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
TODO: FILL

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

1 Introduction and motivation 1

1.0.1 Motivation 1

2 Definitions and terminology 3

2.1 Languages 3

2.2 Operations over languages 3

2.3 Automaton 4

2.3.1 Deterministic Finite Automaton (DFA) 4

2.3.2 Non-deterministic Finite Automaton (NFA) 5

2.3.3 Non-deterministic Finite Automaton with epsilon transitions (ENFA) 5

3 User manual 7

3.1 Installation 7

3.1.1 Compiling JAR yourself 7

3.2 Execution 7

3.3 Syntax of the language 8

3.3.1 Functions 8

3.3.2 Variables 10

3.3.3 Member functions 12

4 Details of Implementation 19

4.1 Used technology 19

4.1.1 JAutomata 19

4.1.2 Graphviz 20

4.1.3 Graphviz-java library 20

4.1.4 Tikz and automata for tikz . . 21

4.2 Details of implementation 21

4.2.1 JASL interpreter 21

4.2.2 Conversion to tikz 23

4.3 Problems with implementation . 23

4.3.1 Problems with syntax errors . 23

4.3.2 Problems with Graphviz-java 24

5 Drawing images - details 25

5.1 Using Graphviz for layout output 25

5.1.1 Graphviz layout engines 26

5.2 TIKZ 27

6 Examples of usage, practice, problems of testing 29

6.1 Defining a NFA automaton 29

6.2 Defining an ENFA automaton . . 30

6.3 Example of Tikz image 31

6.4 Example of executable file 32

7 What to do next? Looking to the future 35

7.1 JASL Syntax 35

7.2 Interpreter 35

7.3 Graph coordinate conversion . . . 35

8 Conclusion 37

Bibliography 39

Figures

3.1 State diagram of the example automaton 9

5.1 Original automaton image 26

5.2 Example of stretched automaton 26

5.3 Image generated using **dot** layout 27

5.4 Image generated using **neato** layout 27

5.5 Image generated using **circo** layout 27

6.1 Image saved in image.png 31

6.2 Image in compiled image.pdf file. 32

Tables

3.1 Transition table of example automaton 9

3.2 Example of conversion of trasition table to list 12

6.1 Transition table of automaton M_1. 29

6.2 Transition table of automaton M_2. 30

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** (Java Automata Syntax Language) and the interpreter for this language. Both jautomata library and JASL interpreter are written in pure object-oriented Java.

1.0.1 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 [5], 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. Sometimes it was not so easy to work with these libraries, because their actual purpose for this project was different from their intended use. More on that in chapter

TODO: CONTINUE

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.

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: $w = a_1a_2 \ldots a_m, a \in \Sigma$.
- **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 $w_1, w_2 \in \Sigma^*, w_1 = a_1a_2 \ldots a_m, w_2 = b_1b_2 \ldots b_n$ the result of the **Concatenation** operation is: $w_1w_2 = a_1a_2 \ldots a_mb_1b_2 \ldots b_n$.
- **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 such:

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\} \subseteq (\Sigma_1 \cup \Sigma_2)^*$$

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

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

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

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

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

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

2.3 Automaton

The term automaton in language theory includes many types of automata, such as Moore automaton, Mealy automaton [6]. 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: $(Q, \Sigma, \delta, q_0, F)$, where:

Q Finite non-empty set of states

Σ Finite non-zero set of inputs

δ Transition function $\delta : Q \times \Sigma \to Q$

q_0 Initial state from Q

F Subset of so-called accepting states from Q

Let $w = a_1a_2 \ldots a_n$ be a word over an alphabet Σ. Automaton M accepts the word w if a sequence of states $r_0, r_1, \ldots, r_n, r \in Q$ exists such that:

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

In other words: We define extended transition function $\delta^* : Q \times \Sigma^* \to Q$ by induction as:

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

Then we can say that automaton M accepts word w if

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

2.3.2 Non-deterministic Finite Automaton (NFA)

Non-deterministic Finite Automaton M is defined the same as DFA automaton, with two differences:

1. It can have more than one initial state: $I \subseteq Q$
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 such:

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

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

1. $X \subseteq \varepsilon\text{-closure}(X)$
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:

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

2.4 Automaton visualisations

. . . TODO . . .

