Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Finite Automata Drawing Platform

Tomáš Hořovský

Supervisor: RNDr. Marko Genyk-Berezovskyj
January 2019

Acknowledgements

I want to thank RNDr. Marko Genyk-Berezovskyj and prof. RNDr. Marie Demlová, CSc. for their help. I am also thankful to my family for their endless support.

Declaration

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.
V Praze dne 7.1.2019

. . . .
Tomáš Hořovský

Abstract

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

Keywords: Java, Automata theory, Regular automaton, Regular expression, Graphviz, JAutomata

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

Abstrakt

Cílem tohoto projektu bylo dokončení knihovny **JAutomata** pro operace s automaty, vývoj nového skriptovacího jazyka **JASL**, který popisuje automaty a operace s těmito automaty a implementace interaktivního konzolového prostředí pro spouštění příkazů tohoto jazyka, které používá knihovnu JAutomata. Skriptovací jazyk JASL měl umožňovat export automatů v různých formátech, včetně kódu pro jazyk TEX, pro kreslení jejich stavových diagramů.

Klíčová slova: Java, Teorie automatů, Regulární automat, Regulární výraz, Graphviz, JAutomata

Překlad názvu: Platforma pro kreslení diagramů konečných automatů

Contents

Figures

Tables

Chapter 1

Introduction and motivation

Java Automata Syntax Language (abbreviated to JASL) is a scripting language that I developed as a part of this project. It allows the user to define and work with acceptor finite state machines. I implemented 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 a format native to TEX.

Both jautomata library and JASL interpreter are written in pure object-oriented Java. The program should be able to run on Linux and Windows operating systems, but the primary support is for Linux.

This project started as a passion of mine for automata. I implemented a library in c++ that could do reduction of automata. I kept adding new functionality until the original automata-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 assignment for my software project and my bachelor's thesis. It was only natural that I would finish and rewrite the whole library properly. **JAutomata** library was the result. I wrote most of the **JAutomata** library in my software project. I finished the library and I started working on **JASL** and the interpreter in my bachelor's thesis .

1.0.1 Motivation

When I wrote my own material for Automata and Grammars in LATEX, I stumbled upon the problem of visualizing automata in the document. I wanted a fast and reliable way to draw automaton diagrams 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 a word is accepted by the automaton, 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 it is complicated to use, it is not public and it cannot output directly to LATEXcode.

The goal of this project is to write a program that would implement an intuitive command line interface for operating my JAutomata library that contains most of the commonly-used algorithms for working with automata. Also, it would 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 thesis to describe the relation of the application and language theory. Some of the definitions in sections 2.1,2.2,2.3 and 2.4 are translations of [1].

2.1 Languages

- **Alphabet** is a finite non-empty set Σ. Elements of Σ are called **terminals**.
- **Word** w **over an alphabet** Σ is a finite sequence of terminals: $w = a_1a_2 \dots a_m, a \in \Sigma, m \geq 0$.
- **Length of word** w, denoted by $|w|$, is the number of terminals in the word w.
- **Empty word** ε is a word that has length $|\varepsilon| = 0$ (i.e. does not contain a terminal).
- **All words over an alphabet** Σ, denoted by Σ^*, is a set of all words that can be created using terminals from Σ (including ε).
- For two words $w_1, w_2 \in \Sigma^*, w_1 = a_1a_2 \dots a_m, w_2 = b_1b_2 \dots b_n$ the result of the **concatenation** operation is: $w_1w_2 = a_1a_2 \dots a_mb_1b_2 \dots b_n$.
- **Language** L over an alphabet Σ is any arbitrary subset of Σ^*.

2.2 Operations over languages

We will use the following operations over languages in this thesis:

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

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

- **Union of languages** $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ is a set that is defined 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 defined 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 $L^0 = \{\varepsilon\}, L^{i+1} = L^iL$ 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 alphabet 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 the following regular 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-empty set of terminals
- δ is a transition function $\delta : Q \times \Sigma \rightarrow Q$
- q_0 is an initial state from the set Q
- F is a subset of so-called accepting states, $F \subseteq Q$.

For the purpose of this thesis, we will interchangegably use the term **state** and **name of state** to call an element of Q and we say that state name can be any string.

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

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

We say that automaton M **accepts** word w iff:

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

For an automaton M over Σ, we define $L(M)$ (language accepted by M) as follows:

$$L(M) = \{w \mid \delta^*(q_0, w) \in F, w \in \Sigma^*\}$$

We say that states $p, q \in Q$ are **equivalent** iff:

$$\forall w \in \Sigma^*, \delta^*(p, w) \in F \iff \delta^*(q, w) \in F$$

We say, that $M = (Q, \Sigma, \delta, q_0, F)$ is **reduced** if it does not have any unreachable states ($\forall q \in Q, \exists w \in \Sigma^*, \delta^*(q_0, w) = q$) and no two states of this automaton are equivalent.

