Imperial College London

Department of Computing

Individual Project

Equivalences Tutor

Author:
Sahil Jain

Supervisor:
Fariba Sadri

Second marker:
Ian Hodkinson

June 13, 2013

Abstract

Acknowledgements

I would like to thank Dr. Fariba Sadri for her continuous support and guidance throughout the course of this project. I would also like to thank Dr. Ian Hodkinson for his value feedback on my interim report.

Above all, I would like to thank my family for being there for me and giving me constant support through my education.

Contents

Chapter 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.

Chapter 2

Background

This section summarises the necessary background knowledge for understanding this report. It also introduces the concepts of Logic Equivalences.

2.1 Propositional Logic

Before we can define what Propositional Logic is, we have to define what a proposition is. Essentially, a proposition is a statement which is either true or false. This then leads to defining Propositonal Logic as a branch of symbolic logic dealing with propositions as units and with their combinations and the connectives that relate them. [1]

2.1.1 Syntax

This is the formal language used in propositional logic. There are three different parts which make up the syntax of propositional logic; atoms, connectives and punctuation.

A propositional atom, or propositional variable, is a symbol which has a truth value, either true or false. Usually, atoms are denoted by letters p, q, r, s, etc.

Connectives are boolean operations which are applied to atoms. There are a total of seven connectives:

- And - written as $\wedge$. Takes two arguments.
- Or - written as $\vee$. Takes two arguments.
- Not - written as $\neg$. Takes one argument.
- If then - written as $\rightarrow$. Takes two arguments.
- If and only if - written as $\leftrightarrow$. Takes two arguments.
- Truth - written as $\top$. Takes zero arguments.
- Falsity - written as $\bot$. Takes zero arguments.

2.1.2 Semantics

This is the meaning of a logic formula. As each atom has a truth value, there can be several combinations of truth values for each atom. To be more specific, if there are n atoms in a logic formula, then there are a total of 2^n combinations of truth values. Each combination is known as a situation.

Each connective has a different meaning. These meanings can be shown using a truth table and using the atoms p and q.

And

p	q	p ∧ q
0	0	0
0	1	0
1	0	0
1	1	1

As we can see, p ∧ q is only true when both p and q are true. Otherwise, p ∧ q is always false.

Or

p	q	p ∨ q
0	0	0
0	1	1
1	0	1
1	1	1

As we can see, p ∨ q is true when either p or q is true. When both p and q are false, p ∨ q is false as well.

Not

p	¬p
0	1
1	0

The truth value of ¬p is always the opposite of p.

If Then

p	q	p → q
0	0	1
0	1	1
1	0	0
1	1	1

p → q is true if p is false or/and if q is true. This means that there is only one situation where p → q is false, i.e p is true and q is false.

If and Only If

p	q	p $\leftrightarrow$ q
0	0	1
0	1	0
1	0	0
1	1	1

p $\leftrightarrow$ q is true if p and q both have the same truth value.

Truth and Falsity

Truth is an atom which is always true, and Falsity is an atom which is always false.

2.2 Equivalences

As we introduced the example of "*If Sahil is a final year student, then he has to do an individual project*" being the same as "*If Sahil is not doing an individual project, then he is not a final year student*", all logic formulae are equivalent to other logic formulae.

We can show this by converting the above example into propositional logic. If we let the atom p denote "*Sahil is a final year student*" and let the atom q denote "*Sahil is doing an individual project*", then the statement "*If Sahil is a final year student, then he has to do an individual project*" becomes p $\rightarrow$ q and the statement "*If Sahil is not doing an individual project, then he is not a final year student*" becomes $\neg$q $\rightarrow$ $\neg$p.

If we construct the truth tables of both of these formulae, we get:

p	q	p $\rightarrow$ q
0	0	1
0	1	1
1	0	0
1	1	1

p	q	$\neg$q $\rightarrow$ $\neg$p
0	0	1
0	1	1
1	0	0
1	1	1

As we can see, both of the truth tables are the same. This means that in any given situation, both of the logic formulae have the same truth value, meaning that they are equivalent.