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 3.3.1.

3.3 Syntax of the language

The **JASL** language allows you to define variables and call functions upon those variables. The commands are parsed line by line. On every line there is one assignment or a command.

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

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

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 Functions

In this section I will describe the functions that are implemented to JASL syntax in more detail.

execute

```
execute(file.jasl)
```

This function executes script on specified path. It uses currently defined variables for the execution and update them.

fromCSV

```
$automaton = fromCSV(file.csv)
```

This function returns new Automaton object, loaded from comma-separated csv file specified in the single argument of this function. The CSV output/input format is specified in greater detail in chapter: TODO.

getExample

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

This function 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:

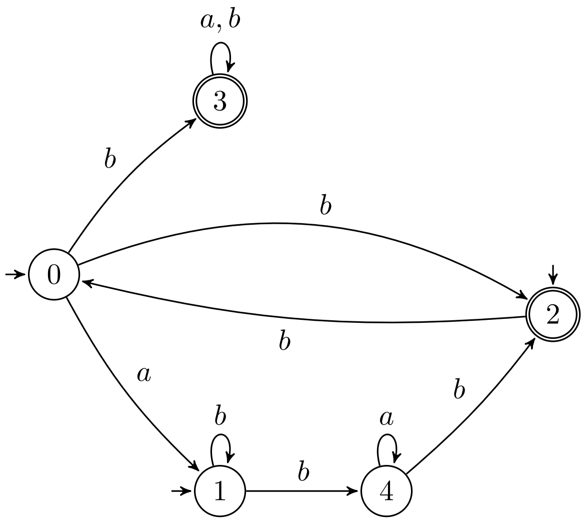

Figure 3.1: State diagram of the example automaton

fromRegex

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

This function returns new Automaton object specified by regular expression passed in as an argument. The regular expression has to be in format specified in chapter: TODO.

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.

getTikzIncludes

```
getTikzIncludes()
```

This 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}
```

3.3.2 Variables

Variables are defined as follows:

```
$variableName = value
```

Variable name can be any string that does not contain '$', ' ' or '.'. Variables can hold objects of these types:

- string - `$thisIsString = hello world`
- list - `$thisIsList = {a, b, c}`
- automaton - `$thisIsAutomaton = Automaton($args)`

Strings

String in JASL can be any sequence of letters, that does not start with open curly bracket or a dollar sign.

Lists

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 or variables. 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}
```

Automata

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

- disjunct 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 $IO = \begin{cases} <>, & \text{if } Q_i \in F, Q_i \in I \\ <, & \text{if } Q_i \in F \\ >, & \text{if } Q_i \in I \\ \text{empty string}, & \text{otherwise} \end{cases}$ and each of $\delta(Q_i, t_j)$ is a list of target states.

In other words this paremeter 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 trasition 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.

```
$alphabet = {a, b}
$row0  = {<>,0,{},2}
$row1  = {>,1,0,{1,2}}
$row2  = {<,2,{1,2,3},1}
$row3  = {3,3,{}}

% Now we can define the nested list:
$nestedList = {$alphabet, $row0, $row1, $row2, $row3}

% And now we can define an automaton:
$automaton = Automaton($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:

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

3.3.3 Member functions

Member function is a function called specifically on automaton object saved in a variable. It can be invoked as such:

```
$result = $automaton.functionName($arg1, $arg2)
```

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

```
$reduced = $automaton.reduced()
$reduced.toPNG(image.png)

% Can be written as:
$automaton.reduced().toPNG(image.png)
```

This is a list of all member functions for automata objects:

accepts

```
$M.accepts(aabbaab)
```

This function returns *true* if automaton 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. Note, that if you have an automaton that has any terminals with more than one character, variant with argument of type string will not work. In that case you need to use list as an argument.

equals

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

This function 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()
```

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

toCSV

```
$M.toCSV(m.csv)
```

This function creates/overwrites csv file on path specified by the argument. The csv will contain description of the automaton in format, that is specified in chapter: TODO

toPNG

```
$M.toPNG(m.png, circo)
```

This function creates/overwrites png file on path specified by the argument. The png will contain 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** will be used as a default. Possible layouts are: **circo**, **neato**, **dot**.

toTexTable

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

This function will output string containing TEXcode to display the transition table of automaton M.

