Equivalences Tutor
BEng Final Report

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

February 7, 2013

Abstract

Contents

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.

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 application which lets the students learn and practice equivalences.

1.1 Report Structure

2 Background

2.1 Propositional Equivalences

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	$A \vee B$	$B \vee A$
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 ∨ B) ∨ C ≡ A ∨ (B ∨ 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 ≡ ¬¬A

A	¬A	¬¬A
0	1	0
1	0	1

2.1.4 Implies Equivalences

1. A → B ≡ ¬A ∨ 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 → B ≡ ¬(A ∧ ¬B)

A	B	¬B	A ∧ ¬B	A → B	¬(A ∧ ¬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. ¬(A → B) ≡ A ∧ ¬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

3 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:

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.

4 Evaluation Plan

5 Appendix

5.1 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'|'_')*;
```

5.2 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.n
  ;

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

orexpr returns [ASTPropositionalNode node]
  : and = andexpr {$node = $and.node;} (OR or = orexpr {$node = new ASTOrNode($and.node, $or.nod
  ;

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

```
  ;

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;}
  ;
```