMARK characters))? | (group ->group) (AS -> ^(ASTERISK
          group) | PL -> ^(PLUS group) | QM -> ^(QMARK group))? | (interval -> interval)
          (AS -> ^(ASTERISK interval) | PL -> ^(PLUS interval) | QM -> ^(QMARK
interval))? ;
group   : LT expression RT->^(GROUP expression);
interval          : '['  from=CHARS '..' to=CHARS ']'   -> ^(CHARINTERVAL $from $to )
          | '[' from=NUMBERS '..' to=NUMBERS ']' -> ^(NRINTERVAL $from $to ) ;
operator          : AS^
          |         PL^
          |         QM^
          ;
characters
          :         CHARS->^(CHAR CHARS)
          |         NUMBERS->^(NUMBER NUMBERS)
          |         SYMBOLS->^(SYMBOL SYMBOLS)
          ;
```

Table 3.1 ANTLR Grammar for each Tree Node

The expressionList's child node can be one or more EXPRESSION nodes. The booleanOr is followed by another BOR node and an ExpressionList or by two Expression List nodes. For example if the program is **a|b|c** the tree is constructed left to right and it will look like the Figure 3.14:

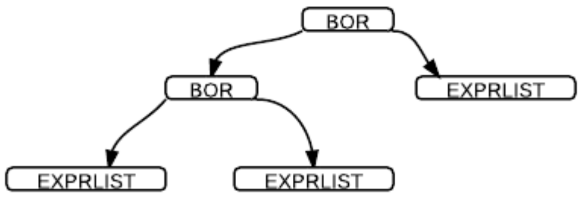


Figure 3.14 Tree construction of a|b|c

Each expression node can have one of the following child nodes: CHAR (character), SYMBOL, NUMBER, NRINTERVAL (number interval), CHARINTERVAL (character interval), GROUP (group of expressions), QMARK (question mark (?)), ASTERISK (*) or PLUS (+).

In case the child node of the expression node is CHAR, SYMBOL, NUMBER, NRINTERVAL or CHARINTERVAL a simple edge from a node to an other node is added

labeled with the corresponding character, number, symbol or interval of characters or numbers. When the node is NRINTERVAL or CHARINTERVAL the program must add separate edge for each element in that interval, otherwise mistakes when converting NFA into DFA will occur, examples of these problems will be introduced later on.
The PLUS, QMARK and ASTERISK nodes are followed by one of the following child nodes: GROUP, CHAR, SYMBOL, NUMBER, CHARINTERVAL or NRINTERVAL. The difference is that they generate different graphs, in the Figure 3.15 are examples of how they are generated.

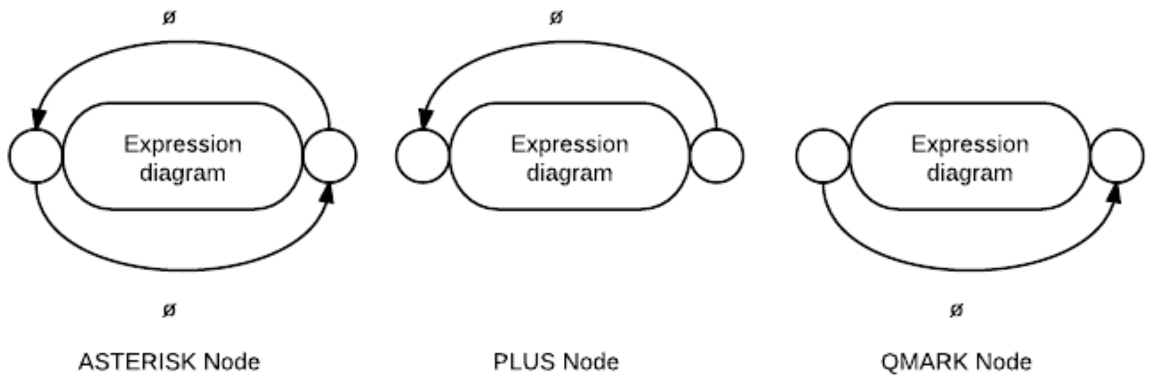


Figure 3.15 Construction of graph for Asterisk, Plus and QMark nodes

The expression diagram represents a subgraph, that can be a group or simple character, number, symbol or a type of interval.

The GROUP node can have one of the following child nodes: EXPRLIST or BOR. Considering the group child node the expression diagram can differ, so it may be the case than a huge graph can be in the expression diagram.

The concatenation of two graphs is made by adding an empty transition from the last node of the graph 1 to the first node of graph 2, but, there is no such easy way to add that transition between two objects in this case graphs. We must create a new graph and add all nodes and edges of graph 1 and graph 2 then we can merge them. The following algorithm has been used for merging the graphs.

```
createNewGraph temporaryGraph
iterate each node of graph 1 and add to temporary graph
iterate over each node of graph 1
    for each node iterate over successors of that node
        for each edge between the node and the successor add an edge into temporary
        graph for these two nodes

iterate each node of graph 2 and add to temporary graph
iterate over each node of graph 2
    for each node iterate over successors of that node
        for each edge between the node and the successor add an edge into temporary
        graph for these two nodes

get the last node of graph 1
get the first node of graph 2
add edge empty from last node of graph 1 to the first node of graph 2
```

Table 3.2 Algorithm for merging the graphs

Since the algorithm for merging the graphs creates many empty transitions, we must implement an algorithm for removing these empty edges which minimize the NFA graph also. By removing the empty transitions we can loose the elegance of the NFA graph, so the algorithm pretends to remove all empty transitions in the same time we carry to keep the rule that each node must have at most 2 predecessors or successors except the cases when PLUS, QMARK or ASTERISK have occurred. The algorithm below explains this step:

```
iterate over each node of graph
if the node is not a accepted state
    get indegree and outdegree
    if indegree and outdegree is equal to 1
        if there is an empty inEdge and outEdge
            link predecessor of this node with the successor of the same node with
            an empty transitions
            remove the node
        else if inEdge is empty and outEdge is not empty
            link predecessor and successor with a transition labeled with the
            same label as outEdge label
            remove the node
        else if inEdge isNot empty and outEdge is empty
            link predecessor and successor with a transition labeled with the
            same label as inEdge label
            remove the node
```

Table 3.3 Algorithm for removing unnecessary empty transitions

3.3 Turning the NFA into an equivalent DFA using the Subset Construction Algorithm

The goal of the Subset Construction Algorithm is converting a NFA graph with zero or more epsilon transitions and multiple transitions on a single character into an equivalent DFA graph with no epsilon transitions and unique transition on a single character that means a unique path for each accepted sequence of input characters.

While translating the NFA into an equivalent DFA the problem is that the DFA automaton's number of states can be exponential in the number of NFA automaton's states. This exponential growth of the states is not a concern for simple NFA's with 3 states because the maximum number DFA states can be 8, but it gets difficult when we have 10 states in NFA that means thousand states in DFA, or it can get even worst with 20 states in NFA that gives million states on DFA. In practice the situation is not so bad as it looks in theory, because most of NFA's when converted, the number of states in DFA does not grow much at all, in fact sometimes it is fewer than in NFA.

As shown by Aho, Sethi and Ullman, [1] the subset algorithm works as follows:

```
The start state of DFA are all states of the NFA that can be reached with empty transition
For each new state of DFA do:
        for each character of alphabet
                move to all reached states in that character
                perform an empty closure for the set of states we got (the result from empty
                closure can be a new state or an already existing state)
                If at least one of the states from the result is accepted state in NFA it is
                accepted state in DFA as well
```

Table 3.4 Subset construction algorithm

The epsilon closure function takes as parameter a set of states and return another set of states containing all those states that can be reached on epsilon transition including the current states.

The algorithm below is equivalent with the algorithm above but expressed with pseudocode for the current problem:

```
emtpyClosures(ListOfNodes)
newListOfNodes
foreach state from ListOfNodes
    Iterate over each successor of the node
        Iterate over all edges between the current node and the successor of the node
            if the edge is an empty transition and newListOfNodes does not
            contain that successor node, add successor node to newListOfNodes
return newListOfNodes
```

Table 3.5 emptyClosures function

The emptyClosure algorithm goes through all states reachable from the states on no input character (so called empty transition). This function returns at least such nodes as the input listOfNodes has. That means that for each state we have at least one state reachable, namely itself. For example if we look on the figure below (3.16), lets assume that we call the epsilonClosure with this set of states as input parameter: {0, 3}the resulting states will be {0, 2, 3, 4}, this is because from state 0 with no input we can reach state 2, as well as from state 3 the state 4 is reached on empty transition.

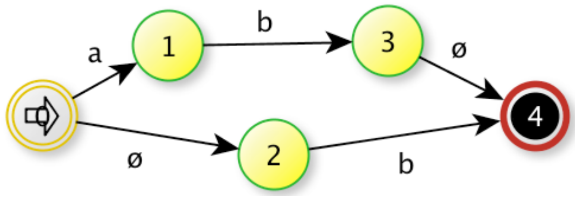


Figure 3.16 Graph automaton with epsilon transitions

Move function takes as parameter a set of states and the input character and return all the nodes that can be reached on the input character from all states in the set.

The algorithm below is equivalent pseudocode with the above algorithm:

```
move(ListOfNodes, inputCharacter)
newListOfNodes
foreach state from ListOfNodes
    Iterate over each successor of the node
        Iterate over all edges between the current node and the successor of the node
            if exist edge which label is equal with inputCharacter and
            newListOfNodes does not contain that successor node, add successor
            node to newListOfNodes
return newListOfNodes
```

Table 3.6 The move function of the automaton

From the Figure 3.16 we can see that if we perform a move function with {1} states as input parameter, and let the input character be “b”, the returning result will be {3}. When an empty closure of the current result is performed the result will be {3, 4} that means the string pattern is accepted since the final state has been reached.

For this algorithm to be more clear below is an example of transforming an NFA into an equivalent DFA. The regular expression is: a*c|bc. The corresponding NFA is shown in Figure below:

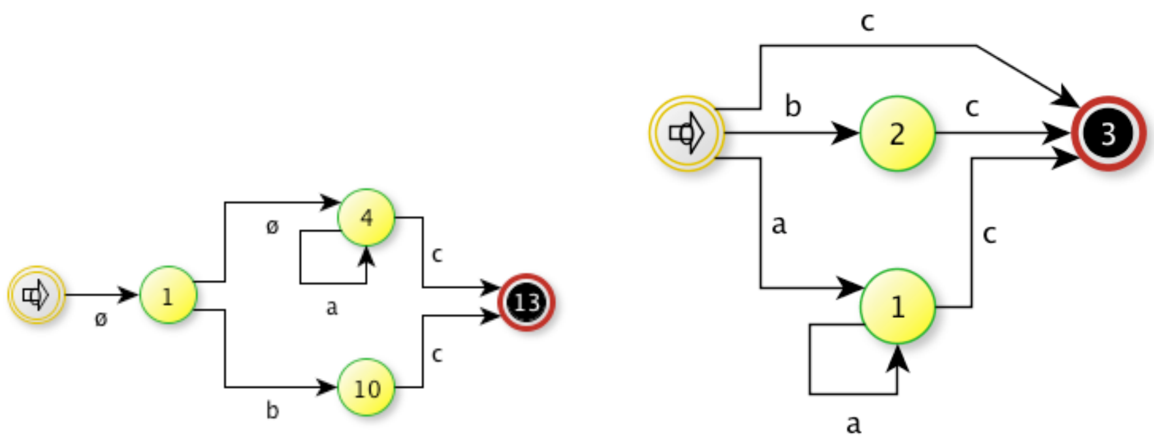


*Figure 3.17 NFA and DFA representing a*c|bc*

The start state of the DFA is the set of states reached in empty transition from start state of NFA, that is {0, 1, 4}. The alphabet of the NFA is: {a, b, c}.

Using the subset construction algorithm the following steps should be done:

1. Create start state of a DFA by taking epsilon closures of the start state of the NFA.
2. Then for each character of the alphabet perform move then epsilon closure of the results returned from the move function.
3. Repeat step 2 until we have no more new DFA states.

The following transitions has been made and the following DFA nodes has been created by following the subset construction algorithm. The empty states are not shown in the list below, these empty states are some non existing states which are returned while trying to move for instance with input character "a" on state 13 of NFA.

[0, 1, 4]	(a) ➞	[4] new state
[0, 1, 4]	(b) ➞	[10] new state
[0, 1, 4]	(c) ➞	[13] new state
[4]	(a) ➞	[4]
[4]	(c) ➞	[13]
[10]	(c) ➞	[13]

*Table 3.7 The transition table for a*c|bc regular expression while converting from NFA into DFA*

The equivalent DFA is shown in Figure 3.17. Note that all states of DFA are named as follows: we keep track of each new state created, for example the first state created that is the start state has number 0, then on each new state we increment this number per one, for example: 0=[0, 1, 4], 1=[4], 2=[10], 3=[13].

The accepted states of the DFA are all those states that contains at least one of the final states of the non deterministic finite automaton, in our example only state q3 of DFA contains state q13.

A description of this form with the list of transitions and new states created can be seen as program output also in the final application we are doing here.

3.4 GraphML and yEd Works

yEd is a powerful free application that can be used to generate high-quality diagrams in a quick and effective way. It runs on all major operating systems as Windows, Unix/Linux, and Mac OS X. It is based on the diagramming library so called yFiles for Java, which provides the intuitive user interface for creating and editing diagrams in an easy way, moreover some automatic layout algorithms and analysis tools. The main file format for saving yEd files is .graphml, that is a XML like document. yEd supports multiple formats export, as PSF, SWF, SVG, JPG, BMP, PNG, HTML image maps. The Regular Expression Code Generator software represent the NFA and DFA output graphs in a graphml form, that can be seen and modified in yEd and then can be exported into any other supported file format. The graphml (xml) file creation and the input FA xml file parsing has been done using the xml document builder respectively DOM parser. An example of how a graphml file looks like has been given in the following example:

```
<graph edgedefault="directed" id="G">
  <node id="n1"> The node properties and the label</node>
  <node id="n0">...</node>
  <edge id="e0" source="n1" target="n1">The edge label</edge>
  <edge id="e1" source="n0" target="n1">...</edge>
</graph>
```

Table 3.8 Graphml XML file example

Inside each node and each edge are the node properties that describe the color, position and the layout of the node respectively the edge.

3.5 Generating equivalent Java Class automaton from an existing DFA graph

By code generation we mean the compiler's process of converting some of intermediate representation of source code (in this case graphs) into an independent Java Class that whenever is executed represent the same graph automaton. As input for a code generator can be parse trees or abstract syntax trees. As stated in section 5 we use abstract syntax trees, that later on are converted into an intermediate language (sequence of instruction) such as graphs.

According to Ullman in [15] a "compiler" for regular expressions is useful to turn the expressions we write into executable code. During this project instead of compiler we will use two other terms: Automatic Programming, and/or Code Generator. According to Mur [16], automatic programming identifies a type of computer programming mechanism that generates a computer program (source code) to allow programmers to write high level code. Code

Generators or Application generators are software tools that helps programmers to generate a complete program or part of it in a very quick way according to the given input specification [17]. Using code generators the programmer can easily edit or modify and execute the output source (program). The major advantages of using code generators are:

- Saving a lot of development time
- Useful as a learning tool for writing code
- Programs are easy to modify and maintain

Each of these graphs represents a deterministic finite automaton that can be easily converted into a Java Class that accepts all the string patterns as the graph itself. According to [12] such programming of automaton is called Automata-based Programming, where program is considered as a model of a Finite State Machine. The reason it is called automata-based programming is because the whole execution of the program (automata-based) explicitly is a cycle of the automaton steps. The programmers style of thinking about representation of finite state machines in this technique is very similar to the style of solving automata in mathematical way, that is an other reason why this style of programming of automata is called automata-based programming.

There are many ways to interpret an automaton using Java Code, but we will talk more about most used ones, and we will describe the ones used in this project. One way of representing FSA is using enums, that essentially are list of classes. Each enum element may have a different implementation. For example, let us assume that we want to implement the automaton in Figure 3.18, that represent this regular expression R = a+b*c* using enum approach.

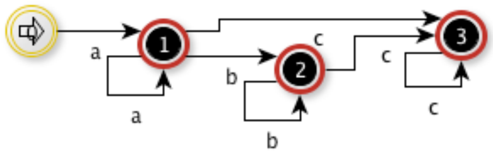


*Figure 3.18 Deterministic Finite Automaton representing a+b*c**

The enum approach works by writing states as enum elements, for example:

```
enum States implements State {
  State0 {
    @Override
    public State next(Input input) {
      switch(input.read()) {
        case 'a': return State1;
        default: return DeadState;}      }
  },
  State1 {
    @Override
    public State next(Input input) {
      switch(input.read()) {
        case 'a': return State1;
        case 'b': return State2;
        case 'c': return State3;
        case '': return null;
        default: return DeadState;}    }
  },
  ...
  DeadState {
    @Override
    public State next(Input input) {
      return DeadState;}
  };
```

Table 3.9 The enum approach example while implementing DFA in Java

Each enum element describes its functionality, by this we mean that each transition is defined. The advantage of using enum approach is that it is clean and simple approach. All the logic of the automaton is in the same place and it is easy to trace the automaton.

An other way of implementing a DFA into Java is by using map-like data structures, where keys represents the inputs and values represent the states. The transitions of a DFA automaton using the map-like approach looks like: HashMap<HashMap<State,Input>, State> that means that for each state on each input we go to an other state, where HashMap<State,Input> is the unique key representing the state from where on an input we go to an other State represented as the value of the map.

During this project two approaches of generating such a Java Class in an automatic way for a specific given regular expression has been introduced:

- if-else statement approach and
- graph approach.

When regular expressions are interpreted and used to generate code at runtime, a non-deterministic finite state machine is generated instead of a deterministic one, because the former can be created more quickly and occupies less memory space than the latter.

3.5.1 If-Else Statement Approach

The if-else approach is an easy way of implementing a DFA automaton, such a way works by creating an if statement for each input and the state of the automaton. The example in the Figure 3.18 using the if-else approach will look like the example below:

```
public class className {
        protected final int state3 = 3; protected final int state2 = 2;protected final int state1 = 1;
        protected final int state0 = 0; protected final int deadState = -1;
        protected int currentState = 0;
        public void update(String edge) {
                if(currentState == state3 && edge.equals("c")) { currentState = state3;
                }else if(currentState == state2 && edge.equals("b")) { currentState = state2;
                }else if(currentState == state2 && edge.equals("c")) { currentState = state3;
                }else if(currentState == state1 && edge.equals("a")) { currentState = state1;
                }else if(currentState == state1 && edge.equals("b")) { currentState = state2;
                }else if(currentState == state1 && edge.equals("c")) { currentState = state3;
                }else if(currentState == state0 && edge.equals("a")) { currentState = state1;
                }else { currentState = deadState; }}
        public boolean matches() {
                 if ((currentState == state3 || currentState == state2 || currentState == state1))
                        return true;
                else
                        return false; }
}
```

Table 3.10 The if-else approach example while implementing DFA in Java

The automaton has two methods, *update* and *matches*. The update method moves the current state of the automaton into another state depending on which input character of the input string is next read. When there is no more characters to be read from the input string the matches method is called, which simply checks if the current state of the automaton is one of the accepted states. The dead synonymously trap state represents a node when there is no transition from one state to another in the given input character, and when the automaton visits this state it never leaves it, so it mean that the given input string will never match.

There must be a main method that starts this automaton, and breaks the input string into characters, that can be done by reading the input string character by character, and for each character the update method of the automaton is called (Appendix A).

Probably this is not the best way to implement a DFA, but for beginners it is a good start.

3.5.2 Graph approach

There are two possible ways of representing graphs in computer sciences, adjacency matrix and adjacency list. Since graph representation is irrelevant for this project, instead of creating new graph representation, grail library will be used while representing directed graphs and while discussing the algorithm that describes the moves of the automaton. This means that when the graph output source is used the grail library must be imported (built to path) to run this program. To represent the automaton the first thing we have to do is build that automaton, so we should create for each state of the DFA a node in the Java class, and for each arc between states and edge should be created. All the nodes and edges are created in the buildGraph() method, where each node and edge is labeled. Unlike other approaches, the transitions are represented by connecting two nodes with an edge, that means that the source node “0” is connected with the target node “1” by an arc “a”, such that from state “0” with input character “a” the a transition to state “1” is made.

First, a global SetBasedDirectedGraph is created, together with all kind of state definitions:

- state - that is the node represented using DirectedNodeInterface,
- a dead state - that is useful for non existing edges between two nodes,
- currentState - that keeps track of the current position in the automaton, and
- edge - that is represented using DirectedEdgeInterface (Table 3.11).

```
private SetBasedDirectedGraph graph = new SetBasedDirectedGraph();
        private static DirectedNodeInterface state;
        private static DirectedNodeInterface deadState;
        private static DirectedNodeInterface currentState;
        protected DirectedEdgeInterface edge;
```

Table 3.11 The required global variables while implementing graph approach DFA in Java

To create a node, you must use the method createNode of the SetBasedDirectedGraph, and as a parameter the id of that node should be given. The properties as label or description is added using the method setProperty. The label is used to name the state, and the description is used to check whether that state is Initial, Accepted, Initial & Accepted or just a normal State.

```
state = graph.createNode(1);
        state.setProperty(GraphProperties.LABEL,"1");
        state.setProperty(GraphProperties.DESCRIPTION,"Accepted");
        graph.addNode(state);
state = graph.createNode(0);
        state.setProperty(GraphProperties.LABEL,"0");
        state.setProperty(GraphProperties.DESCRIPTION,"Initial");
        graph.addNode(state);
```

Table 3.12 Example of creating new nodes, and adding properties while implementing graph approach DFA in Java

The edges are created using the createEdge method of the SetBasedDirectedGraph class, where the source and the target must be specified, the label is added by using setProperty method.