toRegex

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

This function will output 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.

toDot

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

This function will output 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** will be used as a default. Possible layouts are: **circo**, **neato**, **dot**.

toSimpleDot

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

This function will output dot code, that contains description of the automaton state-diagram image. As opposed to toDot function, the dot code will not contain positions of elements, because it has not been run through Graphviz yet.

toTikz

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

This function will output 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** will be used as a default. Possible layouts are: **circo**, **neato**, **dot**. It is recommended not to specify this argument (hence use dot as an engine), because it will generally output the nicest results. Note that you need to add appropriate includes to your TEXcode. You can get these using getTikzIncludes function.

union

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

This member function accepts one other automaton. It will output new automaton M_3 that accepts union of languages accepted by automata M_1, M_2.

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

intersection

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

This member function accepts one other automaton. It will output new automaton M_3 that accepts intersection of languages accepted by automata M_1, M_2.

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

kleene

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

This member function will output new automaton M_2 such that:

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

complement

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

This function will return 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)
```

This function accepts one other automaton as a parameter. It will output new automaton M_3 that accepts the concatenation of languages accepted by automata M_1, M_2.

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

renameState

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

This function accepts two arguments. The old state name as first and the new state name as second argument. It will fail if the original state has not been found in the automaton or if the new name is already taken by some other state of the automaton.

renameTerminal

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

This function accepts two arguments. The old temrinal name as first and the new terminal name as second argument. It will fail if the original terminal has not been 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 mark of epsilon tranisiton. You cannot use this function to add or remove epsilon transitions from the table.

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 invented this algorithm: ... fill ... I implemented this algorithm in AutomatonAcceptor object. Theoretically, it should be much faster, 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 [2] 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 [3] 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 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: [4]

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

Interpreter for JASL language implements three types of statemenets.

- Assignments
- Expressions
- Commands

There is just a handful of commands available in console interpreter. These are: **help**, **helpLong** and **clear**. These commands are described in more detail in sections 3.2 and 3.3

Assignments consist of the variable name on the left side of the equals operator and some expression on the right side. Expressions consist of function calls, variables and their member functions. If an expression is called by itself, the console will display the result of the evaluation. However, if an expression is used as a part of an assignment, it saves the result to the variable and does not output anything to the console.

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.

```
% Code without in-place construction
$automaton1 = fromRegex(a*(b+a))
$exampleAutomaton = getExample()
$automaton1.equals($exampleAutomaton)

% Code with in-place construction
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.

1. Split expression in three parts: variable name **V**, first member function call **F** and the rest of the expression **R**.
2. Evaluate **VF** and save the result in temporary variable **T**

3. Run evaluate on expression **TR**

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 **$TEMP** variable. As a countermeasure I added a little stack-frame just for this variable. I could implement the stack-frame using Java's native stack-frame, because the algorithm uses recursion to evaluate chained function calls. This solution was simple to implement, but it has a problem. If one of the member functions has a syntax error in it, the interpreter throws an InvalidSyntaxException and the stack frame will be lost. It is not recommended using **$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.

```
% Code without chained member function calls
$automaton = getExample()
$reducedAutomaton = $automaton.reduce()
$tikzCode = $reducedAutomaton.getTikz()
$tikzCode.save(test.txt)

% Code with chained member functions
getExample().reduce().toTikz().save(test.txt)
```

4.2.2 Conversion to tikz

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 and then convert them to dot code directly 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 thing if 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 - details

. . . Something here . . .

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 itself does not take care about 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.

Graphviz has many attributes that you can specify in the dot file: styling of the edges, positions of nodes and many more. However, there are some attributes that have no effect on the resulting file because Graphviz often ignores some of the arguments to produce better-looking image.

Size attribute is used to control the size of output images. Graphviz copies size attribute to the result 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 the images.

The output dot file contains coordinates for all elements of the graph. However, these are internal coordinates of Graphviz. I needed to convert these coordinates to tikz internal coordinates. 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 output file that should specify the bounding box of the image. Unfortunately this value does not correspond to the maximum coordinate values of elements. Again, 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. 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. The following two images show png image generated by graphviz and the Tikz code result image in contrast with one another.

