An education tool for writing proofs in CO884

Maxime MERLIN
Advanced Computer Science (Computational Intelligence) – University of Kent
mm711@kent.ac.uk

ABSTRACT

In this paper, I will talk about my project research. The subject of this project is "An education tool for writing proofs in CO884".

The aim of this project is to develop a tool that could be used for the module CO884 – Logic and Logic Programming, within the University of Kent.

This tool has to be focused on the truth tables, which is a part of propositional logic, and which is taught in the CO884 module. It has to provide a way for students to simply practice and work on truth tables.

Through this paper, I will briefly explain what propositional logic is, and what truth tables are, in order to setup the background. I will also talk about the researches I have made, and about the specification I defined for the tool. Then finally, I will detail all the development process, which is split in different important parts and different challenges. But also the issues I encountered during the development process.

1 INTRODUCTION

In this paper, I will give details about my researches, the technologies I will use to develop my project, but also the issues I encountered and the steps/objectives I achieved to lead my project research to a successful end.

The title of my project research is "An education tool for writing proofs in CO884". Thus, the project I will attempt to develop is an education tool that will be used by students who study the module "CO884 – Logic and Logic Programming".

This module, available for computer science students, firstly teaches its students propositional and predicate logic, and resolution. And secondly teaches Prolog programming after students learnt the basics about logic, and what it means. The part I will work on is about truth tables in propositional logic.

Those truth tables are used to see the semantics of a propositional formula. In an informal way to say it, it tells whether a propositional expression containing logical operators is true of false, and this for every combination of input. You can see an example of a truth table in the following Fig 1.

p	q	r	(p ∨ q)	(q ∨ r)	(p ∨ q) ∧ (q ∨ r)
T	T	T	T	T	T
T	T	F	T	T	T
T	F	T	T	T	T
T	F	F	T	F	F
F	T	T	T	T	T
F	T	F	T	T	T
F	F	T	F	T	F
F	F	F	F	F	F

Fig 1. Truth table example

My project research aims to answer several needs for the CO884 module. At the moment I am writing these lines, the whole propositional and predicate logic, and resolution, is entirely

taught by lectures, and the exercices are done using pens and papers. The first objective of my project research is to provide to students a better way to complete exercice sheets given in the module, and also a better way to practice with truth table exercices. It is important to notice that I will only deal with the truth tables part, since other kinds of exercices such as equivalences, are treated by other project researches. As I experienced it myself, I found pretty annoying to use a pen and paper support to deal with truth tables. It takes time, and it is a very repetitive task. It requires for the student to draw tables for each formula he's working with, and it quickly becomes unclear if mistakes are made. Another objective of my project research concerns the teachers of the module. Currently, teachers have to think about new examples for truth tables exercices, my solution will generate this exercices automatically, thus, it means teachers won't have to worry about this anymore. Plus, if students want to do more exercices about truth tables in order to practice, they will be able to generate as much exercices as they want (which is currently not possible using exercice sheets). Also, the teachers currently have to go from table to table in order to check if the students' answers are correct. With an education tool, the correction will be done by the software, which means students will be more independent. The software will be able to recognize where a mistake has been made and give explanations about it. And finally, my software will provide 3 different difficulty levels: easy, medium and hard. Each difficulty level will add challenges for the student. The easy level will only contain basic formulas that are easily resolved, and the hard level will generate formuleas that are more tricky. The amount of information and hints given to the student will decrease as the level of difficulty increase. This will allow the students to practice with easier or harder exercices, depending what the students need.

2 BACKGROUND

2.1 LOGIC

As I am going to talk a lot about propositional logic in this paper, I would like to start by talking about it and explain what it trully is.

The study of logic started at first with the works of Aristotle in 384-322 BCE. He used to work with quantifiers such as "some" or "all". Those quantifiers are not included in propositional logic. However, he defined 2 really important rules called: *Law of Excluded Middle* and *Law of Contradiction*. Those 2 laws state that a statement/proposition is either true or false, and can't be true and false at the same time. Later, some works have been done by the Stoic philosophers about logical operators such as "and", "or" and "if…then". This allows more complexity in the logic statements.

A statement or proposition in propositional logic, can be defined as a declarative sentence, or part of a sentence, that can be true of false.

"Paris is the capital of France."
"In England, most people speak English."
"The sun is green."

Those 3 sentences are statements. The 2 first ones are true, and the last one is false. None of them can be true and false at the same time. Logical operators can be used to generate more complex statements. For example:

"Paris is the capital of France or Paris is the capital of England."

The first part or this sentence (before "or") is true, and the second part is false. Although, the fact that the first part is true makes the whole statement true. This is basically, what propositional logic is. It is a branch of logic that studies ways of combining different statements

together. Those simple statements are linked by what we call "logical operators". [1]

2.2 Logical Operators

There are 5 different logical operators used in propositional logic[2]:

- Conjunction $\wedge$ ("A and B")
  Both parts of the statement must be true for the whole statement to be true.

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

  "Paris is a capital AND Paris is in France" - TRUE
  "Paris is a capital AND Paris is in England" - FALSE
  "Paris is a village AND Paris is in France" - FALSE
- Disjunction $\vee$ ("A or B")
  One part of the statement must be true for the whole statement to be true. It has to be noted that the "or" we use is an "inclusive or".

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

  "Paris is in France OR Paris is in England" – TRUE
  "Paris is a village OR Paris is a capital" – TRUE
  "Paris is a village OR Paris is in England" - FALSE
- Implication $\Rightarrow$ ("if A then B")
  If the first part of the statement is false, then the whole statement will be true whether the second is true or false. The whole statement is also true if both parts are true. But, the whole statement is false if the first part is true and the second is false.

A	B	$A \Rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

  "IF Paris is in England THEN Paris is a village" – TRUE
  "IF Paris is in England THEN Paris is a capital" – TRUE
  "IF Paris is in France THEN Paris is a capital" – TRUE
  "IF Paris is in France THEN Paris is a village" - FALSE
- Bi-implication $\Leftrightarrow$ ("A if and only if B")
  The bi-implication is similar the implication, but the difference is that if the first part is false, and the second is true, the whole statement will be false instead of being true.

A	B	$A \Leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

  "Paris is in England IF AND ONLY IF Paris is a village" – TRUE
  "Paris is in England IF AND ONLY IF Paris is a capital" – FALSE
  "Paris is in France IF AND ONLY IF Paris is a capital" – TRUE
  "Paris is in France IF AND ONLY IF Paris is a village" – FALSE

- Negation ¬ (“not A”)
  The negation is simply the opposite of the statement.

A	$\neg A$
T	F
F	T

NOT “Paris is in France” – FALSE
NOT “Paris is in England” – TRUE

Thus, if you look at the truth table in Fig 1, you can now understand how truth tables work. In this example there are 3 statements: p, q and r. Each of them has a value true or false. The number of different combinations is defined by: $2^{\text{number of variables}}$ (=8 in the example). The first 3 columns on the left are all the different combinations between the 3 statements. The whole formula is split in several parts that are resolved one by one for each combination. The whole formula for this example is: (p ∨ q) ∧ (q ∨ r). So we first need to resolve (p ∨ q) for each combination, then (q ∨ r). And finally, we can use the results to resolve the whole formula (p ∨ q) ∧ (q ∨ r).

p	q	r	(p ∨ q)	(q ∨ r)	(p ∨ q) ∧ (q ∨ r)
T	T	T	T	T	T
T	T	F	T	T	T
T	F	T	T	T	T
T	F	F	T	F	F
F	T	T	T	T	T
F	T	F	T	T	T
F	F	T	F	T	F
F	F	F	F	F	F

Fig 1 bis. Truth table example

2.3 Syntax Trees

Another important part related to truth tables and propositional logic, is the syntax trees. It is a different way to reperesent a logical formula and it gives us more information about the structure of the formula (e.g. inside of parenthesis have to be resolved first).