We can informally describe the operation of an automaton as follows: Suppose we have a word w and an automaton M. We start in initial state of the automaton: q_0. We read the first terminal a of the word and move to the state q_1 that we got as a result of $\delta(q_0, a)$. Now we have only $|w| - 1$ terminals left to read. We read the second terminal b. We move to the state q_3 that is the result of $\delta(q_1, b)$. We repeat this process, until we have no more terminals to read. Suppose we finished in state q_7. We say that word w is accepted by automaton M iff $q_7 \in F$.

2.3.2 Non-deterministic Finite Automaton (NFA)

Non-deterministic Finite Automaton is a tuple: $M = (Q, \Sigma, \delta, I, F)$. Sets Q, Σ, F have the same meaning as in DFA definition and:

- I is a set of states that are considered initial: $I \subseteq Q$
- $\delta : Q \times X \to P(Q), P(Q)$ 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:

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

We say that NFA M accepts word w iff:

$$\exists q_0 \in I, \text{ such that } \delta^*(q_0, w) \cap F \neq \emptyset$$

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

ENFA differs from NFA by introducing so-called ε-transitions. These allow the automaton to move 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 ε-closure. We can say that the epsilon closure of a set of states: N, is the set of all states that can be reached from states in N, using only epsilon transitions.

We define ε-closure formally as:

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

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

We say that ENFA M accepts word w iff:

$$\exists q_0 \in I, \text{ such that } \delta^*(q_0, w) \cap F \neq \emptyset$$

2.4 Regular expression

A regular expression (regex) can describe any regular language (language accepted by some regular automaton). In this section, we will use bold font to denote regular expressions. In [1] 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 $\mathbf{r_1} + \mathbf{r_2}, \mathbf{r_1}\mathbf{r_2}$ and $\mathbf{r_1^*}$ 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 $\mathbf{r_1}\mathbf{r_2}$ represents language $L_1 L_2$.
- If regular expression $\mathbf{r}$ represents language L, then regular expression $\mathbf{r}^*$ represents language L^*.

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

2.5.1 Table

In this thesis, automata are often described by their transition tables. Transition table has full information contained in 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 relation of q to 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 $\delta(q, t)$. We will omit curly brackets of the sets to clean up the notation.

The 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:

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

The resulting transition table is on figure 2.1:

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

Figure 2.1: Transition table of automaton M

2.6 State diagram

One of the many use-cases for JASL is to output TEXcode to draw state diagrams of automata. A state diagram is a visual representation of an automaton. It provides a complete description of the automaton it is visualizing. A state diagram is a directed graph. Ullman and Hopcroft define it in [2, Example 3.1, p.27] as follows:

> "The state diagram consists of a node for every state and a directed line from state q to state p with label a (in Σ) if the finite automaton, in state q, scanning the input symbol a, would go to state p. Final states, i.e., states in F, are indicated by a double circle."

Ullman and Hopcroft use an arrow with label *start*, to mark the initial state. We will omit the label, cleaning up the visuals of state diagrams. If we have multiple oriented edges from state q_0 to state q_1, we use only one edge labeled by the comma-delimited labels of all edges from q_0 to q_1. We will use dotted edges to emphasize ε transitions in the diagram.

Example. A state diagram of the automaton M from Example in section 2.5.1, is in figure 2.2

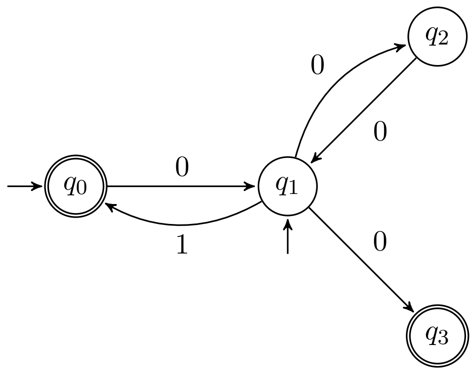


Figure 2.2: State diagram of automaton M

Chapter 3

User manual

3.1 Installation

There are two ways of installing this program. You can either download precompiled .jar file or compile it on your own. If you just want to use the precompiled 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 the source code of the project from the official github repository [12]. After that, you can install it using [13] and [14].

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

```
mvn clean
mvn install
```

After running these commands, you can find the compiled .jar file in the target/ folder. Use the compiled .jar with dependencies (jautomata-1.1-jar-with-dependencies.jar).