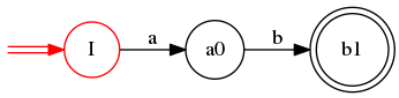

Figure 5.1: Original automaton image

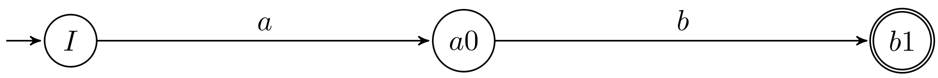

Figure 5.2: Example of stretched automaton

This unwanted effect could be one of the areas to work on in the future. 7.3

5.1.1 Graphviz layout engines

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

- dot
- neato
- circo

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 automaton that accepts only word *ab*. This automaton has four states. These are the images of this automaton generated by different layout engines:

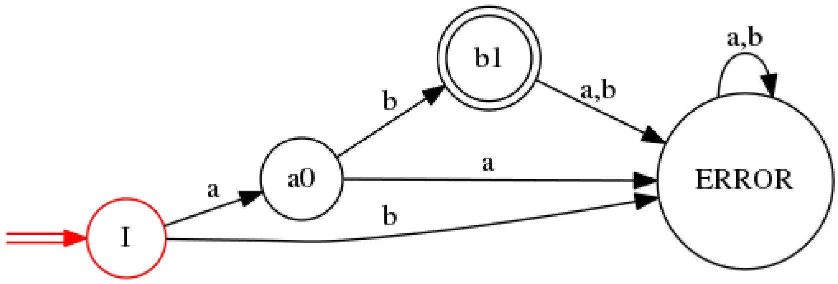

Figure 5.3: Image generated using **dot** layout

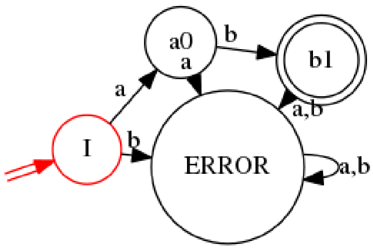

Figure 5.4: Image generated using **neato** layout

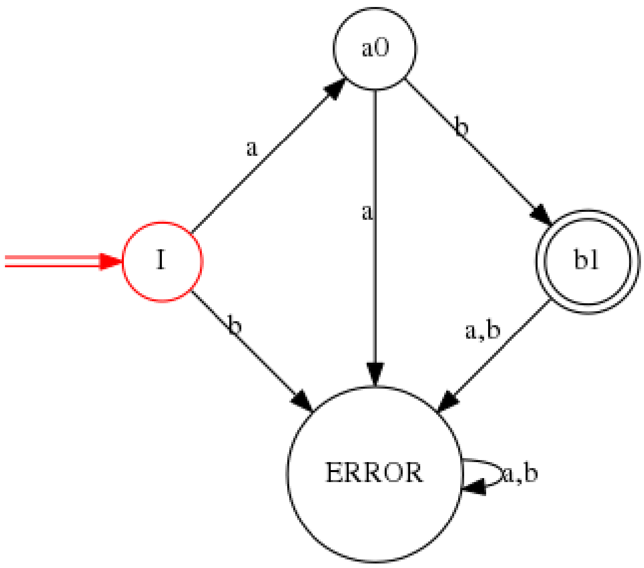

Figure 5.5: Image generated using **circo** layout

5.2 TIKZ

Originally, I wanted JASL to output tikz code that would be the most easy for the user to edit. 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 such:

```
\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. This approach would pose some difficult challenges.

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 have to define a few lists:

```
$alphabet = {a, b}
$row0  = {>, 0, {0,1}, 0}
$row1  = {1, {}, 2}
$row2  = {2, 3, {}}
$row3  = {<, 3, 3, 3}

% Now we can define an automaton:
$M_1  = Automaton({$alphabet, $row0, $row1, $row2, $row3})

% We can get, whether automaton accepts word bbbbaab:
$accepted = $M_1.accepts(bbbbaab)
% Accepted has value: false

% We can get regular expression describing the language L1:
$reg = $M_1.getRegex()
% $reg has value: b*aa*b((bb*aa*b)*)a((a+b)*)

