Equivalence Teaching Tool

Anna Thomas

June 1, 2014

Contents

Introduction

1.1 Motivation

All logics are based on propositional logic in some form, so it is important that new students learn how to use it. Propositional logic consists of syntax, semantics and proof theory; syntax is the formal language which is used to express concepts, semantics provide meaning for the language and proof theory provides a way to convert one formula into another using a defined set of rules.

We know that new students learning propositional logic can struggle to understand the rules and how they should be applied to formulae. To help with this our idea is to create an equivalence teaching tool; this will be a tablet application which will allow a user to apply rules to a formula until they have reached the desired equivalence.

1.2 Approach

We decided early on in the project that the tool should be an Android tablet application as opposed to a web or mobile one; this allows for an intuitive, interactive design while still having a large screen space.

The tool can be divided up into its main component parts: the parser, tree constructor, tree processing and tablet interface. The parser will be generated by the ANTLR4 parser generator, the tree constructor, processing code and the tablet interface will be written in Java with the interface using the Android SDK.

The parser requires a grammar to generate the relevant parser components. This will be used to parse the initial equivalence and return the ANTLR4 tree representation of the string.

The tree constructor is required to take the ANTLR4 representation of a tree and convert it into a more useful data structure which can be modified and displayed easily. The operators and atoms of the formula will be represented as nodes and leaves respectively.

Tree processing will be used to internally calculate which rules are applicable to each node and will allow us to subsequently manipulate the tree by applying these rules.

The interface will have an intuitive design displaying the current formula's formation tree (generated by the tree constructor) and allowing a user to click on the operators to apply a rule.

1.3 Objectives

The application should have some key features. These are outlined below:

1. Graphical tree representation of formulae

   Representing the formula as a tree structure allows a user to see exactly how the formula should be read and can help them understand the order of operations. It also allows the user to click on an operator to select a rule for it; this is much more intuitive than just clicking on the whole formula and not knowing which section the rule would be applied to.

2. Undo/Redo functionality

   Previous equivalent formulae will be displayed above the current formula. When an old formula is selected it will expand into tree form and the formulae below it are faded out *(Undo)*. This will allow a user to perform rules on the old tree or select one of the later faded trees *(Redo)*. When a rule is applied, the faded formulae will be removed from the history allowing the user to continue on from that point (Figure 1.1).

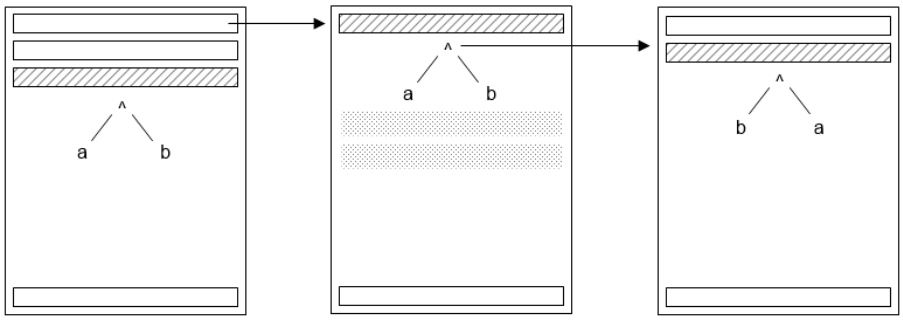

Figure 1.1: Undo functionality - Clicking a previous equivalence shows the formula tree for that equivalence and sets the undone history as translucent. Applying a rule then overwrites the undone history.

3. Generated equivalences

   We want to allow the user to have equivalences automatically generated for them to solve. This would be implemented by running the tool on a generated

formula to give a significantly different equivalent one. The key advantage of having this feature as well as allowing manual entry of equivalences would be that the user will have a continuous supply of new equivalent logic formulae after completing all those set by their lecturer. This also requires generating the initial formula for the tool to be run upon or allowing the user to manually enter the first formula and have the second one generated for them.

4. Difficulty setting

   Extending the idea of generated equivalences, we could allow the user to select a difficulty. This would be calculated by the length of the generated formula and how many rules were applied to get its equivalent formula. We could also provide a recommended difficulty based on how many previous equivalences they have completed and how far from optimal their solutions were.

5. Help

   Providing the user with help is key to their improvement. Once an equivalence has been set up, the tool should calculate the optimal route from the starting formula to the desired end point. Upon finishing the equivalence the user will receive a message telling them how far from the optimal solution they were and give them the option to try again or to view the optimal solution. Throughout there will also be a help button available that will suggest the next step to the user on request; this includes the recommendation to undo certain steps if the user reaches a cycle. Pointing out mistakes could also be enabled, so if the user has completed a cycle or is heading towards a dead end they will be prompted to consider a different strategy.

1.4 Achievements

We have been challenged with developing an android application to help new students learn propositional logic and to complete logical equivalences. In this section we will discuss our motivation, approach and objectives.

Logic

We are assuming a basic understanding of propositional logic, including the operators and rules that are defined in the system. For more information on these please visit the Wikipedia article[1].

2.1 Propositional Logic

Propositional logic is a branch of logic that studies ways of combining and modifying whole sentences, statements or propositions to form more complex propositions. It is a formal system containing logical relations and properties which are derived from these methods of joining or altering statements.

A logical system contains three major parts:

1. Syntax - the formal language that is used to express concepts.
2. Semantics - provide meaning for the language.
3. Proof theory - provides a way to convert one formula into another using a defined set of rules.

Logical definitions:

- *Atomic* - A formula whose logical form is $\top$, $\bot$ or p for an atom p.
- *Negated atomic* - A formula of the form $\neg p$.
- *Negated formula* - A formula of the form $\neg A$ for a formula A.
- *Conjunction* - A formula of the form $A \wedge B$.
- *Disjunction* - A formula of the form $A \vee B$.
- *Implication* - A formula of the form $A \rightarrow B$, where A is the *antecedent* and B is the *consequent*.
- *Literal* - A formula that is either atomic or negated atomic.
- *Clause* - A disjunction of one or more literals.

A *statement* is defined as a meaningful declarative sentence that is either true or false. For example, a statement could be:

- 'Socrates is a man.'
- 'All men live on Earth.'

A statement can be constructed of multiple parts, for example, the above statements can be combined into:

- 'Socrates is a man and all men live on Earth.'

