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 User manual **3**

2.1 Installation 3

2.1.1 Compiling JAR yourself 3

2.2 Execution 3

2.3 Syntax of the language 4

2.3.1 Functions 4

2.3.2 Defining a variable 6

2.3.3 Member functions 8

3 Details of Implementation **13**

3.1 Used technology 13

3.1.1 JAutomata 13

3.1.2 Graphviz 14

3.1.3 Graphviz-java library 15

3.1.4 Tikz and automata for tikz . . 15

4 Drawing images - details **17**

4.1 Problems with Graphviz 17

4.2 TIKZ 18

5 Examples of usage, practice, problems of testing **19**

5.1 Defining a NFA automaton 19

5.2 Defining an ENFA automaton . . 20

5.3 Example of Tikz image 21

5.4 Example of executable file 22

6 What to do next? Looking to the future **25**

7 Conclusion **27**

Bibliography **29**

Figures

2.1 State diagram of the example automaton 5

5.1 Image saved in image.png 21

5.2 Image in compiled image.pdf file. 22

Tables

2.1 Transition table of example automaton 5

2.2 Example of conversion of trasition table to list 7

5.1 Transition table of automaton M_1. 19

5.2 Transition table of automaton M_2. 20

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 (TODO: Algorithms Library Toolkit), but they are complicated to use and they can not output directly to LaTeXcode.

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

User manual

2.1 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 2.2

2.1.1 Compiling JAR yourself

If you want to compile it yourself, you have to get source code of the project from this repository. After, 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.

2.2 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 user variables to memory and user can use them freely. However after terminating the shell environment (by using command: **quit**) all saved variables are lost. We can achieve the same effect 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.

User can execute file from shell, where the interpreter will use variables in current session. This can be done by using execute function 2.3.1.

2.3 Syntax of the language

The **JASL** language allows the user 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.

We can comment JASL code with 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.

2.3.1 Functions

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

execute

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

This function will execute script on specified path. It will use currently defined variables for the execution and update them.

fromCSV

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

This function will return 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 will get example automaton. This automaton is described by table:

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

Table 2.1: Transition table of example automaton

And it's state diagram:

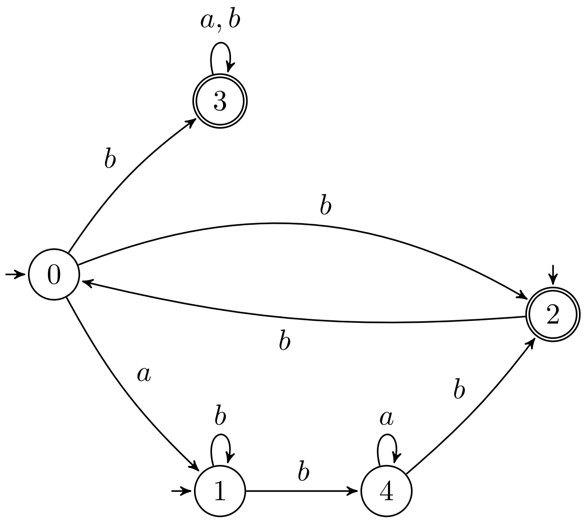

Figure 2.1: State diagram of the example automaton

fromRegex

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

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

There are limitations of this function. It will work only with single characters long letters. Characters can not be escaped, so symbols '(', ')' and '*' can not be used as letters.

getTikzIncludes

```
getTikzIncludes()
```

This will output a couple lines of TEXcode includes, needed for the Tikz diagrams to work.

2.3.2 Defining a variable

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)`

Now we will look at the details of defining lists and automata:

Defining lists

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:

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

TODO: Check and complete

Defining automata

To define an automaton we need to use the constructor function. This function accepts one parameter. This parameter is the transition table of the automaton, enclosed in nested list. Elements of this list are:

1. The alphabeth Σ as an ordered list of letters.
2. $|Q|$ lists. For every state $q \in Q$ we define a list as such:
   a. Whether q is an initial state $q \in I$ (denoted by '$<$') or whether q is a final state $q \in F$ (denoted by '$>$') or both (denoted by '$<>$'). If $q \notin (I \cup F)$, then we can skip this field and not append it to the list whatsoever.
   b. The name of the state q.
   c. For every letter $l \in \Sigma$ we append a list of target states $q \in Q$.

Basically lists in the definition are the rows of transition table read from left to right, separated by commas.