3.2 Execution

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

```
java -jar <path-to-jar> [-f <path-to-file>]
```

Square brackets are optional. If the switch -f is not specified, the program will enter interactive shell mode where you can type in your command and get an immediate response. The environment will store your variables in the interpreter memory and you can manipulate them as described in section 3.3. However after terminating the shell environment (by using the **quit** command) all variables in the interpreter memory are lost. The same effect can be achieved even without closing the environment by using the **clear** command.

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 JASL script files from the live interpreter environment, by using the execute function (full description in section 3.3.3).

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.

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. Starting non-terminal is linethe starting non-terminal.

line	→	expression | assignment | comment | command | ε
variable	→	$**variableName**
comment	→	%**any**
expression	→	**string** | list | functionCall | variable.memberFCall | variable
list	→	{listItems} | {}
assignment	→	variable = expression
listItems	→	expression | expression, listItems
functionCall	→	**functionName**([args])[.memberFCall]
args	→	expression[, args]
memberFCall	→	**memberFName**([args])[.memberFCall]
command	→	help | helpLong | clear

where non-terminals:

- **any** can be any sequence of characters that does not contain linebreak
- **variableName** can be any non-empty sequence of characters that does not contain any whitespace characters or dots.
- **functionName** is any function name from chapter 3.3.3
- **memberFName** is any function name from chapter 3.3.5
- **string** is any sequence of characters that does not start with any function name, member function name or any of these symbols: **{ , % $** and does not contain linebreak

3.3.1 Syntax details

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

Object. An object is a piece of data.

Variables. Variables are used to store objects. Variable name can be any string that does not contain whitespace or a dot. To access a variable (create variable or use existing variable), use this syntax:

```
$<variableName>
```

Assignment.

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

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

Functions. Function calls consist of the name of the function followed by 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. This means that only a whole line can be a comment. Commenting at the end of a non-comment line is not possible. Every comment starts with the % sign as the first character. 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 command: **helpLong** prints longer, more detailed version with descriptions of functions.

3.3.2 Data types

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

String. Used mainly to describe terminals, words or regular expressions. A string can be any sequence of characters that does not contain any linebreak and does not start with any function name, member function name or any of these symbols: **{ , % $**.

List. List in JASL is an ordered set of elements. 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. This data type is explained in section 3.3.4

3.3.3 Functions

In this section, I will describe the functions that are implemented in JASL in more detail. The code examples use assignments to indicate return type.

execute

```
execute($path)
```

Executes script on the specified path. The argument is a string (or a variable that contains a string) that contains absolute or relative path to a file in filesystem. Execute uses already defined variables for the execution, and updates/overwrites them. This effect is demonstrated in section 6.4.

A JASL script is a file that contains one or several lines of JASL code. It is recommended to give JASL scripts the .jasl extension.

fromCSV

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

Returns new Automaton object, loaded from a comma-separated csv file specified in the single argument of this function. The argument is a string (or a variable that contains a string) that contains absolute or relative path to a file in filesystem. The CSV file should contain the automaton table as described in section 2.5.1.

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 its state diagram:

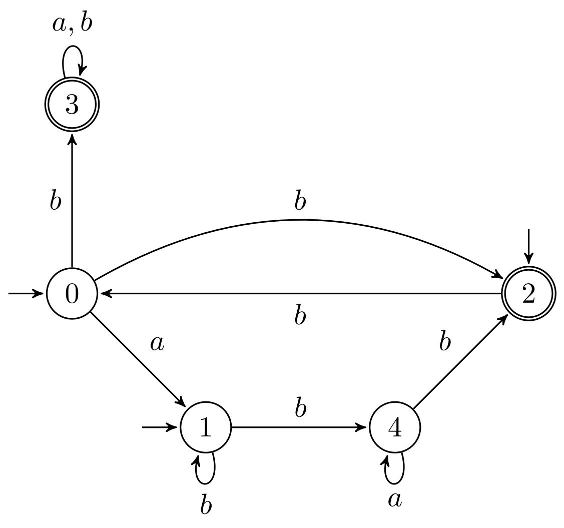


Figure 3.1: State diagram of example automaton

fromRegex

```
$automaton = fromRegex($regex)
```

Returns new Automaton object specified by regular expression passed in as an argument. The argument is a string or a variable that contains a string. The regular expression has to be in the format specified in chapter 2.4 or format described in section 3.3.5.

There are limitations to this function. It works only with terminals that are a single character. Characters cannot be escaped, so symbols '(', ')', {, }, '.' and '*' cannot be used as terminals.

The JAutomata library uses a modified version of the algorithm from [7, p.86 algorithm 2.107], to create automata from regular expressions.

getTikzIncludes

```
getTikzIncludes()
```

Returns the TEXincludes needed, in order to use TikZ and its libraries that are necessary for diagrams of automata to work.