Each part of this statement can be considered a proposition. Propositional logic involves studying the connectives that join these such as *'and'* and *'or'* (to form conjunctions and disjunctions), the rules that determine the truth values of the propositions and what that means for the validity of the statement.

2.2 First Order Logic

2.3 Truth tables

It is necessary to understand the meanings of the symbols used in a language. Truth tables are mathematical tables used in logic to compute the functional values of logical expressions. They can be used to determine whether or not a propositional logic statement is logically valid.

A *situation* determines whether each propositional atom is true or false. A truth table shows all the situations the input variables can be in. We write 1 for true and 0 for false as shown below:

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

Truth tables can be used to define any operators, including any new ones which might be desired.

Related Work

3.1 Logic Daemon

Created in Texas A&M University, the Logic Daemon[5] is an online logic proof checker. It comprises a simple web page with two small text input boxes for the premises and conclusion and then one large text box for applying primitive rules (Figure: 3.1).

While this tool does allow us to apply rules to prove equivalences we do not find it very intuitive to use at all. The interface itself looks quite unclear and isn't user friendly so it would not be suited to new students. Applying the rules to the premises is also quite confusing as it shows all the rules and does not alert the user to the fact they cannot be applied until they have already clicked on it. We found this a frustrating way of attempting a proof; as such, we intend to only display the rules that can be applied in the current situation.

However, it is a useful tool for those more knowledgeable about logic for checking proofs once they have got to grips with the interface. It also has a *'Get Help'* button to suggest which rule to use next. We hope to use the idea of a help button in our project but to build upon it in order to provide more extensive help such as showing when an action has been repeated (i.e. a cycle has been reached).

We hope that our tool will be more intuitive to use because we will display formulae as formation trees with interactive nodes to apply rules with. This should provide a more straight forward way of presenting the rules to new students.

The website also provides a simple equivalence checker, well-formed formula checker and countermodel checker. These are all similar tools useful in logic but again the interface is not particularly intuitive or user friendly.

The Logic Daemon tool also provides support for first order logic which we are not planning on covering in our main tool. However, this could be implemented as an extension at the end of our project.

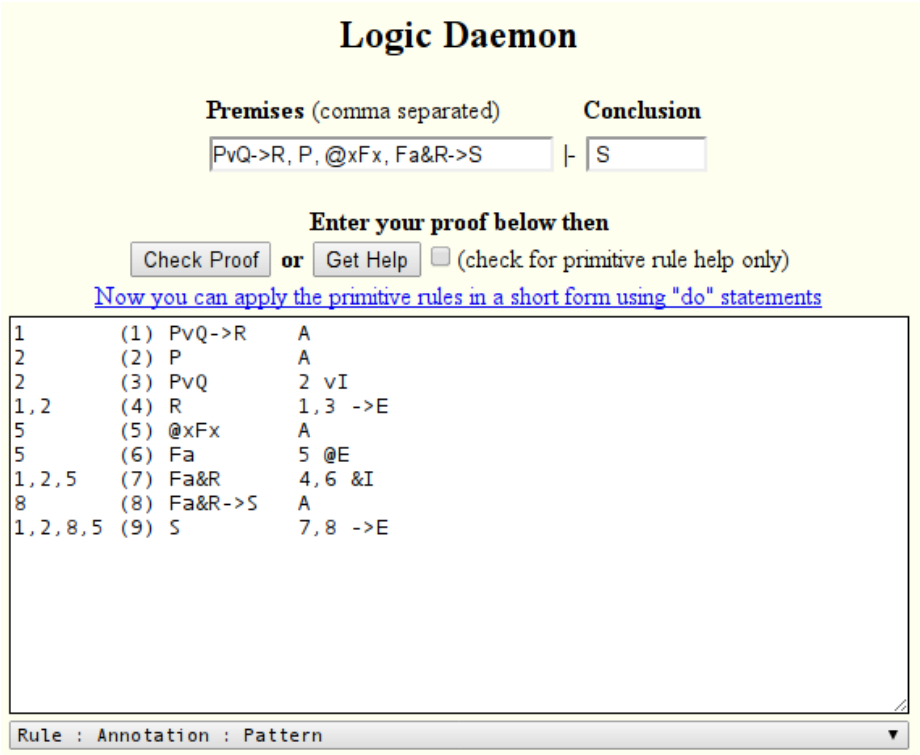

Figure 3.1: Logic Daemon

3.2 Pandora

Pandora[6] is a tool created by Imperial College London; it stands for 'Proof Assistant for Natural Deduction using Organised Rectangular Areas'. It can be used to prove that a goal formula follows from the given formulae. Pandora allows the user to repeatedly apply natural deduction rules (Figure: 3.2).

We do not intend to include natural deduction in our tool as we want to focus more on solving propositional logic equivalences. Pandora is a much nicer tool than the Logic Daemon and we find it much more intuitive and easy to use.

Pandora allows a user to apply rules forwards or backwards. We know from using Pandora that some proofs are easier to complete working backwards; as such, this is an idea we hope to incorporate into our tool by allowing the user to expand either the top or bottom formula to show its formation tree and to apply rules.