For example, if I stick to my truth table example, I can represent the formula “(p ∨ q) ∧ (q ∨ r)” with the following syntax tree:

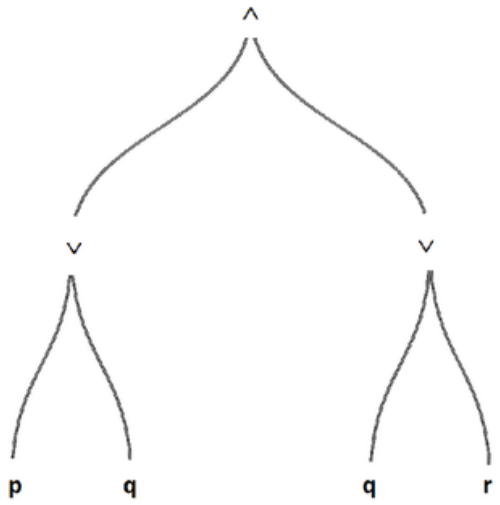


Fig 2. Syntax Tree example

With this syntax tree, it is easy to see that the highest-level connective is “∧”. Which means that before this operator, I will have to deal with the lower-level conectives that are both “∨”.

3 Specification & Needs

In this part of the paper, I am going to explain the specification of my software, and the needs that it will have to fulfilled.

3.1 Specification

First of all, the software has to be easily runable by each student. The easiest way that came to my mind was obviously to have the software installed on every computers present in the rooms where CO884 is taught during the year. Since all the campus computers are running Windows 7, a common Windows program will

perfectly correspond with what I need. In order to enlarge my range, it would be nice for the software to be downloadable by the students who desire to run it on their own computer.

Another important property of my software is that it has to be easy to use. Everyone can use pens and papers, but when it comes to use a program, people react differently when facing difficulties. Some of them will be experimented enough to use a complex user interface, some others will not. Thus, the software should have a couple of different pages, not to much so the users won't get lost easily, and should also be user friendly. Let me explain quickly what I mean by "pages".

Most application with a user interface running on Windows have at least one window, but might as well have several ones. For my project, I think only one window is needed. But in this window, different contents will have to be displayed. This means that I will need a sort of menu that the user can click to change the content he wants to display. What I call a page, is a section of this menu. It splits the different parts of the software, and it is able to display different contents in those parts.

3.2 Needs

In addition to these important points, the software will have different features that will be able to fulfilled the needs of the students and teachers.

The student will need to:

- Access information about the courses related to the truth tables
- Practice on truth tables resolution
- Evaluate himself
- Access corrections
- Print (exercices or courses information)

The teacher needs the software to:

- Auto generate logical formulas
- Accept logical formulas input
- Correct filled truth tables and display errors

Knowing all ot this, I determined that my software could implement no more than 3 main pages in order to fulfilled the students and teachers needs in the simplest possible way.

3.3 User Interface Design

The first page concerns the courses contents that are related to truth tables. The software will implement a "Classroom" page. It will able the students to access information about truth tables, logical operators, syntax trees and, more generally, logic. Those information will be fetched from the Powerpoints and PDFs provided by the teachers. Implementing such a thing will allow the students to stay on the software while accessing courses information. Each page of information will have a button allowing the student to display the content in another window, so he can keep an eye on what he wants while practicing. Although, those windows won't be accessible during an exam (seperated or not). Each page will also have a printing button.

The second page will be called "Practice" and will provide tools for the students to practice. One truth table at the time will be presented to the user, and will ask him to fill it. The truth tables will be either auto generated or generated from an input from the user (a logical formula). A settings page will be displayed before every creation of a practice. The settings will make the user choose: a name for the practice, wether the formulas should be auto generated or not, the level of difficulty, the display of tool tips on the truth tables, the display of syntax trees, the display of logical operators truth tables. Afterward, the student will fill a truth table, he will have to confirm before moving to the next

one. After confirmation, the incorrect answers in the truth table will be highlighted. Then, another truth table is presented to the student. This exercices have no marks, and no end. They are provided to the student just in order to practice. This, brings me to the next page.

The last page needed by the software is the "Exam" page. It will operate quite similarly to the "Practice" page. Unless, in this page, the student will not have any help from the software and the exam ends when every present truth tables are answered (they might be correct or not). After the end of the exam, the correction of each truth table is displayed to the user, but also his mark for the exam. He will have the choice between, restart (different formulas), print, and exit.

In order to achieve my goals, I took the time to choose wisely which technologies I was going to use. Which brings me to the next part.

4 TECHNOLOGIES

In this part I am going to list the technologies I chose to use for my software.

As I said before, my software has to run easily on Windows. C# and WPF are part of the well known languages used to develop on Windows. I also could have used Java (which is probably the most used one), but I am already familiar with C# and WPF, and I appreciate it.

C# is a modern programming language used to build application using Visual Studio (Integrated Development Environment)[5] and the .NET[4] Framework[3]. WPF (Windows Presentation Foundation) is a graphical subsystem that works using the .NET Framework. It is one of the most used presentation system used for graphical user interfaces when developing with C#/.NET on Windows[6]. The combination of these tools will allow me to develop easily my application and its user interface for any Windows system.

Being already familiar with these tools, I know from my past experiences that they are enough to produce a decent GUI (Graphical User Interface), but I chose to add another tool on top of it. This tool is called ModernUI[7].

ModernUI add a set of controls and templates to WPF in order to make good looking and user friendly GUI.

The first appendix shows the difference between a Windows program GUI with WPF and a Windows program GUI with ModernUI for WPF. You can see that the design is better looking, and the layout is more practical and user friendly. The tabs are redesigned to display something less strict than with WPF only, and the buttons are personalisable. Also, a previous button is added at the top left of the window. The navigation between different pages is managed by ModernUI.

And that is everything I will use for the GUI of my software.

Finally, after some researches and thoughts about my project, I realised that I will need another tool, which is not related to GUIs. Indeed, my program will have to deal with inputs (logical formulas) in order to create empty truth

tables that the user will be able to fill. Those inputs will be provided by the user, or will be auto generated by the program. In a program that deals with input that can be correct or incorrect (e.g. "q ∨ r" is correct, but "q ∨ ∨ r" is not), those inputs will have to be checked at one point before the generation of a truth table. A good and safe way to do so, is to use a "Lexical analysis" (Parser/Lexer)[8]. In a lexical analysis process, the lexer will translate a sequence of characters in a sequence of tokens. In my case, with logical expressions, my tokens could be for example: VARIABLE, OPERATOR, PARENTHESIS, ENDOFLINE… Then, after each character has been tokenised, the sequence of tokens is sent to a parser. The lexer itself is kind of a parser, but the actual parser will take care of the analysisation of the sequence of tokens and the determination of whether the sequence is correct or not. Plus, in order to correct the user, my software will have to fill any truth table itself to compare the user's answer with its. Computingly speaking, I think the best way to do that is to use the syntax trees (see previous Background part). The lexical analysis of my inputs will allow me to easily build trees from them. So for this part of the job, I chose to use ANTLR[9]. ANTLR is a parser/lexer generator. It is usable by different languages, including C#. With this computer-based language recognition, I will be able to define what a correct logical expression is, to build a tree from a correct input, and to resolve it using this tree.

5 Development

Now that I talked about my objectives and the technologies I will use to reach them, it is time to start the development part. In this part I will go through details for every part of the development. I will explain how I managed to implement all the needed features, but I will also talk about the eventual issues that I encountered.