This code is:

```
\usepackage{tikz}
```

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

3.3.4 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 all terminals in Σ.
- n lists where each list $l_i = \{\mathrm{IO}, q_i, \delta(q_i, t_1), \delta(q_i, t_2), \ldots, \delta(q_i, t_k)\}, \forall q_i \in Q, k = |Q|$, where 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. We can also omit empty lists from the end of each row. This is demonstrated on lines 5-8:

```
$alphabet = {a, b}
$row0  = {<>,0,{},2}
$row1  = {>,1,0,{1,2}}
$row2  = {<,2,{1,2,3},1}
% The full definition of row 3:
$row3  = {3,3,{}}
% Shortened definition of row 3:
$row3 = {3,3}

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

% Finally, 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 the terminals. That terminal then signifies an ε transition. The alphabet of some ENFA automaton could be:

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

3.3.5 Member functions

Member function is a function called on an object saved in a variable. There member function for types string and automaton. These can be invoked as follows:

```
$variable.functionName(<list of args>)
```

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)
```

String objects have only the **save** member function:

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. The argument is a string (or a variable that contains a string), that specifies a file in filesystem.

What follows is a list of member functions for automata objects. The notation for examples is the same as in section 3.3.3

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 the argument of type string. In that case you need to use a list as an argument.

Note, that this function works even if $w \notin \Sigma^*$, where Σ is the alphabet of automaton M. In that case it returns *false*.

equals

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

Returns *true* if $L(M_1) = L(M_2)$ (if M_1 is equivalent to M_2). It outputs *false* otherwise. In other words this function checks, whether two automata accept the same language.

reduce

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

Returns new automaton M_2 that is the reduced version of automaton

M. Note that this function creates a new automaton object, so the original automaton remains unchanged.

If the automaton M is an ENFA automaton or an NFA automaton, it is first converted to DFA automaton, before the reduction. All of the described operations are implemented using respective algorithms from [1].

toCSV

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

Exports automaton M to CSV format. It creates/overwrites csv file on the path specified by the argument. The argument is a string that contains absolute or relative path to a file in the filesystem. The created CSV file contains the transition table of the automaton M.

toDot

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

Outputs attributed dot code (described in chapter 5), that contains a 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**.

toPNG

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

Exports automaton M to PNG format. It creates/overwrites png file on the path specified by the argument. The argument is a string that contains absolute or relative path to a file in the filesystem. The png contains an 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**.

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 section 2.4. In the output, concatenation of languages generated by regular expressions $\mathbf{r_1}, \mathbf{r_2}$ is denoted by $\mathbf{r_1} \cdot \mathbf{r_2}$, instead of $\mathbf{r_1 r_2}$. This format is also accepted by the fromRegex function 3.3.3.

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

toSimpleDot

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

Outputs dot code, that contains a description of the automaton state-diagram image. As opposed to toDot function, the dot code is not attributed, so it does not contain positions of elements.

toTexTable

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

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

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

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.

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

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:

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

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.

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

renameState

```
$M.renameState(q, p)
```

Renames state q of automaton M to p. 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 automaton object saved in variable $M.

renameTerminal

```
$M.renameTerminal(a, b)
```

Renames terminal a of automaton M to b. 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 transition. You cannot use this function to add or remove epsilon transitions from the table. This function modifies automaton object saved in variable $M.

3.3.6 Consequences of string as default data type

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

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

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. The programmer documentation is present in the code in the form of a javadoc.

4.1 Used technology

4.1.1 JAutomata

JASL interpreter needed some backend that would implement automata and the operations executed on them. 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, in-depth knowledge about inner workings and I could make changes to the way the library works.

I had to fix some bugs that were in the CSV loading code, finishe the regex conversions 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 ENFA automaton [8]. I implemented modified version of this algorithm in AutomatonAcceptor object. I intended to use the *long* datatype to hold the bitmaps. Java's *long* is 64-bit long, so I could use AutomatonAcceptor object on any automaton with $|Q| \leq 64$.

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 actually got a bit slower (about 10%) when using the newly implemented algorithm. The object-oriented way 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 and great optimizations on object-oriented approach. 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 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 used ENFA object for all automata defined in JASL except for reduced automata. The user of JASL cannot distinguish between inner types of automaton. This could be re-implemented in the future, if users request it as a feature.

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 only a couple attributes (nodes, edges, colors of the edges, . . .) and 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 toTikz 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

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 $\TeX$which give $\TeX$more variability. One of these native packages is the TikZ package.

4.1.5 TikZ and automata for TikZ

TikZ is a native $\TeX$package 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 $\TeX$code to generate images.