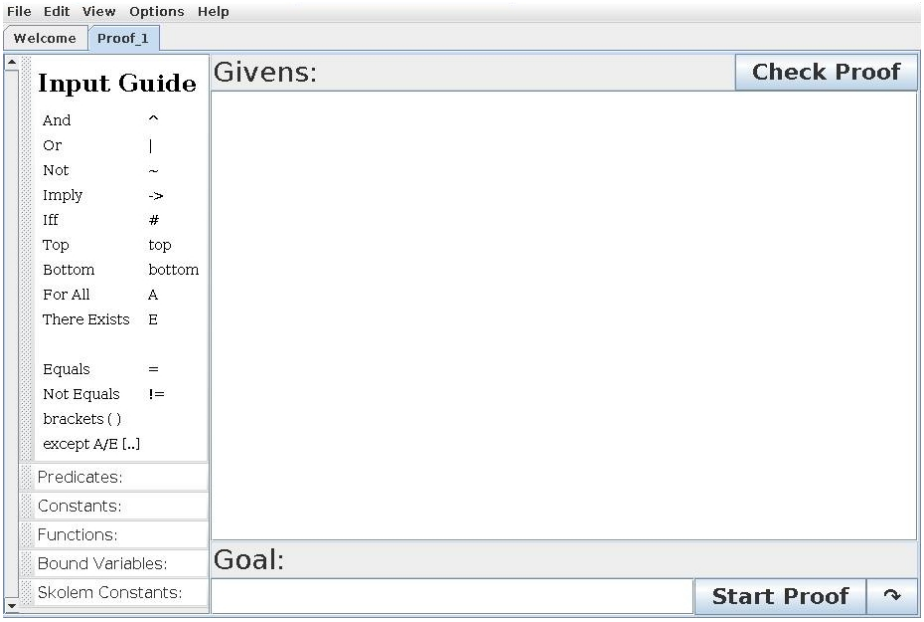

Figure 3.2: Pandora

3.3 Logic Solver

This is an Android application to show truth tables and possible logical equivalences. It looks like it has been made as a phone application as on a tablet it fills very little of the screen and it is difficult to click on some of the links (Figure 3.3).

The user can apply rules to the formula they enter into the application; this is done by selecting the rule they want out of a list which is then applied to the formula. However they cannot use this application to solve an equivalence as they can only enter one formula and then apply rules to that. We want to improve upon this in our own application by allowing the user to enter two formulae and work from either end to solve it.

The list of rules that is offered to the user only shows the rules that can be applied to the current formula. We want to also offer this behaviour as there are far too many rules to display them all to the user and expect them to sort through which could be applied.

Formulae are displayed simply in the application; we think this could be improved upon and want to display our current formula as its formation tree.

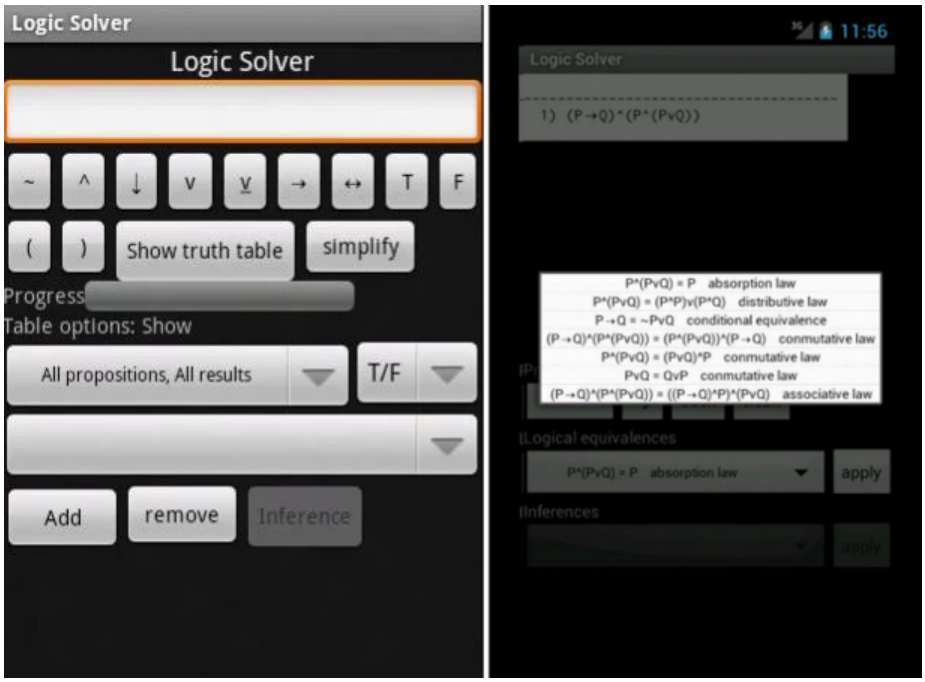

Figure 3.3: Logic Solver Android Application

3.4 Truth Tables

Truth Tables is also an Android application mainly for use on a phone. It simply generates and displays truth tables based on a logical formula (Figure 3.4).

We currently do not plan on adding truth tables to our application as we do not believe it is as useful for learning and understanding propositional logic as our other ideas. However, this could be added as an extension at the end of the project depending on how much time is left.

3.5 LogicCalc

LogicCalc is an Android application for solving problems with propositional logic. It allows the user to create workbooks to save and print their proofs (Figure 3.5). However we had trouble running it on our Nexus tablet.

We like the idea of saving proofs in workbooks for future reference. This is something we had not considered before and are curious to explore as an extension. This functionality would be useful for students completing exercises that needed a paper hand in as they would be able to save and print them off.

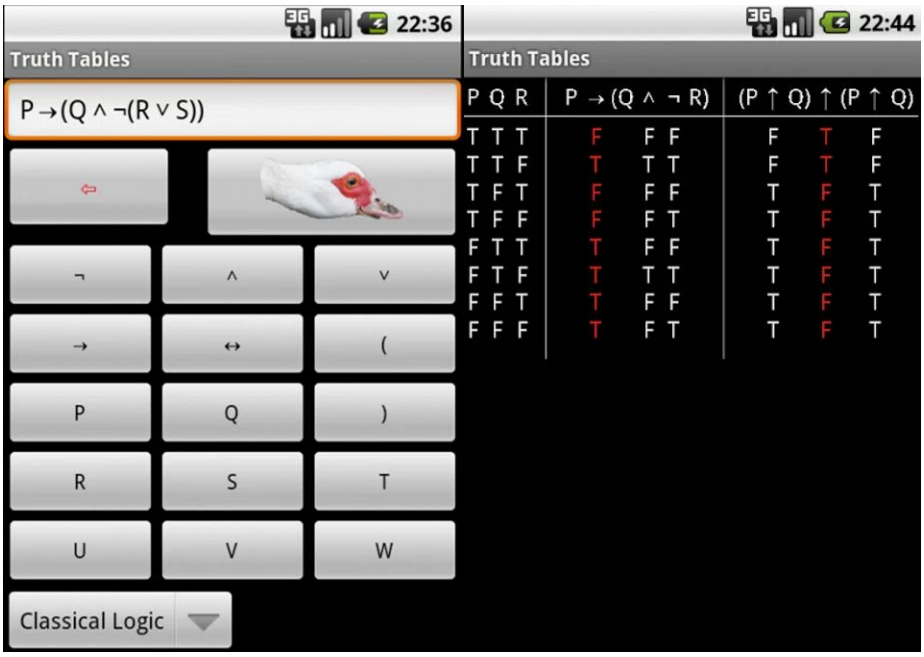

Figure 3.4: Truth Tables Android Application

3.6 Propositional Logic Calculator

The Propositional Logic Calculator[7] finds all of the models of a given propositional formula. The website tells us that the only limitation for this calculator is that we have only three atomic propositions to choose from: p,q and r (Figure: 3.6).