5.1 Main Layout

First things first, I had to find a good point where to start my software from scratch. When creating a new WPF project in Visual Studio, the only thing that is provided is an empty window. As I said before, one window is enough for my project, which means I didn't have to deal with this. The window is created, and I don't need any other window. Below you can see a screenshot of how an empty window looks like.

Fig 3. Empty WPF window

You can see that this empty WPF window has the default appearance and does not implement the ModernUI appearance (like shown in the Appendix 1). To give this window its new appearance, I had to add a reference to ModernUI in my Visual Studio project and change the code a little bit.

After the reference to ModernUI is added, I can access new "controls" in my project. "Controls" are all the elements a WPF GUI can contain. For example, a button is control, or a line of text, or a table… The window itself is a control !

At first, the window is actually a "Window" control. What I needed there, was to change this control by a "ModernWindow" control, which is contained in the controls provided by ModernUI and which looks like the right part of the Appendix 1.

Here is the differences in the code:

```
<Window>
</Window>
```

```
<mui:ModernWindow xmlns:mui="http://firstfloorsoftware.com/ModernUI">
</mui:ModernWindow>
```

Fig 4. Window and ModernWindow tags

On the first piece of code is the default Window tag. On the second one is the ModernWindow tag. The main difference here, beside the name of the control, is that I need to notify the program that the control I am looking for is in "mui", because it is not a default control integrated in WPF. In the tag, the "xmlns:mui" line allows me to define what "mui" is. Here it is a reference to ModernUI.

And here is the result on my empty window:

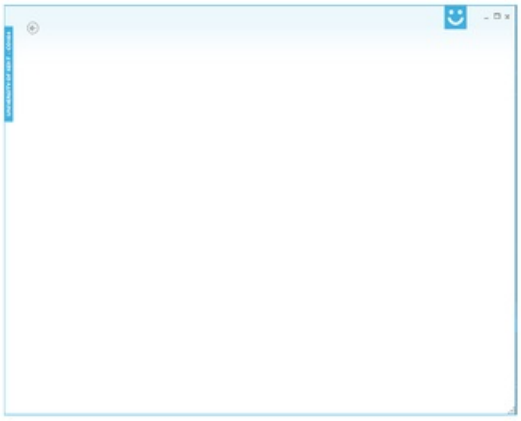


Fig 5. Empty ModernUI window

After I've done this, I was sure that the reference to ModernUI was correctly done. And also, what I called my "MainWindow" was set. The next step was the implementation of my several pages. I knew that I needed 3 different pages plus a homepage. Each of them has an undefined number of different contents.

To do so, I used another Control from ModernUI, which is the "MenuLinkGroups" Control. It is actually a modified version of the default WPF "Tab" Control. This means that, each page I have will be a tab, with different contents.

The MenuLinkGroups control is composed of several LinkGroups, and each LinkGroups is composed of several Links. The LinkGroups are actually my tabs, and the Links are the contents that will be displayed in my tabs.

Here is an example with one tab (LinkGroup) that contains several contents (Links):

```
<mui:ModernWindow.MenuLinkGroups>
        <mui:LinkGroup DisplayName="Lorem Ipsum">
            <mui:LinkGroup.Links>
                <mui:Link Source="/Pages/LoremIpsum.xaml" />
            </mui:LinkGroup.Links>
        </mui:LinkGroup>
    </mui:ModernWindow.MenuLinkGroups>
```

Fig 6. MenuLinkGroups tag

The example above will display a tab "Lorem Ipsum" that can display the content from "LoremIpsum.xaml". And so here is the result when I adapt it to the pages I want:

Fig 7. Pages setup

From that point, I was able to add pages, and navigate easily through them. The main layout part was over and I moved to the next part of the development.

5.2 Truth Tables

One of my first concern when I started my project was to find a way to represent a truth table as an class in my code. So I decided to start by this. My objective was to be able to instanciate a "TruthTable" object for any given logical formula, and to implement a method allowing me to display correctly this truth table in my GUI.

Since the truth table is constructed from the formula, it is the only information needed when generating a truth table. Thus, my constructor contains only one parameter, which is a string representing the formula. For practical reasons, I chose to replace the logical operators characters by more common ones, and implemented a method that gives me access to the "formated" formula, where I switch each logical operator for the correct character. So below are the substitution characters when operating with a logical formula.

- '∧' switched for '&'
- '∨' switched for '|'
- '⇒' switched for '->'
- '⇔' switched for '<->'
- '¬' switched for '~'

5.2.1 Lexical Analysis

The first step in the generation of a truth table is to make sure that the inputed formula is actually correct. As I said before, the solution I chose is a lexical analysis. I used ANTLR4 to do this, so I had to reference another library like I had to do for MordernUI before.

This lexical analysis is composed of 2 strongly linked parts: Lexer and Parser. They both use a grammar to define if an input fits correctly with the expected patterns. It might sounds a bit confusing, but I'll try to explain the role of each part.

Lexer

The lexer first defines the grammar. This grammar represents all the expected characters, or character sequences, and attribute them a token. One rule is then fixed, any character contained in the analysed input has to be in the grammar, otherwise, the input is incorrect, and doesn't fit with the expected pattern. The rules for my lexer are the followings:

TOKENS	CHARACTERS
BOOL	[0-1]|[a-zA-Z]
AND	'&'
OR	'|'
NOT	'~'
IMPLI	"->"
BIIMPLI	"<->"
IGNORED	' ' | '\r' | '\n'
ERROR	.

Note that for the BOOL, END and ERROR tokens I used regular expressions, in order to match a collection of characters. BOOL will match any single digit and any single letter. IGNORED will match a space or a new line character. ERROR will match any character that is not matched by the previous rules. From now on, any input analysed by the lexer will be tokenised and ready to be sent to the parser if no errors are detected.

Parser

The parser takes its input from the tokenised output of the lexer. Its role is to analyse each token contained in its input and determine if the order is whether correct or not, if it fits the desired pattern. Just like the lexer, the parser follows rules. My parser contains only 2 rules. The first one is quite simple:

```
prog: expr+ ;
```

"prog" represents the input. So, this rule defines that the input can only be composed of 1 or more "expr".

For now, *expr* is not defined. But, this rule already says that if an input is not composed by whatever *expr* is, it will not be considered as correct.

Obviously, the second rule is the definition of *expr*, and here is how it can be defined:

EXPR	DEFINITION
'(' *expr* ')'	parens
'~' *expr*	NOT
expr '&' *expr*	AND
expr '|' *expr*	OR
expr '->' *expr*	IMPLI
expr '<->' *expr*	BIIMPLI
BOOL	bool

And you can see the Appendix 2, for the actual code in my grammar file.

So, this means that each distinct part of the tokenised input has to respect one of these rules to be correct. The last line is a bit different from the previous ones. It says that *expr* can be defined as BOOL. I previously defined BOOL in the lexer. BOOL is a token that says the tokenised string is equal to 1 single digit or 1 single letter. It has to be noticed that it is called BOOL but it can represent a letter. The reason for this is that, a logical formula contains letters, but each line of the truth table is actually the formula with each letter replaced by true or false.

The key to understand how the parser works is to think about recursion. Over the time I passed on this software, I realised that recursion is fairly present when dealing with input parsing. Here are 2 examples that help to clarify how it works:

(p & q) -> r

This formula is split in 2 parts by the implication operator "->". My parser's rules say that "->" fits only between 2 *expr*. So each side is being checked. The right side matches with the BOOL rule. The left side contains parenthesis. A rule says that the parenthesis are valid if around an *expr*. So the middle gets checked. It contains a "&" operator. A rule says that it has to be wrapped between 2 *exprs*. So both sides are being checked. Both of them match BOOL, which make them *expr*. Everything has been checked, so the parser uses recursion to notify the previous operations if the next content is valid or not. So, when it goes back to "&", yes both sides are *exprs.* Then, the parenthesis, yes the middle is *expr*. And finally the "->", again, both sides are *exprs.* This logical expression is valid.

p -> & q

With this formula, the first operator "->" is found. A rule says each side has to be *expr.* Left side matches BOOL. Right side matches the AND rule. The rule for "&" says the left side has to be *expr*. But the left side is not a valid *expr* since there are no matching rule for "expr ->". The logical formula is then invalid.

