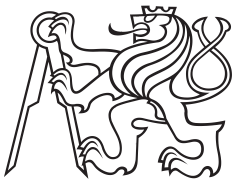
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Platforma pro kreslení diagramů konečných automatů

Tomáš Hořovský

Školitel: RNDr. Marko Genyk-Berezovskyj
Leden 2019

Poděkování

I thank to my family and my supervisor for support in dire times. ...

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt

Klíčová slova:

Školitel: RNDr. Marko Genyk-Berezovskyj
Praha, Na Zderaze 269/4, room: G-9

Abstract

The goal of this project was to develop new coding language for description of automata and operations with them, implement interactive shell interface for executing the commands and finalize the **jautomata** library for operations on automata. The language operates the **jautomata** library and implements export of automata to various output formats including LaTeXcode to display the automaton.

Keywords:

Title translation: Finite Automata Drawing Platform

Contents

Figures

Tables

Chapter 1

Introduction and motivation

This project started as a passion of mine for Automata, where I needed a tool to do simple Automata operations and I kept adding new functionality, until the original jautomata-cpp library was so messy I could not orient in the code very well. After some time of struggling with the code I needed to choose the subject of my software project and my bachelor's thesis. It was only natural that I would rewrite the whole library properly. **jautomata** library was the result. In my software project I wrote most of the **jautomata** library, while in my bachelor's thesis I finished the library and I started working on **JASL** and the interpreter for this language. Both jautomata library and JASL interpreter are written in pure object-oriented Java.

1.0.1 Description of the project

Java Automata Syntax Language (abbreviated to JASL) is a scripting language, developed as a part of this project, that allows the user to define and work with acceptor finite state machines. This project implements an interpreter for this language, that functions as a live console environment. One of the main features of JASL is the ability to export state machines to diagrams in format native to TEX.

1.0.2 Motivation

When I wrote my own material for Automata and Grammars, I stumbled upon the problem of visualising automata in the document. I wanted fast and reliable way to draw automaton diagrams in place in code, not having to include image files to the compilation folder. I searched for a suitable way to do so and I found **TikZ**. TikZ is a powerful image drawing library that has many features. I tried drawing automaton directly with TikZ, but the

code was unnecessarily long and tedious to write. After a couple of diagrams I started looking for another option. Then I found a library for TikZ called **automata**. It was just what I was looking for. It could draw nodes and edges nicely, while keeping the code simple and clear.

Next problem on the line was to draw these diagrams, so that they are as simple as possible. Mostly eliminating crossing edges did the trick. However the more complex the diagram got, the harder it was to eliminate those by hand. I used *Graphviz* to do the layout work for me. Then it was all about the process of converting Graphviz output to the TikZ code.

Automata have a few common operations associated with them. These include reduction, deciding whether $w \in L$, constructing automaton that accepts language $L = L_1 \cup L_2$ or even automaton that accepts L^*. I decided to create a library that would implement all of these operations and more. There are libraries that can do these operations such as Algorithms Library Toolkit [6], but they are usually complicated to use or they are not public and they can not output directly to LaTeXcode.

The goal of this project is to write a program that would implement intuitive command line interface for operating my jautomata library that contains most of the commonly-used algorithms for working with automata. It would also allow the user to convert automata to various output formats including LaTeXcode.

The implemented solution uses various other programs and libraries to make the codebase smaller. It uses tools such as **Graphviz** or **graphviz-java** library.

Chapter 2

Definitions and terminology

In this section I will define terminology used in this document to describe the relation of the application and language theory. Most of the definitions in 2.1,2.2,2.3 and 2.4 are translations of [2].

2.1 Languages

- **Terminal** a is a sequence of one or more characters.
- **Alphabet** is a finite non-empty set Σ of terminals a.
- **Word** w **over an alphabet** Σ is a finite sequence of terminals: $w = a_1 a_2 \dots a_m, a \in \Sigma, m \geq 0$.
- **Length of word** w, written as $|w|$ is the number of terminals in the word w.
- **Empty word** ε is a word that has length $|\varepsilon| = 0$.
- **All words over an alphabet** Σ, written as Σ^* is a set of all words that can be created using terminals from Σ and ε.
- For two words $w_1, w_2 \in \Sigma^*, w_1 = a_1 a_2 \dots a_m, w_2 = b_1 b_2 \dots b_n$ the result of the **concatenation** operation is: $w_1 w_2 = a_1 a_2 \dots a_m b_1 b_2 \dots b_n$.
- **Language** L over an alphabet Σ is an arbitrary subset of Σ^*.

2.2 Operations over languages

We will use these operations over languages in this document:

- **Concatenation of languages** $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ is a set that is generated as follows:

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \subseteq (\Sigma_1 \cup \Sigma_2)^*$$

- **Union of languages** $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ is a set that is generated as follows:

$$L_1 \cup L_2 = \{w \mid w \in (L_1 \cup L_2)\} \subseteq (\Sigma_1^* \cup \Sigma_2^*)$$

- **Intersection of languages** $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ is a set that is generated as follows:

$$L_1 \cap L_2 = \{w \mid w \in L_1, w \in L_2\} \subseteq (\Sigma_1^* \cap \Sigma_2^*)$$

- For any language L we define $L^0 = \{\varepsilon\}, L^{i+1} = L^i L$ for $i \geq 0$. We define the result of **Kleene operation** as:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

- For any language L over an alphabeth Σ we define its **complement** as: $\overline{L} = \{w | w \notin L, w \in \Sigma^*\}$

2.3 Automaton

The term automaton in language theory includes many types of automata, such as Moore automaton, Mealy automaton or others [7]. However, JASL and jautomata library implement only these types of automata:

2.3.1 Deterministic Finite Automaton (DFA)

Deterministic Finite Automaton M is defined as a tuple with five elements: $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite non-empty set of states
- Σ is a finite non-zero set of inputs
- δ is the transition function $\delta : Q \times \Sigma \to Q$
- q_0 is an initial state from the set Q
- F is a subset of so-called accepting states from the set Q

Let $w = a_1 a_2 \dots a_n$ be a word over an alphabet Σ. Automaton M accepts the word w if a sequence of states $r_0, r_1, \dots, r_n, r \in Q$ exists such that:

$$\begin{aligned} q_0 &= r_0, \\ r_{i+1} &= q(r_i, a_i), i = 0, 1, \dots, n-1 \\ r_n &\in F \end{aligned}$$

In other words: We can say that automaton M accepts word w if

$$\delta^*(q_0, w) \in F$$

Where $\delta^* : Q \times \Sigma^* \to Q$ is an extended transition function that is defined by induction as:

$$\begin{aligned} &1 : \delta^*(q, \varepsilon) = q, && q \in Q \\ &2 : \delta^*(q, wa) = \delta(\delta^*(q, w), a), && a \in \Sigma, w \in \Sigma^*, q \in Q \end{aligned}$$

2.3.2 Non-deterministic Finite Automaton (NFA)

Non-deterministic Finite Automaton ia a tuple: $M = (Q, \Sigma, \delta, I, F)$. Q, Σ, F definitions are the same as in DFA automaton.