Propositions are entered using the keyboard they provide on the calculator and the reasoning process is initiated by clicking 'ENTER'. It then calculates the truth value assignments that will make the formula true in the 'MODELS' section and the truth value assignments making the formula false in the 'COUNTERMODELS' section.

This tool is used simply for calculating truth values of a formula. We will not be implementing this in our equivalences tool because we do not think that knowing the truth values for a formula is as useful for learning and understanding propositional logic as our other ideas.

3.7 Previous Year Projects

Once we have made more progress on our tool we will look into finding out what previous years did when they were assigned similar projects. We do not want to look

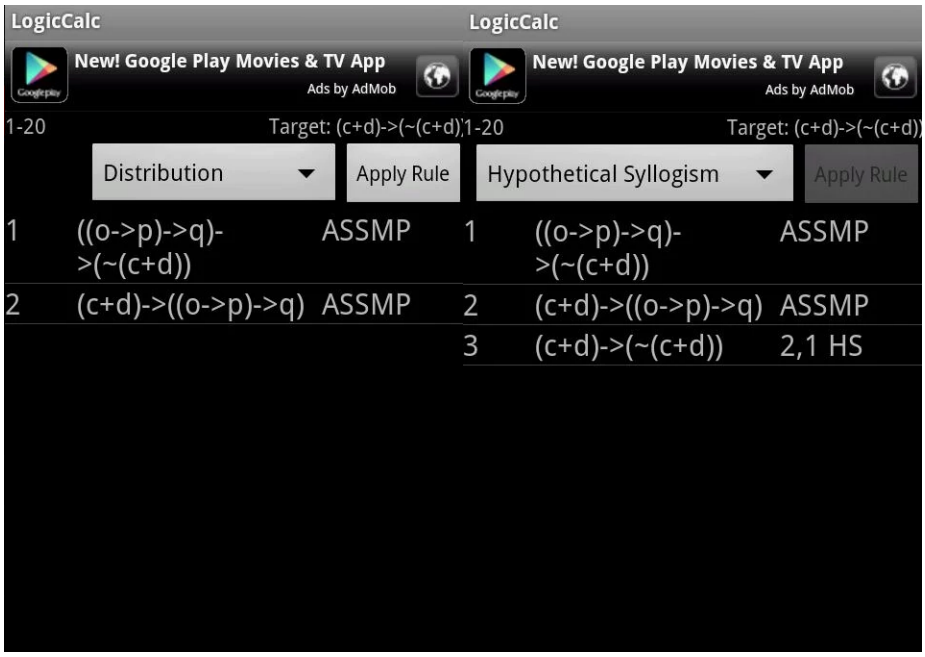

Figure 3.5: LogicCalc Android Application

into these this early in the project so we can generate our own implementation ideas as these projects will be very similar to ours.

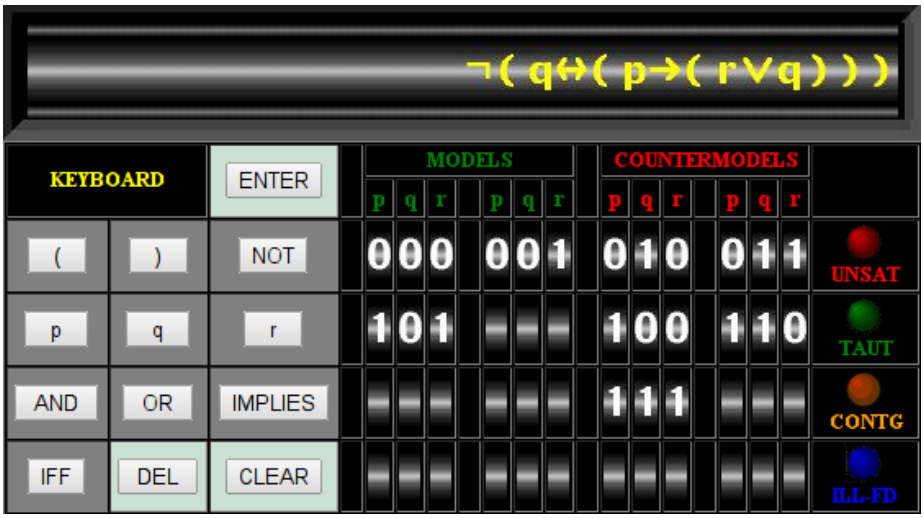

Figure 3.6: Propositional Logic Calculator

Parser

4.1 Parser Generation

A parser generator is a tool which creates a parser from an inputted grammar. This is necessary so that we can parse the logic equivalences entered in to our application and create an internal tree representation.

4.1.1 ANTLR4

ANTLR is a widely used powerful parser generator that can be used to read, process, execute or translate structured text. It is used by a variety of companies such as Twitter for query parsing and Hadoop in data analysis systems. ANTLR supports a wide range of languages including Java making it a very suitable choice for our project.

From a grammar ANTLR generates a parser for that language and has the ability to automatically build parse trees representing the input. More interestingly, ANTLR also generates tree walkers which can be customised to visit the nodes of those trees and perform actions on those nodes. We use this functionality to build our own custom tree as described in Section 4.2.3.

ANTLR uses EBNF for its input grammar notation and uses an LL(*) parsing algorithm. This allows direct left recursion so that expressing our required syntax is easy and natural.

4.1.2 Alternatives to ANTLR4

ANTLR4 provides many benefits over previous versions and alternatives. ANTLR4 uses a new variation of the LL(*) parsing algorithm that was used in ANTLR3. It uses an adaptive algorithm that is much stronger than the static grammar analysis algorithm used in ANTLR3. This means that ANTLR4 will accept much more natural looking grammars and provides a lot more flexibility than in previous versions. The

ability to write much more natural expression rules increases the usability as in previous versions it was often a struggle to write a grammar that ANTLR would accept. ANTLR4 aimed to focus more on ease-of-use and chose simplicity over complexity where possible. This is a large reason for choosing this parser generator over the alternatives.

