Equivalences Tutor
BEng Final Report

Sahil Deepak Jain
Department of Computing
Imperial College London
Supervisor: Dr. Fariba Sadri

February 8, 2013

Abstract

Contents

1 Introduction **4**
- 1.1 Current Logic Tutorial Applications . . . 4
- 1.2 Requirements . . . 5

2 Background **6**
- 2.1 Propositional Equivalences . . . 6
  - 2.1.1 And Equivalences . . . 6
  - 2.1.2 Or Equivalences . . . 6
  - 2.1.3 Not Equivalences . . . 7
  - 2.1.4 Implies Equivalences . . . 7
  - 2.1.5 If and Only If Equivalences . . . 8
  - 2.1.6 De Morgan Equivalences . . . 8
  - 2.1.7 Distributibivity of And and Or . . . 9
- 2.2 Equivalence Examples . . . 9

3 Design and Implementation Overview **11**
- 3.1 Lexer and Parser . . . 11
- 3.2 Logic Tree Structure . . . 11
- 3.3 Graphical User Interface . . . 11

4 Project Plan **12**

5 Evaluation Plan **13**
- 5.1 User Feedback . . . 13
  - 5.1.1 Ease of Use . . . 13
  - 5.1.2 Difficulty . . . 13
- 5.2 Testing . . . 14

6 Appendix **15**
- 6.1 Lexer Grammar . . . 15
- 6.2 Parser Grammar . . . 16

1 Introduction

Logic refers to the study of different modes of reasoning conducted or assessed according to strict principles of validity. Due to Logic being one of the most fundamental aspects of Computer Science, it is taught to every student pursuing a Computing degree at Imperial College London. During the first term in university, every Computing student is taught the Logic course, which aims to provide the students with knowledge of the syntax and semantics of Propositional and Predicate logic. Students can apply this knowledge to complete equivalences and natural deduction proofs.

A logical system is made up of three things:

1. Syntax - this is the formal language specified to express different concepts.
2. Semantics - this is what provides meaning to the language.
3. Proof theory - this is a way of arguing in the language. This allows us to identify valid statements in the language.

In logic, two statements are logically equivalent if they contain the same logical content. Mendelson stated that "two statements are equivalent if they have the same truth value in every model." This can be illustrated in the following example:
Statement 1: *If Sahil is a final year student, then he has to do an individual project*

Statement 2: *If Sahil is not doing an individual project, then he is not a final year student*

As we can see, both statements have the same result in same models. When two logic statements are equivalent, they can be derived by each other, with the use of equivalences which we know to be true.

1.1 Current Logic Tutorial Applications

At the moment, there are a two programs which specifically built to support the Logic course at Imperial College London. These are:

1. Pandora - learning support tool designed to guide the construction of natural deduction proofs.
2. LOST - application which helps in the learning of logic semantics.

There is no current tutorial tool for logic equivalences.

1.2 Requirements

The aim of this project is to provide a web based equivalence tutorial tool with the following features:

- Input start and end states of a propositional or predicate equivalence, and perform the equivalence.
- Choose a difficulty of equivalence and practice an auto-generated level.
- Choose between a logic tree view and a logic expression view while completing a level.
- Give the student the ability to view previously completed levels.
- Give the student the ability to save uncompleted levels, which they can complete at a later time.
- Automatically suggest a higher level to the student if they are progressing well.

2 Background

2.1 Propositional Equivalences

The first thing I had to look at were the logic equivalences which the students are taught in their respective logic courses. These equivalences are described in the next few subsections.

2.1.1 And Equivalences

1. $A \wedge B \equiv B \wedge A$ *(Commutativity of And)*

A	B	$A \wedge B$	$B \wedge A$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

2. $A \wedge A \equiv A$ *(Idempotence of And)*

A	$A \wedge A$
0	0
0	0
1	1
1	1

3. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ *(Associaticity of And)*

A	B	C	$(A \wedge B) \wedge C$	$A \wedge (B \wedge C)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

2.1.2 Or Equivalences

1. $A \vee B \equiv B \vee A$ *(Commutativity of Or)*

A	B	A ∨ B	B ∨ A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

2. $A \vee A \equiv A$ *(Idempotence of Or)*

A	A ∨ A
0	0
0	0
1	1
1	1

3. $(A \vee B) \vee C \equiv A \vee (B \vee C)$ *(Associaticity of Or)*

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

2.1.3 Not Equivalences

1. $A \equiv \neg\neg A$

A	¬A	¬¬A
0	1	0
1	0	1

2.1.4 Implies Equivalences

1. $A \rightarrow B \equiv \neg A \vee B$

A	B	¬A	A → B	¬A ∨ B
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

2. $A \rightarrow B \equiv \neg(A \wedge \neg B)$

A	B	$\neg B$	$A \wedge \neg B$	$A \rightarrow B$	$\neg(A \wedge \neg B)$
0	0	1	0	1	1
0	1	0	0	1	1
1	0	1	1	0	0
1	1	0	0	1	1