Tree output

When a potential formula goes through the lexer and the parser without any error, the produced output is a tree. This tree is actually the computed representation of a logical formula. I can use it to iterate through each part of the formula, and get the correct answers in the truth tables.

Here is an example of a tree generated from my parser and lexer:

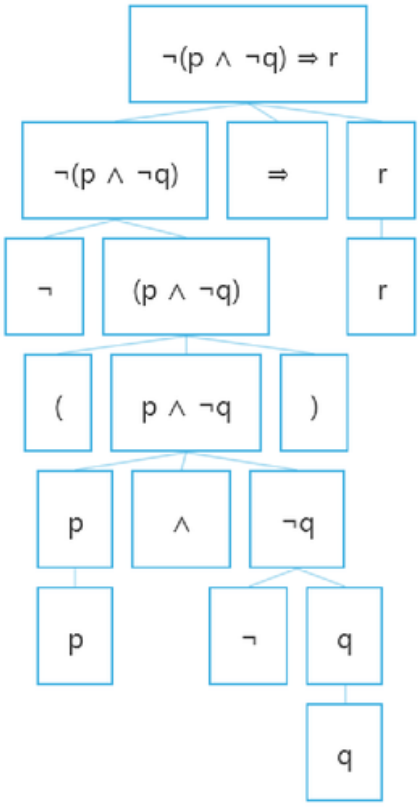


Fig 8. Output Tree

5.2.2 Computed Representation and Generation

After this point, I was able to correctly represent a logical formula. But I still had to figure out how to represent a whole truth table. This is what I am going to explain in this part.

Back to the constructor of my truth table. After my formula is stored, I am using my lexer/parser to store the produced output, which is a tree.

Each truth table has a number of line that depends on the number of variables (see the Background part). Each line represents the formula with each variable replaced by true or false, and 1 corresponds to 1 possible combination.

For example, for the formula "(p ∨ q) ∧ (q ∨ r)", one line will represent "(true ∨ false) ∧ (false ∨ true)", another one "(false ∨ true) ∧ (true ∨ true)", etc…

So what I decided to do is to create a new class called TruthTableLine, which represent one line of answer of a truth table. Then, I implemented a List of TruthTableLine in my TruthTable class. When creating a truth table, all combinations of booleans are generated, and for each combination, a line is added to the list.

The TruthTableLine's constructor takes in argument: the formula, and a List of KeyValuePair representing the boolean the variables with the bollean value they represent. In the constructor all variables are changed by their respective values, and another tree is generated, but this time with the actual boolean values. Having this tree with trues and falses, will allow me get a true or false return value when asking for the result of any part of the formula.

In a truth table, each line has several results. This is why I decided to create another class called TruthTableLineResult, which represents 1 of the results of a truth table line. This class contains several information about the result, and the result itself.

So, each TruthTableLine has a list of TruthTableLineResult. It actually has 2. The first one is filled with the correct results when generating the TruthTableLine. The second one is filled with empty results at first. This will allow me to edit this list later when the user will fill the truth table, and then to compare the actual results and the answers provided.

You can find an example of a simplified version of the computed representation of a truth table in the Appendix 3.

After this point, I was able to instanciate a truth table from any correct logical formula, and also to access the results of any cell of this truth table.

The final step for the truth tables part was to represent it graphically.

5.2.3 Graphical Representation

For the graphical representation of the truth tables in my software, I chose to use a WPF control called Grid. A Grid is a kind of advanced table, which fits perfectly with my need, knowing the a truth table is actually a table. My only problem was that, each truth table is different, and its layout depends on the formula. This means that I had to dynamically generate my Grid.

Thus, I decided to implement a method in my TruthTable class, that returns me a Grid control dynamically generated from the formula. For this method, I had to keep in mind that a truth table can either be displayed with or without its answers. If I call this method in order to display a complete truth table, the Grid has to be non editable. If I call the method in order to ask a user to fill it, the lines of the truth table have to be editable with trues or falses.

So here are the steps for the generation of a truth table Grid:

In this method, a lot of layout related operations are present, but since the purpose is not to discuss the design of the truth table, I will just skip those parts. There is not need to detail how I have setup the sizes, or the alignment of the cells, etc…

The first operation of my method is to create a new Grid. Once this is done, I need to define the columns and rows that my Grid will contain. For the columns, there is one for each variable on the left, and then one for each operator in the formula (if no operators, there is only one column). For the formula "(p ∧ q) ⇒ p", there are 2 variables and 2 operators, this means my truth table needs 4 columns. 2 for the variables, and 2 for the results. Then, for the rows, I just need 1 row for the headers of the truth table, and 1 row for each TruthTableLine contained in my TruthTable. After this, my Grid knows how many rows and columns it is going to contain. The next step is to put content in each cell of my Grid.

I started by the headers, which is pretty simple. The first variable goes to the top of the first column, the second one goes to the top of the second column, etc… When the is no more variable, the next column top is attributed a colspan in order to take all the remaining space, and the entire formula is displayed. Here is at my header look like:

p	q	r	(p ∧ q) ⇒ r

Fig 9. Truth table headers

After this, the next step is to define the content of each row of the truth table Grid. For this, I create a loop that iterates through my TruthTableLine collection. I said earlier that a TruthTableLine instance contains a List of KeyValuePair that contains all the variables, and their value for the current line (true or false). So, another loop is started to iterate through my variables and display them in the correct column of my current line. Here is my result at this point:

p	q	r	(p ∧ q) ⇒ r
true	true	true	
true	true	false	
true	false	true	
true	false	false	
false	true	true	
false	true	false	
false	false	true	
false	false	false	

Fig 10. Truth table variables combination

And finally, the last contents to set are the answers of the truth table in the remaining cells. To do this, I started another loop right after the previous variables loop. This last loop iterates through each TruthTableLineResult contained in the current TruthTableLine. If the truth table has to display the correct answers, I simply have to

display the result stored in my TruthTableLineResult, and pass to the next one and to the next column. If the truth table is generated for a student to fill it, then I display a cell with an empty content. But, I attach an event to this cell that allows a user to click it. When such a cell is clicked, the content is being checked. If the content is "true" it's changed by "false" and opposite. If the content is empty, it's changed by "true". When setting the new content, I also need to modify my List of TruthTableLineResult that represents the user inputs.

Finally, when everything is set, I return the Grid. Here is the code needed to instanciate a truth table and display it:

```
<Grid x:Name="TruthTable" />
```

This is the tag in the XAML page.

```
TruthTable tt = new TruthTable("( p & q ) -> r");
this.TruthTable.Children.Add(tt.getTruthTableGrid());
```

This is the C# instruction in the code behind file linked to the XAML page.

And here is the final result:

p	q	r	(p ∧ q) ⇒ r	
true	true	true	true	true
true	true	false	true	false
true	false	true	false	true
true	false	false	false	true
false	true	true	false	true
false	true	false	false	true
false	false	true	false	true
false	false	false	false	true

Fig 11. Truth Table with results

And this is it for the truth table part. After this point, my TruthTable class was able represent a truth table of a formula and to display it in my GUI. This was a significant advancement in my project since it clearly represents the core of my application. The success of the TruthTable development made me realise that I was heading to the the right way.