Although there are a few other tools that can generate source code similar to ANTLR that you can load into a debugger and step through, none of them have adaptive LL(*) parsers as ANTLR4 does. Which means if we chose an alternative such as JavaCC we would need to configure our grammar to suit a weaker LL(k) parsing strategy. JavaCC is limited to the LL(k) class of grammars because it generates top-down parsers which means left recursion cannot be used. This would make generating the grammar more difficult and the result appear less natural.

Through our research we established that ANTLR4 provided the most sophisticated and easy-to-use parser generator for our project.

4.2 Grammar

4.2.1 Propositional Logic

The grammar we use with ANTLR can be seen in Listing 1. We only use propositional atoms between 'a' and 'r' because we don't need a whole alphabet of variables available and extending it to the whole alphabet would cause confusion with the logical or ('∨') symbol.

```
grammar Expr;

prog: expr;

expr: '(' expr ')'          #EXPR
    | '¬' expr              #NOT
    | expr BINOP expr       #BINOP_
    | expr '→' expr         #IMPLIES
    | expr '↔' expr         #IFF
    | ATOM                  #ATOM_
    |                       #ERROR
    ;

BINOP: ('∧' | '∨');
ATOM:  ('a'..'r' | '⊤' | '⊥');
```

Listing 1: Expr.g4 grammar for Propositional Logic to be used by ANTLR

4.2.2 Order of Precedence

Precedence rules can be used as a way of reducing the number of necessary parentheses. We use these in our grammar to decide which connective is the main connective when parsing a non-atomic formula. The precedence of logical operators is defined in Table 4.1.

Operator	Precedence
$\neg$	1
$\wedge$	2
$\vee$	3
$\rightarrow$	4
$\leftrightarrow$	5

Table 4.1: Order of Precedence for Propositional Logic

4.2.3 Parser, Lexer and Walker

ANTLR automatically generates a parser and lexer for the grammar. The lexer tokenises the input source (the logical formula) into a token stream which can then be passed to the parser. A parse tree is generated by the parser which can be walked through by a custom walker.

The custom walker extends the ANTLR generated listener to define methods for when the parser enters each rule defined in the grammar. For example, when the parser enters the `NOT` rule, `enterNOT()` is called. Equally, when that rule is exited, `exitNOT()` is called. This allows us to walk through the parse tree and create our custom tree as we go.

An empty `FormationTree` is passed through to the walker alongside the parse tree so the walker can build the tree up as the parse tree is walked through. Each time a grammar rule is entered the appropriate method in the walker is called to create that node in our tree. The structure of the tree is defined in more detail in Section 5.2.

4.2.4 Error Handling

ANTLR provides an `ANTLRErrorStrategy` interface for implementing a custom error handler to deal with syntax errors encountered during a parse by ANTLR-generated parsers. However, this has limited documentation and is a complex feature. Therefore, we have opted to relax the grammar by defining an error case so we can treat incorrect syntax as a semantic error instead.

As shown in Listing 1 we define an `ERROR` case which accepts any incorrect syntax. Once parsing is complete we walk over the tree using an `ExprWalker` (described in Section 4.2.3) and when an error case is entered we can set an error flag in the tree.

This enables us to manage the error later by displaying an error message to the user without the application throwing up errors.

4.2.5 Extension to First Order Logic

Internal Tree Representation

5.1 Compiler

The compiler combines all of the generated and custom parts of the parser. We define a `compile()` method to take the logic formula as input and parse it to create its tree representation. This is shown in Listing 2.

```
public FormationTree compile(String expr) {
    CharStream input = new ANTLRInputStream(expr);
    ExprLexer lexer = new ExprLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ExprParser parser = new ExprParser(tokens);

    FormationTree tree = new FormationTree(null);
    ParseTree parseTree = parser.prog();

    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(new ExprWalker(tree), parseTree);

    return tree;
}
```

Listing 2: compile() function to convert a logical formula into a FormationTree

5.2 Formation Tree

To build our internal representation of the logical formula we define a class `FormationTree`. This is a slightly modified binary tree that stores the root node and an error flag as well as performing all of the tree operations required - such as adding a new node (detailed in Section 5.2.2).

5.2.1 Node Index

In the final application we will be manipulating individual nodes in the tree. If there are two nodes with the same values and subtrees we need to be sure we're applying a rule to the correct node. Therefore, it is essential that all nodes in the tree have a unique index.

[Key, Depth] Pairs

We represent this index as a [key, depth] pair. We define this pair using the three following principles:

Principle 1: A node's depth is defined as the level at which it is found in the tree.

Principle 2: A node's key is defined by its horizontal position in its level with the leftmost node beginning at 0 and increasing incrementally.

Principle 3: Index's are also applied to gaps in the tree - where nodes could be placed in the future. So the leftmost leaf will only have a key of 0 if it is the child of all the leftmost nodes in the tree.

For example, a formula 'a∧b' would only have three nodes - the 'and' connective (∧) would have a key and depth of 0 and the atoms 'a' and 'b' would be at depth 1. As 'a' is the leftmost node it will be assigned a key value of 0, and 'b' will be assigned 1. The example's key and depths are shown in Figure 5.1.

By Principle 3 the incrementation of keys includes all gaps where it is possible for future nodes to be placed - so in our example it were possible for the node 'a' to not exist, b would still have the index [1, 1] and would not be upgraded to [0, 1] as there is still a gap for a left child to exist in.

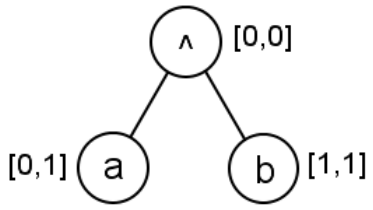

Figure 5.1: Tree representation of 'a∧b' showing the key and depth at each node

Binary Representation of Index

The index of a node can be represented in binary to enable us to navigate around the tree easily. The depth represents how many bits should be read and the key defines the binary number. The depth is required so that two nodes with the same key cannot be read as the same.

In the above example '∧' and 'a' both have the same key, but their binary representations would be different. '∧' does not have a representation as it is the root (length of binary read is zero) and 'a' would be 0. If 'a' had a children the left child's index would be read as 00 and the right child's as 01. We go into more detail about navigating the tree using the binary representation of indexes in Section 5.2.2.