3. $\neg(A \rightarrow B) \equiv A \wedge \neg B$

A	B	$\neg B$	$A \rightarrow B$	$\neg(A \rightarrow B)$	$A \wedge \neg B$
0	0	1	1	0	0
0	1	0	1	0	0
1	0	1	0	1	1
1	1	0	1	0	0

2.1.5 If and Only If Equivalences

1. $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

A	B	$A \rightarrow B$	$B \rightarrow A$	$A \leftrightarrow B$	$(A \rightarrow B) \wedge (B \rightarrow A)$
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	1	1	1	1

2. $A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

A	B	$A \wedge B$	$\neg A$	$\neg B$	$A \leftrightarrow B$	$(A \wedge B) \vee (\neg A \wedge \neg B)$
0	0	0	1	1	1	1
0	1	0	1	0	0	0
1	0	0	0	1	0	0
1	1	1	0	0	1	1

2.1.6 De Morgan Equivalences

1. $\neg(A \wedge B) \equiv \neg A \vee \neg B$

A	B	$\neg(A \wedge B)$	$\neg A \vee \neg B$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

2. $\neg(A \vee B) \equiv \neg A \wedge \neg B$

A	B	$\neg(A \vee B)$	$\neg A \wedge \neg B$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

2.1.7 Distributibivity of And and Or

1. $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

A	B	C	$A \wedge (B \vee C)$	$(A \wedge B) \vee (A \wedge C)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

2. $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

A	B	C	$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

2.2 Equivalence Examples

1. $\neg(p \rightarrow q) \equiv p \wedge \neg q$

   (a) $\neg(p \rightarrow q)$

   (b) $\neg(\neg p \vee q)$ — a, Implication rule

   (c) $\neg\neg p \wedge \neg q$ — b, De Morgan's law

   (d) $p \wedge \neg q$ — c, Double Negation rule

2. $\neg((p \wedge q) \rightarrow r) \equiv (p \wedge q) \wedge \neg r$

   (a) $\neg((p \wedge q) \rightarrow r)$

   (b) $\neg(\neg(p \wedge q) \vee r)$ — a, Implication rule

   (c) $\neg\neg(p \wedge q) \wedge \neg\, r$ — b, De Morgan's law

   (d) $(p \wedge q) \wedge \neg r$ — c, Double Negation rule

3. $\neg(p \vee \neg q) \vee (\neg p \wedge \neg q) \equiv \neg p$

   (a) $\neg(p \vee \neg q) \vee (\neg p \wedge \neg q)$

   (b) $\neg(p \vee \neg q) \vee \neg(p \wedge q)$ — a, De Morgan's law

   (c) $\neg[(p \vee \neg q) \wedge (p \wedge q)]$ — b, De Morgan's law

   (d) $\neg[p \vee (\neg q \wedge q)]$ — c, Distributivity

   (e) $\neg(p \vee \bot)$ — d, And rule

   (f) $\neg p$ — e, Or rule

3 Design and Implementation Overview

3.1 Lexer and Parser

As we have had previous experiences with writing lexers and parsers, this was not a hard part of the development process. With the use of ANTLR, which is a lexer and parser generator, I have achieved a fully functional parser for propositional logic, which can easily be extended to account for predicate logic in future development.

3.2 Logic Tree Structure

To generate the logic tree, I chose not to use the ANTLR generated Abstract Syntax Trees. Instead, I wrote a tree structure of my own. The parser grammar can be seen in the appendix, which would help with the understanding of the tree structure as well.

3.3 Graphical User Interface

I am now starting to develop the GUI for the application. As the main implementation of the application is in Java, I have chosen to use the Swing library to develop the GUI.

4 Project Plan

As we are building a completely new application, we have to start from the very basics of what to do. The inital stages of the project consisted of the following:

1. Meeting up with project supervisor to discuss what she would like as an end result of the project.
2. Reading through equivalence notes from previous years.
3. Talk to a few first year students to know where they found problems with equivalences.
4. Research about the best way to model logic expressions so they could be easily manipulated in future development.

Once these steps had been carried out, I had a rough idea of where to begin the development aspect of the application. After thorough research, I had realised that the best way to model the logic expressions was to use a tree structure, which was suggested by Fariba as well. To create the tree structure, I chose to use ANTLR. This stage consisted of:

1. Writing the Lexer rules.
2. Writing the Parser rules.
3. Research more about the Abstract Syntax Tree (AST) returned by ANTLR.

After this further research, I realised that using the ANTLR generated ASTs would not be the most convenient tree structure to use in the long run, so I decided to create a tree structure of my own.

