# Practical Automata and Process theory

## Table of Contents



## Introduction

This document describes the assignments for the course ALE2 (3 ECTS) and the innovation route Academic Preparations as part of the pre-master program for the Tue (5 ECTS).

Make sure that your lecturer is properly informed about all smart activities which you have done during this course. This could be done for example by writing an accompanying document. Please note: when your lecturer is only looking at your code, he might quite easily overlook your intelligent solutions, which would be a pity.

The assignments can be done in any modern object oriented language (we advise C# and Java). The assignments differ in difficulty. The next table gives a global indication (1 = relatively easy; 4 = relatively difficult), together with an advised week planning. (YMMV)

| assignment | difficulty | week (for 5 ECTS) | week (for 3 ECTS) |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 |
| 3 | 2 | 2 | 3 |
| 4 | 3 | 3 | 4 |
| 5 | 3 | 4 | 6 |
| 6 | 2 | 5 | 7 |
| 7 | 2 | 6 | - |
| 8 | 4 | 7 + 8 | - |

## Grading

When all assignments are working: grade = 6

Additions on the grade (each makes a grade++):

- thorough test vectors in applicable automata
- good software design (classes, interface, SLID principles, Design Patterns)
- clear documentation
- spectacular new features (flex, bison, …)

## References


- https://portal.fhict.nl/es/MAT2/Forms/AllItems.aspx (languages part only, not the linear algebra part)
- Seymout Lipschutz and Marc Lipson
  Discrete Mathematics
  (Schaum's Outline Series)
- http://www.tutorialspoint.com/automata_theory/


## Abbreviations

| | |
|---|---|
| DFA | Deterministic Finite Automaton |
| NDFA | Non Deterministic Finite Automaton |
| PDA | Push Down Automaton |

RE Regular Expression

## Finite state automaton

### File format for finite automata

A line starting with '`#`' contains comments that may be ignored. Use this comment to give a short description of your constructed automaton.
The following lines appear in the input: '`alphabet:`', '`states:`', '`final:`' and '`transitions:`'. Empty lines are allowed as well, for better readability.

'`alphabet:`' is followed by a row of lowercase characters, the alphabet. The symbols are concatenated without spaces or other separators. Note: the epsilon symbol is not part of the alphabet, so it should not be listed here.

'`states:`' is followed by a list of strings, separated by commas. The first listed state is the initial state of the automaton.

'`final:`' is followed by a list of states (which must appear in the earlier mentioned state list), separated by commas. Those are the possible end states. Note: maybe there are no end states!

'`transitions:`' is at a separate line. The next lines contain each one name of an state, a comma, an alphabet symbol (or symbol '`_`' for an epsilon transition), the symbols '`->`' and then a name of state The meaning of such a line should be evident. The alphabet symbol and the states must have been listed in the corresponding lines of the file. After the last transition, a separate line contains '`end.`'.

As a summary, the input file looks like:

```
# comments

alphabet: abcdefghijk…
states: A1,A2,A3,A4,A5,…
final: A2,A3,A4

transitions:
A1,a --> A2
end.
```

The input file contains test vectors for this automaton as well. See the paragraph File format for test vectors for the syntax.

An example of a non-deterministic automaton corresponding to regular expression `(a|b*)` is for example:

```
# automaton for regular expression (a|b)*

states: Z,A,B
final: A,B

transitions:
Z,a --> A
Z,b --> B
Z,_ --> B
B,b --> B
end.
```

Note: symbol '_' indicates an epsilon transition.

It is not obliged, but it is perhaps convenient to use lower case letters for the alphabet and capitals for the states (and avoid '`R`' and '`T`' for your states because they appear for dedicated functionality as described in Assignment 7.

## File format for test vectors

It is extremely convenient to add test cases to your automata. In the same input file, test vectors can be added like:

```
dfa:n
finite:n

words:
bbbbbbbbbb,y
a,y
ab,n
end.
```

'`dfa:`' indicates if the described automaton is a DFA ('`y`') or not ('`n`'). Please note that the example automaton of the previous paragraph is indeed *not* a DFA. See Assignment 1.