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

4.2 Interpreter implementation

One of the main goals of this project was to create live console environment. There are 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 its nature, the live console environment needs to handle many exceptions. Exceptions could occur because of a non-existent file, a corrupted CSV file or a syntax error in a command. I created two custom exceptions: *SyntaxException* (further shortened to *SE*) and *ParsingException* (further shortened to *PE*. These exceptions are used all throughout the interpreter code to distinguish between syntax errors and incorrect parsing methods.

4.2.1 Details of parsing code

Every line of the code is parsed by the *parseLine* function. There are two fundamental types of lines in JASL. A line can be either an expression or an assignment. The *parseLine* function attempts to parse as an assignment first, by calling the *parseAssignment* function. If it was not an assignment, *parseAssignment* function throws *PE*. Then the line is parsed as an expression by the *parseExpression* function.

Parsing an assignment. As mentioned above, assignments in JASL have the form of:

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

If the '=' operator have been parsed correctly, all the following errors will result in throwing *SE* instead of *PE*.

Parsing an expression. Expressions in JASL can be of three types, which are evaluated by respective parsing methods in this order:

1. Member function of an variable call
2. List definition
3. Function call

If a parsing method failed before any determining parameters were found, it throws *PE*. Then the expression is parsed by the next parsing method. If all of the parsing methods fail, then the expression is evaluated as a string.