Future steps include (in order of implementation):

1. Designing the GUI
2. Developing the GUI
3. Writing the equivalence rules which alter the tree structure
4. Writing the logic expression generator
5. Writing the progress tracker
6. Extending the application to support predicate equivalences

5 Evaluation Plan

5.1 User Feedback

As this is a tutorial tool, the best way to evaluate the application is to see if the people who use it find it useful in learning equivalences. There are a number of aspects which would need to be evaluated. These are discussed below.

5.1.1 Ease of Use

For a tutorial tool, or any application, to be a good application, it has to be user friendly. When talking about this application specifically, we have to consider many different areas when talking about ease of use:

1. Inputting start and end states for an equivalence - this would be the first step for a user to complete an equivalence, unless they are using the auto-generated equivalences. Input of logic expressions should be simple and fast. If this aspect of the tutorial tool is not easy to use, the user would not want to waste time on the application as it would be too time consuming.

2. Solving equivalences - this part is the main feature of the application. This aspect of the application should be very simple and not require many clicks or lots of typing, and should provide an easy way of undoing rules that the user might have applied. Both the tree method and the expression method of solving equivalences should be very easy and fast to use.

3. Saving and opening uncompleted equivalences - this feature of the tutorial tool would be essential for a user who might have a specific equivalence too hard to complete at a given point, so they can return to it in the future when they are more comfortable with equivalences. It should be easy to see the start and end states of the uncompleted levels, and the users should be allowed to sort it by date or difficulty if need be.

4. Generating new levels - if the user does not have any equivalences to enter themselves, they can generate levels of their chosen level, or generate a level which the application thinks is suitable. This again, should be done within a few clicks of the mouse, as the user would not want to waste time on it.

5. Viewing Progress Report - the user can keep track of their progress with equivalences. This should be shown with easy to read and understand statistics of the user's progress.

5.1.2 Difficulty

Every tutorial tool needs to give the user tougher challenges as they progress. In this application, we keep track of certain statistics of each user, such as the number of completed levels, number of incomplete levels, average steps needed to complete a level, number of

easy/medium/hard levels completed, etc. As there is a level generator, we need to keep track of these statistics, and after analysing the data, the application would generate a suitable level for the user. To test this aspect of the application, we would have a few users who would complete generated levels, and with time, they would see the levels getting harder and harder. After a certain time, we would ask the users if they felt that the levels were getting harder and if they felt under-challenged or over-challenged. This would provide the basis of evaluation of the level generation aspect of the tutorial tool.

5.2 Testing

Thorough testing is essential to any application. Many tests will be written to check that no completed part of the application is broken once new parts are added, making sure that every feature of the application works.

6 Appendix

6.1 Lexer Grammar

```
lexer grammar LogicLexer;

options {
  language = Java;
}

@header {
  package eqtutor;
}

WHITESPACE: ( '\t' | ' ' | '\r' | '\n' | '\u000C' )+ { $channel = HIDDEN; };

AND : '&';
OR : '|';
IFTHEN : '->';
IFF  : '<>';
NOT  : '!';

LPAREN  : '(';
RPAREN  : ')';

ID : ('A'..'Z'|'a'..'z') ('A'..'Z'|'a'..'z'|'_')*;
```

6.2 Parser Grammar

```
parser grammar LogicParser;

options {
  tokenVocab = LogicLexer;
}

@header {
  package eqtutor;

  import java.util.LinkedList;
  import java.util.List;
  import AST.*;
}

@members {
  private boolean hasFoundError = false;

  public void displayRecognitionError(String[] tokenNames, RecognitionException e) {
    hasFoundError = true;
  }

  public boolean hasFoundError() {
    return this.hasFoundError;
  }
}

program returns [AST tree]
  : e = iffexpr {$tree = new AST(new ASTProgramNode($e.node));} EOF
  ;

iffexpr returns [ASTPropositionalNode node]
  : ifthen = ifexpr {$node = $ifthen.node;}
  (IFF iff = iffexpr {$node = new ASTIffNode($ifthen.node, $iff.node);})*
  ;

ifexpr returns [ASTPropositionalNode node]
  : or = orexpr {$node = $or.node;}
(IFTHEN ifthen = ifexpr {$node = new ASTIfThenNode($or.node, $ifthen.node);})*
  ;

orexpr returns [ASTPropositionalNode node]
```

```
  : and = andexpr {$node = $and.node;}
(OR or = orexpr {$node = new ASTOrNode($and.node, $or.node);})*
  ;

andexpr returns [ASTPropositionalNode node]
  : not = notexpr {$node = $not.node;}
(AND and = andexpr {$node = new ASTAndNode($not.node, $and.node);})*
  ;

notexpr returns [ASTPropositionalNode node]
  : NOT not = notexpr {$node = new ASTNotNode($not.node);}
  | id = identifier {$node = $id.node;}
  ;

identifier returns [ASTPropositionalNode node]
  : ID {$node = new ASTIdentifierNode($ID.text);}
  | LPAREN iffexpr RPAREN {$node = $iffexpr.node;}
  ;
```