There are several defined equivalences for each operator.

2.2.1 And Equivalences

1. $A \wedge B \equiv B \wedge A$ *(Commutativity of And)*
2. $A \wedge A \equiv A$ *(Idempotence of And)*
3. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ *(Associaticity of And)*

2.2.2 Or Equivalences

1. $A \vee B \equiv B \vee A$ *(Commutativity of Or)*
2. $A \vee A \equiv A$ *(Idempotence of Or)*
3. $(A \vee B) \vee C \equiv A \vee (B \vee C)$ *(Associaticity of Or)*

2.2.3 Not Equivalences

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

2.2.4 Implies Equivalences

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

2.2.5 If and Only If Equivalences

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

2.2.6 De Morgan Equivalences

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

2.2.7 Distributibivity of And and Or

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

2.3 Examples of Equivalence Problems

2.3.1 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

2.4 Current Logic Tutorial Applications

In the first year Logic course, several different concepts are taught to the students. Some of the concepts have applications installed on the lab machines which are produced to help students understand these concepts better. Firstly, there is an application called *Pandora*, which is a learning support tool designed to guide the construction of natural deduction proofs. The second application the department provides is *LOST*, which helps students understand logic semantics better.

Equivalences is a large part of the Logic course, but currently, there is no tutorial application which the students can use to understand how to apply equivalences better, or which provides a user friendly interface for the students to use.

Chapter 3

Implementation and Features

3.1 Modelling Logic Formulae

The first part that had to be implemented was modelling logic formulae so that they could be manipulated when applying equivalences. In the first year Logic course taught by Ian Hodkinson, the students are taught the Logic formation tree.

The example given in the first year slides is:

Using this formation tree, we can model logical formulae in a tree structure in Java by creating our own data structure.

3.1.1 Lexer

The first step in modelling the logical formulae into a tree structure is lexical analysis. This is the process of converting a sequence of characters into a sequence of tokens which can be parsed in the future.

As writing a lexer from scratch would have been very time consuming and tedious, I researched about the tools which had been developed for generating lexers after inputting the lexical grammar. The best tool which I came across was ANTLR, as it provides a very easy to use debugger where you can load the generated code and step through it.

Lexical 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'|'_')*;
```

3.1.2 Parser

Once the formulae has been tokenised by the lexer, it then has to be parsed. Parsing is the process where the tokens are used to build a data structure, which is usually a hierarchical structure. This data structure gives a structural representation of the input.

ANTLR provides both lexer and parser generators. Due to having the ability of doing both of these steps using one tool which I had become familiar with, I decided to use the ANTLR parser generator.

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;}
  ;
```

3.2 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is the tree representation of the structure of the logic expression. Each node of the tree represents either a connective or an identifier of the logic expression.

ANTLR provided an implementation of the AST as well, but I decided not to use it since manipulation of the trees provided was tough and confusing. Due to this, I created my own data structure to store the logic expression.

Once the tree is parsed, we obtain an object of type AST. The tree contains a node of type ASTProgramNode, which is the highest level of the tree. The ASTProgramNode node contains an ASTPropositionalNode. ASTPropositionalNode is an abstract class which contains the abstract methods for all of the propositional operators. The ASTPropositionalNode class is shown below:

Listing 3.1: Methods held in a Propositional Node

```
public abstract class ASTPropositionalNode extends ASTNode {

        /*--Checks if the current node is equal to the input node--*/
        public abstract boolean equals(ASTPropositionalNode node);

        /*--Returns the string representation of the node--*/
        public abstract String toString();

        /*--Returns the string representation of the node which can be parsed--*/
        public abstract String toParserString();

        /*--Returns a tree map of the identifiers in the node--*/
        public abstract TreeMap<String, Integer> numIdentifiers(TreeMap<String, Integer>
            identifiers);

        /*--Returns the value of the node, e.g TRUTH and TRUTH = TRUTH--*/
        public abstract int value(TreeMap<String, Integer> id);

        /*--Creates the JPanel for the node which is used to solve equivalences--*/
        public abstract JPanel createJPanel(NewPersonalEquivalenceListener l, boolean side);

        /*--Copy constructor for the node--*/
        public abstract ASTPropositionalNode copy();

}
```

3.3 Equivalences Data Structure

The equivalences the user perform have to be used in several different features. To make it easy to use and manipulate the equivalences, a data structure in which the equivalences could be stored had to be decided upon.