```
edge = graph.createEdge(null,(DirectedNodeInterface) graph.getNode(3),
                            (DirectedNodeInterface) graph.getNode(3));
                    edge.setProperty((GraphProperties.LABEL), "c");
```

Table 3.13 Example of creating new edges, and adding properties while implementing graph approach DFA in Java

The algorithm on how the automaton changes from one node to another is described with the pseudocode below, or a Java working example is given in Appendix B.

```
Iterate over successors of the currentState
  Iterate over all edges between the currentState and the successors
    if the edge is the same as the character input
      return true
after each successor has been iterated an no edge has been found change the current state to
the trap state and return false.
```

Table 3.14 Pseudocode that describes the move function while implementing graph approach DFA in Java

After the end of the input string has been reached, the matches method is called, that simply checks if the description of the current node is Accepted or Initial & Accepted.

The source code output looks very good and simple until interval ranges have occurred in the regular expression input, the code gets too long because if between two nodes an “a..z” edge is presented, that means that 26 edges must be created, so for each character in that interval a separated edge is created. Because of the simplicity and readability in the NFA and DFA separate edges will not be added, instead a simple “from..to” edge will be created (be aware that in some cases the DFA is not correct, but the source code will be correct always), but the data structure behind works with separate edges. The reason this approach is used is the correctness of the source code. A better solution is required to simplify and minimize the source code, but so far it has not been proven that they are correct for each case, so that is a work to be done in future. If the interval would be treated as simplified version presented in the .graphml NFA and DFA files, the following problem will occur: assuming we want to filter a text and check the words that start and end with an **a**, the regular expression will look like: a[a..z]*a. If **a** and **a..z** will be treated as different edges, by this the automaton with an **a** goes to one state and with an **a..z** to another, without knowing that **a** is inside **a..z** interval, when converted into a DFA automaton a problem will occur, by which the DFA automaton will not be a DFA, but an NFA because with **a** you could move into two different states. Theoretically an attempt to fix this has been done by excluding the next variables from the interval, but even then there will be cases when this approach will not be useful. It will be a good approach in cases when there will be examples like this: [a..z]a?b, but it will not be a useful case when there will be something like this: [0..9]1?2*3*4*5?6?7*8?, in such case by excluding the next variables from different states again will be complicated ([0..9] except 1,2,3,4,5,6,7,8) that means only 9.

4. Summary

This chapter is a summary where are represented the results of the thesis, followed by the related and future work regarding this project.

4.1 Summary

Regular Expression Code Generator is an interactive software for visualizing finite automata, and converting these automatons in Java executable Source Code in a quick, fast and effective way.

It can be found very helpful tool within any course in Automata Theory or some Compiler courses. We intend Regular Expression Code Generator to be used by Students, Teachers and Programmers. For students it will be a good starting point to better understand RE, FA and how these FA are implemented in Java in different ways. For teachers it will be a perfect tool to check and correct student's assignments. Programmers could find it very useful to generate source code in a quick and fast way providing accessibility, performance, and improved programmer productivity while reducing time consumption.

Several examples, including the ones shown in this report have been used to test the application, and the results were successful as expected.

4.2 Related Work

There are some tools that converts the Regular expression into an equivalent NFA or DFA, and there are also many tools that generates source code from a given input, but the Regular Expression Code Generator, is the first tool that except converting the regular expression into NFA and DFA, it also generates equivalent Java Source Code that represents the automaton for the given regular expression as input.