4.2.2 Enhancing syntax

I wanted to develop a way to make JASL more compact so that the scripts do not have to be unnecessarily long and verbose. 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 often 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. 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 flaw: If one of the member functions has a syntax error in it, the interpreter throws an *SyntaxException* and the original stack frame will be lost (hence the original contents of $TEMP 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)
```

4.3 Problems with Graphviz-java

I thought that using Graphviz-java library would save me a lot of work on parsing of the 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 objects and the meaning of those fields, 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 knot points in the spline, it would linebreak 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.

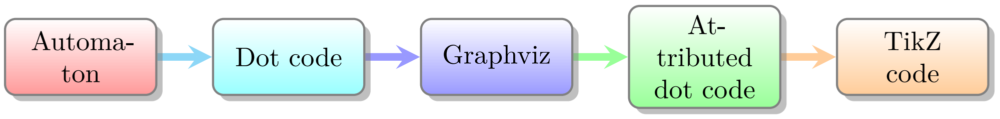
So using graphviz-java for this use-case may not have been the best idea. It might have been easier to get plain text file as an output from Graphviz and write my own parsing code.

Chapter 5

Drawing images

Drawing state diagrams in such a way that they are readable and nice is a very complicated problem. Apart from special cases, removing edge crossings produces a better-looking image. This is a NP-hard problem as is shown in [17].

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. The attributed dot code is then converted to TikZ code.



First, JASL creates the dot code to draw the automaton, then it runs Graphviz on this code. Graphviz outputs attributed dot code. At last, JASL converts the attributed dot to TikZ code. However, many problems arose from this conversion. Some of these problems are described in next section.

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 of page size. If input coordinates exceed page size, it will draw the image cropped. I wanted the output of *toTikz* conversion to fit on a 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 attributes, 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 the size attribute to the attributed dot file, but it does not affect the coordinates of the elements. It is included to the attributed dot only to instruct the rendering engine on how to scale/orient the resulting image. JASL reads positions of elements from the attributed dot, so the size attribute does not have any effect on the resulting TikZ code.

The *rankdir* attribute in the dot code allows the user to instruct Graphviz on the direction he wants the graph to grow in. As most publications use the portrait orientation, I tried to instruct Graphviz to prefer vertical growth. Using this setting, labels would often overlap with edges. Even without the overlaps I did not like these images so I discarded the idea and used horizontal growth instead.

The attributed dot file contains coordinates for all elements of the graph. However, these are in Graphviz coordinate system. 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 specifies the bounding box of the image. Using this bounding box as mapping parameter, resulted in distorted images.

Using minimum/maximum coordinates in each axis yielded visually better results even for larger graphs. As a side-effect, I lost the ratio between sizes of the elements and length of the edges. This means that the output TikZ code draws small graphs unnecessarily stretched. This is caused by the mapping of small image to a larger area. The following figures show png image generated by Graphviz and the TikZ result image in contrast with one another.

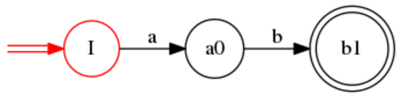


Figure 5.1: Original automaton image. Code A.1.1

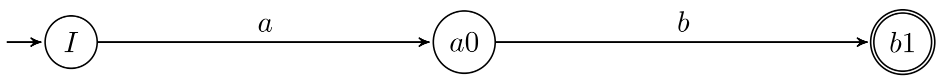


Figure 5.2: Example of a stretched automaton. Code A.2.1

This unwanted effect could be one of the areas to work on in the future (section 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:

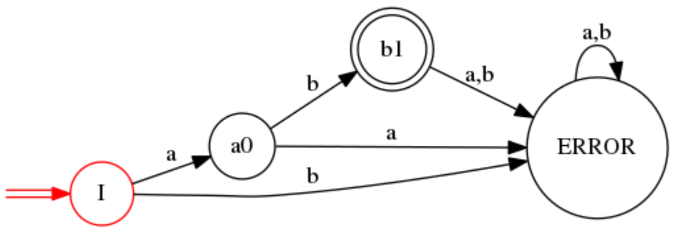


Figure 5.3: Image of automaton M generated using **dot** layout

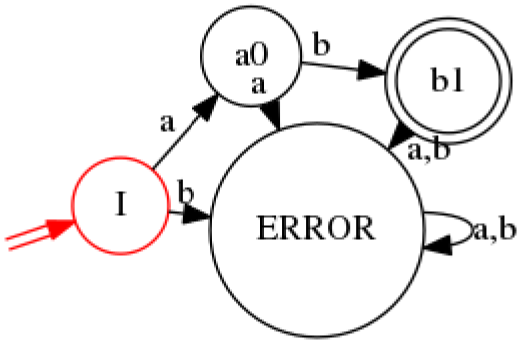


Figure 5.4: Image of automaton M generated using **neato** layout

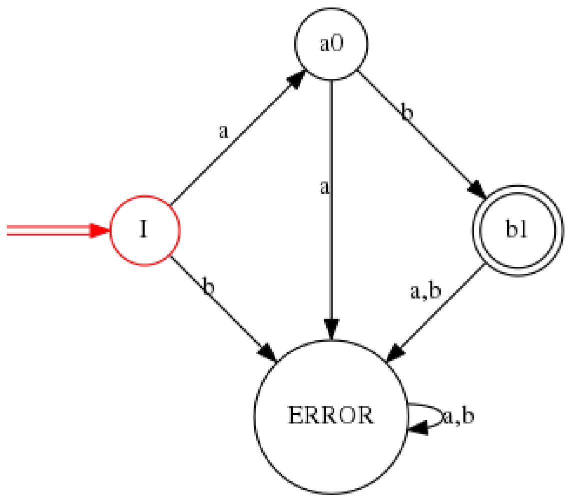


Figure 5.5: Image of automaton M generated using **circo** layout

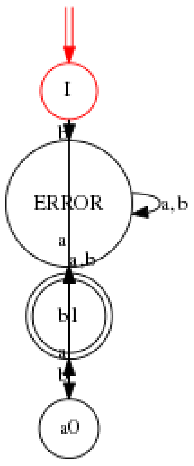


Figure 5.6: Image of automaton M generated using **twopi** layout

Graphviz has an interesting feature. It varies the size of nodes according to the number of incoming/outgoing edges. This effect is apparent on foregoing figures.

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. One of the consequences of this feature is that sequence (below right, above right) does not give the same position as (right, right). 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 destroys the layout completely at times.

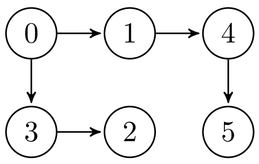
5.2.1 Relation chains

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

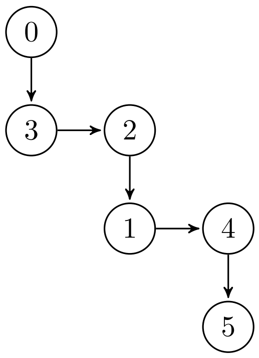
In this simple example, we have some graph, which is constructed using these relations:

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

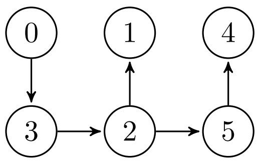
We can visualize these relations like this:



Suppose that 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 more complicated 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.

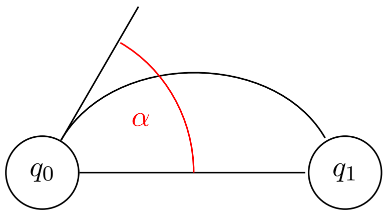
Because of these difficulties, I discarded the idea of relative positions and used absolute positions in the final version of the program.

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 TikZ 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° and 360°, which specifies the angle α depicted in the following picture:



Bend left at 270° produces the same output as bend right at 90°. I decided to keep TikZ code readable and unambiguous by using both directions (left and right), and limiting the angle to between 1° to 179°.

I defined an function f in the code of the application, that outputs this angle as well as the edge orientation. It takes three arguments: $\vec{q_0}, \vec{q_1}, P$, where q_0 is the source node center, q_1 is the destination node center and P is a set of spline knot points. Function f should return an angle. If the angle is negative, the curve is to the right, if it is positive, the curve is to the left, otherwise, it is to the right.

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

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

This function outputs angle γ between q_0, q_1 and point $\vec{p_*} \in P$, which maximizes γ.

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

This function can be improved in the future (see 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:

```
$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.toRegex()
% $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 ∅. 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 an ENFA automaton M_2 that accepts all words generated by regular expression: $a^* + b^*$.