Before I could decide on which data structure to use, I had to delve into the details of what I had to store in order to complete equivalences, and how exactly an equivalence was done. In an equivalence, you only ever have to use the previous formula to obtain the next equivalent formula. This is illustrated by the example below:

1. $\neg(p \rightarrow q)$
2. $\neg(\neg(p \wedge \neg q))$ — 1, Implication rule
3. $\neg(\neg p \vee q)$ — 1, Implication rule
4. $\neg\neg p \wedge \neg q$ — 3, De Morgan's law
5. $p \wedge \neg q$ — 4, Double Negation rule

As we can see, line 2 is unneccesary in this equivalence, although true. We also notice that line 4 is obtained using line 3, and line 5 from line 4.

Due to the presence of this feature, I considered the use of linked lists to store equivalences. A linked list is essentially a list implemented by each item having a link to the next item [3]. In the equivalences case, this means that every formula would have a link to a logically equivalent formula which has been derived by applying a single equivalence.

Before I could create the Linked List class, I had to create the structure of each node. Below are the fields stored in a node:

Listing 3.2: Fields held in a list node

```
/*--line number of the formula--*/
private int lineNumber;

/*--the Abstract Syntax Tree of the formula--*/
private AST tree;

/*--the next node in the equivalence--*/
private EquivalenceLinkNode next;
```

Once the node class was written, the list class could be implemented. Below are the fields and methods in the list class:

Listing 3.3: Methods and fields in the Linked List class

```
public class EquivalenceLinkedList {
```

```
    /*--the first node of the list--*/
    private EquivalenceLinkNode head;

    /*--the number of nodes in the list--*/
    private int size;

    /*--constructor--*/
    public EquivalenceLinkedList();

    /*--returns true if the list is empty, false otherwise--*/
    public boolean isEmpty();

    /*--adds a node to the end of the list--*/
    public void add(EquivalenceLinkNode node);

    /*--removes the last element of the list--*/
    public void removeLast();

    /*--returns the first element of the list--*/
    public EquivalenceLinkNode getHead();

    /*--sets the parameter node as the head of the list--*/
    public void setHead(EquivalenceLinkNode head);

    /*--returns the size of the list--*/
    public int getSize();

    /*--sets the parameter size as the size of the list--*/
    public void setSize(int size);

    /*--returns the last node of the list--*/
    public EquivalenceLinkNode getLast()

}
```

3.4 Implementation of Equivalences

As formulae are stored as a tree, to apply an equivalence to a formula, the tree structure has to be altered. As nodes had to be altered, the tree had to be traversed, which lead to a few problems. In the method which applied equivalences, which is discussed later, the node to be changed and the new node were passed in as parameters. The node was then altered. A problem arose due to the fact that there could be several

instances of the node in the tree. For example, consider the formula $(p \wedge p) \rightarrow (q \rightarrow (p \wedge p))$. Suppose we wanted to apply to equivalence $(p \wedge p) \equiv p$ to the first instance of $p \wedge p$. The method would initially return the tree with formula $p \rightarrow (q \rightarrow p)$. To overcome this problem, a key had to be designated to each node, so when an equivalence was applied, the key of the node would be a parameter in the method. This allows for the equivalence to be correctly applied to the formula.

For each equivalence, there is a different method. An example method is shown below. The example is for the equivalence $p \wedge q = \neg(\neg p \vee \neg q)$.