1. I is a set of states that are considered initial: $I \subseteq Q$.
2. $\delta : Q \times X \to P(Q), P(Q)$ is a set of all subsets of states: $P(Q) = \{X \mid X \subseteq Q\}$

We define extended transition function for NFA as $\delta^* : Q \times \Sigma^* \to P(Q)$ by induction as:

$$\begin{aligned} &1 : \delta^*(q, \varepsilon) = \{q\}, && q \in Q \\ &2 : \delta^*(q, wa) = \bigcup \{\delta(p, a) \mid p \in \delta^*(q, w)\}, && a \in \Sigma, w \in \Sigma^*, q \in Q \end{aligned}$$

2.3.3 Non-deterministic Finite Automaton with epsilon transitions (ENFA)

ENFA differs from regular NFA automaton by introducing so-called ε-transitions. These allow the automaton to transition between states without reading any terminal. Formally we change the transition function as follows:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$$

To define extended transition function for ENFA, we need another function called ε-closure. This can be defined as:

1. $X \subseteq \varepsilon\text{-closure}(X), X \subseteq Q$
2. If $p \in \varepsilon\text{-closure}(X)$, then $\delta(p, \varepsilon) \subseteq \varepsilon\text{-closure}(X)$

Now we can define extended transition function as:

$$1 : \delta^*(q, \varepsilon) = \varepsilon\text{-closure}(q), \qquad q \in Q$$
$$2 : \delta^*(q, wa) = \cup\{\varepsilon\text{-closure}(\delta(p, a)) \mid p \in \delta^*(q, w)\}, \qquad a \in \Sigma, w \in \Sigma^*$$

2.4 Regular expression

Regular expressions are used to describe regular languages. In [2] regular expressions are defined as follows:

Suppose we have an alphabet Σ. The set of all regular expressions over Σ is defined as:

- $\emptyset$ is a regular expression
- ε is a regular expression
- $\mathbf{a}$ is a regular expression for every terminal $a \in \Sigma$
- If $\mathbf{r_1}, \mathbf{r_2}$ are regular expressions, then $\mathbf{r_1} + \mathbf{r_2}, \mathbf{r_1 r_2}, \mathbf{r_1^*}$ are regular expressions

Every regular expression over an alphabet Σ represents a language over an alphabet Σ as follows:

- Regular expression $\boldsymbol{\emptyset}$ represents language $\emptyset$
- Regular expression $\boldsymbol{\varepsilon}$ represents language $\{\varepsilon\}$.
- If $a \in \Sigma$ then regular expression: $\mathbf{a}$ represents language $\{a\}$
- If regular expression $\mathbf{r_1}$ represents language L_1 and regular expression $\mathbf{r_2}$ represents language L_2, then regular expression $\mathbf{r_1} + \mathbf{r_2}$ represents language $L_1 \cup L_2$ and regular expression $\mathbf{r_1 r_2}$ represents language $L_1 L_2$
- If regular expression $\mathbf{r}$ represents language L, then regular expression $\mathbf{r}^*$ represents language L^*

In addition, brackets in regular expressions define the order of operations. Regular expression $\mathbf{r}$ describes the same language as regular expression $(\mathbf{r})$.

2.5 Notation of automata

In this document automata are often described by their transition tables. These tables contain full information needed to construct the tuple $(Q, \Sigma, \delta, I, F)$. We construct this table from the tuple as follows:

The table has $k+2, k = |\Sigma|$ columns and $n+1, n = |Q|$ rows. In the first row of the table there are two empty cells followed by all unique elements of Σ. The next n rows are each for one state $q \in Q$. In the first column is $IO(q)$, which holds information about I and F:

$$IO(q) = \begin{cases} \rightarrow, & \text{if } q \in I \text{ and } q \notin F \\ \leftarrow, & \text{if } q \notin I \text{ and } q \in F \\ \leftrightarrow, & \text{if } q \in I \text{ and } q \in F \\ \text{empty}, & \text{otherwise.} \end{cases}$$

In the second column is the name of state q. The remaining cells describe the function δ. For every column header $t \in \Sigma$ and state q the cell contains the set $\delta(q, t)$.

Table that is the result of this process describes fully all of the elements of the tuple and can be easily reconstructed into the tuple.

Example. Suppose we have a non-deterministic finite automaton $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0, q_1\}, \{q_0, q_3\})$, where δ is defined as:

$$\begin{aligned} \delta(q_0, 0) &= q_1, & \delta(q_0, 1) &= \emptyset, \\ \delta(q_1, 0) &= q_2, q_3, & \delta(q_1, 1) &= q_0, \\ \delta(q_2, 0) &= q_1, & \delta(q_2, 1) &= \emptyset, \\ \delta(q_3, 0) &= \emptyset, & \delta(q_3, 1) &= \emptyset. \end{aligned}$$

We can construct the transition table:

		0	1
$\leftrightarrow$	q_0	q_1	$\emptyset$
$\rightarrow$	q_1	q_2, q_3	q_0
	q_2	q_1	$\emptyset$
$\leftarrow$	3	$\emptyset$	$\emptyset$

Chapter 3

User manual

3.1 Installation

There are two ways of installing this program. You can either download pre-compiled .jar file or compile it on your own. If you just want to use the pre-compiled jar, skip right to the running section 3.2

3.1.1 Compiling JAR yourself

If you want to compile it yourself, you have to get source code of the project from this repository. After that, you can install it using Maven and JDK.

Open console in the root directory of the downloaded project and run these commands:

```
mvn clean
mvn install
```

After building the project, you can find the compiled .jar file in the target folder. Use the compiled .jar with dependencies.

3.2 Execution

The program can be executed from the console with this command:

```
java -jar <path-to-jar> <args>
```

If no args are specified, the program will enter interactive shell mode where you can type your commands and get immediate response for every command. The shell will store your variables to memory and you can use them freely. However after terminating the shell environment (by using command: **quit**) all saved variables are lost. The same effect can be achieved even without closing the environment by using command **clear**.

If switch -f is specified, the program will look for its argument, which should be the path to an existing file. JASL will then execute commands from this file line by line. Note, that all variables are lost after terminating the program.

You can execute file from shell, where the interpreter uses variables from current session. This can be done by using execute function 3.3.2.

3.3 Syntax of the language

The **JASL** language allows you to define variables and call functions upon those variables. Commands are parsed line by line. On every line there either an assignment, an expression, a comment or a command. There are three commands in JASL: **clean**, **help** and **helpLong**. These are described in more detail later.

Grammar of the language. The following is an abstract grammar describing the JASL language. Terminals are shown in red color. Square brackets enclose optional parameters. Vertical bars separate alternatives.

line	→	expression | assignment | comment | ε
variable	→	$variableName
variableName	→	ANY
comment	→	%ANY
expression	→	string | list | functionCall | variable.memberFCall | variable
list	→	{listItems} | {}
listItems	→	expression | expression, listItems
functionCall	→	functionName([args])[memberFCall]
args	→	expression[, args]
memberFCall	→	memberFName([args])[memberFCall]

Where non-terminal **ANY** can be any string that does not contain linebreak, non-terminal **functionName** is any function name from chapter 3.3.2, non-terminal **memberFName** is any function name from chapter 3.3.4 non-terminal **string** is any string that:

- Does not start with any keyword or any of these symbols: **{ , % $**
- Does not contain linebreak

Expressions. Expression is a function call, member function call, string or list definition. Expressions are evaluated before assignments. This evaluation typically produces new value of type string that contains the result of the evaluation. If an expression is called by itself, the console prints the return value.

Variables. Variables are used to store objects. Variable can be any string that starts with **$** and does not contain whitespace or a dot.

```
$<variableName>
```

Assignment. Assignments save <expression> return value to an <variable>. If the variable does not exist, it creates it. This is done using '=' operator.

```
<variable> = <expression>
```

Functions. Function calls consist of the name of the function followed by a comma-separated arguments enclosed in a pair of parentheses.

```
<functionName>(<arguments>)
```

Comments. You can add comments to your JASL code by the means of line comments. Every line that should be a comment starts with % sign. Everything that follows the percent sign will not be parsed and the whole line will be skipped.

```
%<commentString>
```

Help. Help for the JASL syntax can be displayed with command **help** while **helpLong** prints longer, more detailed version with descriptions of functions.

