Imperial College London

Department of Computing

Third Year Individual Project

Equivalences Tutor

Author:
Sahil Jain

Supervisor:
Fariba Sadri

Second marker:
Ian Hodkinson

June 7, 2013

Abstract

Acknowledgements

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

Chapter 3

Approach and Implementation details

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

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;}
  ;
```

Chapter 4

Evaluation

Chapter 5

Conclusions and Future Work

5.1 Learning outcomes

5.2 Potential improvements

5.3 Potential extensions

Bibliography