Automata and Logic Engineering 1

Contents

Introduction

This document describes the assignments for the course ALE1 (3 ECTS).
They can be done in any object oriented language of your preference.
The assignments differ in difficulty; the next table gives a global indication (1 = relatively easy; 3 = relatively difficult), together with the week planning.

assignment	difficulty	week
1	3	1+2
2	1	3
3	2	4+5
4	2	6
5	1	7

Prerequisites

While working on the assignments, the following general aspects of software engineering play a role:

- UML modelling
- refactoring (in particular when your initial UML modelling was not that optimal)
- testing (module tests and system tests)
- code analysis (coverage, complexity)
- user interface design

Working methods

In this module, there will be no lectures or extra learning materials, except for this Lab Manual. The students are supposed to solve the given tasks by doing research, and experiment on their own.
Every student will have a weekly meeting with the teacher, unless otherwise agreed. In this meeting, the student can ask questions to clarify the assignments, show the progress, and maybe get some extra hints when stuck in a problem.
Be aware that the teacher will not solve your problems for you, but only give suggestions how you could go forward.
It is of course strictly forbidden to share code between students. On the other hand, you are allowed as a group to set up a shared set of test cases.

Grading

There are five assignments, in which you develop one single, large program.
In order to get a pass (6 or higher), all five assignments must be finished and you must proof the correct working of your program by providing some test cases and their results. The program must have a basic but clear user interface (either console or GUI) that fulfills the minimal requirements.
Only the final program will be graded, not the intermediate versions.

To achieve a grade above 6, the following suggestions are given:

- Add documentation that fully and clearly explains the working of your program, including a motivation of important design decisions..
- Give the application a well-designed GUI with a professional look and feel.
- Make the application foolproof, and handle any input errors graciously.

- Add extra convenience options for the user (like reading input from file instead of typing, etc).
- Add extra levels of testing, like code coverage, automated code review, etc.
- Use tools to examine the complexity of your code, and refactor if necessary.
- Be creative and add other interesting, relevant, well documented and tested functions to the application.
- …

Proposition logic

Input format for proposition logic

In Proposition Logic, Formulas are created by using:

- Predicates (also called “variables”):
  - each capital letter is a Predicate
- Connectives (also called “Boolean operators”), in our case that will be:
  ¬, ⇒, ⇔, Λ and V.

In this course, we use the following ASCII prefix notation for the Connectives:

Infix notation	ASCII prefix notation
`¬A`	`~(A)`
`A ⇒ B`	`>(A,B)`
`A ⇔ B`	`=(A,B)`
`A Λ B`	`&(A,B)`
`A V B`	`|(A,B)`

So the infix Formula

`(A⇒B)⇔(¬AVB)`

will be written in ASCII prefix notation as

`=( >(A,B), |( ~(A) ,B) )`

There are two special Predicates: `0` which always has value `false`, and `1` which always has value `true`.

Real life relevance

Formulas can be realized with electronic logic circuits. In this way, a complete processor can be build. NAND circuits (see

Assignment 5) are used in chip development because they are smaller than other circuits (and they use less energy).
See also:

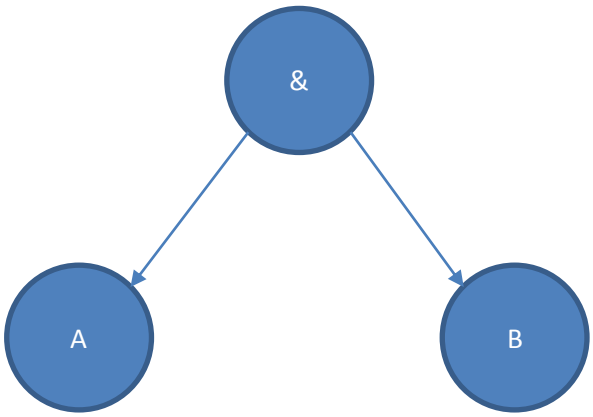
- https://en.wikipedia.org/wiki/Logic_gate
- https://en.wikipedia.org/wiki/Adder_%28electronics%29
- http://www.neuroproductions.be/logic-lab/ (and various other logic gate simulators)

Test vectors

For a proper test of your system you need a set of Formulas for which you know the results, including Formulas which touch all the difficult aspects of your software. When handing in the program, the correct working must be shown from the user interface. It is advised to also use unit-tests as much as possible, because then, you can test your code in the earliest possible stage of development. As mentioned before, it is allowed to set up a shared set of test cases.