3.3.1 Data types

Variables can hold objects of types: string, list or automaton.

String. This data type can hold any string of characters that does not contain linebreak and does not start with any keyword or any of these characters: **{ $**

List. List in JASL is an ordered set of objects. Lists are enclosed in pairs of curly brackets. Elements are separated by commas. Elements can be any objects. Lists can be empty and they can be nested. They are used for defining automata. Some examples of lists are:

```
{a, b, c}
{}
{a, {b, {}}, c}
```

Automaton. Automaton type will be further explained in 3.3.3

3.3.2 Functions

In this section I will describe the functions that are implemented to JASL syntax in more detail. Example code uses assignments to indicate return type.

execute

```
execute($path)
```

Executes script on specified path. The argument is a string that contains absolute or relative path to a file in filesystem. It uses currently defined variables for the execution and update them.

A script is a file that contains lines with JASL code. It is recommended to give JASL scripts the .jasl extension.

fromCSV

```
$M = fromCSV($path)
```

Returns new Automaton object, loaded from comma-separated csv file specified in the single argument of this function. The argument is a string that contains absolute or relative path to a file in filesystem. In the CSV file should be the automaton table.

getExample

```
$automaton = getExample()
```

Returns example automaton. The example automaton is described by this transition table:

		a	b
$\rightarrow$	0	1	2, 3
$\rightarrow$	1		1, 4
$\leftrightarrow$	2		0
$\leftarrow$	3	3	3
	4	4	2

Table 3.1: Transition table of example automaton

And it's state diagram:

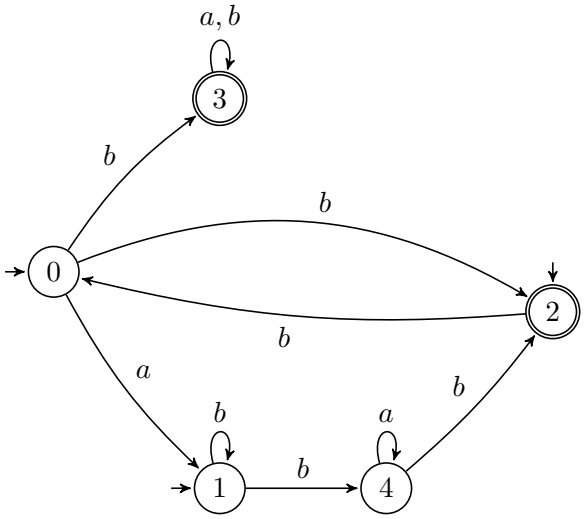


Figure 3.1: State diagram of the example automaton

fromRegex

```
$automaton = fromRegex(a*b(a+b)*)
```

Returns new Automaton object specified by regular expression passed in as an argument. The regular expression has to be in format specified in chapter 2.4 or format described in 3.3.4.

There are limitations of this function. It works only with single character

long terminals. Characters can not be escaped, so symbols '(', ')', '·' and '*' can not be used as terminals.

To create an automaton from regular expression, the JAutomata library uses a modified version of the algorithm from Melichar p.86 Algorithm 2.107 [7]

getTikzIncludes

```
getTikzIncludes()
```

Returns TEXcode to use TikZ package and libraries necessary for diagrams of automata to work.

These are:

```
\usepackage{tikz}
\usetikzlibrary{shapes,angles,calc,quotes,arrows,automata,positioning}
```

3.3.3 Automata

To define an automaton you need to use the Automaton function. This function accepts a single parameter: nested list $L = \{l_s, l_1, l_2, \ldots, l_n\}, n = |Q|$, where Q is the set of states of the automaton we want to define. Elements of list L are:

- disjoint list of terminals: $l_s = \{t_1, t_2, \ldots, t_k\}, t_i \in \Sigma, i = 1 \ldots |\Sigma|$
- n lists where each list $l_i = \{\mathrm{IO}, q_i, \delta(q_i, t_1), \delta(q_i, t_2), \ldots, \delta(q_i, t_k)\}$, where each of $\delta(q_i, t_j)$ is a list of target states and IO is defined as:

$$IO = \begin{cases} <>, & \text{if } q_i \in F, Q_i \in I \\ <, & \text{if } q_i \in F \\ >, & \text{if } q_i \in I \\ \text{empty string}, & \text{otherwise} \end{cases}$$

In other words this parameter is the transition table of the automaton. Lists in the definition are the rows of transition table read from left to right, separated by commas.

Example of conversion:

		a	b
$\leftrightarrow$	0	$\emptyset$	2
$\rightarrow$	1	0	$1,2$
$\leftarrow$	2	$1,2,3$	1
	3	3	$\emptyset$

$\rightarrow$

		a	b
<>	0	{}	2
>	1	0	$\{1,2\}$
<	2	$\{1,2,3\}$	1
	3	3	{}

$\rightarrow$

$\{a,b\}$
$\{<>,0,\{\},2\}$
$\{>,1,0,\{1,2\}\}$
$\{<,2,\{1,2,3\},1\}$
$\{3,3,\{\}\}$

Table 3.2: Example of conversion of transition table to list

So the argument to construct this automaton is:

```
{{a,b},{<>,0,{},2},{>,1,0,{1,2}},{<,2,{1,2,3},1},{3,3,{}}}
```

The automaton specified by the transition table is NFA automaton. It is created by using the Automaton constructor. The definition of the nested list can be split into multiple list variables for the sake of clarity.

```
1   $alphabet = {a, b}
2   $row0 = {<>,0,{},2}
3   $row1 = {>,1,0,{1,2}}
4   $row2 = {<,2,{1,2,3},1}
5   $row3 = {3,3,{}}
6
7   % Now we can define the nested list:
8   $nestedList = {$alphabet, $row0, $row1, $row2, $row3}
9
10  % And now we can define an automaton:
11  $automaton = Automaton($nestedList)
```

Note about ENFA automata. ENFA automata can have ε-transitions. These are defined using keyword eps as one of terminals. That terminal then signifies an ε transition. The alphabet of some ENFA automaton could be:

```
1   $alphabet = {eps, a, b}
```

3.3.4 Member functions

Member function (method) is a function called on an object saved in a variable. There are no member functions for objects of type list. It can be invoked as follows:

```
% If the function returns some value
$returnVal = $variable.functionName(<list of args>)
% If the function does not return any values
$variable.functionName(<list of args>)
```

Note that member function calls can be chained on one line:

```
1  $reduced = $automaton.reduced()
2  $reduced.toPNG(image.png)
3
4  % Can be written as:
5  $automaton.reduced().toPNG(image.png)
```

There is only one member function defined for string type objects:

save

```
$myString.save($path)
```

Saves string saved in the variable $myString to document at $path. If the document does not exist, it creates it. If the document already exists, it appends the string to the end of the document.