JFLAP is a software tool that mainly operates with FSA, one of its functionalities is converting regular expression into a NFA or DFA. The difference is that in RegExCodeGen the regular expression specification is little bit extended, so that range of characters as [A..z] or range of numbers as [0..9] can be added, the question mark and plus operator (UNIX like) are introduced, and for future work it can be extended even more by introducing the A{n,m} or A{n} operators that represent at least **n** and no more than **m** occurrences of A, respectively exactly **n** occurrences of A. For example the regular expression a?bc is a shortcut for (!+a)bc in JFLAP where '!' stands for empty. The main idea of using '?' instead of introducing an other operator symbol is that users are more used with UNIX operators, and there is no need to learn new regular expression specification rules that can be used only for that specific application. As we mentioned JFLAP operates only with Finite State Automatons and there is no source code generation part yet.

Acceleo [18] (cross-platform / Eclipse Java) is a Model to Text tool that has the same aim as the Regular Expression Code Generator, the aim of helping developers to write the code in a faster way, a tool that is implemented in Java which as input takes user defined EMF based models (UML, Ecore, user defined metamodels) and generates any textual language as output. Jostraca is an other software tool that generates any code into Java or JSP code, this kind of tools are called source-to-source generators. Other than some of the code generator tools, RegExCodeGen is a platform independent software.

4.3 Future work

Extending the regular expression grammar in Regular Expression Code Generation Software is one of the features will be added later on, this one will provide users to describe more complex problems.

Generating a more optimized code in cases when range intervals are used will increase the maintainability and productivity of the generated source code, and will decrease the complexity even more.

Providing the enum and the map like approach while implementing finite automata using the source code generator will help programmers increase the productivity.

References

[1] A.V. Aho et al., “Lexical Analysis” in Compilers: principles, techniques, & tools, 2nd ed. Boston: Pearson/Addison Wesley, 2007, pp.128-163
[2] A. V. Aho, and J. D. Ullman “Patterns, Automata and Regular Expressions” in Foundations of Computer Science, New York: W.H. Freeman & Company, 1994, pp.530-571
[3] P. Linz “Introduction to Theory of Computation; Finite Automata; Regular Languages and Regular Grammars” in An Introduction to Formal Languages and Automata, 3rd ed. Massachusetts: Jones & Bartlett Learning, 2001, pp. 3-30; 37-70; 71-98
[4] J.E. Hopcroft and J.D. Ullman “Grammars; Finite Automata and Regular Grammars; Operations on Languages” in Formal languages and their relation to automata, New York: Addison-Wesley Pub. Co, 1969, pp.8-24; 26-43; 120-134
[5] Russ Cox (January 2007), Regular Expression Matching can be Simple and Fast (but slow in Java, Perl, PHP, Python, Ruby,...), [Online], Available: http://swtch.com/~rsc/regexp/regexp1.html
[6] ANTLR (2005), Abstract Syntax Tree, [Online], Available: http://www.antlr2.org/doc/trees.html
[7] Ashley J.S Mills (2005),ANTLR, [Online], Available: http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.html
[8] Amer Gerzic (november 2003), Write Your Own Regular Expression Parser, [Online], Available: http://www.codeproject.com/Articles/5412/Writing-own-regular-expression-parser
[9] Chia-Hsiang Chang , “From Regular Expressions to DFA’s Using Compressed NFA’s”, Courant Institute of Mathematical Sciences, New York University, October 1992.
[10] J.A. Storer “Pattern Matching” in An Introduction to Data Structures and Algorithms, 1ed. Boston: Birkhäuser, 2000, pp.374-379
[11] Bellur Ashwin, Pratik Mehta, Regular Expressions to DFA, [Seminar], Available: http://drona.csa.iisc.ernet.in/~deepakd/fmcs-06/seminars/presentation.pdf
[12] Automata-based programming, Bulletin of St Petersburg State University of Information Technologies, Mechanics and Optics 53. 2008
[13] A. V. Aho and J.D. Ullman “An Introduction to compiling; ” in The Theory of Parsing, Translation, and Compiling: Parsing, Texas: Prentice-Hall, 1972, pp
[14] Jeffrey Ullman, Video Lecture “Informal Introduction to finite automata”, Stanford University, May 2012, www.coursera.com