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:

```
$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:

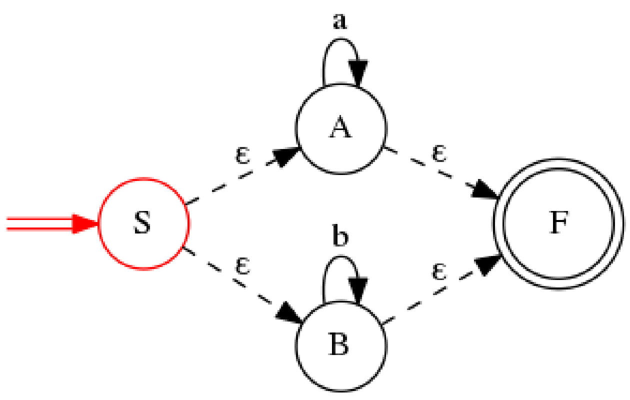


Figure 6.1: Image saved in image.png

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 can use JASL to construct this automaton and create TEXfile to display it:

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

% Now append these parts in the 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:

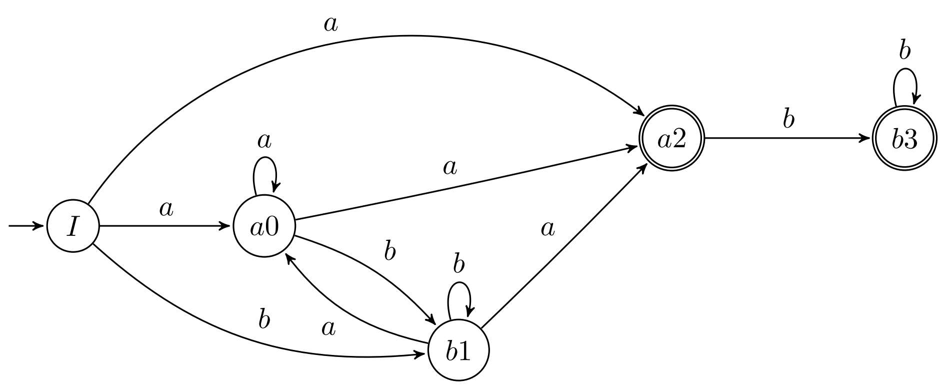


Figure 6.2: Image in compiled image.pdf file.

6.4 Example of a JASL script file

In this example, we can see the execution of JASL script file and the side effects of using execute function.

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:

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

We can check whether the function worked correctly:

```
$i = fromRegex(bba)
% We define the append variable
$j = hello
$shouldBe = fromRegex((bba)(ab*a))

execute(append.jasl)

% We can see that the contents of the variable k were
    overwritten:
$j.equals($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

Looking to the future

JASL and its interpreter do not yet implement some of the features that I would want them to. The ultimate goal of this application is to make tasks regarding automata simple. 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) because these types of automata are often used in practice. 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 thesis differs from the format that is usually used in literature. It does not implement powers of regular expressions and positive iteration operator.

Power of regular expression

Powers of regular expressions are often used to shorten regular expressions with repeating symbols.

Definition of power of regular expression. Suppose we have regular expression $\mathbf{r}$ that describes language L. Then regular expression $\mathbf{r^i}$ describes language L^i which can be defined recursively as:

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

Positive iteration

Another operator missing from the implementation is the positive iteration of regular expressions.

Definition of positive iteration. Suppose we have a language L that is described by regular expression $\mathbf{r}$. Then regular expression $\mathbf{r}^+$ describes language L^+ which is the positive iteration of the language L and can be defined as:

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

Both of these operators are only shortened versions of regular expressions. For example, regular expression $\mathbf{r}^+$ can be denoted by $\mathbf{rr}^*$, 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 minimization is computationally hard (PSPACE-complete as shown in [18]). This could be implemented using already existing algorithms that yield "good enough" results [16].

7.2 JASL

JASL could implement additional features to make the enhance the user experience. In this section I will focus on those improvements.