What follows is a list of member functions for automata objects. The notation for examples is the same as in section 3.3.2

accepts

```
$M.accepts($w)
```

Returns *true* if automaton M (saved in variable $M) accepts word passed in argument ($w \in L(M)$). It outputs *false* otherwise. The argument of this function can be a string or a list of terminals. If the argument is of type string, it is parsed character by character into terminals. Note, that if you have an automaton that has any terminals with more than one character, you cannot use the variant with argument of type string. In that case you need to use list as an argument.

equals

```
$M1.equals($M2)
```

Returns *true* if $L(M1) = L(M2)$. It outputs *false* otherwise. In other words this function checks, whether two automata accept the same language.

reduce

```
$M2 = $M.reduce()
```

Returns reduced automaton $M2$. Note that this function creates a new automaton object, so the original automaton remains unchanged.

toCSV

```
$M.toCSV($path)
```

Creates/overwrites csv file on path specified by the argument. The argument is a string that contains absolute or relative path to a file in filesystem. In the created CSV file there is transition table of the automaton.

toPNG

```
$M.toPNG($path, circo)
```

Creates/overwrites png file on path specified by the argument. The argument is a string that contains absolute or relative path to a file in filesystem. The png contains image of the state diagram of the automaton M.

The second argument of toPNG is optional. It is the layout (engine) that Graphviz will use to organize the graph. When no layout is specified, **dot** is used as a default. Possible layouts are: **circo**, **neato**, **dot** and **twopi**.

toTexTable

```
$M.toTexTable()
```

Outputs string containing TEXcode to display the transition table of automaton M.

toRegex

```
$M.toRegex()
```

Outputs regular expression describing language $L = L(M)$. Because no regular expression simplifier is implemented, the output of this function can be quite complicated. Nevertheless, it describes the language L. The output has a slightly different format from the definition in 2.4. In the output, concatenation of regular expressions $\mathbf{r_1}, \mathbf{r_2}$ is written as $\mathbf{r_1} \cdot \mathbf{r_2}$ instead of $\mathbf{r_1 r_2}$. This format is also accepted by 3.3.2.

To convert automaton to regular expression the JAutomata library uses a modified algorithm described in Melichar p.98 Algorithm 2.120 [7].

toDot

```
$M.toDot(neato)
```

Outputs dot code, that contains description of the automaton state-diagram image. It accepts one, optional argument. The argument is the layout (engine) that Graphviz will use to organize the graph. When no layout is specified, **dot** is used as a default. Possible layouts are: **circo**, **neato**, **dot** and **twopi**.

toSimpleDot

```
$M.toSimpleDot()
```

Outputs dot code, that contains description of the automaton state-diagram image. As opposed to toDot function, the dot code does not contain positions of elements, because it was not run through Graphviz yet.

toTikz

```
$M.toTikz(dot)
```

Outputs TikZ code to display the state diagram of automaton M. It accepts one parameter, that is the layout (engine) graphviz will use to organize the graph. When no layout is specified, **dot** is used as a default. Possible layouts are: **circo**, **neato**, **dot** and **twopi**. It is recommended not to specify this argument (hence use dot as an engine), because it mostly outputs the nicest

results out of the options. Note that you need to add appropriate includes to your TEXcode. You can get these using getTikzIncludes function.

union

```
$M3 = $M1.union($M2)
```

Outputs new automaton M_3 that accepts union of languages accepted by automata M_1, M_2. It accepts one argument of type automaton.

$$L(M_3) = L(M_1) \cup L(M_2)$$

intersection

```
$M3 = $M1.intersection($M2)
```

Outputs new automaton M_3 that accepts intersection of languages accepted by automata M_1, M_2. It accepts one argument of type automaton.

$$L(M_3) = L(M_1) \cap L(M_2)$$

kleene

```
$M2 = $M1.kleene()
```

Output new automaton M_2 such that:

$$L(M_2) = L(M_1)^*$$

complement

```
$M2 = $M1.complement()
```

Returns automaton that accepts language, that is the complement to the language of the original automaton.

$$L(M_2) = \overline{L(M_1)}$$

concatenation

```
$M3 = $M1.concatenation($M2)
```

Outputs new automaton M_3 that accepts the concatenation of languages accepted by automata M_1, M_2. It accepts one argument of type automaton.

$$L(M_3) = L(M_1)L(M_2)$$

renameState

```
$M1.renameState(0, 2a)
```

Renames state of caller automaton. This function accepts two arguments. The old state name as first and the new state name as second argument. It fails if the original state is not found in the automaton or if the new name is already taken by some other state of the automaton. This function modifies caller automaton object.

renameTerminal

```
$M1.renameTerminal(a, css)
```

Renames terminal of caller automaton. This function accepts two arguments. The old temrinal name as first and the new terminal name as second argument. It fails if the original terminal is not found in the automaton or if the new name is already taken by some other terminal of the automaton. Also you cannot use 'eps' or ε as a terminal because that is a keyword of epsilon tranisiton. You cannot use this function to add or remove epsilon transitions from the table. This function modifies caller automaton object.

Chapter 4

Details of Implementation

In this chapter I will describe various details of the implementation and some problems that I found when implementing the application.

4.1 Used technology

4.1.1 JAutomata

JASL interpreter needed some backend that would execute algorithms on automata and which would house the automata themselves. I developed **jautomata** library just for this reason. It is a library that allows the user to define automata and execute various operations on them. Because I wrote the library, I had the source code and I could make certain changes to the way the library works. For example, I had to fix some bugs that were in the CSV loading code and I had to implement new constructor functions for the Automaton object. For this reason I decided to work with the code directly and not to pack it into separate .jar file.

The **jautomata** library was developed by test-driven development. This meant that most of the functionality in the library was already unit-tested so I could rely on the algorithms to work properly.

Acceptor object

When I wrote the library I encountered an interesting problem with word accepting. Previously I used a function that would use Java objects like HashMaps and Lists to find out if a word was accepted by the automaton. While working on the library I was concerned about the speed of this operation.

I found an algorithm for bit-vector implementation of running an ε-NFA automaton [8]. I implemented this algorithm in AutomatonAcceptor object. Theoretically, it should be much faster, than the original solution, because it did not use any objects. Computers generally do bitwise operations very fast, so I thought that this would be much faster than my previous version. I tested both versions on multiple automata and various lengths of words. To my surprise, the operation got actually slower when using the newly implemented algorithm. The object-oriented way was faster, even on automata with fewer than 32 states.

I came to the conclusion that the bitwise algorithm was slower, because of Java's inner workings. This algorithm is much faster when implemented in C++. In the final version of the library I used the object-oriented way.

Automaton types

The jautomata library distinguishes between deterministic, non-deterministic and epsilon non-deterministic Finite Automata. In the library there is an object for each of those types. I originally implemented separate constructor functions for these types to the JASL language, but soon I realized that functionally they were indistinguishable from each other. Because of that, I refactored the language to have only one constructor function for all automata. Because both NFA and DFA automata are special cases of ENFA automaton, I use ENFA object for all automata defined in JASL except for reduced automata. The user of JASL cannot distinguish between inner types of automaton.

4.1.2 Graphviz

Graphviz [3] is an open source tool for graph visualization. I used graphviz to organize state diagrams of automata. Graphviz uses dot language to describe graphs. It has several output formats which include png image, plain text or even dot code. The user can pass in dot code where only a couple attributes are specified (nodes, edges, colors of the edges, ...). Graphviz will output dot code with all attributes specified (node position, size, edge anchor points, ...). JASL interpreter uses graphviz to get PNG images and to get layouts for getTikz conversions. More on that in section 5.1.

4.1.3 Graphviz-java library

Graphviz-java [4] is a library that parses dot code into objects in Java and vice versa. I used Graphviz-java for parsing and extracting attributes from

dot code. I encountered several problems with this library, more on that in section 4.3.2

4.1.4 TEX

TEX [11] is a typesetting system that is widely used to publish academic texts mainly in the fields of mathematics, physics and computer science. There are many extensions, packages and software bundles for TEXwhich give TEXmore variability. One of these native packages is the TikZ package.