[15] J. E. Hopcroft, R. Motwani and J. D. Ullman “Finite Automata and Regular Expressions” in Introduction to automata theory, languages, and computation, New York: Addison Wesley, 2006, pp.13-45
[16] Ricardo Aler Mur, "Automatic Inductive Programming", ICML 2006 Tutorial. June 2006.
[17] Arora, Sh. Bansal and A. Arora “ Application Generators” in Comprehensive Computer and Languages, New York: Firewall Media, 2005, ch. 2 sec. 4.1, pp. 41
[18] http://www.eclipse.org/acceleo/

[19] M.V. Lawson "Introduction to finite automata; Non-deterministic automata; Kleene's Theorem" in Finite Automata, 1 ed. New York: Chapman and Hall/CRC, 2003, pp.1-14; 53-60; 97-114

Appendix A - ReadMe

Introduction

The objective of this document is demonstrating the use of this software. The aim of this software is to generate Finite Automata graphs or Java Code Classes for any Regular Expression. The following steps describes the functionality of the tool:

1. Generating Nondeterministic Finite Automata from the RegularExpression input.
2. Translating the NFA into an equivalent Deterministic Finite Automata.
3. Generating Java Code Classes for the DFA automaton.

The first window that will be shown upon the start of the software (Figure 1) will offer two options:

- Regular Expression to Finite Automata (FA) and Java Code
- Finite Automata to Java Code

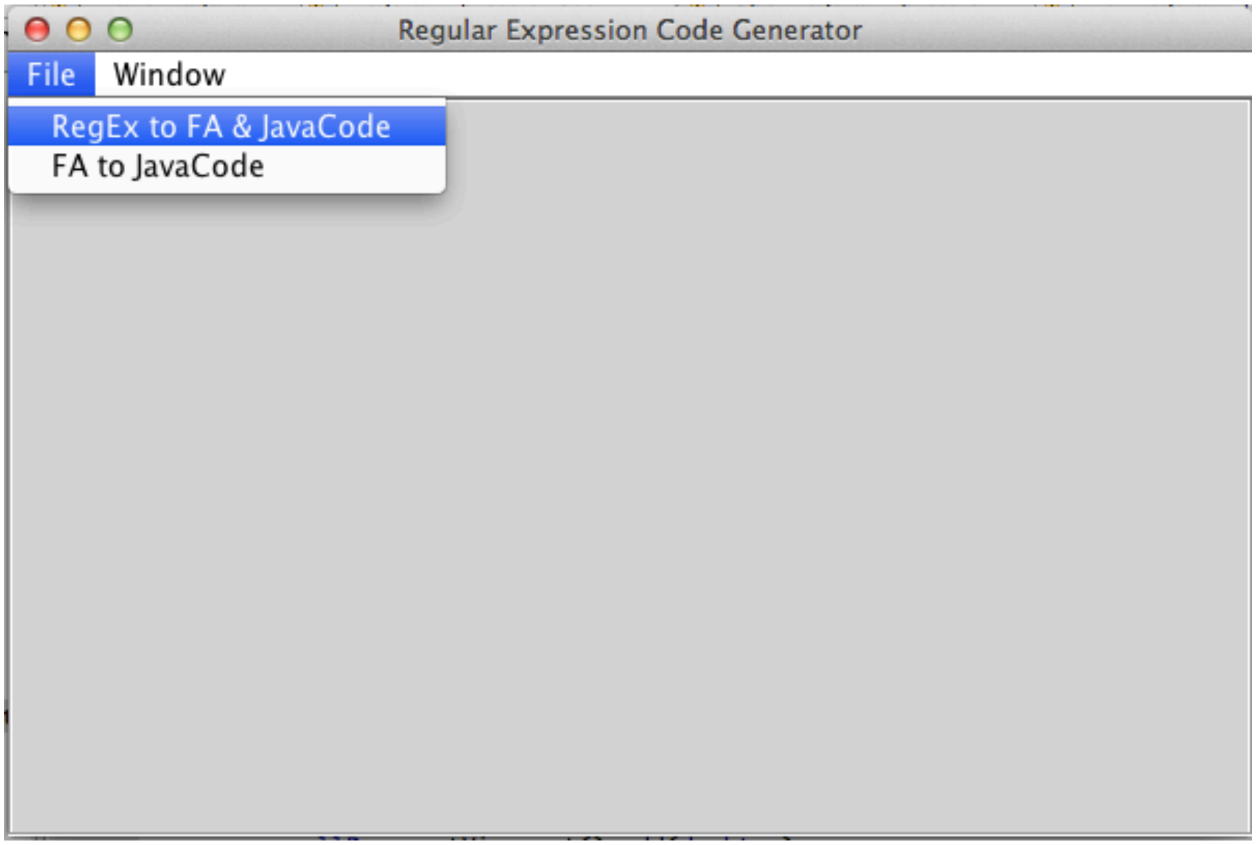
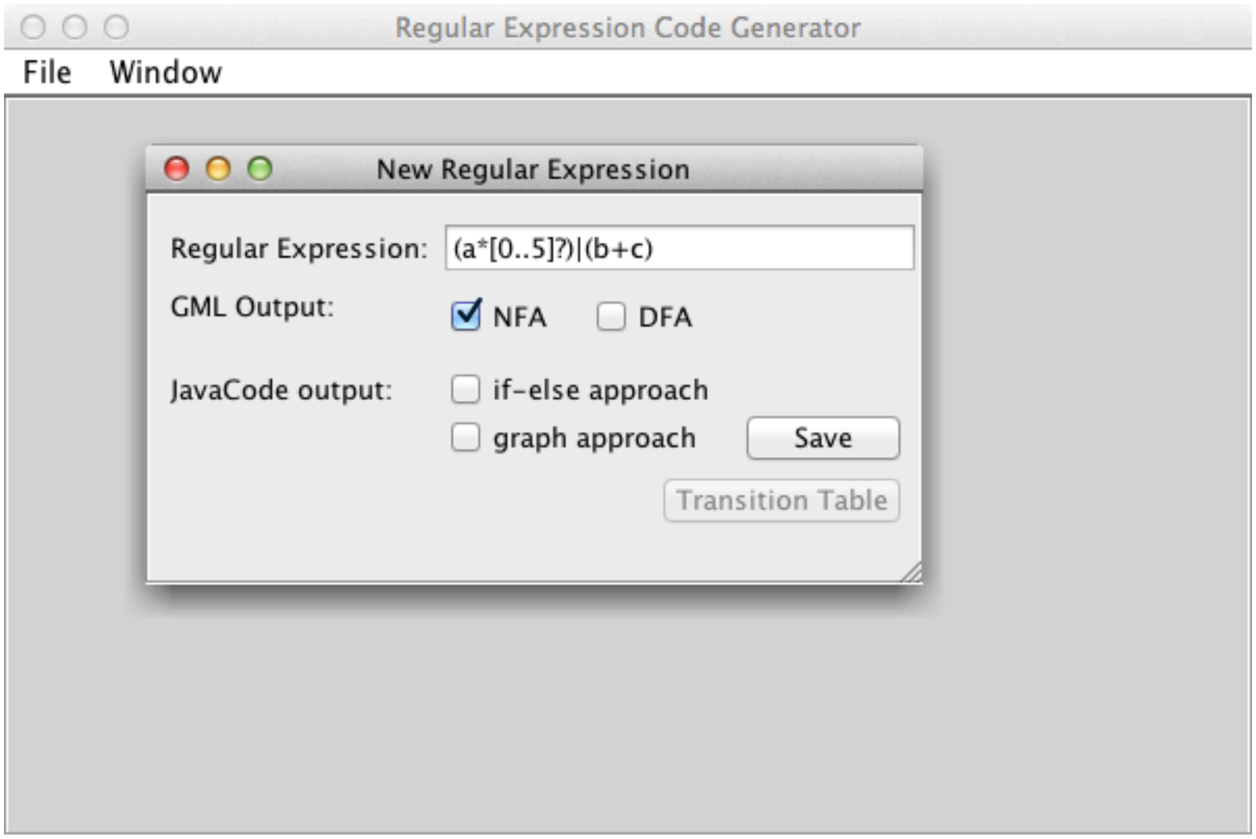


Figure 1

1. Regular Expression to NFA

After ‘RegEx to FA & JavaCode’ option from File menu of the main window has been selected a new window will appear (Figure 2), that expects as an input a valid regular expression (eg. (a*[0..5]?)|(b+c)) as described in Section 1.1.



Regular Expression: (a*[0..5]?)|(b+c)

GML Output: [✓ NFA] DFA

JavaCode output: if-else approach; graph approach

Figure 2

After the regular expression has been entered, in order to get the NFA, the NFA checkbox should be selected, and then click on Save button, then the save dialog will appear asking you to select the name and the directory of the output.

Then, in the selected directory a folder named GMLFiles will be created containing the NFA graph in form of .graphml file, that can be opened using yEd Works (Section 5).

The corresponding generated NFA graph for the given regular expression example in Figure 2 looks like the graph shown in figure 3.

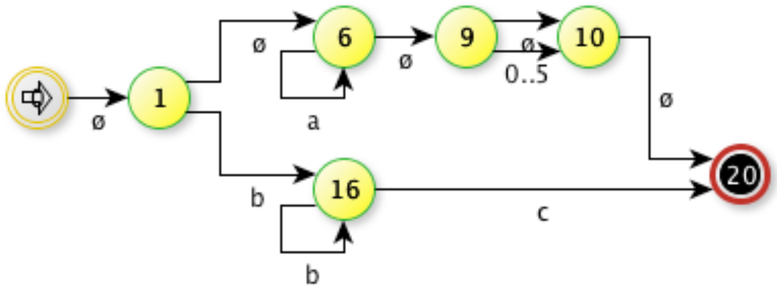


Figure 3

where:

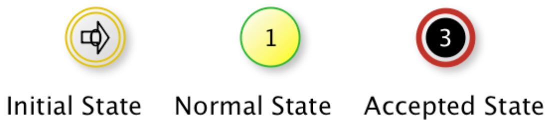


Figure 3.1

1.1. Regular Expression Convention

The default convention has been kept during this software, that means that you can write regular expression in the right way without character escape symbols. Conventionally the Kleene closure is written by adding the star '*' symbol after the subexpression you want, e.g. a*, (a|b)*, [A..z]*, etc. The positive closure is written in the same way as Kleene closure but here instead of '*' you write '+', e.g.: a+, (a|b)+, [A..z]+, etc. The character that represents zero or one subexpression is the question mark symbol '?', so you can create subexpression by adding it at the end, e.g. a?, (a|b)?, [A..z]?, etc. The union closure is expressed by the vertical bar symbol '|', e.g. a|b, (ab)*|c, etc. The range between two characters is expressed in this way: a..z, 0..9

The symbols like ?, +, *, | are reserved by the grammar, but you can use every other symbol as: '@', '#', '$', '%', '!', '~', '`', '^', '.', ',', ';', '<', '>', '/', '-'. And of course you can use any letter or number as alphabet.

An example of how a regular expression that accepts all float values is:

([1..9][0..9]*|0?.[0..9]+)|-(([1..9][0..9]+)|0.0*[1..9][0..9]*)

or the regular expression that validates email addresses:

[A..z]([A..z]|[0..9])+@([A..z]|[0..9])+(.[a..z]+)+

2. NFA to DFA

There isn't any additional step to do to generate the corresponding DFA graph, except selecting the DFA checkbox from Figure 2 and click Save.

The DFA graph will be generated in the same folder with the DFA suffix (e.g. tutorialDFA.graphml). The corresponding automaton for the NFA graph from Figure 3 is shown in Figure 4:

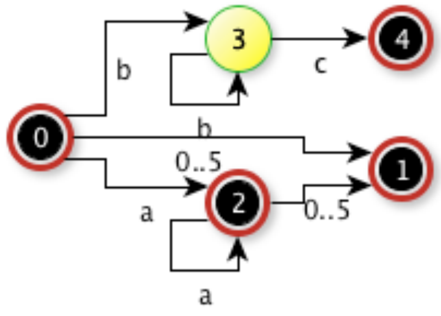
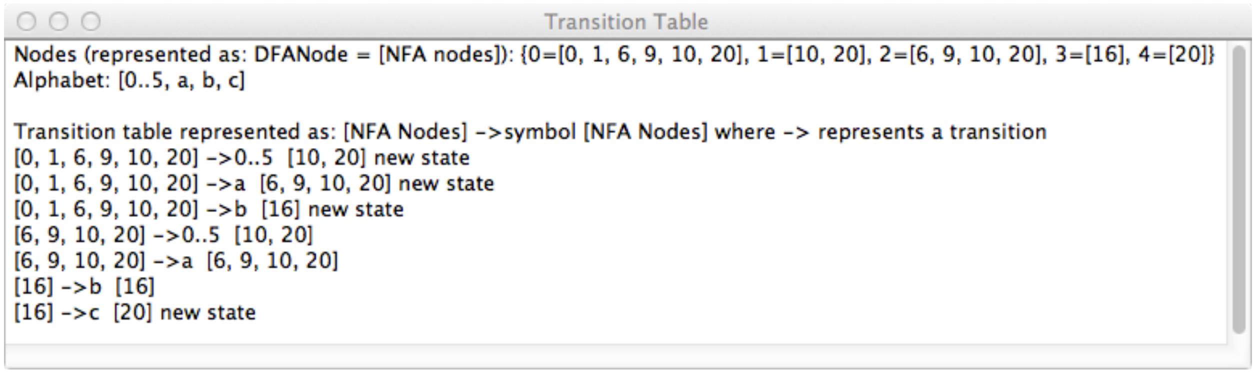


Figure 4

In cases where Initial State is also an Accepted State, the node looks like each other accepted node, but it can be recognized by label number 0(zero) or by clicking over the Initial State and reading the description of that node (Initial and Accepting State).

When the DFA checkbox is selected after clicking the Save button, the Transition Table button is clickable that provides the transition table (Figure 5) description how the DFA has been built from the NFA.