Example 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 2.2: 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. We can create this automaton object using the Automaton constructor. For clarity we can split the definition of the nested list into multiple list variables.

```
$alphabeth = {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:
$nestedList = {$alphabeth, $row0, $row1, $row2, $row3}

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

Note about ENFA automata. ENFA automata can have ε-transitions. We mark these as another letter of the alphabeth. The letter **eps**. So the alphabeth of ENFA automaton could be:

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

2.3.3 Member functions

We can call member functions of objects saved in variables. Member functions are defined for automata objects. We call these member functions like this:

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

Note that we can chain member function calls on one line:

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

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

Now we will list 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 letters. Note, that if you have an automaton that has letters 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 to 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**.

TODO: THIS IS JUST A COPY OF THE ABOVE TEXT! DEAL WITH IT?

toSimpleDot

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

This function will output dot code, that contains description of the automaton state-diagram image. As opposed to toDot 2.3.3, 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 to not 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 2.3.1.

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 one 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.

renameLetter

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

This function accepts two arguments. The old letter name as first and the new one as second argument. It will fail if the original letter has not been found in the automaton or if the new name is already taken by some other letter of the automaton. Also you cannot use 'eps' or ε as a letter because that is a mark of epsilon tranisiton. You cannot use this function to add or remove epsilon transitions from the table.

Chapter 3

Details of Implementation

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

3.1 Used technology

3.1.1 JAutomata

JASL interpreter needed some backend that would execute algorithms on automata and which would house the automata themself. 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 or I had to implement new constructor functions for the Automaton object. I For this reason I decided to work with the code directly and not 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.

In textbook I found this algorithm: 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 JASL language, but soon I realized that functionally they were indistinguishible 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.

3.1.2 Graphviz

Graphviz is a tool for graph layout . . . INFO . . . 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. User can pass in dot code where only a couple attributes are specified (nodes, edges, colors of 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. Graphviz takes many arguments as to how it should organize the layout. Unfortunately some of those arguments have no effect on the resulting image. Graphviz often ignores some of the arguments to produce better image. That ended being quite problematic for my particular use-case. More about that in section 4.1.

3.1.3 Graphviz-java library

Graphviz-java is a library that parses dot code into objects in 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 edge anchor poits extraction. If dot code from Graphviz contained any edge, that was long enough to have more than 9 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.

3.1.4 Tikz and automata for tikz

Tikz is a library . . . build on PGF . . . and it 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.

Chapter 4

Drawing images - details

4.1 Problems with Graphviz

Graphviz has many attributes that one can specify in the dot file. User can specify styling of the edges or even positions of nodes. However there are some attributes that Graphviz ignores. Unfortunately these happened to be the most important for JASL. I needed a way to tell graphviz maximum dimensions of the output image. These attributes were in the result dot, but they did not alter the coordinates of the elements. They were in the final dot only as an instruction for the dot-drawing platform on how to scale the resulting image. Graphviz does not have support for aspect ratio of the output layout. As one of the main goals of JASL is to convert automata to Tikz code, I needed to instruct Graphviz to fit graphs to page size.

Graphviz uses internal coordinates for layout in the output dot file, so I needed to convert them 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 had to know the bounding box of the dot coordinates. Graphviz has attribute in the output file under **bb** keyword that should specify the bounding box of the image. Unfortunately this value did not correspond to the maximum coordinate values of elements. The **bb** argument specified (bounding box after being scaled by scale attribute, but the values did not match even after scaling the attributes).

My only choice was to calculate the bounding box myself. By doing that I sacrificed the size constraints of the original image. This means that tikz ...

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

- dot
- neato

- circo

These engines produce vastly different images. I got the best results using dot engine, but for some particular examples I got nicer images with **circo**.

4.2 TIKZ

Originally I wanted to output tikz code that would be the most easy for the user to edit. Automata library was built to use the **relation** model. In this model every element of the graph is placed in relation to some other element. . . .

Chapter 5

Examples of usage, practice, problems of testing

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

5.1 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 5.1: Transition table of automaton M_1.

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

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

% Now we can define an automaton:
$M_1  = Automaton({$alphabeth, $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.

5.2 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 5.2: 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:

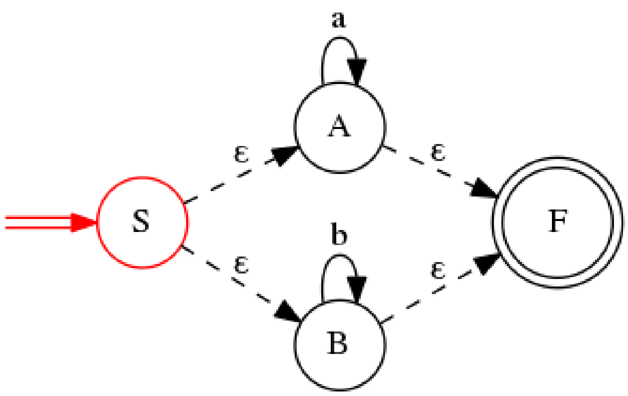

Figure 5.1: Image saved in image.png

5.3 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:

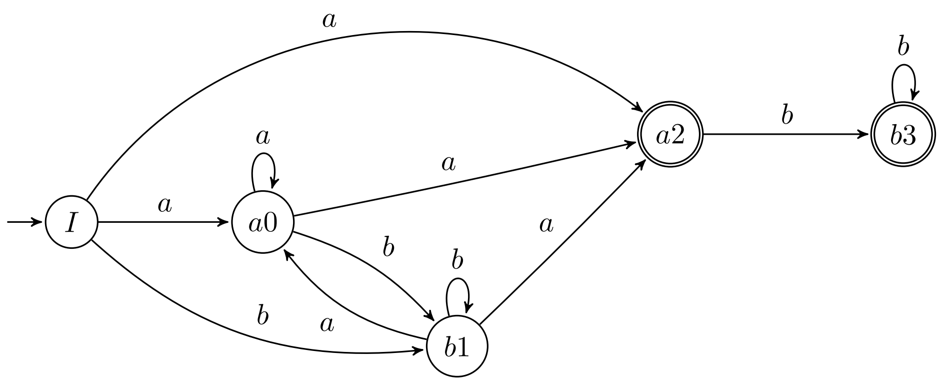

Figure 5.2: Image in compiled image.pdf file.

5.4 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 **??**

Chapter 6

What to do next? Looking to the future

TODO: FILL

Chapter 7

Conclusion

Lorep ipsum [1]

Bibliography

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