4.1.5 TikZ and automata for TikZ

TikZ is a native TEXpackage for creating vector graphics that is built on top of PGF package. It allows user to draw diagrams and graphs in an intuitive way.

TikZ has a library made for drawing automata. This library is very well documented. I have used this library extensively to write my own texts about Automata so I was very familiar with the syntax. I decided to use it as an output platform, because of it being user-friendly and immediately usable in TEXcode to generate images.

The syntax of the code to draw automata in TikZ is very well described in this tutorial: [5]

4.2 Details of implementation

This project can be divided into 2 main sections: the interpreter and the convertor. Each of these parts had it's intricacies. In this section I want to focus on those.

4.2.1 JASL interpreter

JASL language does not use "" to define strings as most programming languages do. This has an interesting side effect, where if name of a function is mistyped, the whole expression is evaluated as a string:

```
>> $a = frmRegex(a*babb*a)
>> $a
frmRegex(a*babb*a)
```

I wanted to develop a way to make JASL more compact, so that the scripts do not have to be unnecessarily long and verbose. The first step was to allow in-place constructors. To achieve this I added two features to the interpreter: in-place constructors and member function chaining.

In-place constructors

In-place constructors allow the user to define and immediately use objects in expressions. I solved this by evaluating expressions recursively. For example, if a function call is being evaluated, the interpreter splits the function call into $n + 1$ parts: function name and it's n arguments. On each of those n arguments the interpreter calls evaluate function again. This allows the user to write condensed code where it is needed. It can also reduce unnecessary variable declarations which could potentially be quite problematic in larger programs.

```
1  % Code without in-place construction
2  $automaton1 = fromRegex(a*(b+a))
3  $exampleAutomaton = getExample()
4  $automaton1.equals($exampleAutomaton)
5
6  % Code with in-place construction
7  getExample().equals(fromRegex(a*(b+a)))
```

Member function chaining

The second step was to make chaining of member function calls possible. Calling a member function of an object in JASL results in new object. This feature allows the user to call member functions on these resulting objects on the same line as they were defined. Again, this reduces the number of unnecessary variables in the environment.

To do this, the interpreter splits the expression into member function calls and then evaluates them one by one. The actual implementation of this process is rather intriguing.

Suppose we have an expression:

```
$automaton.reduce().toPNG(test.png)
```

We run evaluate on this expression.

Evaluate first splits the expression into three parts:

- **V** - variable, which in this case is $automaton
- **F** - first member function call, which in this case is .reduce()
- **R** - rest of the expression, which in this case is .toPNG(test.png)

Then it evaluates expression **VF**, which in this case is $automaton.reduce(). It saves the result in temporary variable **T**. JASL uses $TEMP as a temporary variable. At last it runs evaluate on expression **TR**, which in this case is $TEMP.toPNG(test.png).

In each step the interpreter evaluates the front-most function call. Each time it saves the result in temporary variable **$TEMP** and removes the first function call from the expression. Then it replaces this function call by $TEMP and runs the evaluation again on the shortened expression.

As a side-effect, it would overwrite $TEMP variable. As a countermeasure added a little stack-frame just for this variable. Adding a little stack-frame just for this variable solved this issue. This stack frame was implemented using Java's native stack-frame. The implementation is simple, because the algorithm uses recursion to evaluate chained function calls. However, this solution has a little flaw. If one of the member functions has a syntax error in it, the interpreter throws an InvalidSyntaxException and the stack frame will be lost. It is not recommended using $TEMP as a variable name.

The chaining feature was added to the JASL language after the main framework had been completed and this process was the easiest to implement. The way it is implemented is taxing on computation time because of all the unnecessary variable name parsing and it should be upgraded in the future.

```
1  % Code without chained member function calls
2  $automaton = getExample()
3  $reducedAutomaton = $automaton.reduce()
4  $tikzCode = $reducedAutomaton.getTikz()
5  $tikzCode.save(test.txt)
6
7  % Code with chained member functions
8  getExample().reduce().toTikz().save(test.txt)
```

4.2.2 Conversion to TikZ

. . . TODO . . .

4.3 Problems with implementation

While implementing JASL I had to solve many problems. Some of these problems were caused by tools I was working with, or syntax checking.

4.3.1 Problems with syntax errors

One of the main goals of this project was to create live console environment. There are some libraries for Java that can create such environment, but they do not allow the user to define his own syntax. After searching the internet for a suitable framework, I decided to implement the console environment myself. Due to the nature of live console environment, I needed to handle many exceptions. Exceptions could occur because of non-existent file, invalid CSV files and syntax errors of the JASL language. Because of this, the code had to have solid exception handling. This turned out to be quite complex.

4.3.2 Problems with Graphviz-java

I thought that using this library would save me a lot of work on parsing dot code. Graphviz-java is originally meant to be used to construct graphs by the means of Java objects, then convert those object to dot code and vice versa. However, I wanted something different from the library. I wanted to use it only for parsing of the dot code. Then, I wanted to extract only layout information (x and y coordinates) from the objects.

Graphviz-java does not have any documentation on its classes or functions. So I had to reverse-engineer most of the fields of the object and their meaning, using java reflection and debugger. Fortunately, I found the attributes in the classes created by graphviz-java and was able to extract them.

I struggled on one particular bug in Graphviz-java library related to knot points extraction of the edges. If dot code from Graphviz contained any edge that was long enough to have more than nine anchor points, it would line-break the dot file in the middle of **pos** attribute. This caused Graphviz-java to parse it incorrectly. Fixing this issue made me think, whether using Graphviz-java was worth it in the first place.

So using graphviz-java for this use-case may not have been the best idea. It might have been easier to get plain text file as an output from Graphviz and parse it myself.

Chapter 5

Drawing images

Drawing state diagrams in such a way that they are readable and nice is a very complicated problem. I used Graphviz to create the layout of the graph. It draws nice diagrams, but I needed to convert the output to TikZ code. To do that I instructed graphviz to output so-called attributed dot code. This reproduces the input with added information about the layout of the graph. This dot code is then converted to TikZ code.

This gave birth to many problems with sizes of images, angle conversion and label positions.

5.1 Using Graphviz for layout output

One of the main problems of converting from dot code to TikZ code was the size of the output image. TikZ iteslf does not take care of page size. If input coordinates exceed page size, it will draw cropped image. The output of toTikz conversion should fit on regular A4 page so I needed a way to tell Graphviz the maximum dimensions of the output image.

There are many attributes that you can specify in the dot file: styling of the edges, positions of nodes and many more [9]. However, there are some, that have no effect on the attributed dot code because Graphviz often ignores some of the attributes to produce a better-looking image.

Size attribute is used to control the pixel size of output images. Graphviz copies size attribute to the attributed dot file, but it does not affect the

coordinates of the elements. It is included to the final dot only to instruct the rendering engine on how to scale/orient the resulting image. Because JASL reads position attributes from dot output format, the size attribute does not have any effect on them.

The attributed dot file contains coordinates for all elements of the graph. However, these are in Graphviz coordinate system. I needed to convert these coordinates to TikZ coordinate system. I measured the maximum feasible width/height of TikZ image to fit regular A4 page. I tried using linear mapping, but for that I would have to know the bounding box of the dot coordinates. Graphviz has bounding box attribute in the attributed dot file, that should specify the bounding box of the image. Unfortunately this value does not correspond to the maximum coordinate values of elements. It is only an instruction for the rendering engine that does not correspond to the element coordinates.