```
Nodes (represented as: DFANode = [NFA nodes]): {0=[0, 1, 6, 9, 10, 20], 1=[10, 20], 2=[6, 9, 10, 20], 3=[16], 4=[20]}
Alphabet: [0..5, a, b, c]

Transition table represented as: [NFA Nodes] ->symbol [NFA Nodes] where -> represents a transition
[0, 1, 6, 9, 10, 20] ->0..5  [10, 20] new state
[0, 1, 6, 9, 10, 20] ->a  [6, 9, 10, 20] new state
[0, 1, 6, 9, 10, 20] ->b  [16] new state
[6, 9, 10, 20] ->0..5  [10, 20]
[6, 9, 10, 20] ->a  [6, 9, 10, 20]
[16] ->b  [16]
[16] ->c  [20] new state
```

Figure 5

3. DFA to Java Code

To generate the Java Code Classes for the corresponding regular expression there are two options to select: if-else approach and graph approach.

A folder JavaCode will be created in the selected directory, containing the generated Java Classes. In our example while selecting the if-else approach the following code will be generated:

Main.java

```
import java.util.Scanner;

public class Main {

        public static void main(String[] args) {

                tutorial automaton = new tutorial();
                char nextInt;
                System.out.print("Enter the input string: ");
                Scanner scanner = new Scanner(System.in);
                String input = scanner.nextLine();
                for (int i = 0; i < input.length(); i++) {
                        nextInt = input.charAt(i);
                        automaton.update(nextInt + "");
                }
                String message = "";
                if (automaton.matches()) {
                        message = "The input string: " + input       + " is accepted by this
automaton.";
                } else {
                        message = "The input string: " + input       + " is NOT accepted by
this automaton.";
                }
                System.out.println(message);
        }
}
```

tutorial.java

```
/** @author: Suejb Memeti
 *The automaton is generated for this regular expression: (a*[0..5]?)|(b+c)
 */
public class tutorial {
	protected final int state4 = 4;
	protected final int state3 = 3;
	protected final int state2 = 2;
	protected final int state1 = 1;
	protected final int state0 = 0;
	protected final int deadState = -1;
	protected int currentState = 0;

	public void update(String edge) {
			if(currentState == state3 && edge.equals("b")) {
				currentState = state3;
			}else if(currentState == state3 && edge.equals("c")) {
				currentState = state4;
			}else if(currentState == state2 && ( edge.charAt(0) >= '0' && edge.charAt(0)
<= '5')) {
				currentState = state1;
			}else if(currentState == state2 && edge.equals("a")) {
				currentState = state2;
			}else if(currentState == state0 && ( edge.charAt(0) >= '0' && edge.charAt(0)
<= '5')) {
				currentState = state1;
			}else if(currentState == state0 && edge.equals("a")) {
				currentState = state2;
			}else if(currentState == state0 && edge.equals("b")) {
				currentState = state3;
			}else {
				currentState = deadState;
			}
	}
	public boolean matches() {
			 if ((currentState == state4 || currentState == state2 || currentState == state1 ||
currentState == state0))
				return true;
			else
				return false;
	}
}
```

If we select the graph approach the following Java code will be generated that represents the same automaton:

tutorial.java

```
import grail.interfaces.DirectedEdgeInterface;
import grail.interfaces.DirectedNodeInterface;
import grail.iterators.EdgeIterator;
import grail.iterators.NodeIterator;
import grail.properties.GraphProperties;
import grail.setbased.SetBasedDirectedGraph;
public class tutorial {

        private SetBasedDirectedGraph graph = new SetBasedDirectedGraph();
        private static DirectedNodeInterface state;
        private static DirectedNodeInterface deadState;
        private static DirectedNodeInterface currentState;
        protected DirectedEdgeInterface edge;


        public  tutorial() {
                buildGraph();
                deadState = graph.createNode(-1);
                deadState.setProperty(GraphProperties.DESCRIPTION, "DeadState");
                graph.addNode(deadState);
        }
        protected void buildGraph(){
                //Creating Graph Nodes (Automaton States)
                createNode("4", 4,"Accepted");
                createNode("3", 3,"null");
                createNode("2", 2,"Accepted");
                createNode("1", 1,"Accepted");
                createNode("0", 0,"Initial & Accepted");
                currentState = state;
                //Creating Graph Edges (Automaton Transitions)
                createEdge(3, 3, "b");
                createEdge(3, 4, "c");
                createEdge(2, 1, "0");
                createEdge(2, 1, "1");
                ...
public boolean update(String edge){

        NodeIterator succ = currentState.getSuccessors();
                while(succ.hasNext()){
                        succ.next();
                        EdgeIterator edges = graph.getAllEdges((DirectedNodeInterface)
                        currentState, (DirectedNodeInterface)succ.getNode());
                                while(edges.hasNext()){
                                edges.next();
                                if(edges.getEdge().getProperty(GraphProperties.LABEL).
                                equals(edge)){
                                        currentState = (DirectedNodeInterface) succ.getNode();
                                        return true;
                                }
                        }
                }
                currentState = deadState;
                return false;
        }

void createNode(string label, int id, String description){
                state = graph.createNode(id);
                state.setProperty(GraphProperties.LABEL, label);
                state.setProperty(GraphProperties.DESCRIPTION,description);
                graph.addNode(state);
        }

void createEdge(int from, int to, String label){
                edge = graph.createEdge(null,(DirectedNodeInterface) graph.getNode(from),
                (DirectedNodeInterface) graph.getNode(to));
                edge.setProperty((GraphProperties.LABEL), label);
                graph.addEdge(edge);
        }