% But does this regex really describe language L1?
% This one definitely does:
$regex = (a+b)*aba(a+b)*
$M_2 = fromRegex($regex)
$M_2.equals($M_1)
% 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 such:

```
$Sigma = {eps, a, b}
% We can even shorten the definition by the last empty transitions
$stateS = {>, S, {A, B}}
$stateA = {A, F, A}
$stateB = {B, F, {}, B}
$stateF = {<, F}
$M_2 = Automaton({$Sigma, $stateS, $stateA, $stateB, $stateF})

% Now we can save png image of automaton M_2:
$M_2.toPNG(image.png)
```

The resulting image is:

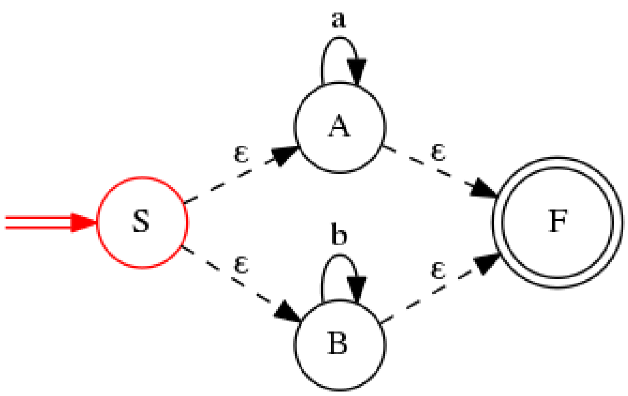

Figure 6.1: Image saved in image.png

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 construct this automaton and create tex file to display it in JASL:

```
$a = fromRegex((a+b)*ab*)
% Tex document parts
$class = \documentclass{article}
$includes = getTikzIncludes()
$beginning = \begin{document}
$tikzCode = $a.toTikz()
$end = \end{document}

% Now save these parts to image.tex file
$class.save(image.tex)
$includes.save(image.tex)
$beginning.save(image.tex)
$tikzCode.save(image.tex)
$end.save(image.tex)
```

After compiling image.tex file we get this image:

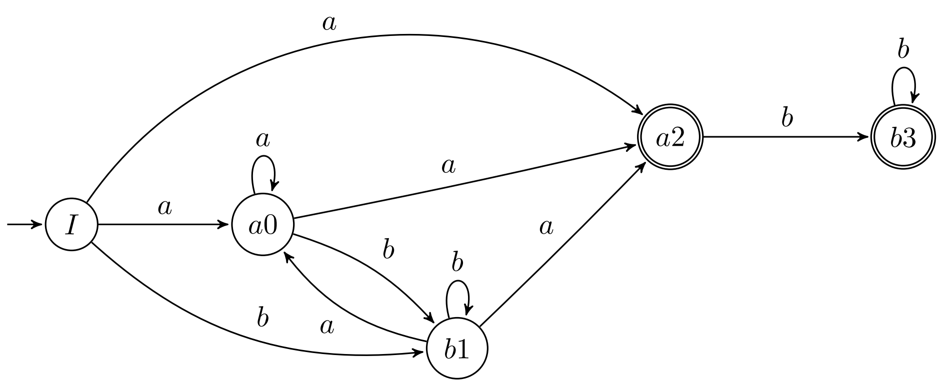

Figure 6.2: Image in compiled image.pdf file.

6.4 Example of executable file

Suppose we have a file append.jasl, that contains the code that will concatenate 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:

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

We can check whether the function worked correctly:

```
$i = fromRegex(bba)
execute(append.jasl)
$shouldBe = fromRegex((bba)(ab*a))
$j.equals($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 $append. The user has to be aware of this side effect. Stack frames might be implemented later 7.2

Chapter 7

What to do next? Looking to the future

...These areas could use some work in the future ...

7.1 JASL Syntax

...Other data types ...

7.2 Interpreter

...Stack frames, exceptions handles, etc... ...

7.3 Graph coordinate conversion

...Stretched problem, relativistic coordinates ...

Chapter 8

Conclusion

Lorep ipsum [1]

Bibliography

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

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

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

[4] 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`

[5] Private Gitlab repository of Algorithms Library Toolkit developed by FIT CVUT, accessible with CVUT credentials. `https://gitlab.fit.cvut.cz/algorithms-library-toolkit/automata-library`

[6] Melichar

[7] Wikipedia page of Deterministic Finite Automata `https://en.wikipedia.org/wiki/Deterministic_finite_automaton`