My only choice was to calculate the bounding box myself by finding the largest used coordinate on both axies. As a side-effect I lost information about spacing of the elements and maximum size. This means that the output TikZ code draws small graphs unnecessarily stretched. This is caused by mapping small image onto larger area. The following two images show png image generated by graphviz and the TikZ code result image in contrast with one another.

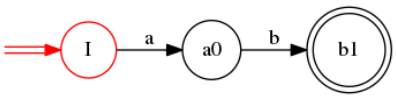


Figure 5.1: Original automaton image. Code A.1.1

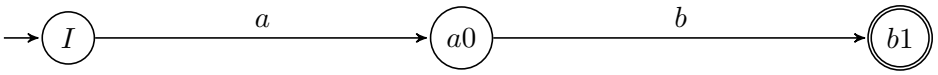


Figure 5.2: Example of stretched automaton. Code A.2.1

This unwanted effect could be one of the areas to work on in the future. 7.2.3

5.1.1 Graphviz layout engines

Graphviz has four main layout engines that can be used to draw automaton state diagrams:

- dot
- neato

- circo

- twopi

There are also other engines: sfdp, fdp. These engines are not implemented in JASL, because sfdp and fdp are for undirected graphs.

These engines produce vastly different images. I got the best results using dot engine, but for some particular examples **circo** yields more visually appealing images.

Example of layout difference

We have an automaton M. This automaton has four states. These are the images of automaton M generated by different layout engines:

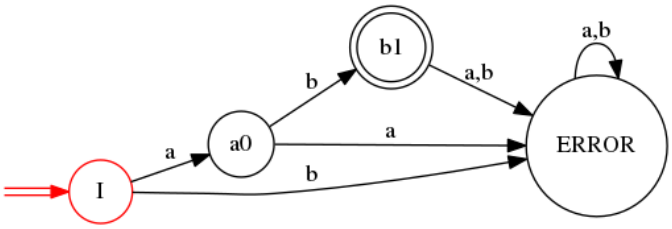


Figure 5.3: Image of automaton M generated using **dot** layout

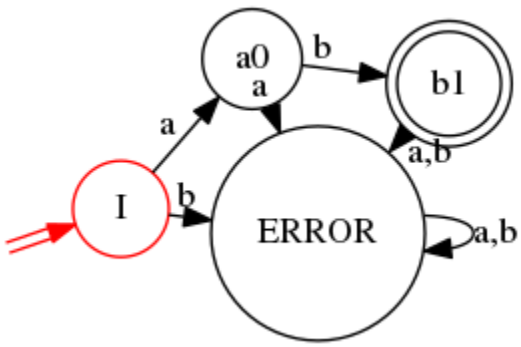


Figure 5.4: Image of automaton M generated using **neato** layout

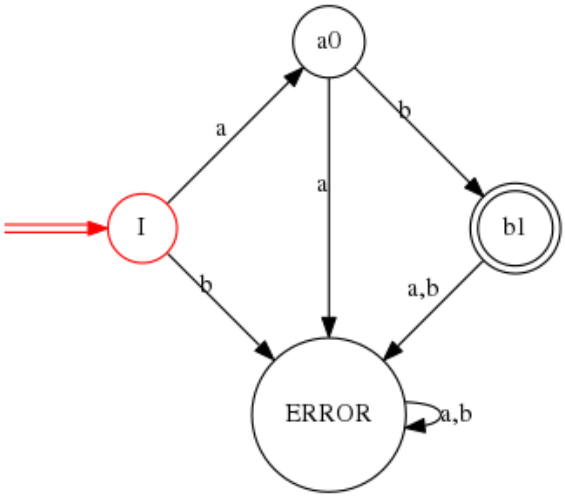


Figure 5.5: Image of automaton M generated using **circo** layout

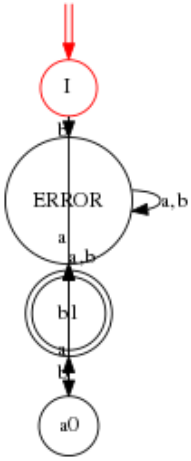


Figure 5.6: Image of automaton M generated using **twopi** layout

5.2 Relative position model

Originally, I wanted JASL to output TikZ code that would be the most easy for the user to edit. TikZ automata library was built to use the **relative** position model. In this model every element of the graph is placed in relation to some other element. This allows for very easy editing of the image. For example, nodes of the graph on figure 5.5 could be described as follows:

```
\node[state, initial]  (0)  {I};
\node[state]  (1) [above right of=0]  {a0};
```

```
\node[state]  (2) [below right of=0]  {ERROR};
\node[state, accepting] (3) [below right of=1]  {b1};
```

This code can be quickly read and edited as opposed to absolute coordinates. TikZ allows nodes to be in eight different directions: above, above right, right, below right, below, below left, left, above left. Distance between nodes can be specified, but that clutters the code. TikZ places nodes on outer edge of a circle with diameter equal to the distance between nodes. There are only three types of edge shapes: bend left, bend right and straight.

Using simplifications of the relative position TikZ code, poses several constraints on the layout generator. Graphviz does not have support for these types of constraints. Graphviz has no attribute for ideal edge length or grid alignment of nodes. So the only option is to calculate/approximate these positions. However, this approximation would destroy the layout completely at times.

5.2.1 Relation chains

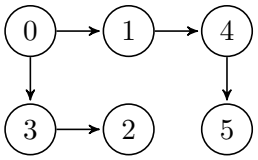
The relative model is easily editable, but only if the relations are connected right. The grouping of relations is a problem, because the program has to approximate, which nodes should be connected to which.

A simple example:

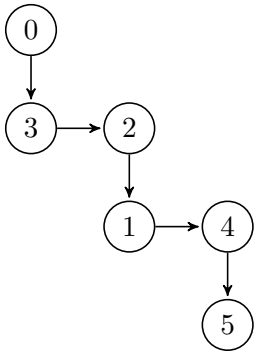
Suppose we have some graph, which is constructed using these relations:

1. 1 is to the right of 0
2. 4 is to the right of 1
3. 5 is under 4
4. 3 is under 0
5. 2 is to the right of 3

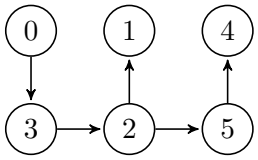
We can visualise these relations like this:



For some reason, we want to move nodes $1, 4, 5$ below node 2. It is very easy, just by changing the relation of node 1 to be: 1 is below of 2. The result will look like this:



Now suppose we have these relations in the graph:



Now it is much harder to do this simple operation, because to produce the wanted result, we have to change the relation of three nodes: $1, 4, 5$.

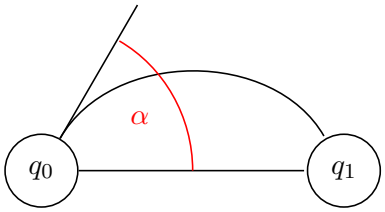
Because we do not know, if and how the user wants to reorganize the output graph, there would have to be some utility, that would re-route the edges. Without this utility, the relation model would be of no use.

Because of these difficulties, I discarded the idea of relative positions and used absolute positions in the final version of the program.

5.2.2 Edge angle calculation

The graphviz output dot file specifies edge shapes in the pos attribute. This attribute contains coordinates of several knot points, specifying a spline curve. As stated in previous section, there are three possible shapes of edges in automata library: straight, bend left, bend right. Approximation to those three edge shapes in TikZ produced mostly illegible images.