Listing 3.4: Example of an equivalence method

```
public AST deMorgan() {
  try {
    AST tree = getTree();
    int key = getKey();

    /*--finds the node with the given key in the tree--*/
    ASTNode node = find(tree.getRoot(), key);

    /*--only applies the equivalence if the found node is an instance
        of an And node--*/
    if(node instanceof ASTAndNode) {
      ASTAndNode andNode = (ASTAndNode) node;
      ASTPropositionalNode left = andNode.getLeft();
      ASTPropositionalNode right = andNode.getRight();

      /*--creates a not node of the left child of the and node--*/
      ASTNotNode notLeft = new ASTNotNode(tree.getKey(), left);
      tree.setKey(tree.getKey() + 1);

      /*--creates a not node of the right child of the and node--*/
      ASTNotNode notRight = new ASTNotNode(tree.getKey(), right);
      tree.setKey(tree.getKey() + 1);

      /*--creates the or node needed in the equivalence--*/
      ASTOrNode orNode = new ASTOrNode(tree.getKey(), notLeft, notRight);
      tree.setKey(tree.getKey() + 1);

      /*--creates the not node which is equivalence to the initial and node--*/
      ASTNotNode notNode = new ASTNotNode(tree.getKey(), orNode);
      tree.setKey(tree.getKey() + 1);

      /*--replaces the and node with the not node--*/
      ASTPropositionalNode p = replace(tree.getRoot().getLeaf(), notNode, key);
      ASTProgramNode program = tree.getRoot();
```

```
      program.setLeaf(p);

      /*--creates and returns the new logically equivalent tree--*/
      AST t = new AST(tree.getKey(), program);
      return t;
    }
  }
  catch(Exception e) {
    return null;
  }
  return null;
}
```

The same structure is used for other equivalence methods.

Clearly, the find and replace methods are essential to this method. The find method is shown in 3.5, and the replace method is shown in 3.6

Listing 3.5: Find method for equivalences

```
public static ASTNode find(ASTNode node, int key) {
  if(node.getKey() == key) {
    return node;
  }
  if(node instanceof ASTPropositionalBinaryNode) {
    ASTPropositionalBinaryNode binary = (ASTPropositionalBinaryNode) node;
    ASTNode left = find(binary.getLeft(), key);
    ASTNode right = find(binary.getRight(), key);
    if(left != null) {
      return left;
    }
    if(right != null) {
      return right;
    }
  }
  if(node instanceof ASTPropositionalUnaryNode) {
    ASTPropositionalUnaryNode unary = (ASTPropositionalUnaryNode) node;
    ASTNode ret = find(unary.getLeaf(), key);
    if(ret != null) {
      return ret;
    }
  }
  return null;
}
```

Listing 3.6: Replace method for equivalences

```
public static ASTPropositionalNode replace(ASTPropositionalNode prop, ASTNode node, int key) {
  if(prop.getKey() == key) {
    return (ASTPropositionalNode) node;
  }
  if(prop instanceof ASTPropositionalBinaryNode) {
    ASTPropositionalBinaryNode binary = (ASTPropositionalBinaryNode) prop;
    if(find(binary.getLeft(), key) != null) {
      binary.setLeft(replace(((ASTPropositionalBinaryNode) prop).getLeft(), node, key));
    }
    if(find(((ASTPropositionalBinaryNode) prop).getRight(), key) != null) {
      binary.setRight(replace(((ASTPropositionalBinaryNode) prop).getRight(), node, key));
    }
    return binary;
  }
  if(prop instanceof ASTPropositionalUnaryNode) {
    ASTPropositionalUnaryNode unary = (ASTPropositionalUnaryNode) prop;
    unary.setLeaf(replace(unary.getLeaf(), node, key));
    return unary;
  }
  return null;
}
```

3.5 Graphical User Interface

3.6 Logic Expression Generator

3.7 Equivalence Solvers

Two different interfaces are implemented which the user can use to solve equivalences. Both of the modes have several features in common. Firstly, regardless of what mode the user selects to perform an equivalence, the application is divided into two different distinct panels. Once the start and end state have been entered, the start state will be visible on the left panel, and the end state will be available on the right panel. The advantage of having the interface laid out like this is that it allows the user to work from the start state and from the end state, apply equivalences, and eventually meet in the middle. Once the final equivalences on each side match, the equivalence is complete. When performing these equivalences, the user can switch modes as they please.