Determining Child and Parent Indexes

Using this implementation of indexing not only guarantees that no two nodes will have the same index, but also we can easily work out the index of any child node, given we have the parent's index. For example, if we have a subtree of 'c→d' and we know the parent's (→) index is [3, 2] and we want to calculate the index of the child node 'c' we perform the following actions:

1. **Key**: We perform a logical left-shift on the parent key (3 « 1 = 6)
2. **Depth**: Increment the parent depth (2++ = 3)

So the resulting index becomes [6, 3]. To calculate the right child 'd' we can simply increment the key of the left child to become [7, 3].

We can also find the parent node's index given a child node by performing the opposite operations. A logical right-shift finds the parent key and decrementing the depth finds the parent depth.

TODO: Draw diagram of left-shift example

5.2.2 Tree Operations

Finding a Node

Finding a node based on it's index is extremely straight forward using our node indexing scheme. We define a method `findNode()` which simply steps through the tree using the binary representation of the index as a map.

For each level in the tree starting at the root node we take the left most bit of the key and this tells us whether we should take the left or right child route. An unset bit and a set bit represent the left and right children respectively. After we have taken that path and moved to the next level we use a mask to remove the left most bit from the key so that it does not conflict with the next level's left most bit.

Once the required depth is reached the current node is returned.

To find the node defined by index [1, 1] in Figure 5.1 we would begin at the root node and take the left most bit of the binary representation to determine our path. The binary representation of this node is 1. As the left most bit of this is 1 we take the right path putting us at the node 'b'. This is at the required level so the tree walk completes and we return node 'b'.

Adding a New Node

We define a method `addNode()` to handle the addition of new nodes to the tree. The node passed in will already have its index set up so here we simply need to assign it to the correct position.

If the root of the tree does not exist then the node becomes the new root. Otherwise, we calculate the parent node's index as described in Section 5.2.1 and use `findNode()` to obtain the parent node that will already exist in the tree. We set this as the new node's parent.

Either the parent node is a unary operator and we set the child of that node to be our new node, or it is a binary operator and we must determine if the the new node is a left or right child. If the node's key is divisible by 2 then the node is a left child, else it's a right child. We then set parent's child respectively.

5.3 Nodes

Due to the nature of logical formulae we need to be able to differentiate between unary operators such as $\neg$, binary operators such as $\rightarrow$ and atoms. We define a class `Node` which all of these will extend to define the core functionality required from every node.

Every node in the tree must contain a key, a depth, a value and a parent (unless it's the root in which case the parent is null).

We define `BinaryOperator`, `UnaryOperator` and `Atom` separately so we can set customised methods for each of them. Binary operators possess a left and right child, unary operators possess one child and atoms have none. The custom methods that require separate cases include a `clone()` function to duplicate the node (used in applying certain rules - Section 6.1), custom `toString()` functions and methods for determining truth values (Section 6.3).

Tree Manipulation

The application requires a few key features concerning tree manipulation: the ability to apply logical equivalence rules to a tree, compare the equivalence of two trees and randomly generate trees.

6.1 The Rule Engine

To handle the application of logical equivalences we define a series of rules that can be applied to individual nodes so long as they meet the required criteria for that rule. The `RuleEngine` manages the application and selection of these rules employing a `RuleApplicator` and `RuleSelector` respectively.

TODO: Add Appendix with rules list

6.1.1 Rule Selection

The `RuleEngine` defines a method `getApplicableRules()` to compute which rules are applicable to an individual node in a tree. This returns a `BitSet` where each set bit represents a rule that can be applied.

Most of the rules can be put into categories (by operator) so only a subset of the rules need to be tested for a particular node. An example category is the 'and' connective ($\wedge$) - if the selected node is an 'and' connective then only the test for these rules should be performed as we know no other rules outside of that category will apply.

Within categories we define the tests for each rule individually. Some rules do not need any extra tests other than being assigned to a category. For instance the commutativity rule can be applied to any 'and' or 'or' operator node without any further tests being conducted. Whereas, the distributivity rule could be applied to an 'and' operator only so long as one of its children was an 'or' operator.

The `RuleSelector` defines methods of testing for the properties required by the rules such as testing for idempotence which proceeds by checking that both subtrees

of the node are equivalent. Each rule that can be applied to the current node has its index set in the returned `BitSet`.

6.1.2 Rule Application

When the user selects an applicable rule the `RuleApplicator` is used to apply that rule to the selected node. A method `applyRuleFromBitSet()` is defined in the `RuleEngine` to choose the correct manipulation to apply to the tree. It implements a large switch statement over the index of the rule to be applied with each case calling the necessary functions from the `RuleApplicator`.

Some rules can require user input, for example in '$\bot \equiv a \wedge \bot$' we require some input to decide which variable to use in the place of 'a'. The selected variable can be passed in when applying the rule from `BitSet`.

6.2 Generating Equivalent Formulae

6.2.1 Random Formula Generation

When generating a logical formula we should be able to specify certain parameters such as how many variables and connectives are used. These parameters are used to generate different complexities of formulae for different difficulties.

Our `Compiler` is used to generate our random formula, `generateRandomEquivalence()` takes the number of variables we want to use and the maximum depth of the tree to be generated. A selection of variables that can be used is then chosen (Section 6.2.2) and passed through to a method `generateSubEquivalence()`. This recursively generates subformulae until the end criteria are fulfilled.

The end criteria are either the maximum depth or a random criterion has been reached. The random criterion adds an extra random factor so that we don't always get a balanced tree. This provides us with more interesting equivalences to work with. Currently the random criterion has a $\frac{1}{5}$ chance of succeeding because this produces nice looking results - trees which are generally not completely imbalanced but have a good level of variation to them.

During each recursion if the end criteria has been met then we can randomly choose an operator and variables to assign to it and set this as a final subformula. Else, instead of assigning variables we call the function again and assign the returned results to the operator.

Random operators and variables are selected using instances of `Random` from their respective lists. All the operators and a reduced selection of variables are available for use in the formula.