The ultimate goal was to keep the TikZ code simple and easy to read. I decided to use the angles library for TikZ, that allows the user to specify the angle of the bend. The angle is a numeric value between $0 - 360$, which specifies the angle α depicted in this picture:



Splines generated by graphviz are generally simple, mostly elliptical. I used this to approximate the curve angle using this function:

$$f(\vec{q_0}, \vec{q_1}, P) = angle(\vec{q_1}, \vec{q_0}, \vec{p}) \cdot ((\vec{q_1} - \vec{q_0}) \times (\vec{p} - \vec{q_0})), \text{ where}$$
$$\vec{p} = \arg\max_{\vec{p} \in P}(angle(\vec{q_1}, \vec{q_0}, \vec{p}))$$
$$angle(\vec{a}, \vec{b}, \vec{c}) = \frac{180 \cdot \arccos \vec{x} \cdot \vec{y}}{\pi \cdot |\vec{x}| \cdot |\vec{y}|}, \vec{x} = (\vec{a} - \vec{b}), \vec{y} = (\vec{c} - \vec{b})$$

where q_0 is the source node center, q_1 is the target node center and P is a set of spline knot points. This function returns an angle. If it is negative, the curve is to the right, if it is positive, the curve is to the left.

While this function has good results when graphviz uses elliptical edges, it is not so good otherwise. Graphviz uses ellipses only if the edges are short. If the edge exceedes certain point, it will flatten the spline curve. In these cases, angles output by function f are not optimal.

This function could be improved in the future 7.2.3

Chapter 6

Examples of usage, practice, problems of testing

Here are some examples of usage of the **JASL** language:

6.1 Defining a NFA automaton

Suppose we have regular language:

$$L_1 = \{w \mid w \text{ contains } aba \text{ as substring } \}, L_1 \subseteq \{a, b\}^*$$

We design regular automaton M such that $L(M) = L_1$. Example of such automaton could be this non-deterministic automaton:

	M_1	a	b
$\rightarrow$	0	0, 1	0
	1		2
	2	3	
$\leftarrow$	3	3	3

Table 6.1: Transition table of automaton M_1.

In order to define automaton M_1 in JASL language we define a few lists:

```
1   $alphabet = {a, b}
2   $row0 = {>, 0, {0,1}, 0}
3   $row1 = {1, {}, 2}
4   $row2 = {2, 3, {}}
5   $row3 = {<, 3, 3, 3}
6
7   % Now we can define an automaton:
8   $M_1 = Automaton({$alphabet, $row0, $row1, $row2, $row3})
9
10  % We can get, whether automaton accepts word bbbbaab:
11  $accepted = $M_1.accepts(bbbbaab)
12  % Accepted has value: false
13
14  % We can get regular expression describing the language L1:
15  $reg = $M_1.getRegex()
16  % $reg has value: b*aa*b((bb*aa*b)*)a((a+b)*)
17
18  % But does this regex really describe language L1?
19  % This one definitely does:
20  $regex = (a+b)*aba(a+b)*
21  $M_2 = fromRegex($regex)
22  $M_2.equals($M_1)
23  % Outputs: true
```

Note that we use nested lists for definitions of sets of target states. We can use {} to denote $\emptyset$. The output of .getRegex() can be quite complicated. That is because no real regular expression simplifier has been implemented yet.

6.2 Defining an ENFA automaton

Suppose we have a ENFA automaton M_2 that accepts language L such that:

$$\underline{r} = a^* + b^*, \qquad L_{\underline{r}} = L = L(M_2)$$

Such automaton can be described by this transition table:

M_2		ε	a	b
$\rightarrow$	S	A, B		
	A	F	A	
	B	F		B
$\leftarrow$	F			

Table 6.2: Transition table of automaton M_2.

We can define this automaton in JASL as follows:

```
1   $Sigma = {eps, a, b}
2   % We can even shorten the definition by the last empty
        transitions
3   $stateS = {>, S, {A, B}}
4   $stateA = {A, F, A}
5   $stateB = {B, F, {}, B}
6   $stateF = {<, F}
7   $M_2 = Automaton({$Sigma, $stateS, $stateA, $stateB,
        $stateF})
8
9   % Now we can save png image of automaton M_2:
10  $M_2.toPNG(image.png)
```

The resulting image is:

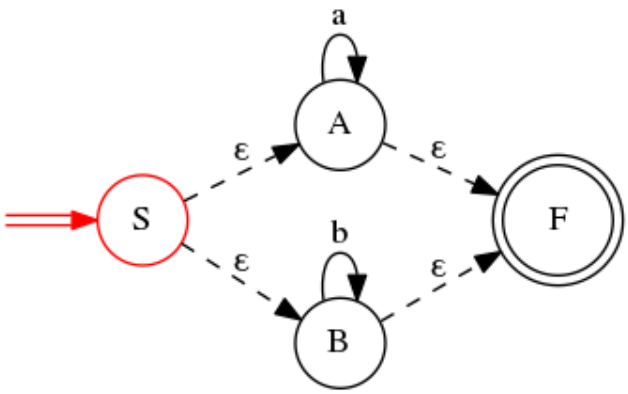


Figure 6.1: Image saved in image.png

6.3 Example of TikZ image

Suppose we have automaton M_3. This automaton accepts language $L = L(M_3)$. This language is also described by regular expression $\underline{r_2}$.

$$\underline{r_2} = (a+b)^*ab^*, \qquad L(M_3) = L_{\underline{r_2}} = L$$

We can use JASL to construct this automaton and create tex file to display it:

```
1   $a = fromRegex((a+b)*ab*)
2   % Tex document parts
3   $class = \documentclass{article}
4   $includes = getTikzIncludes()
5   $beginning = \begin{document}
6   $tikzCode = $a.toTikz()
7   $end = \end{document}
8
9   % Now append these parts in the image.tex file
10  $class.save(image.tex)
11  $includes.save(image.tex)
12  $beginning.save(image.tex)
13  $tikzCode.save(image.tex)
14  $end.save(image.tex)
```

After compiling image.tex file we get this image:

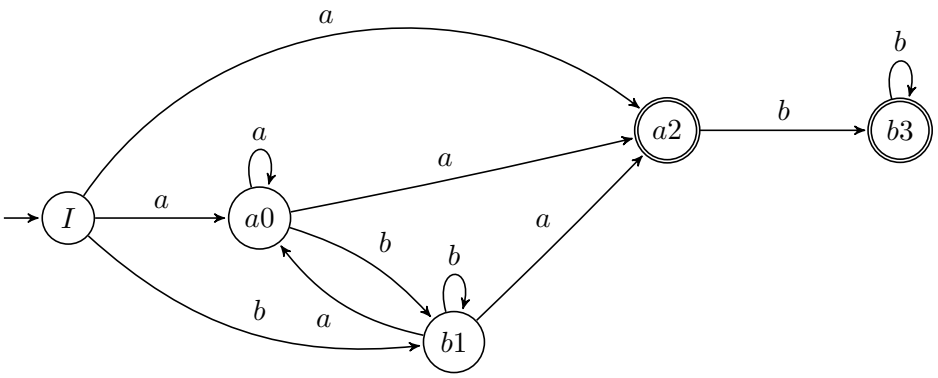


Figure 6.2: Image in compiled image.pdf file.