After this, I first thought about starting the development of the "Classroom" part of the software, but I figured out that I still needed something else. Indeed, I was now able to represent a truth table, but not a syntax tree. You saw earlier that the tree generated by my lexical analysis (Fig 8) is quite different from the syntax trees used in propositional logic. I told myself that it would be a good idea to be able to also represent a syntax tree before going any further.

5.3 Syntax Trees

To display a correct syntax tree, I decided to create a WPF UserControl. A UserControl is a WPF Control, just like a Grid, or a Button. The difference is that the UserControl isn't a default Control, but a customised Control created by the developer.

After researches about WPF and trees, I found a good way to display a tree using a TreeContainer Control. So I created my UserControl which simply contains 1 TreeContainer. I also had to add a property to my UserControl: the formula. I managed to be able to pass the formula as an argument to the UserControl. When the UserControl is initialised, it generates a TruthTable from the formula, and a new method of TruthTable is called. This method is called "drawSyntaxTree.

"drawSyntaxTree" is a recursive method. Its aim is to iterate through the tree produced by the lexical analysis and display the needed nodes in a correct order. Here is how it operates:

Again, just like for the truth table Grid generation, I won't go through details for the design since it is not what matters here.

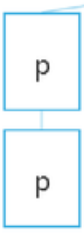
The first parameter of the method is the root of the TreeContainer Control, it won't be modified

at all during the execution of the function. The second parameter is the IParseTree and the third one is a TreeNode representing the last node that has been added to the TreeContainer. TreeNode is a class defined in TreeContainer, and represents a node of the tree.

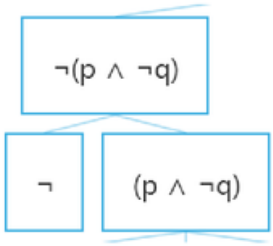
First, a new TreeNode is created. Its content will be set depending on the currently analysed IParseTree node.

If the current IParseTree node is null or doesn't have any child, the function ends. This is actually the recursion limit, when a node matches this condition, it means it is a terminal node (leaf), so there is nothing left after it.

If the current IParseTree has only one child, it implies that this child will be a terminal node, so I am setting the new TreeNode content to this terminal node content.

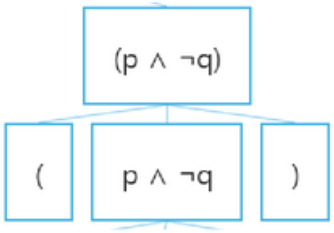


If the current IParseTree node has 2 children, it implies that the next child has a negation before it, so I simply need set the content of the new TreeNode to the second child and to display a negation node before the next.

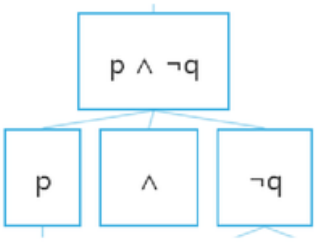


If the current IParseTree node has 3 children, it implies that this node represents either:

An expression between parenthesis.



Or an operator wrapped by 2 expressions.



Parenthesis are not displayed in syntax trees, so I skip them when encountering some during the parsing of the tree, until I find an operator (which is the second case). When an operator wrapped by 2 expressions is found, the new NodeTree content is set with this operator.

Once the new NodeTree has a content, I simply add the node to the TreeContainer. If the previous added TreeNode was null, it means I need to add a root to my TreeContainer (because nothing is inside yet, so it has no root). Otherwise, the new NodeTree is added as a child of the previous added TreeNode.

Once the new NodeTree is added, it is time to recursively call the drawSyntaxTree function with each child encountered. The 3rd parameter will not be null anymore, but will be the TreeNode we just added.

And here is the result when the recursion is over (with the same formula than Fig 8):

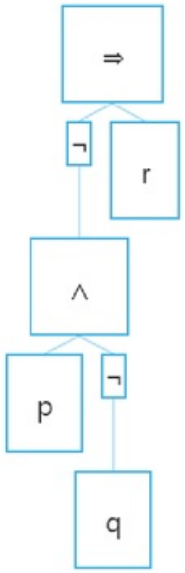


Fig 12. Generated Syntax Tree

At this point, I had everything I needed to start the development of the "Classroom" page.

5.4 CLASSROOM

The classroom part is designed to display course related information about truth tables. I selected the relevant parts from the Powerpoints and PDFs of the CO884 module. The challenge here, was to display all those information in a nice way.

I have set up my "Classroom" page earlier in the Main Layout part. What I needed there, was to divide my Classroom page into several parts, which I knew was possible with the MenuLinkGroups ModernUI control.

This control displays a menu that allows a user to navigate through different sections of the software, but I've shown you earlier that it was also possible to add sub pages (Fig 7). So I decided to put 4 sub pages in the Classroom instead of 1 only: Overview, Truth Tables, Operators and Syntax Trees.

With ModernUI, each sub page can use a different layout to display information. Here are the different types:

- Basic Layout
  The basic layout is a simple page without anything special.

Fig 13. Basic Layout

- Split Layout
  The split layout is a page split in 2 parts, both of them being scrollable.

Fig 14. Split Layout

- List Layout
  The list layout displays a list on the left side, and each element of the list is linked to a different content that will be displayed on the right side.

Fig 15. List Layout

- Tab Layout
  The tab layout shows a tab list on the top right corner, and each element of this tab list is linked to a different content that is displayed below.

Fig 16. Tab Layout

The first Classroom sub page is the Overview page. This part explains the basics about propositional logic and is split in 3 parts: “What is logic ?”, “Why logic ?” and “Logical Connective”. I used a Tab Layout with 3 tabs and attributed 1 content to each tab.

The second Classroom sub page is the Truth Tables page. It explains what truth tables are and gives examples of how to use them. Since this part is not split in sub parts, I simply used a Basic Layout and set its content. In this content, truth tables are displayed, so it was the first page where I was about to implement my TruthTable class. For each truth table displayed in the content, I inserted a Grid with the name of the truth table, and initialized it in the code behind during the initialization of Truth Tables page. In that way, I am making sure that each truth table in the software is displayed in the same way.

The next page is the Operators page. This page displays information about each logical operator. Those information contain a description, a truth table, some useful links, and some examples. I figured that the best layout to use for this page would be the List Layout. So each operator is listed in the left part, and its information are displayed on the right side when selected. Again, I had to instanciate TruthTable for each truth table displayed in the information.

And the last part is the Syntax Trees part. It contains information about Syntax Trees and what they mean. Just like the Truth Tables part, the Syntax Trees part contains only 1 content, so I used a Basic Layout. This time, instead of instanciating TruthTables, I had to instanciate Syntax Trees using the UserControl I created earlier.

For all the contents contained in Classroom, I tried to display them in the nicest possible way. I made sure to keep everything simple and clear. During the development, I thought about making those contents editable, but then I realised that it was a long feature to develop for something that probably will not be used more than once. By this, I mean that once the contents are set, they are not likely to change again.

For each page, except the Overview page, I implemented a button on the top right. This button can be used to open the current content in a seperated window. This will allow users to keep an eye on the information while doing something else in the software. When this button is clicked, it basically opens a similar brand new window, implementing only 1 page and 1 content. In the Operators part, if a user

clicks on a useful link, a similar seperated window is also opened with the page that has been clicked. It avoids the user to open his web browser and to stay focused on the software. I also planned to implement another button for printing. But I encountered some development issues when I tried it. This issue will be discussed later in this paper.

After I was done with all the design issues of the contents, I was able to move on to the next pages.

5.5 Practice & Exam

I decided to discuss the Practice and Exam pages in the same part here, since they are both pretty similar and operate in the same way.