The user also has the ability to save an equivalence which they are doing, either to file or on a database. The user can undo their last equivalences from both panels as well.

3.7.1 Checking if two formulae are logically equivalent

When a user starts a new equivalence, he has to first put input the start state and the end state. These two formulae are then checked to be equivalent or not, as the user should not be allowed to perform an equivalence problem if the start and end state are not equivalent.

If we were doing this on paper, we would just construct the truth table for the two formulae, and then compare them to be equal. The same idea is programmed in the AST class. As the AST only contains the structure, we have to assign each identifier with a truth value, so that every situation is accounted for while creating a truth table.

To achieve this, firstly a Tree Map of the identifiers is obtained. A Tree Map is used as it sorts the map according to the natural ordering of its keys [2], which are strings in this case, which means that the keys are stored alphabetically. This means the order of identifiers in the map of both formulae will be the same, allowing for easy comparison of truth tables. Once the tree map of the identifiers is obtained, the size of the map can be used to determine every situation. This is illustrated by the example below:

Consider the formula p $\wedge$ q $\wedge$ r, the situations are:

p	q	r
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

If we look at each situation in order, we can see that they are binary numbers starting from 0, going to 2^n where n is the number of identifiers. This connection can be used to assign values to identifiers. The pseudocode is shown below:

```
public int[][] createTruthTable() {
  TreeMap<String, Integer> identifiers = this.identifiers();
  int numIdentifiers = identifiers.size();
  int temp = (int) Math.pow(2, numIdentifiers);
  int[][] truthTable = new int[temp][numIdentifiers + 1];
  for(int i = 0; i < truthTable.length; i++) {
    String binary = Integer.toBinaryString(i);
    if(binary.length() < numIdentifiers) {
      int numZeroes = numIdentifiers - binary.length();
      String zeroes = "";
      while(zeroes.length() < numZeroes) {
             zeroes += "0";
```

```
      }
      binary = zeroes + binary;
    }
    for(int j = 0; j < truthTable[i].length - 1; j++) {
      char c = binary.charAt(j);
      int n = Integer.parseInt(Character.toString(c));
      truthTable[i][j] = n;
    }
  }
  for(int i = 0; i < truthTable.length; i++) {
    Set<String> keys = identifiers.keySet();
    TreeMap<String, Integer> id = new TreeMap<String, Integer>();
    Iterator<String> it = keys.iterator();
    int j = 0;
    while(it.hasNext()) {
      String key = (String) it.next();
      id.put(key, truthTable[i][j]);
      j++;
    }
    truthTable[i][j] = value(id);
  }
  return truthTable;
}
```

3.7.2 Easy Equivalence Solver

The first mode for solving equivalences is the easy mode. In this interface, the user can click on either an identifier or a connective. Once the user clicks on an identifier, a dialog opens up with all possible equivalences that can be applied to the identifier, for example, p $\equiv \neg\neg$p. If the user clicks on an connective instead of an identifier, the dialog which opens up shows the equivalences that can be applied to the clicked connective, for example, if $\wedge$ was clicked from the formula a $\wedge$ b, then one of the equivalences that would be seen would be the commutativity equivalence, i.e a $\wedge$ b $\equiv$ b $\wedge$ a.

3.7.3 Hard Equivalence Solver

The hard equivalence solver is an interface for solving equivalences where the student enters logic expressions into a text field. This expression is then compared to the previous expression to see if it is logically equivalent, and to see if only one of the given equivalences has been applied.

3.8 Overview of Packages

In this section, a description of each package written for the implementation of the application is given.

3.8.1 AST

This package contains all of the classes needed for the Logic tree structure. The top level class is the AST class, which contains all of the data about the tree, such as the top level node.

