Automata and Logic Engineering 1

Logic in Professional Practice

Contents

Introduction

This document describes the assignments for the course ALE1 (3 ECTS) and the innovation route Academic Preparations as part of the pre-master program for the Tue (6 ECTS).

Make sure that your lecturer is properly informed about all smart activities that 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 6 ECTS)**	**week (for 3 ECTS)**
1	3	1	1+2
2	1	2	3
3	3	2	4+5
4	2	3	6
5	1	3	7
6	3	4	-
7	2	5	-
8	4	6	-

Grading

To pass this course (grade 6), all assignments are implemented and work properly with an easy-to-use GUI (even on another machine (in particular: the lecturer's)).

For higher grades: incorporate the following aspects:

- good software design (classes, interfaces, SOLID principles, Design Patterns, …)
- clear documentation of your actual design and your design decisions
- proof of the robustness of your code (thorough test cases, code analysis, code coverage)
- robust recovery for incorrect user input
- other smart inventions and spectacular new features

All assignments have to be submitted in Canvas before their deadlines. For all but the last assignment, a snapshot of your software project is sufficient. After the last deadline, there will be an individual meeting with the lecturer with a discussion and explanation of your work.

Learning goals

Besides the specific contents of 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

Proposition logic

Input format for proposition logic

Each capital letter is a proposition variable. Formulas for proposition logic are built with proposition variables and connectives like ¬, ⇒, ⇔, ∧ and ∨. For simplification in this course we use the following ASCII prefix notation of the formulas:

Logic notation	ASCII
`¬A`	`~(A)`
`A ⇒ B`	`>(A,B)`
`A ⇔ B`	`=(A,B)`
`A ∧ B`	`&(A,B)`
`A ∨ B`	`|(A,B)`

So the formula

```
(A⇒B)⇔(¬A∨B)
```

will be written as

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

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

Spaces are allowed in proposition formulas, but they must be ignored.

Tips

Probably, you will make Assignment 1 with a text box and a button, and the results will be put in another text box. For all next assignments, it's convenient to start them by the same button (or by a new button) and to put the results in separate text boxes.

Real life examples

Propositions 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, preferably formulas which touch the difficult aspects of your formulas.

Testing can be done by module tests, but in this case every participant of this course has to write the tests again. It would be better if you all have a system, in which you can exchange your test formulas (and their results) with each other. This system should be easily extendible with new formulas and can be executed by each implementation.

Develop such a system within your group (for example: a common database or shared Dropbox files with a well described syntax)

Assignment 1

In the Logic and Set Theory course you have learnt that formulas are in fact tree structures. Write a program that reads a formula in ASCII format (as described in Input format for proposition logic) and builds a tree of objects internally. Note: spaces are allowed but must be ignored.

Furthermore: make a method which returns a list of all proposition variables of the formula (each variable should appear only once).

And show a picture of the tree (see Additional assignments

Assignment A

Convert a proposition formula (so without quantifiers) into CNF (Conjunctive Normal Form). For example:

```
E ∧ (A ⇒ (B ∧ C) ∨ (D ∧ ¬C))
```

is converted into:

```
(¬A ∨ B ∨ ¬C) ∧ E ∧ (¬A ∨ C ∨ D)
```

Furthermore, print this CNF in a shorter format like: `[ aBc, E, aCD ]`, and make sure that a formula can be entered in this format as well.

Please note:

- the tree is not a binary tree anymore, and its depth is 2 (one ∧ at the top layer, and only ∨'s at the second layer (or: depth is 3 if you consider possible ¬'s)) (tip: you might need new classes like MultiOr and MultiAnd)

- the CNF *must* be constructed by manipulating the formula, and *not* via the truth table

Assignment B

Check if a proposition formula is satisfiable, and if yes, find the appropriate values for the proposition variables.

Apply the Davis-Putnam algorithm.

During execution, show the internal actions of the algorithm (with clear indication where it happens in the recursion).

Afterwards: show the final assignment like "100110" (in alphabetic order of the variables) and make an explicit check that the assignment is indeed correct.

Provide at least 5 hand made proposition formula's that cover all aspects of the algorithm.

Assignment C

Perform a Tseitin transformation on a proposition formula and apply the Davis-Putnam algorithm (as described in another assignment).

Note: take care that the new formula may have too many variables to show its truth table.

Assignment D

Generate a random proposition with random number of variables and random operators.

Apply all possible checks (as described in the other assignments) on this proposition.

You may consider an upper bound of max 5 variables.

Tip: have a button that is repeatingly generating random propositions for an endurance test.

Graphical representation of a logic formula).

Tips:

- 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
- as the next assignments build on this first assignment: spend a serious amount of time to make a clean design; you will benefit from it

Assignment 2

Extend your program such that the truth table of a given proposition formula is calculated.