The first common point between those 2 pages, is that they both need to generate logical formulas, and this, depending on a selected level of difficulty. Let me explain how I managed to do so.

5.5.1 Formula Generation

The first idea that crossed my mind for this task was to use ANTLR4 again. Knowing the rules I defined for the lexer and the parser, I tried to find a way to use them in a reversed way in order to randomly generate a logical expression while respecting the lexical rules previously defined. Unfortunatelly, I haven't find any way to word that out.

So here is what I have done. I figured that I might just start by the easy level, and then adjust my code to make it able to increase or descrese the difficulty of the generated formula. First, I defined a grammar. This grammar is a table of strings that contains only strings that can be contained in a logical formula:

E
~E
(E & E)
(E | E)
(E -> E)
(E <-> E)

This is quite similar than when I defined my lexer rules ealier. Then I defined some variables in a table of strings as well, I started with 2: P and Q.

When a formula has to be generated, it is first created as "E". "E" actually represent an expression (similar to the parser rules defined earlier). Then, I start a loop in which I replace 1 occurrence of "E" in the formula, by a string randomly picked from my grammar. Everytime the loop starts, the formula is being checked first. If the check says that enough variables and operators are present in the formula, the replacement of "E" occurences has to stop. This, is the first part of the formula generation. Here is an example of the several steps:

E	default string
(E & E)	"E" replaced by "(E & E)"
(E & ~E)	second "E" replaced by "~E"
((E | E) & ~E)	first "E" replaced by "(E | E)"
((E | E) & ~(E -> E))	last "E" replaced by "(E -> E)"

This process would be endless without a check of the formula before every loop. But this check, allows me to consider the level of difficulty. The function that checks the formula everytime is actually making sure that there are enough variables in the formula. To set up this limit, I am counting the number of present variables. If the number is lower than the number of variables in my grammar plus a random number picked between 0 and the number of variables in my grammar, then it needs more variables and the loop continues. A look at the code might be easier to understand:

```
Random rnd = new Random();
int nbE = Regex.Matches(expr, "E").Count;
if (nbE >= this.Variables.Length + rnd.Next(0, this.Variables.Length))
        return true;
return false;
```

As the level of difficulty increases, the number of variables defined in my grammar increases as well, which makes the formula longer and containing more operators.

Once this is done, the second part is the replacement of all the "E" occurences by variables that are defined in my grammar. Then I finally remove useless parenthesis (e.g. "(A & B)") and negations (e. g. "~~~A" or "~~~~A").

I finally encapsulated my formulas in a class, and created a GenerateFormula method that takes the level of difficulty in argument.

5.5.2 Practice

At first, the only sub page accessible in the Practice part is the settings panel. Each practice can have different parameters, and this setting panel is provided to the user in order to create those practices.

This settings panel asks the user several information:

- A name for the practice (just so several practices can be distinguished).
- How are generated the truth table ? From auto generated formulas, or from user inputs.
- The level of difficulty, in case the user chose auto generated formulas
- And the activation of 3 different kind of helps and tips: tooltips, truth tables, and syntax trees.

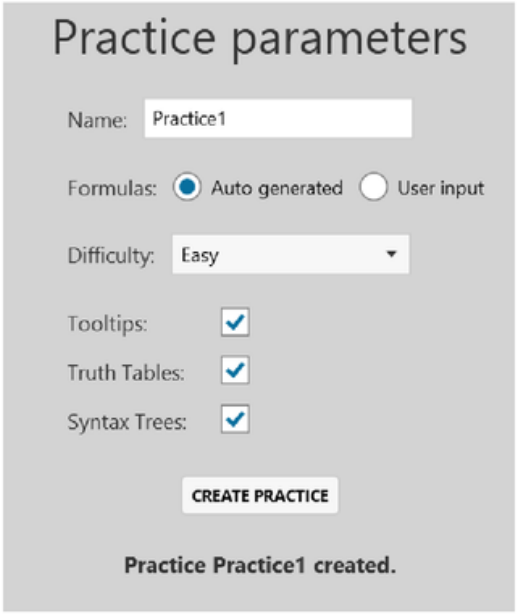


Fig 17. Practice Settings Panel

After those information are provided, the user is able to create its practice. When it is created, a new sub page is dynamically added.

For the practice page, I used a Split Layout. The left part called "Workspace" displays an empty truth truth table. The student has to fill this truth table, and can either validate or close the practice, or go to the next truth table (if validated the current one).

I said earlier that each TruthTableLine has 2 lists of TruthTableLineResult. One is for the results, the other one is for the student's answer. So when a student clicks a cell in the truth table to change its content to true or false, it is the second one that is modified. In that way, when the student clicks that validation button, the code checks if both lists are the same, if not, the differences are mistakes, this is how the cells that have to be highlighted are distinguished from others.

Fig 18. Corrected Truth Table

In the case the student chose to generate truth tables from his inputs, a additional step is displayed before each truth table is shown. A TextBox is displayed to the user, asking for him to type the formula he wants. When validated, the formula is being checked, and the truth table is generated if the formula is correct. Otherwise, the user is asked to type another formula.

Fig 19. Practice User Input

The right side of the layout is dedicated to the helps and tips enabled by the user in the settings panel. Here is a description for each of them:

- Truth Tables
  The Truth Tables help will display the truth table of each operator present in the analysed formula.
- Syntax Trees
  The Syntax Trees help will display the syntax tree of the analysed formula.
- Tooltips
  The tooltips help will display a tooltip on each cell of the truth table in order to show to which part of the formula is related this cell.

It is important to be noted that by "analysed formula", I don't mean the entire formula. When pointing a cell, the user determines which part of the formula he is working on. This is the analysed formula.

You see all helps and tips working together in the Appendix 4.

5.5.3 Exam

The Exam part was a bit simpler than the Practice part. The operation is quite similar except on certain points.

First, the Exam part also uses a settings panel just like the Practice part. It is a bit different but operates the same way.

Fig 20. Exam Settings Panel

The Exam part also uses the formula auto generation. It actually uses only this, since it is not possible for the user to input his own formula during an exam.

When the exam is created, the first thing done is to generate a list of truth tables depending on how many the user wants. Then, the start sceen is displayed to the user and notifies him that an exam is about to start.

The user is then in exam, he can not access any other part of the software until he is done. Each truth table is displayed alone, and the user can navigate through them with arrow buttons. For

this, I used a Basic Layout in which I'm displaying and desired part of the exam, and hidding the others. When the exam reaches the end, the user is notified again, and has the choice to go back to the truth tables, or validate all of them. Finally, the user moves to the correction page, where all truth tables are displayed with the user's answers, but also with the correct answers. A grade is attributed depending on how many truth tables are correct, and 2 buttons give the choice to restart a new exam, or exit the current exam.

A printing button was supposed to be implemented (just like for the Classroom pages), but then again, I encountered some issues I will discuss in the next part of this paper.

You can see the Appendix 5 for a screenshot of the correction page of an exam.

And this is how I reached the end of the development process. By the time, all my features were implemented, except a couple of them which put me into troubles. This is what I'm going to discuss in the next part.

6 Development Issues

The development process worked out quite well, and I haven't had to face significant difficulties and issues.

As I said earlier, I had troubles with the printing feature. In C#, it is possible to directly print the content of a Control, I was actually able to do so. The problem I faced was the pagination. Some content being to long for a single A4 page, I had to deal with pagination issues. I've been able to print my contents on several pages, but my issue was to avoid parts of my content being cut out. When my content was printed, if the end of the A4 page happened in the middle of one of my Controls (e.g. a truth table, or syntax tree), this content wasn't moved to the next page. Being unfamiliar with the printing tools contained in C#, I haven't been able to figure out a good way to print my contents, and with the deadline getting closer and closer, I decided to cancel the development of this feature, since it was not a high priority task.