'`finite:`' indicates if the automaton generates a finite number of words ('`y`') or not ('`n`'). Please note that the language of the example automaton is indeed *not* finite. See Assignment 4.

Each line after '`words:`' contains a word with an indication if it is accepted by the automaton ('`y`') or not ('`n`'). Please note that words '`bbbbbbbbbb`' and '`a`' indeed belong to the language of the example automaton and that word '`ab`' does *not* belong to that language. The last word is followed by '`end.`', on a separate line. See Assignment 2.

For all mentioned assignments of this course: make at least 2 input files with an automaton together with suitable test vectors, and share them with your fellow students at a convenient location.

### Assignment 1

Write a program that reads a finite automaton (according to the input format as described above) and which indicates if it is a DFA or not.

Show a picture of the automaton (see Appendix Graphical representation of an automaton).

### Assignment 2

Extend the program such that it indicates if a given string is accepted by the automaton or not.

## Regular expressions

### Input format for Regular Expressions

For REs we describe the following prefix notation:

| regular expression | ASCII |
|---|---|
| ε | `_` |
| a | `a` |
| a.b | `.(a,b)` |
| a\|b | `\|(a,b)` |
| a* | `*(a)` |

So RE `a.b*|c*` is written as

```
|(.(a,*(b)),*(c))
```

and `((a.b)*|c)*` is written as

```
*(|(*(.(a,b)),c))
```

### Assignment 3

Extend your program such that a NDFA is constructed from a given RE with the Thompson's Construction.

This NDFA must be written to a file. Format: see File format for finite automata.

### Assignment 4

Extend your program such that it indicates if the number of generated words is finite or not. If it is finite, list the words as well.

### Assignment 5

Extend your program such that a NDFA is converted into a DFA with the Powerset Construction. This DFA must be written to a file; format: see File format for finite

automata. The test vectors of the original automaton need to be written to the new file as well.

## Push down automaton

### File format for push down automata

The input format for PDA differs in two aspects:

1) After the '`alphabet:`' line there is a line with '`stack:`'. This line contains the symbols that can by pushed onto the stack.
2) A transition looks like: `S1,a [x,y] --> S2`, where `x` is a stack symbol that is removed from the stack, and `y` is a stack symbol that is pushed on the stack.
   Instead of a stack symbol, symbol '_' can be used that no symbol is removed from or put on the stack. If the stack is unaffected by a transition (i.e. `[_,_]`), the stack notation may be omitted.

A full description of a PDA to accept strings like $a^n b^m c^{n+m}$ is for example:

```
alphabet: abc
stack: x
states: S,B,C
final: C
transitions:
S,a [_,x] --> S
S,_       --> B
B,b [_,x] --> B
B,_       --> C
C,c [x,_] --> C
end.

dfa: n
finite: n

words:
,y
abcc,y
aacc,y
bbbccc,y
aaabbcccc,n
aabbccccc,n
bbaccc,n
aaaabbbacccccccc,n
end.
```