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 possible the modification of generated PNG images (colors of edges, size of the image, etc.). These changes would make JASL more flexible for the user.

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
- Live console code completion

Apart from stak frames, all of these upgrades are a quality-of-life improvements. For example the live preview of generated images would certainly enhance the user experience. Live code completion for the live console environment would probably require to implement custom console.

7.2.3 Graph conversion

The JASL language could implement the relative position model, as described in section 5.2. The problem is complicated to fine-tune 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.

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 experimnetation.

Angle of edges. Upgrade of the function f, described in section 5.2.2 would improve diagrams of automata with longer edges. This function should be implemented to **cz.cvut.fel.horovtom.jasl.graphviz.DotToTex.getCurveAngle** function. Graphviz does not curve longer edges evenly. Such an edge can be seen on the following picture from state 0 to state 4.

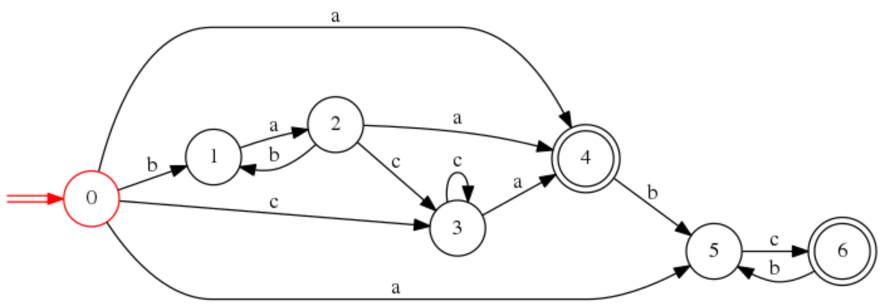
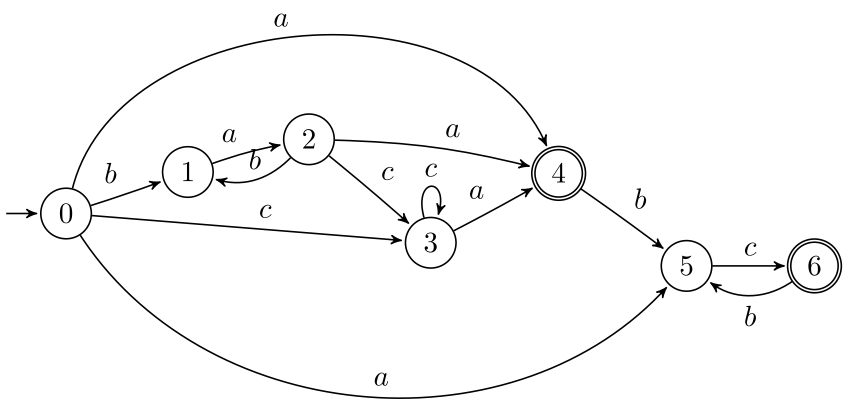
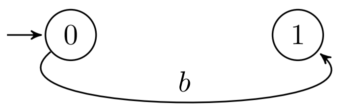


Figure 7.1: Image of longer edge

When converting this automaton to TikZ, we can see, that the current function does a decent job:



However, on some pictures, the curvature is necessary to maintain clarity in the layout. In these cases TikZ can visualize edges of this shape much better with edge angle greater than 90°:



This could one of the possible improvements of the function f.

Chapter 8

Conclusion

The goals of this project were:

- To finish implementation of regular expressions in JAutomata library
- To create a language, that could be used to define/work with regular automata
- To implement a live-console interpreter for this language
- To implement a mechanism for creation of dot code for drawing state diagrams of regular automata
- To implement a mechanism for conversion of attributed dot code to TikZ code.

All of these goals have been reached. Some areas might be improved in the future (see chapter 7), but the application fulfills all set goals at the moment. Newer versions of the application code are accessible in the project github repository [12].

On the appended CD there is the project root directory and the pom.xml file, which allows for easy compilation of the project (as described in section 3.1.1) and a pre-compiled .jar file of the project.

Bibliography

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

[2] John E. Hopcroft, Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley Pub. Co., 1969.

[3] Graphviz official website. `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] The official repository for JAutomata library and JASL projects https://github.com/Horovtom/jAutomata

[13] The official website of Apache Maven https://maven.apache.org/

[14] JDK8 download page https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

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

[16] 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

[17] Michael R. Garey, David Johnson. *Crossing Number is NP-Complete.* SIAM Journal on Algebraic and Discrete Methods. 4. 312-316. 10.1137/0604033, 1983.

[18] A.R. Meyer, L.J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, in: Proc. 13th Ann. IEEE Symp. on Switching and Automata Theory, 1972, pp. 125–129

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}
```