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. . . . TODO: FILL

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.
TODO: EDIT/FILL

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. 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 1

2 Motivation and the rest of the world 3

3 User manual 5

3.1 Installation 5

3.2 Syntax of the language 5

3.2.1 Functions 5

3.2.2 Defining a variable 7

3.2.3 Member functions 9

4 Details of Implementation 13

5 Drawing images - details 15

6 Examples of usage, practice, problems of testing 17

6.1 Defining a NFA automaton 17

7 What to do next? Looking to the future 19

8 Conclusion 21

Bibliography 23

Figures

3.1 State diagram of the example automaton 6

Tables

3.1 Transition table of example automaton 6

3.2 Example of conversion of trasition table to list 8

6.1 Transition table of automaton M_1. 17

Chapter 1

Introduction

TODO: FILL

It uses tools such as **Graphviz** (TODO: LINK) or **graphviz-java** (TODO: LINK) library.

Chapter 2

Motivation and the rest of the world

When I wrote my own material for Automata, Grammars and Language theory, 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.

TODO: CONTINUE

Chapter 3

User manual

3.1 Installation

TODO: FILL

3.2 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. TODO: CHECK AND FILL MBY

3.2.1 Functions

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

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 3.1: Transition table of example automaton

And it's state diagram:

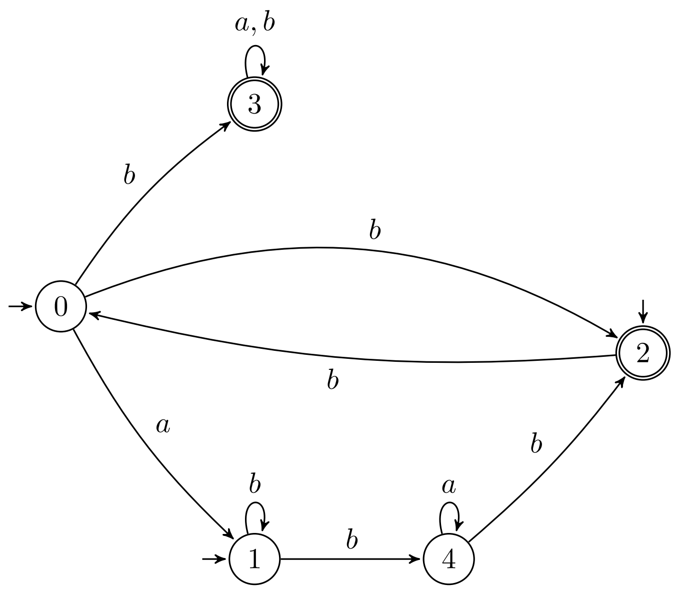

Figure 3.1: State diagram of the example automaton

fromRegex

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

This function 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.

getTikzIncludes

```
getTikzIncludes()
```

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

3.2.2 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 - `$thisIsAutomaton = ENFA($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

JASL implements these types of automata:

- DFA (deterministic finite automaton)
- NFA (non-deterministic finite automaton)
- ENFA (epsilon non-deterministic finite automaton)

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 3.2: Example of conversion of trasition table to list

So the argument to construct this automaton is:

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

The automaton specified by the transition table is a NFA automaton. So we create this automaton with respective constructor function. 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:
$automaton = NFA($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}
```

3.2.3 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 3.2.3, 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 3.2.1.

Chapter 4

Details of Implementation

TODO: FILL

Chapter 5

Drawing images - details

TODO: FILL

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	$\emptyset$	2
	2	3	$\emptyset$
$\leftarrow$	3	3	3

Table 6.1: Transition table of automaton M_1.

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

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

% Now we can define an automaton:
$M_1  = NFA({$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. TODO: FILL

Chapter 7

What to do next? Looking to the future

TODO: FILL

Chapter 8

Conclusion

Lorep ipsum [1]

Bibliography

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