6.2.2 Variables

Initially all the variables would appear in a generated formula with the same probability. However, this proved to be a poor approach because you often ended up with many truth values ($\top$ and $\bot$) in the formula. This made the equivalences very easy to solve as you could usually just reduce both to their equivalent truth value. Proving the formulae were equivalent this way doesn't promote the users learning experience as they will repeatedly only use a very small subset of basic equivalences such as '$a \vee \top \equiv \top$'.

The solution to this problem was to assign probabilities to the variables. We define an enum `Variable` (see Listing 3) where all the entries have a value and a weight. The weight is defined on a scale of 0 to 10 with the higher weights being more likely to be chosen.

We define all atoms apart from the truth values with a weight of 10 and the truth values with a weight of 3.

The enum provides the method `randomVariable()` to return a random variable based on their weights. A simple analogy for this method is to imagine a ruler where the maximum value is the sum of the weights and each weight is dashed on the ruler. If you randomly placed your finger somewhere along the ruler then the segment you land on becomes the variable you return. So variables with smaller weights are less likely to be selected.

```
public enum Variable {
    A("a", 10), B("b", 10), C("c", 10), D("d", 10),
        E("e", 10), F("f", 10), G("g", 10), H("h", 10),
        I("i", 10), TRUTH("⊤", 3), FALSE("⊥", 3);

    ...
}
```

Listing 3: Variable enum defining values and weights for variables

6.2.3 Random Rule Application

Applying random rules to a formula is used in generating an equivalent formula. When only one logical formula has been specified the user can request an automatically generated equivalence.

The second formula can be generated easily by repeated application of random rules. This guarantees that the generated formula will be equivalent to the first and also allows us to set the difficulty of the proof based on how many steps there are between the two formulae.

There are two sections to the rule applications: choosing a random node and choosing a random rule.

In our tree we define a method `randomNode()` to randomly select a node. This method calculates the maximum depth of the tree by walking through the tree and returning the maximum depth found. A random level of the tree is chosen using an instance of `Random` and the maximum depth before walking down the tree to that level - at each level the method arbitrarily picks a child to follow. When the selected random level is reached the current node is returned.

Once a node has been selected to perform a rule on we call `getApplicableRules()` (see Section 6.1.1) to obtain the `BitSet` of applicable rules. From this the number of applicable rules is used with another instance of `Random` to select a rule to apply. Some of the rules require user input, so we define a special case for these rules to randomly select a variable to be passed through to `applyRuleFromBitSet()` (see Section 6.1.2) when applying the rule. If no user input is required then we pass through null.

6.3 Truth Tables

For a user to be able to complete a proof of their own custom equivalences we need to ensure that they're actually equivalent and the proof is completable before they begin. To do this we can use truth tables (defined in Section 2.3).

6.3.1 Generating Truth Tables

Creating Input Truth Values For Atoms

Determining Truth Value For Formula

6.3.2 Testing Equivalence of Tables

Replacing Variables

Differently Sized Tables

Android VS Web Applications

7.1 Android vs Web app

7.1.1 Intuitive to use

7.1.2 Learning experience

7.1.3 None currently available for Android

7.2 Tablet vs Phone app

7.2.1 Screensize

7.2.2 Formation Tree would not feasible on phone app

Android

8.1 Technical Background

8.2 Features

8.2.1 Multi-Touch Gestures

Android is an operating system based on the Linux kernel which was designed primarily for touchscreen mobile devices such as smartphones and tablet computers[3].

The user interface is based around direct manipulation. This means using touch inputs that loosely correspond to real-world actions, for example swiping, tapping and pinching to control items on screen. The response to user actions is designed to be immediate and provides a fluid touch interface. It often provides haptic feedback (forces, vibrations or motions) to the user using the vibration capabilities.

Android takes advantage of internal hardware such as accelerometers, gyroscopes and proximity sensors in some applications to respond to other user actions, such as re-orientating the screen from portrait to landscape.

8.3 Market Share

Android's main competitor in the mobile platform market is Apple's iOS. In research conducted in the fourth quarter of 2012, Kantar Worldpanel Comtech showed sales of all Android smartphones worldwide outpacing the iPhone by a huge margin: 70 percent to 21 percent of the smartphone market[4].

In the tablet market the iPad dominated Android's 7" tablets with 53.8 percent to 42.7, which is lower than Android's smartphone market but steadily increasing.

The App

9.1 Truth Tables

9.1.1 Ensure equivalences are equal before beginning

9.2 Random Generation of Equivalences

9.2.1 Generating single equivalence

9.2.2 Generate equivalence formula - repeated application of random rules

9.3 Custom Keyboard

9.3.1 Logic symbols and variables

9.4 Undo/Redo Functionality

9.4.1 Stack of trees maintained

9.4.2 OnClick listeners

9.5 Formation Tree Representation

9.5.1 The Formation Tree

Each logical connective in an formula can be written enclosed in parentheses. For example:

$(a \rightarrow b) \wedge \neg a$ can be written as $((a \rightarrow b) \wedge (\neg a))$

Due to this property we can create a binary tree unique to each proposition; this is called a formation tree. Each connective and propositional atom is represented by a node and a leaf respectively. This provides a clear and attractive way of displaying the formula but is too expensive for every day use. In our application we wish to display the formation tree of the current equivalences.

Displaying the formation trees allows a user to see an interactive representation of the equivalences where touching the nodes and leaves in the tree allows the user to apply rules to various parts of the tree.

9.5.2 The NP-Complete Problem

When planning to draw the formation trees to the app we assumed there would be a simple, classic algorithm for drawing neat, aesthetically pleasing trees. However, we discovered the problem was not that simple. Drawing an attractive Tree layout is an NP-complete problem[11] and there are many tree-drawing algorithms attempting to solve the problem of drawing attractive trees. We reviewed many of these algorithms to find one suitable to use.

9.5.3 What Makes An Attractive Tree?

Although generating an attractive tree is a matter of taste, certain principles are widely agreed upon to be key to drawing an aesthetically pleasing tree. The first three are taken from Wetherell and Shannon's tree drawing algorithm for producing tidy drawings using the smallest width possible[8].

Aesthetic 1: Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.

Aesthetic 2: A left son should be positioned to the left of its father and a right son to the right.

Aesthetic 3: A father should be centered over its sons.