Another secondary priority issue I encountered was the display of the whole formula at the top of a truth table. The truth tables being dynamically created depending on the formula, I wasn't able to find a way to correctly align each operator or variable with its dedicated column. What I thought about was to split my formula is several parts, and to play with the colspan of the formula row. But, unfortunately, I wasn't able to find a reliable way to do it.

And finally, the last issue I encountered was the formula generation. The feature is working correctly, but I am not satisfied by the way th formulas are generated. I think it would be great if the generation was bit more random, but it is always a bit delicate to deal with randomness when programming. Plus, the technique I used tends to generate a lot of parenthesis in the formulas. It doesn't make them incorrect, but it does make them less pleasant to look, and might be a bit confusing sometimes. I couldn't find another good way to generate the formulas, so I also decided to leave it as it was, since it was actually working correctly.

7 Conclusion

At the end of the development process, and after a period of tests, during which I tried to imagine all the possible scenarios that could happen during the usage of the software, I can say that it is able to answer the needs that has been defined. I have been able to research and find adapted tools in order to help me during the development. I tried to stick as much as possible to my guidelines so the specification I defined will be respected. The GUI is simple and user friendly, the truth tables and the syntax trees,

which are a high percentage part of the project, are operational and probably will not need to be modified. Except maybe for the alignement of the formula with the column of its truth table. The formula generation is operational as well, I regret that I wasn't able to find a more evolved way to do it, but the result is satisfying. The Practice and Exam pages are correctly implemented and also respect the previously defined specification.

The only specification that hasn't been respected is the printing feature. Being unfamiliar with it, I underestimated the time needed to implement it, and it took me too much time to get through it.

The code and a Doxygen documentation are provided with this dissertation. The documentation contains information about the every part of the code (classes, methods, attributes…). The software doesn't need to be installed. It is an executable named "TruthTablesToolPR.exe", which is located in a directory with all the dynamic-link libraries, and the needed files. The best way for it to look like installed on a computer is to copy the entire folder somewhere in Windows, and to create a shortcut that will redirect to the executable file.

8 References

[1] - Klement, K. C. 2005. Propositional Logic. Available at: http://www.iep.utm.edu/prop-log/

[2] - Koehler, K. R. 2012. Logical Operations and Truth Tables. [online] Available at: http://kias.dyndns.org/comath/21.html.

[3] - Visual C# Resources. 2014. Visual C# Resources. [ONLINE] Available at: http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx. [Accessed 13 August 2014].

[4] - .NET Downloads, Developer Resources & Case Studies | Microsoft .NET Framework. 2014. *.NET Downloads, Developer Resources & Case Studies | Microsoft .NET Framework*. [ONLINE] Available at: http://www.microsoft.com/net. [Accessed 13 August 2014].

[5] - Visual Studio - Microsoft Developer Tools. 2014. *Visual Studio - Microsoft Developer Tools*. [ONLINE] Available at: http://www.visualstudio.com/. [Accessed 13 August 2014].

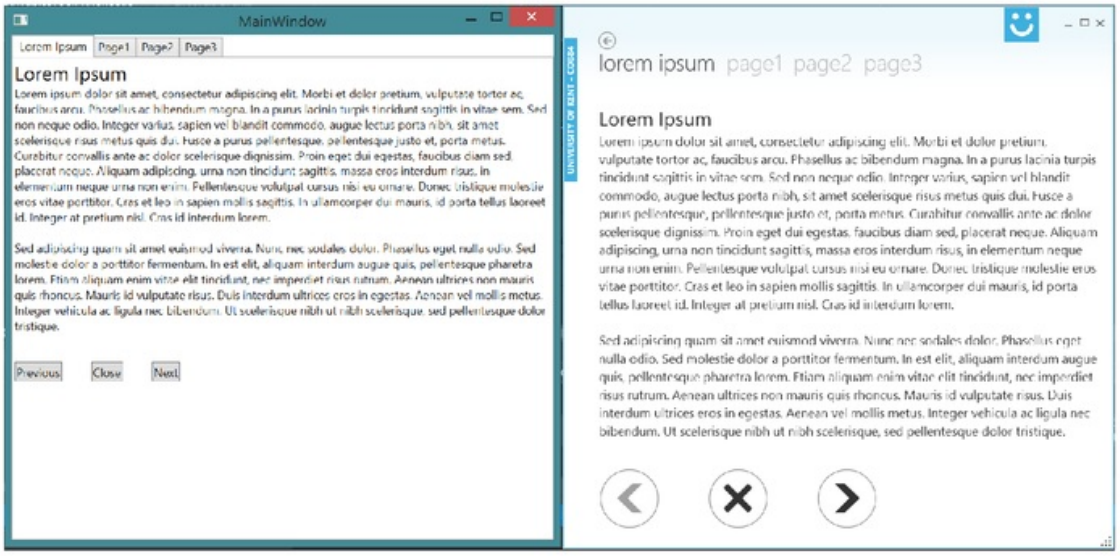
[6] - Introduction to WPF. 2014. Introduction to WPF. [ONLINE] Available at: http://msdn.microsoft.com/en-us/library/aa970268%28v=vs.110%29.aspx. [Accessed 13 August 2014].

[7] - Modern UI for WPF - Home. 2014. Modern UI for WPF - Home. [ONLINE] Available at: https://mui.codeplex.com/. [Accessed 13 August 2014].

[8] - Lexical analysis - Wikipedia, the free encyclopedia. 2014. Lexical analysis - Wikipedia, the free encyclopedia. [ONLINE] Available at: http://en.wikipedia.org/wiki/Lexical_analysis. [Accessed 13 August 2014].

[9] - ANTLR. 2014. ANTLR. [ONLINE] Available at: http://www.antlr.org/. [Accessed 13 August 2014].