## Transition priorities

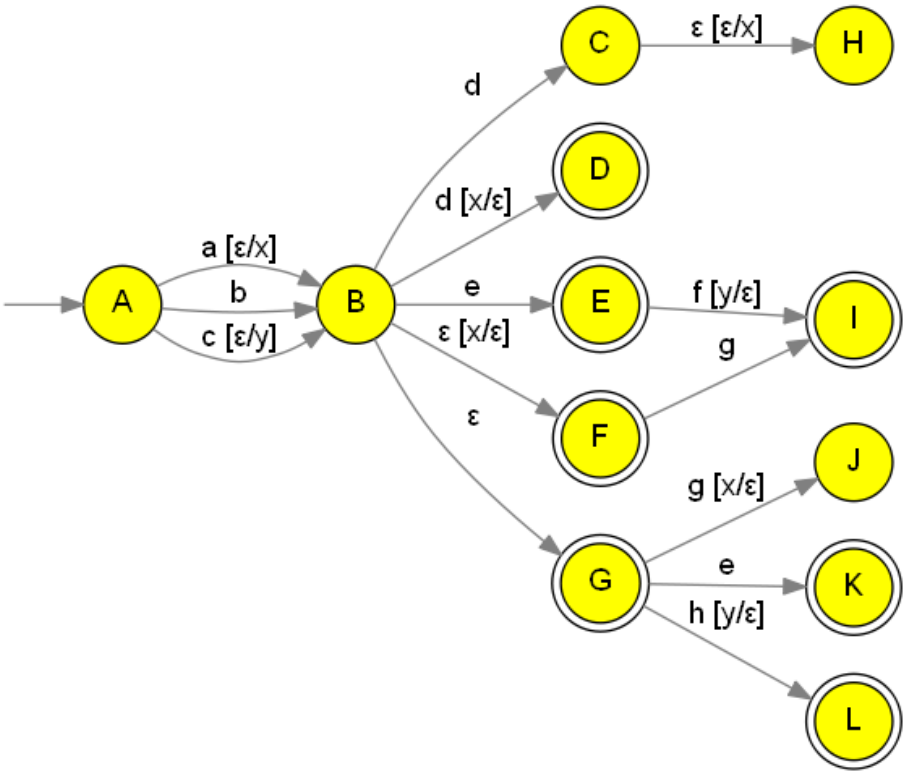
It might occur that at a given moment several transitions are possible, but the following precedence of rules enforces a strict deterministic behavior:

1. transition whose [symbol + pop stack] matches the [current input symbol + current top stack] (if the stack is not empty)
2. transition with epsilon pop stack whose symbol matches the current input symbol
3. transition with epsilon symbol whose pop stack matches the current top stack (if the stack is not empty)
4. transition with epsilon symbol and epsilon pop stack

(you can read this as: "look for the longest fit").

Having multiple transitions from the same state with the same priority is not allowed, since this would interfere with the determinism.

Examples of those rules, given the following automaton:




- for input word `ad`, rule 1 is applied when handling input character `d` to reach final state D (so: not state H by following rule 2)
- for input word `be`, rule 2 is applied when handling input character `e` (note that the stack is empty) to reach state E (so: not state K by following rule 4)
- for input word `cef`, rule 2 is applied when handling input character `e` (note that the stack is *not* empty)

- for input word `ag`, rule 3 is applied when handling input character `g` to reach final state I (so: not state J by following rule 4)
- for input word `b`, rule 4 is applied to reach final state G
- for input word `ch`, rule 4 is applied when handling input character `h` (note that the stack is *not* empty) to reach final state L

(please check that all rules are applied properly)

### Assignment 6

Extend your program such that a PDA specification can be read, and that it can test if the PDA accepts a given string.

Note that a word only belongs to the language when the PDA has reached a final state ***and*** when the stack is empty.

### Assignment 6a (*)

Define a PDA to accept:

- regular expressions as described in Input format for Regular Expressions
- algebraic expressions, like `(a+b)*a` or `a` or `(a*b)`, etc.

## Parallel automaton

### File format for parallel automata

Now we want to run a simulation of two communicating automata. To make this happen, we extend the input format of the two parallel NDFA (so not the PDA) in the following way: for transitions we accept (besides the regular alphabet symbols) also synchronization symbols '`R`' and '`T`'. Those symbols do not belong to the respective alphabets.

Such a synchronization transition can only be executed by both automata together (so either both make an '`R`' step, or both make a '`T`' step).

Two parallel automata (called Automaton A and B) are combined in the Communicating Automaton (called Automaton C), which has a slightly modified format, see the example below.

For ease of implementation, we allow the following simplifications:

- no epsilon transitions
- a state in automaton A or B doesn't have multiple transitions for an input symbol
- the states of automata A and B are disjoint
- the alphabets of automata A and B are disjoint

Automaton C contains a line with ‘`deadlock:`’. This indicates if there is a danger of deadlock ('`y`') or not ('`n`'). See Assignment 8.

Automaton C lists the words to be checked. A line contains the two words separated by a '`|`', and followed by a '`y`' or '`n`' indication. Automata A and B do not contain words to be checked.