6.4 Example of JASL script file

In this example we can see the execution of JASL script file and the side effects of using Suppose we have a file append.jasl, that contains the code that will concatenate the language produced by regular expression: ab^*a to language accepted by automaton saved in variable **$i**. It will save the result to variable **$j**. Such file could contain for example this code:

```
1   $append = fromRegex(ab*a)
2   $j = $i.concatenation($append)
```

We can check whether the function worked correctly:

```
1   $i = fromRegex(bba)
2   % We define the append variable
3   $j = hello
4   $shouldBe = fromRegex((bba)(ab*a))
5
6   execute(append.jasl)
7
8   % We can see that the contents of the variable k were
        overwritten:
9   $j.equals($shouldBe)
```

The last command will print true to console. Note that by executing code in append.jasl we have overwritten anything that might be in the variable $j. The user has to be aware of this side effect. Stack frames might be implemented later 7.2.2

Chapter 7

What to do next? Looking to the future

JASL and its interpreter does not yet implement some of the features that I would want it to. The ultimate goal of this application is to make tasks regarding automata simply. There are many quality-of-life improvements that yet wait to be implemented. I will describe some of those features in this chapter.

7.1 JAutomata

The JAutomata library could implement other types of automata (Mealy, Moore, Push-down). The implementation could use the structure of the project and its classes with minor tweaks.

7.1.1 Operations over regular expressions

The regular expression format used in JAutomata library, which is also defined in this document differs from the format that is usually used in literature. It does not implement powers of regular expressions. The syntax for powers is simple:

Powers of regular expressions. Suppose we have regular expression r that describes language L. Then regular expression r^i describes language L^i which can be defined recursively as:

- $L^0 = \{\varepsilon\}$
- $L^{n+1} = LL^n$

Another operator missing from the implementation is the positive iteration of regular expressions. Positive iteration can be defined as:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Both of these operators are only a simplification of regular expressions. For example regular expression r^+ can be written as rr^* without changing the resulting language.

7.1.2 Regular expression simplifier

For the moment, the library is missing any regular expression simplifier. Problem of regular expression simplification is a "hard" problem. This could be implemented using already existing algorithms that yield "good" results [13]. There are some frameworks that can simplify regular expressions, but I do not know of any that works with regular expression format specified in 2.4. This format is commonly used in literature as opposed to other (enhanced) formats commonly used in programming languages.

7.2 JASL

7.2.1 JASL Syntax

JASL syntax could use some more advanced features. Such as element extraction from a list or a string, user-defined functions, other data-types or allowing the user to use some of the more advanced features of JAutomata library. It could also make modification of PNG images possible (colors of edges, size of the image, etc.).

Standardization of the syntax would help users that are used to working with regular programming languages. One example of such standardization would be encapsulation of strings in double quotes. This would allow for better exception handling and syntax error detection.

7.2.2 Interpreter

Examples of possible upgrades to the JASL interpreter are:

- Making error messages more clear

- Implementing command that would print all used variables
- Implementing stack frames for script executions
- Saving current workspace (state of all variables) to a file
- Live preview of generated images

7.2.3 Graph conversion

The JASL language could implement the relative position model, as described in section 5.2. The problem is hard and it would be necessary to implement some tool to change relations in the graph. If this feature is implemented, adding live preview of graphs would be advisable.

The conversion to TikZ code as it is, has two major flaws:

Coordinate conversion. As described in section 5.1, the current implementation of dot to Tikz coordinate conversion stretches small graphs. One way of dealing with this stretching is to calculate the required distance between nodes. The calculation would compare the size of the node from the dot code to the distance between nodes. Based on the result it would decide, how distant the nodes should be. There are lots of constants in such calculation, that would require fine tuning and a lot of experminetation.

Angle of edges. ... Stretched problem, relativistic coordinates ...

Chapter 8

Conclusion

Lorep ipsum [1]

Bibliography

[1] J. Doe. *Book on foobar.* Publisher X, 2300.

[2] Marie Demlová. *Jazyky, automaty a gramatiky.* Materials for CTU course Jazyky, Automaty a Gramatiky, 2017. `http://math.feld.cvut.cz/demlova/teaching/jag/jag7dohromady.pdf`

[3] Graphviz official webiste. `https://www.graphviz.org/`.

[4] Graphviz-java official repository. `https://github.com/nidi3/graphviz-java`

[5] Satyaki Sikdar. *Drawing Finite State Machines in LaTeXusing TikZ A Tutorial.* University of Notre Dame, 2017. `https://www3.nd.edu/~kogge/courses/cse30151-fa17/Public/other/tikz_tutorial.pdf`

[6] Jan Travnicek and col. Private Gitlab repository of Algorithms Library Toolkit developed at FIT CVUT, accessible with CVUT credentials. `https://gitlab.fit.cvut.cz/algorithms-library-toolkit/automata-library`

[7] Bořivoj Melichar. *Jazyky a Překlady.* CVUT, 2003.

[8] Borivoj Melichar, Jan Holub, Tomas Polcar. *Text searching algorithms.* CTU Prague, 2005. Algorithm p.203. `http://www.stringology.org/athens/TextSearchingAlgorithms/tsa-lectures-1.pdf#page=203`

[9] Manual for DOT language `https://www.graphviz.org/doc/info/lang.html`

[10] Manual for TikZ automata library `https://www.bu.edu/math/files/2013/08/tikzpgfmanual.pdf` p.175-180

[11] TeXUsers Group website. `http://tug.org/`

[12] Tomas Horovsky. Notes for the A4B01JAG CVUT course https://github.com/Horovtom/SchoolNotes/blob/master/A4B01JAG/A4B01JAG.pdf

[13] H. Gruber, S. Gulan. *Simplifying Regular Expressions A Quantitative Perspective*. Universitat Gießen, 2009. https://www.researchgate.net/profile/Hermann_Gruber3/publication/228529267_Simplifying_Regular_Expressions_A_Quantitative_Perspective/links/02e7e517274e7a8f6e000000/Simplifying-Regular-Expressions-A-Quantitative-Perspective.pdf

Appendix A

Used code

A.1 Used DOT code

This appendix includes all dot codes used to draw images.

A.1.1 5.1

```
digraph automaton {
   graph [bb="0,0,306,55", rankdir=LR, size="8,3"];
   node [color=black, label="\N", shape=circle];
   qS0   [color="", height=0.5, label="", pos="27,27.5",
      shape=none, width=0.75];
   I   [color=red, height=0.5, pos="109,27.5",
      width=0.5];
   qS0 -> I [color="red:invis:red",
      pos="e,90.826,27.5 54.195,27.5 62.654,27.5 72.051,27.5
          80.595,27.5"];
   a0  [height=0.52778, pos="189,27.5", width=0.52778];
   I -> a0  [label=a,
      lp="148.5,35",
      pos="e,169.92,27.5 127.31,27.5 136.8,27.5 148.81,27.5
          159.63,27.5"];
   b1  [color="", height=0.76389,
      pos="278.5,27.5",
      shape=doublecircle,
      width=0.76389];
   a0 -> b1 [label=b, lp="229.5,35",
      pos="e,250.75,27.5 208.13,27.5 217.56,27.5 229.44,27.5
          240.69,27.5"];
}
```

A.2 Used TikZ code

A.2.1 5.2

```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node
    distance=2.8cm,semithick,initial text=$  $]
  \tikzset{every state/.style={minimum size=0pt}}
  \node[state] (0) at (12.27,1.78) {$a0$};
  \node[state, accepting] (1) at (18.07,1.78) {$b1$};
  \node[state, initial, initial where=left] (2) at (7.07,1.78)
      {$I$};
  \path
    (0)

      edge  node {$b$} (1)
    (2)

      edge  node {$a$} (0);
\end{tikzpicture}
```