Put the names of the variables in alphabetic order when printing the table. Concatenate the digits in the last column and print them as a hexadecimal number (where the first line is the LSB). We call this the *hash code* and it is an identification of the formula. Example:

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

The last column yields: 10100010, or: A2. See also https://en.wikipedia.org/wiki/Hexadecimal.

Assignment 3

If a truth table contains two lines with the same truth value where all-but-one variables have the same value then the value of this predicate with the different values is not important. Both lines can be rewritten as one new line with a `*` as value for that variable (also called 'don't care'). The truth table of `(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.

Assignment 4

From a truth table you can construct unambiguously a disjunctive normal form of the formula (DNF). Take for every line with a truth value of 1 the conjunction of the variables with value 1 with the conjunction of the negation of the variables 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 the formulas `(¬A ⋀ ¬B)`, `(¬A ⋀ B)` and `(A ⋀ B)`. Grouping them as disjoints together 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 truth table of `(A ⋁ B) ⋁ C` yields `(A ⋁ B) ⋁ C`. This formula is of course simpler than the disjunctive normal form of the full truth table.

Extend your program such that the disjunctive normal form is displayed (for both original and simplified truth tables). Those disjunctive normal forms can be read again by your program into another proposition tree and must deliver the same truth table and hash code.

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

Assignment 5

Each truth table can be created only with either the connectives `~` and `&`, or only with the connectives `~` and `|` (e.g. apply De Morgan on a disjunctive normal form). You can obtain a formula with only `~` and `&` by rewriting the connectives as follows:

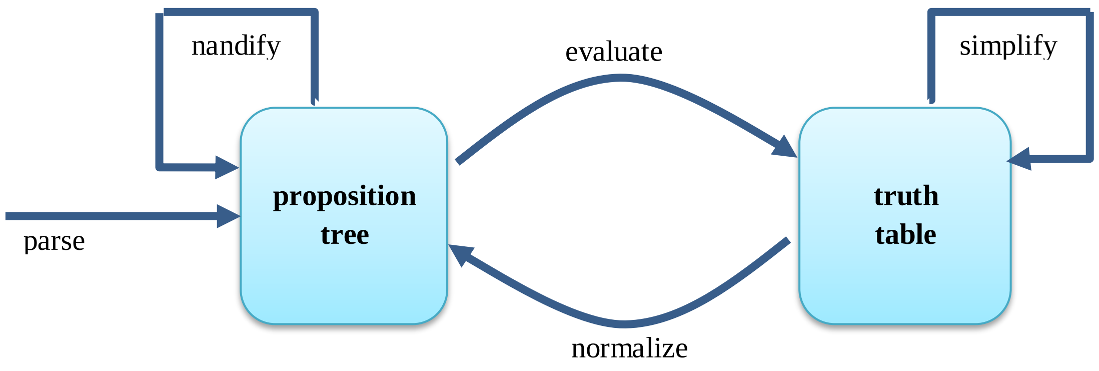
formula	becomes
`A ⇒ B`	`¬A ⋁ B`
`A ⇔ B`	`(¬A ⋀ ¬B) ⋁ (A ⋀ B)`
`A ⋁ B`	`¬(¬A ⋀ ¬B)`

If we define the connective NAND ("*Not AND*") as `¬(A⋀B)`, then we can create all truth tables only with this connective. We use the following ASCII notation: `%(A,B)`.

Extend your program to convert a proposition tree in an equivalent proposition tree with only NANDs. The new NAND proposition tree can be handled again by your program and must deliver the same truth table and hash code.

Recapitulation

Your program has probably two 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, calculate the hash code of 6 truth tables which are constructed as follows:

1. original + evaluate
2. original + evaluate + simplify
3. original + nandify + evaluate + simplify
4. original + evaluate + normalize + evaluate
5. original + evaluate + simplify + normalize + evaluate
6. original + evaluate + simplify + normalize + nandify + evaluate

Of course, install an automated test to validate that all hash codes are identical.

Semantic Tableaux

Assignment 6

Implement the proof procedure with semantic tableaux for proposition formulas. The output indicates if the formula is a tautology, with a complete GraphViz-tree of the steps.

Input format for predicate logic

Besides proposition logic we are going to work with predicate logic. Therefor we enhance our input syntax:

A predicate is an uppercase letter with 1 or more object variables listed in parenthesis, like: (`P(a)` or `Q(x,y)`). An object variable is a lower case letter. The number of variables for a predicate symbol is not pre-determined, but it stays the same in the formula (so: it's not allowed to have both `P(a)` and `P(b,c)` in one formula).