Assignment 1

Formulas are in fact tree structures. For example, the prefix Formula “&(A,B)” can be represented by the following Proposition Tree:



Such a Proposition Tree corresponds one-on-one with an infix Formula, in this case “A&B”

Write a program that reads a Formula in ASCII prefix notation (as described above in Input format for proposition logic) and builds a Proposition Tree of objects internally.

Furthermore:

- Spaces must be allowed in the input, but have no impact.

Show a picture of the Proposition Tree (see Of course, you already installed an automated test to validate that all hash codes are identical.

- Graphical representation of a logic Formula below).
- For testing purposes, you could print the input Formula in infix notation.

Tip: You might want to use recursion to read the Formula. Use one base class (or interface) for each node and let method `ToString()` return the Formula in infix notation.

Assignment 2

Extend your program such that the Truth Table for a given Formula is calculated and shown.

For example the Truth Table of (A ∨ ¬B) ∧ C is:

A	B	C	(A ∨ ¬B) ∧ C
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Put the names of the Predicates in alphabetic order when printing the table.
Tip: make a method, which returns an ordered list of all Predicates of the Formula (each Predicate must appear only once in the list).

The bits in the last column are called the Truth Values. Concatenate the Truth Values from bottom to top (so the first line is the LSB), and print them as a hexadecimal number. We call this the Hash Code and it is an identification of the Formula.
The Truth Values in the above Truth Table are (from bottom to top): 10100010, so the Hash Code is: A2.
See also https://en.wikipedia.org/wiki/Hexadecimal.

Let your program also show the Hash Code.

Assignment 3

If a Truth Table contains two lines with the same Truth Value where all-but-one Predicates have the same value, then the value of this Predicate is apparently not important. Both lines can be rewritten with a `*` as value for that Predicate (also called 'don't care').
For example, the Truth Table of Formula `(A ∨ B) ∨ C` can be simplified from

A	B	C	(A ∨ B) ∨ C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

into

A	B	C	(A ∨ B) ∨ C
0	0	0	0
*	*	1	1
*	1	*	1
1	*	*	1

Extend your program such that the Simplified Truth Table is shown as well.
Obviously, if you take the Truth Values of the Simplified Truth Table from bottom to top (in this case: 1110, or hexadecimal C), it is not the same as the Hash Code of the original Truth Table (which is: 11111110, or hexadecimal FE).
Find a way to calculate the Hash Code of a Simplified Truth Table in such a way that it is equal to the Hash Code of the original Truth Table.
Show the Hash Code of the Simplified Truth Table.

Assignment 4

From a Truth Table, we can construct a Disjunctive Normal Form of the Formula. Take for every line with a Truth Value of 1 the conjunction of the Predicates with value 1 with the conjunction of the negation of the Predicates with value 0. Finally, take the disjunction of those Formulas. For example:

A	B	A ⇒ B
0	0	1
0	1	1
1	0	0
1	1	1

When applying the rules, we first get these Formulas for the lines with Truth Value 1:

- `(¬A ⋀ ¬B)`
- `(¬A ⋀ B)`
- `(A ⋀ B).`

Grouping them together as disjoints gives us the Disjunctive Normal Form:

- `¬A ⋀ ¬B) ⋁ (¬A ⋀ B) ⋁ (A ⋀ B).`

When we strictly apply the rules ('`*`' is not '`0`' and not '`1`') then we can use it for simplified Truth Tables as well.
The Disjunctive Normal Form from the Simplified Truth Table of `(A ⋁ B) ⋁ C` yields `(A ⋁ B) ⋁ C`. This Formula is much simpler than the Disjunctive Normal Form of the full Truth Table.

Extend your program such that the Disjunctive Normal Form is displayed for both the original and the Simplified Truth Table.
Those Disjunctive Normal Forms can be read again by your program and must of course deliver the same Truth Table and Hash Code.

Tip: a contradiction and a tautology are nice test cases.

Assignment 5

Each Formula can be rewritten by using only the Connectives ~ and &, or only the Connectives ~ and | (e.g. apply De Morgan on a Disjunctive Normal Form).
To obtain a Formula with only ~ and &, rewrite the Connectives as follows:

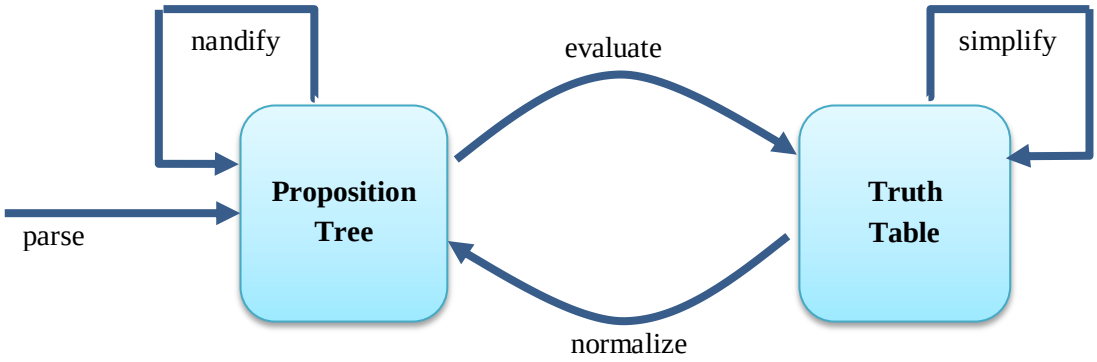
Formula	becomes
A ⇒ B	¬A ⋁ B
A ⇔ B	(¬A ⋀ ¬B) ⋁ (A ⋀ B)
A ⋁ B	¬(¬A ⋀ ¬B)

If we define the Connective NAND ("Not AND") as ¬(A⋀B), then we can rewrite all Formulas with only this Connective.
We use the following ASCII prefix notation for a NAND: %(A,B).

Extend your program to convert a Formula into an equivalent Formula with only NANDs.
The new NAND Formula can be read again by your program and must of course deliver the same Truth Table and Hash Code.

Recapitulation

Your program probably has 2 data structures: a Proposition Tree and a Truth Table. In the assignments, you might have written methods to convert one data structure into the other. Perhaps you have different names, but you might see the following relationships between data structures and methods:



Given the original Proposition Tree, the Hash Code of the 6 Truth Tables, which are constructed as follows, should all be the same:

- original + evaluate
- original + evaluate + simplify
- original + nandify + evaluate + simplify
- original + evaluate + normalize + evaluate
- original + evaluate + simplify + normalize + evaluate
- original + evaluate + simplify + normalize + nandify + evaluate

Of course, you already installed an automated test to validate that all hash codes are identical.

Graphical representation of a logic Formula

If you want to add a graphical picture of your automaton, you can follow the following steps:

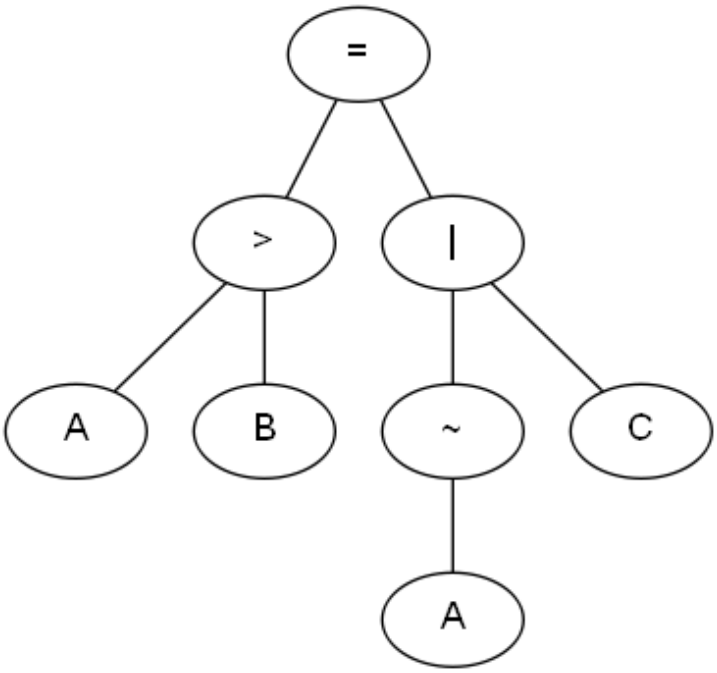
1. install GraphViz
2. adapt your `$PATH` environment variable
3. in your application:
   3.1. generate a text file (e.g. `abc.dot`), similar to this:

```
graph logic {
   node [ fontname = "Arial" ]
   node1 [ label = "=" ]
   node1 -- node2
   node2 [ label = ">" ]
   node2 -- node4
   node4 [ label = "A" ]
   node2 -- node5
   node5 [ label = "B" ]
   node1 -- node3
   node3 [ label = "|" ]
   node3 -- node6
   node6 [ label = "~" ]
   node6 -- node13
   node13 [ label = "A" ]
   node3 -- node7
   node7 [ label = "C" ]
}
```

   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 a proposition tree
                      // into a dot-format file
```

```
Process dot = new Process();
dot.StartInfo.FileName = @"C:\Program Files\<your path>\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 = { "C:\\Program Files (x86)\\<your path>\\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)

Ideas for other nice features of GraphViz are welcome.