For nodes, there is an abstract class called ASTNode, which each node extends. Every node has a key, so the getter and setter for the keys are abstract methods in the ASTNode class. Then, there is a AST-PropositionalNode abstract class, which every propositional node extends. As discussed in the Background section, there are two different types of propositional nodes; unary and binary. Unary nodes include Truth, Falsity and Not, and the binary nodes are the Or, And, If then, If and only if nodes. To accomodate for these two types of propositional nodes, two abstract classes were creates; ASTPropositionalBinaryNode and ASTPropositionalUnaryNode. Both of these classes contain respective methods needed for the manipulation of the trees.

For testing purposes, the Visitor design pattern is used to print the tree to console. The Visitor design pattern allows the separation of an algorithm from an object structure on which it operates.

3.8.2 gui

This package contains all of the classes which are needed for the Graphical User Interface (GUI). This package also contains the main method in the InitialiseGui class.

It also contains classes which model all aspects of the GUI. Most of the functionality of the GUI has been places in other packages, so the functionality does not interfere with how the GUI looks. As there are two modes in which the equivalence can be performed, the methods they contain are essentially the same with different functionalities. Therefore, an abstract class is created which handles all equivalences.

The package also contains how the login, register and load from database page look.

3.8.3 database

This package handles all of the database interaction that occurs in the program. As we are provided with PostgreSQL databases, I decided on using JDBC to interact with the database.

Whenever the application makes a query to the database, it has to connect to it. To make this easy, a class DatabaseAdaptor contains a static method which connects to the database. Every query and update method calls this static method.

During the implementation of the application, the database had to be reset several times. It became tedious to open terminal and clear and then initialise the database several times. To overcome this problem, a class DatabaseInitialisor was created, which resets and initialises the database with one method call.

The other classes in the package contain all necessary methods for queries and updates to the database. Whenever a query is made, a ResultSet object is returned to the method. To keep all database interaction away from the main implementation of the code, the ResultSet is converted to another object which is suitable for the query.

3.8.4 buttonlisteners

This package contains Action Listeners for the buttons which are clicked when completing an equivalence in the Easy mode.

There is a listener class for each different type of node. Every listener creates a new dialog for its respective node.

3.8.5 dialogs

This package contains all of the custom dialogs which open when a connective or identifier is clicked while completing an equivalence in the Easy mode.

In each dialog, there is a button of which equivalence can be applied to that node. These buttons perform the equivalences whose methods are in the equivalence package.

When an equivalence is applied which cannot be applied, the dialog realises it and updates the feedback panel on the page to tell the user what the problem is.

3.8.6 eqtutor

This is the package which contains the lexer, parser and logic expression generator.

The lexer and parser classes are generated using ANTLR, and have been imported into the eqtutor package.

The other part of the package is the expression generator. The propositional expression generator firstly generates either a DNF (disjunctive natural form) or a CNF (conjunctive natural form). Once this has been generated, it applied random equivalences several times to give a logically equivalent expression, which the student can complete.

3.8.7 equivalence

This package handles all of the equivalences and manipulation of the logic trees. There is one major abstract class, which all of the Equivalence classes extend. This class contains the find and replace methods, which have been explained earlier in this chapter.

For every different type of node, there is a different class which contains the methods for all of its related equivalences. All of these methods return the new logically equivalent AST.

This package also contains the data structure of the linked list and link node which are needed to store an equivalence which the user performs.

Chapter 4

Evaluation

Chapter 5

Conclusions and Future Work

Bibliography

Appendix A

Equivalences Truth Tables

A.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

A.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	$A \vee A$
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	$(A \vee B) \vee C$	$A \vee (B \vee C)$
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

A.3 Not Equivalences

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

A	$\neg A$	$\neg\neg A$
0	1	0
1	0	1

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

A	B	¬B	A → B	¬(A → B)	A ∧ ¬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

A.5 If and Only If Equivalences

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

A	B	A → B	B → A	A ↔ B	(A → B) ∧ (B → 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 ∧ B	¬A	¬B	A ↔ B	(A ∧ B) ∨ (¬A ∧ ¬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

A.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

A.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