public boolean matches() {
                if (currentState.getProperty(GraphProperties.DESCRIPTION).
                equals("Accepted"))
                        return true;
                else
                        return false;
        }
}
```

The Main.java class is exactly the same as in if-else approach. When the graph approach has been used you should keep in mind that the grail library should be built to path.

4. Finite Automata to Java Code

In case you have the NFA or DFA but you are missing the Regular Expression and you need the JavaCode, the software provides the opportunity to import the drawn graph in form of (yEd) .graphml file.

To draw the graph using yEd Works Software you should follow the rules below:

- Initial state should be labeled as 'Initial', 'I', 'Start' or 'S'.
- Initial & Accepted State is defined by adding the description of the node as 'Initial & Accepted State'.
- Accepted State should be labeled as 'Accepted', 'A', 'Final' or 'F'.
- The empty edge should be labeled with 'empty' or 'ø'
- Make sure all edges are labeled.
- The other states can be labeled anyway.
- The interval edges should be labeled as: 'A..Z' or '0..9' (represent the range between two characters)

In the Figure 7 is shown an example of a drawn graph using yEd.

When you are ready with the graph simply choose from file menu FA to JavaCode then the windows as in Figure 6 will appear from where you can choose the graph file and select the desired output as in examples before.

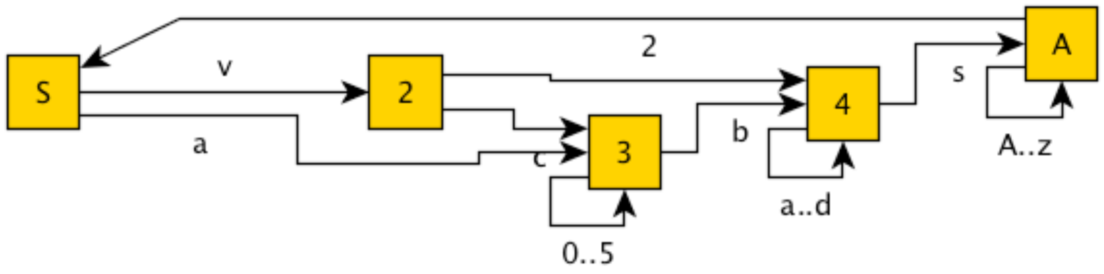


Figure 6

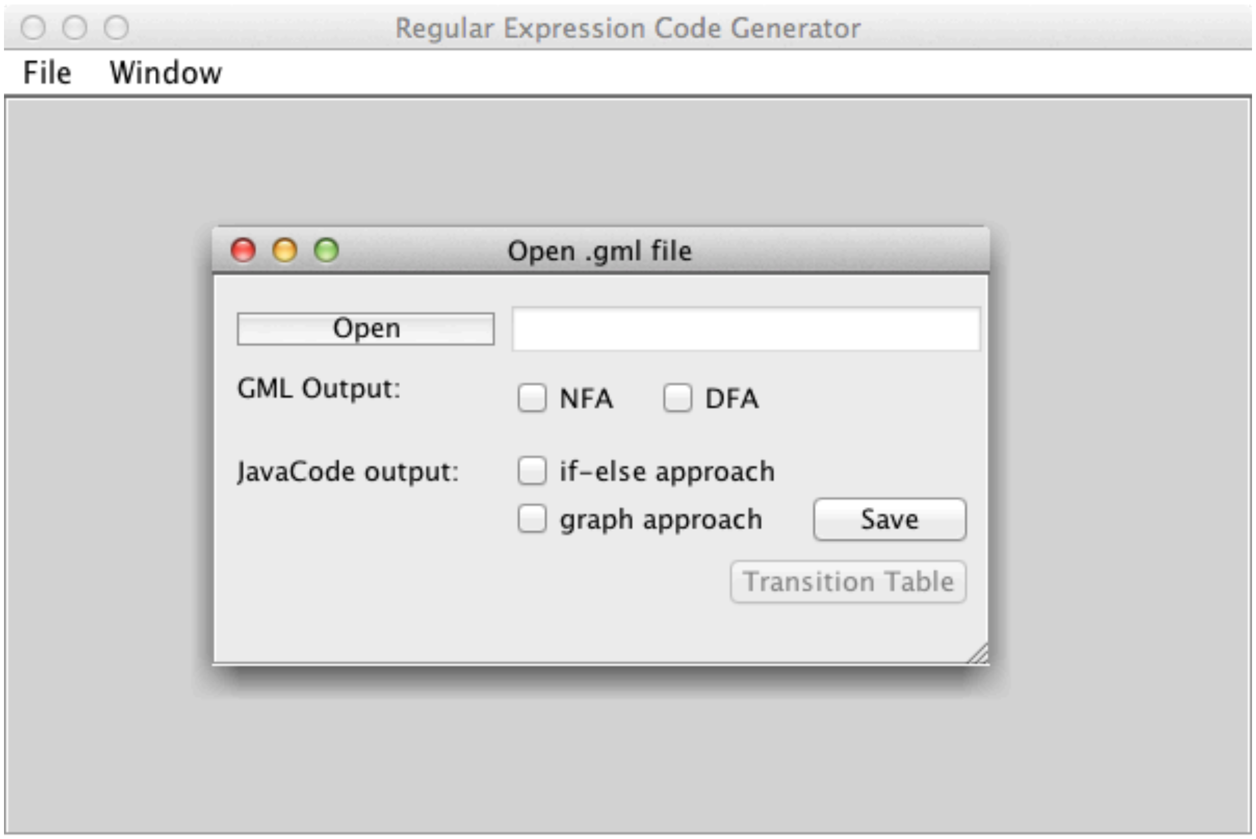


Figure 7

5. Requirements

To open and draw new graphml files you will need yEd Works, for better performance use yEd Works version 3.9.x.

To run graph approach Java Code Classes you need grail library that can be downloaded from the project website (www.s-solutions.info) and from official library web site.

Appendix B - Graph Approach automaton complete example

Following will be shown the automaton implementation of tennis score game described in Section 2.2 using the graph approach in Java.

The main class is the same one as in Appendix A (DFA to Java Code). The automaton class is shown in Table 1.

tennisScore.java

```
import grail.interfaces.DirectedEdgeInterface;
import grail.interfaces.DirectedNodeInterface;
import grail.iterators.EdgeIterator;
import grail.iterators.NodeIterator;
import grail.properties.GraphProperties;
import grail.setbased.SetBasedDirectedGraph;
public class tennisScore {
private SetBasedDirectedGraph graph = new
SetBasedDirectedGraph();
private static DirectedNodeInterface state;
private static DirectedNodeInterface deadState;
private static DirectedNodeInterface currentState;
protected DirectedEdgeInterface edge;
public  tennisScore() {
buildGraph();
deadState = graph.createNode(-1);
deadState.setProperty(GraphProperties.DESCRIP
TION, "DeadState");
graph.addNode(deadState);
}
protected void buildGraph(){

//Creating Graph Nodes (Automaton States)
createNode("19", 19,"null");
createNode("18", 18,"null");
createNode("17", 17,"null");
createNode("16", 16,"null");
createNode("15", 15,"null");
createNode("14", 14,"Accepted");
createNode("13", 13,"null");
createNode("12", 12,"null");
createNode("11", 11,"null");
createNode("10", 10,"Accepted");
createNode("9", 9,"null");
createNode("8", 8,"null");
createNode("7", 7,"null");
createNode("6", 6,"null");
createNode("5", 5,"null");
createNode("4", 4,"null");
createNode("3", 3,"null");
createNode("2", 2,"null");
createNode("1", 1,"null");
createNode("0", 0,"Initial");
currentState = state;
```

```
//Creating Graph Edges (Automaton Transitions)

createEdge(19, 17, "A");
createEdge(19, 14, "B");
createEdge(18, 10, "A");
createEdge(18, 17, "B");
createEdge(17, 18, "A");
createEdge(17, 19, "B");
createEdge(16, 17, "A");
createEdge(16, 14, "B");
createEdge(15, 10, "A");
createEdge(15, 17, "B");
eateEdge(13, 16, "A");
createEdge(13, 14, "B");
createEdge(12, 15, "A");
createEdge(12, 16, "B");
createEdge(11, 10, "A");
createEdge(11, 15, "B");
createEdge(9, 13, "A");
createEdge(9, 14, "B");
createEdge(8, 12, "A");
createEdge(8, 13, "B");
createEdge(7, 11, "A");
createEdge(7, 12, "B");
createEdge(6, 10, "A");
createEdge(6, 11, "B");
createEdge(5, 8, "A");
createEdge(5, 9, "B");
createEdge(4, 7, "A");
createEdge(4, 8, "B");
createEdge(3, 6, "A");
createEdge(3, 7, "B");
createEdge(2, 4, "A");
createEdge(2, 5, "B");
createEdge(1, 3, "A");
createEdge(1, 4, "B");
createEdge(0, 1, "A");
createEdge(0, 2, "B");
}
```

```
public boolean update(String edge){
        NodeIterator succ =
        currentState.getSuccessors();
        while(succ.hasNext()){
                succ.next();
                EdgeIterator edges =
graph.getAllEdges((DirectedNodeInterface)curr
entState,
(DirectedNodeInterface)succ.getNode());
        while(edges.hasNext()){
                edges.next();
if(edges.getEdge().getProperty(GraphProperties.
LABEL).equals(edge)){
currentState = (DirectedNodeInterface)
        succ.getNode();
        return true;
                        }
                }
        }
        currentState = deadState;
        return false;
}
void createNode(String label, int id, String
description){
    state = graph.createNode(id);
    state.setProperty(GraphProperties.LABEL,
label);
state.setProperty(GraphProperties.DESCRIPTIO
N,description);
graph.addNode(state);
}
```

```
void createEdge(int from, int to, String label){
  edge = graph.createEdge(null,
(DirectedNodeInterface) graph.getNode(from),
(DirectedNodeInterface) graph.getNode(to));
edge.setProperty((GraphProperties.LABEL), label);
 graph.addEdge(edge);
}

public boolean matches() {
if(currentState.getProperty(GraphProperties.DESC
RIPTION).equals("Accepted") ||
currentState.getProperty(GraphProperties.DESCRI
PTION).equals("Initial & Accepted"))
        return true;
else
        return false;
        }
}
```