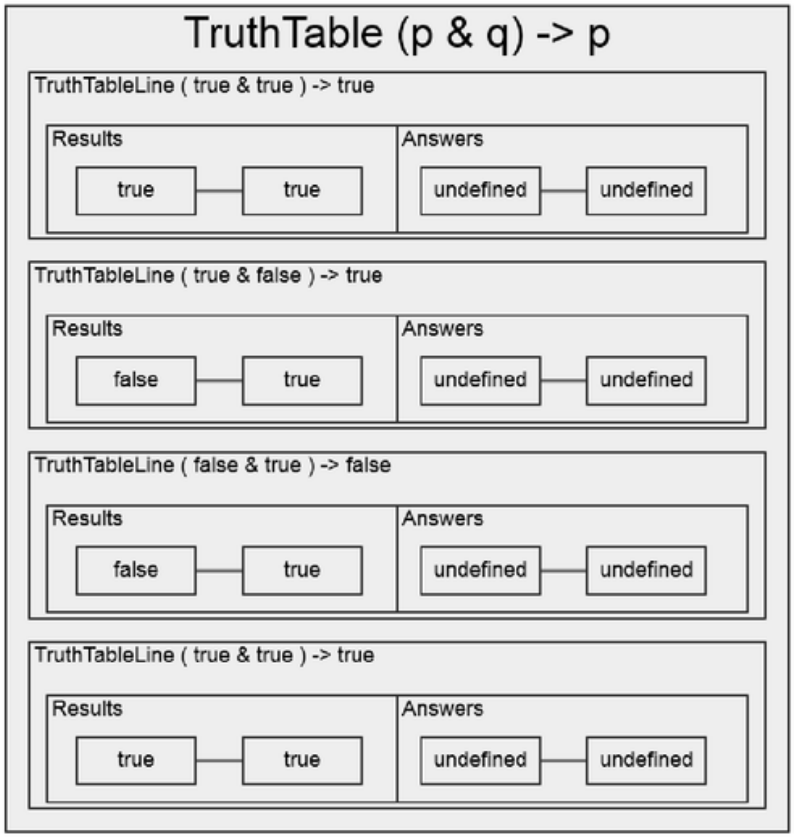
9 APPENDIX



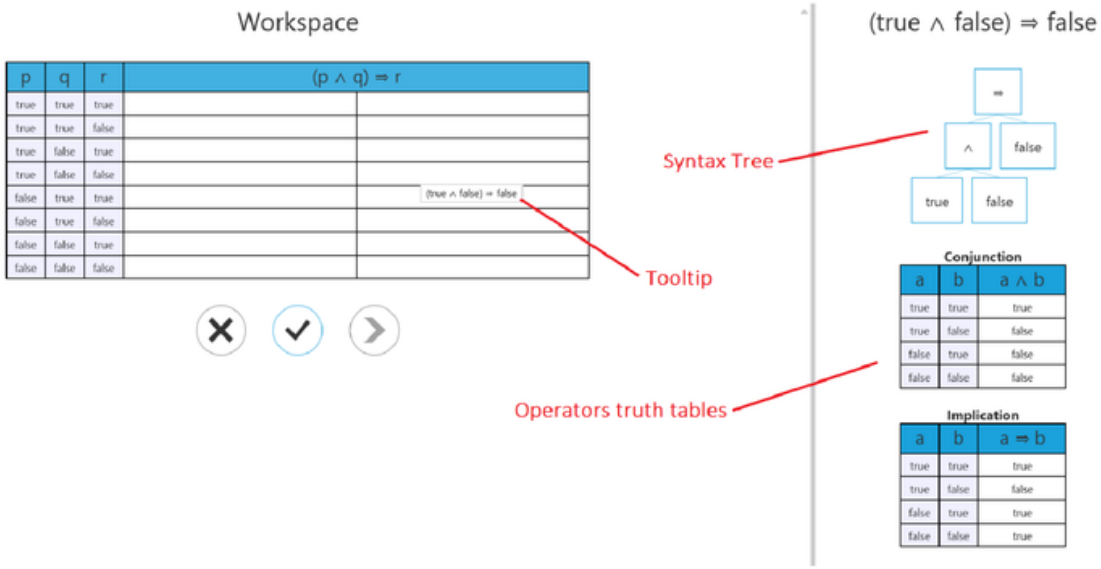
Appendix 1. WPF on the left, ModernUI WPF on the right

```
grammar LogicFormula;

/*
 * Parser Rules
 */

prog: expr+ ;

expr : '(' expr ')'          # parens
     | op='~' expr           # NOT
     | expr op='&' expr      # AND
     | expr op='|' expr      # OR
     | expr op='->' expr     # IMPLI
     | expr op='<->' expr    # BIIMPLI
     | BOOL                  # bool
     ;

/*
 * Lexer Rules
 */
BOOL : [0-1]|[a-zA-Z];
AND : '&';
OR : '|';
NOT: '~';
IMPLI: '->';
BIIMPLI: '<->';
WS
    :   (' ' | '\r' | '\n') -> channel(HIDDEN)
    ;
ErrorCharacter : . ;
```

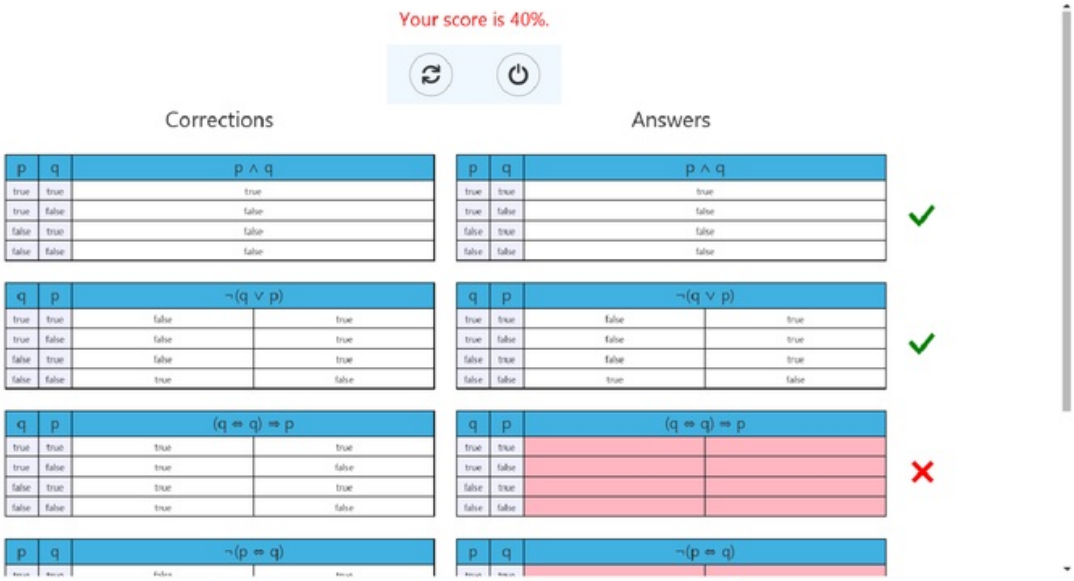
Appendix 2. Grammar file with parser and lexer rules



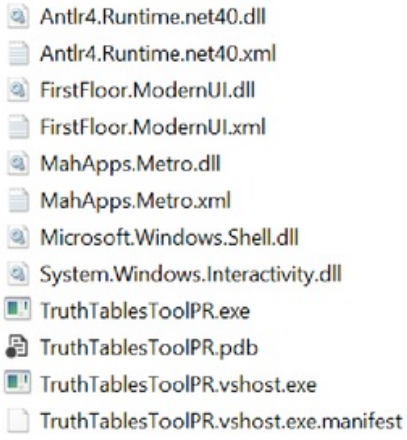
Appendix 3. Computed representation of a truth table



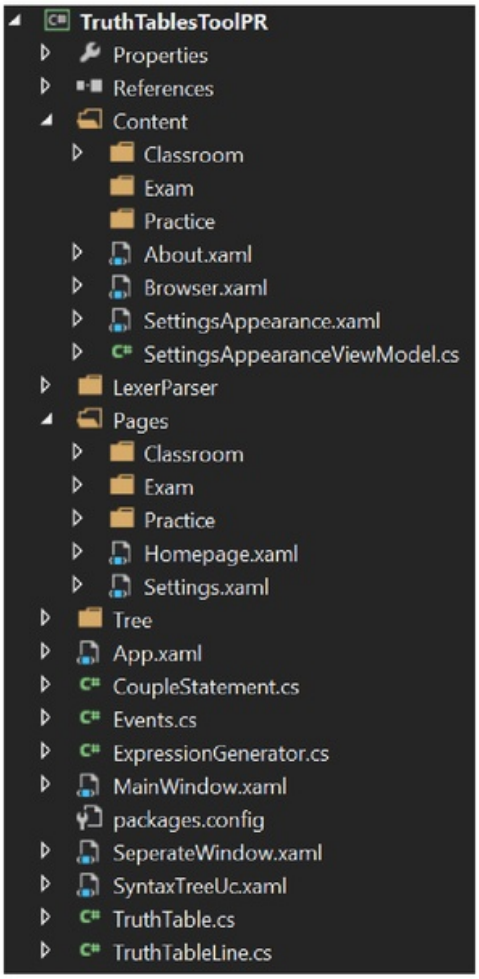
Appendix 4. Practice helps and tips



Appendix 5. Exam Correction Page



Appendix 6. Software's Folder



Appendix 7. Directory Structure

Project Research - An Education Tool for Writing Proofs in CO884

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26