The following formulas are valid formulas for predicate logic:

1. proposition variables are valid formulas
2. predicates (applied on 1 or more object variables) are valid formulas
3. formulas build from (smaller) formulas and connectives ¬, ⇒, ⇔, %, ⋀ and ⋁ are valid formulas
4. if `x` is a variable and `F` is a valid formula (which might contain `x`), then `∀x.(F)` (read: “for all `x` yields `F`”) and `∃x.(F)` (read: “there exists an `x` such that `F` holds”) are valid formulas as well.

The ASCII notation for quantifier formulas are:

Formula	ASCII
`∀x.(F)`	`@x.(F)`
`∃x.(F)`	`!x.(F)`

Assignment 7

Extend your program such that predicate formulas can be read and can be presented as a tree. Moreover, write methods to:

a. indicate if the formula is a proposition formula (so: it doesn't contain predicates, object variables nor quantifiers)
b. in a formula: change all appearances of an unbound variable into another so if we change `x` into `y`, then `∀z.(P(x,z))` becomes `∀z.(P(y,z))`

Assignment 8

Extend your program to proof predicate formulas with the help of sematic tableaux. The output indicates if the formula is a tautology, with a complete tree of the steps.

Additional assignments

Assignment A

Convert a proposition formula (so without quantifiers) into CNF (Conjunctive Normal Form). For example:

```
E ⋀ (A ⇒ (B ⋀ C) ⋁ (D ⋀ ¬C))
```

is converted into:

```
(¬A ⋁ B ⋁ ¬C) ⋀ E ⋀ (¬A ⋁ C ⋁ D)
```

Furthermore, print this CNF in a shorter format like: `[ aBc, E, aCD ]`, and make sure that a formula can be entered in this format as well.

Please note:

- the tree is not a binary tree anymore, and its depth is 2 (one ⋀ at the top layer, and only ⋁'s at the second layer (or: depth is 3 if you consider possible ¬'s)) (tip: you might need new classes like MultiOr and MultiAnd)
- the CNF *must* be constructed by manipulating the formula, and *not* via the truth table

Assignment B

Check if a proposition formula is satisfiable, and if yes, find the appropriate values for the proposition variables.

Apply the Davis-Putnam algorithm.

During execution, show the internal actions of the algorithm (with clear indication where it happens in the recursion).

Afterwards: show the final assignment like "100110" (in alphabetic order of the variables) and make an explicit check that the assignment is indeed correct.

Provide at least 5 hand made proposition formula's that cover all aspects of the algorithm.

Assignment C

Perform a Tseitin transformation on a proposition formula and apply the Davis-Putnam algorithm (as described in another assignment).

Note: take care that the new formula may have too many variables to show its truth table.

Assignment D

Generate a random proposition with random number of variables and random operators.

Apply all possible checks (as described in the other assignments) on this proposition.

You may consider an upper bound of max 5 variables.

Tip: have a button that is repeatingly generating random propositions for an endurance test.

Graphical representation of a logic formula

If you want 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[1]

[1] see https://java.com/EN/DOWNLOAD/HELP/PATH.XML

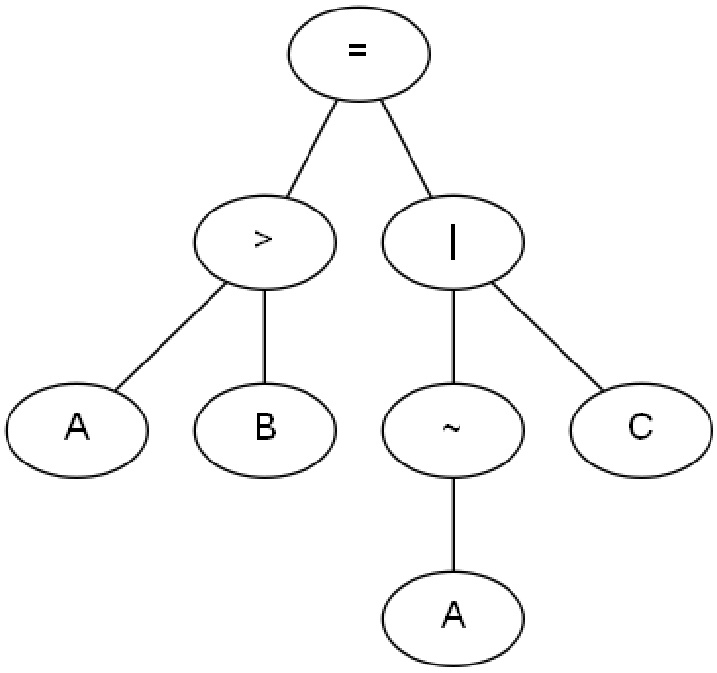
3. in your application:
   3.1. generate a text file (e.g. `abc.dot`), similar to this:

```
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 3.1 - 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 = @"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.