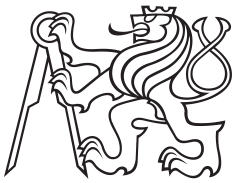
Bakalářská práce



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

Figures

Tables

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

TODO: FILL

3.2.1 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:

```
1  {a, b, c}
2  {}
3  {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 3.1: 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 construct this automaton with constructor function of the same name. For clarity we can split the definition of the nested list into multiple list variables.

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

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

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:

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

Note that we use nested lists for definitions of sets of target states. We can use {} to denote ∅. 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.