File for Communicating Automaton C:

```
automatonA: autA.txt
automatonB: autB.txt

words:
pR|Rcc,y
pRp|Rcc,n
pRppRppR|RccRccRcc,y
end.

deadlock: n
finite: n
dfa: n
```

File `autA.txt` for automaton A:

```
alphabet: p
states: A,B,C
final: C
transitions:
A,p --> B
B,R --> C
C,p --> A
```

File `autB.txt` for automaton B:

```
alphabet: c
states: X,Y,Z
final: X
transitions:
X,R --> Y
Y,c --> Z
Z,c --> X
```

## Assignment 7

Extend your program such that a Communication Automaton specification can be read, and that it constructs one combined automaton.

In GraphViz, show the combined automaton, but also both original automata A and B.

Indicate if this combined automaton accepts a given input. For ease of implementation, this input might be split into two GUI textboxes for strings: one for the (original) automaton A and one for the (original) automaton B.

### Assignment 8

Determine if there is a danger of deadlock in the combined automaton. If so: indicate the path to the deadlock situation.

## Graphical representation of an automaton

To add a graphical picture of your automaton, please follow the following steps:

1. install GraphViz
2. adapt your `$PATH` (Linux) or `%Path%` (Windows) environment variable
3. in your application:
   3.1. generate a text file (e.g. `abc.dot`), similar to this:

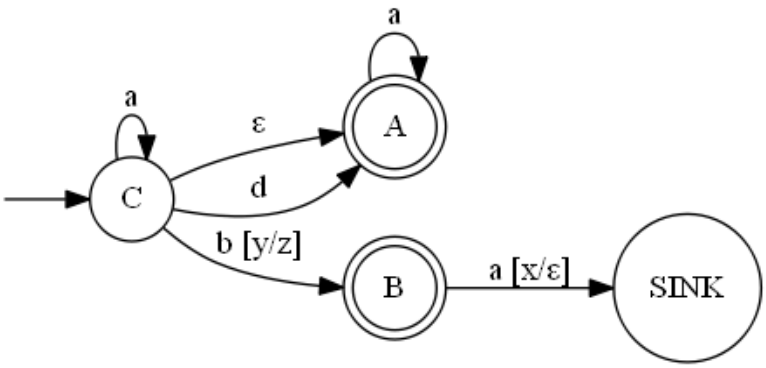
```
digraph myAutomaton {
  rankdir=LR;
  "" [shape=none]
  "A" [shape=doublecircle]
  "B" [shape=doublecircle]
  "C" [shape=circle]
  "SINK" [shape=circle]

  "" -> "C"
  "A" -> "A" [label="a"]
  "B" -> "SINK" [label="a [x/ε]"]
  "C" -> "B" [label="b [y/z]"]
  "C" -> "A" [label="ε"]
  "C" -> "A" [label="d"]
  "C" -> "C" [label="a"]
}
```

   3.2. start the GraphViz executable

```
dot -Tpng -oabc.png abc.dot
```

   3.3. (wait until this executable is finished)
   3.4. show a picture (e.g. `abc.png`), for example in a `PictureBox`




In C#, the steps 3.2 - 3.4 *could* look like:

```
WriteDot("abc.dot");  // your method to write an automaton
                      // into a dot-format file
Process dot = new Process();
dot.StartInfo.FileName = "dot.exe";
dot.StartInfo.Arguments = "-Tpng -oabc.png abc.dot";
dot.Start();
dot.WaitForExit();
myPictureBox.ImageLocation = "abc.png";
```

In Java, the steps *could* look like:

```
String[] cmd = { "dot.exe", "-Tpng", "-oabc.png", "abc.dot" };
Process p = Runtime.getRuntime().exec(cmd);
p.waitFor();
File file = new File("abc.png");
Image image = new Image(file.toURL().toString());
myWidget.setImage(image);
```

(you can do some experiments for steps 3.1 - 3.4 in a text editor and on the command line, or you can check it on http://www.webgraphviz.com/)

Ideas for other nice features of GraphViz are welcome.