Although not mentioned in the original article, Aesthetic 1 was also meant to guarantee the edges in the tree do not intersect except at the nodes by requiring that the relative order of nodes over any level be the same as the level order traversal of the tree. Wetherall and Shannon's algorithm is fairly basic and has a deficiency that compromises the overall attractiveness of the resulting tree. It produces drawings that could be made narrower within the constraints of the aesthetics and are not entirely pleasing to the eye - nodes in certain subtrees are drawn too far apart due to the fact that their shape is influenced by the positioning of nodes outside that subtree. This leads to the asymmetry of the resulting tree. Therefore when Reingold and Tilford[9] set out to create a better tree drawing algorithm they introduced a fourth aesthetic:

Aesthetic 4: A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

Fulfilling this aesthetic requires us breaking Wetherell and Shannon's aim of minimum width so that all all isomorphic subtrees are drawn the same. This is considered a more important principle than using the minimum width as our main aim is to create an attractive tree. This is an acceptable drawback as Supowit and Reingold proved that determining the minimum width under these aesthetics is NP-hard[10]. Therefore, we can use Reingold and Tilford's algorithm to draw our formation tree to produce an attractive result.

9.5.4 Reingold and Tilford's Algorithm

The proposed algorithm was based on the following heuristic:

"Two subtrees of a node should be formed independently, and then places as close together as possble"[8].

This is applied during a postorder traversal of the tree by taking the two subtrees of a node and calculating the contours of those subtrees. The two subtrees are then positioned so they overlap at their roots before moving them apart until no two points are touching. This is done recursively at each level moving down the levels of the tree until the bottom of the shortest subtree is reached. Once the process is finished the position of the subtrees can be fixed relative to their parent node which is centered over them. Once the postorder traversal is complete a preorder traversal occurs to convert the relative positions to absolute coordinates.

9.5.5 Walker's Algorithm

Extending Reingold and Tilford's algorithm to to rooted ordered trees of unbounded degree as described in their paper produces layouts where some subtrees of the tree could have been dispersed better to create a more attractive layout. In 1990 Walker's algorithm solved this problem by spacing out subtrees whenever possible, however in the initial algorithm presented the runtime was quadratic. Walker's algorithm was improved upon by Christoph Buchheim, Michael Jünger and Sebastian Leipert so it would run in linear time[13]. This is the algorithm we use in our application.

9.5.6 Abego TreeLayout

Abego TreeLayout is an external library which efficiently creates compact and highly cusomisable tree layouts. It is based on Walker's algorithm with the enhancements previously mentioned suggested by Buchheim, Jünger, and Leipert[14]. The software builds tree layouts in linear time so that even trees with many nodes are built quickly.

TreeLayout separates the layout of the tree from the actual rendering, therefore it is suitable for drawing trees in an Android application because it is not limited to a specific output or format.

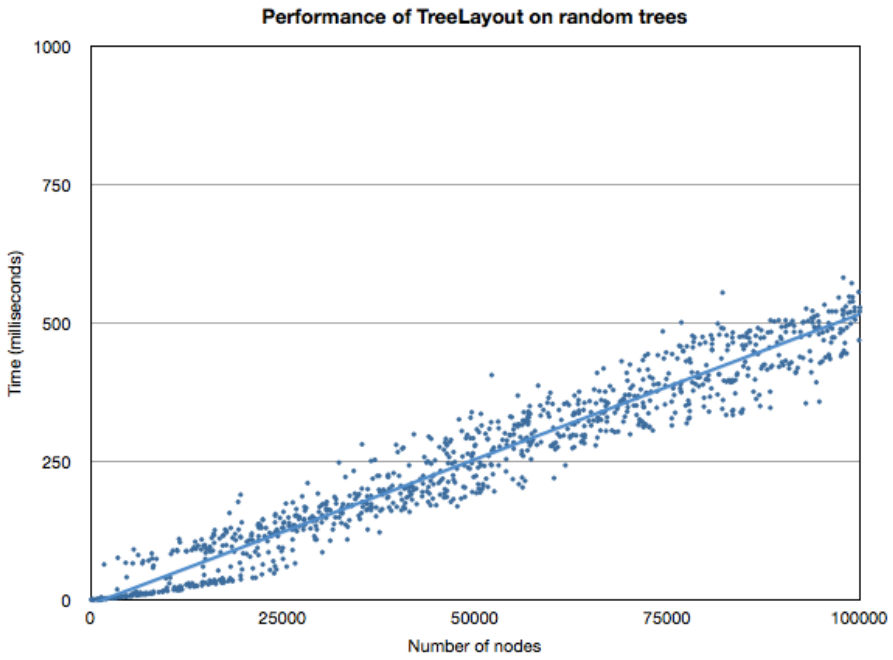

Figure 9.1: Illustrates the linear time behaviour of the Abego TreeLayout algorithm[15] and shows the applicability for large numbers of nodes.

9.5.7 Drawing the Formation Tree

We created a custom view DrawView to draw the trees to the screen in the application. This is created by simply extending the Android View class and overriding the `onDraw()` function. To add a custom view to our user interface we also need to define custom attributes in a resource element as shown in Listing 4.

```
<declare-styleable name='DrawView'>
    <attr name='showTree' format='boolean' />
</declare-styleable>
```

Listing 4: DrawView resource element

In `onDraw()` we determine whether we are drawing the top or bottom formation tree and call `setUpTreeLayout()` (shown in Listing 5) which uses the Abego TreeLayout to manage the layout.

To use TreeLayout we create a `TreeLayout` instance with our tree (accessible through the `TreeForTreeLayout` inferface), a `FixedNodeExtentProvider` (defines a fixed size for each node) and a `DefaultConfiguration` (configures the layout by defining parameters such as `gapBetweenNodes` and `gapBetweenLevels`).

```
public void setUpTreeLayout(Location location) {
    DefaultConfiguration<Node> configuration = new
            DefaultConfiguration<Node>(gapBetweenLevels,
            gapBetweenNodes, location);

    FixedNodeExtentProvider<Node> nodeExtentProvider = new
            FixedNodeExtentProvider<>(20, 20);
    treeLayout = new TreeLayout<Node>(tree,
            nodeExtentProvider, configuration);

    ...
}
```

Listing 5: setUpTreeLayout() is called by onDraw() in DrawView

TreeLayout calculates the position of each node so we can then paint the nodes and edges to the screen using Android Canvas.

9.5.8 Responding to Touch Events

By overriding `onTouchEvent()` in DrawView we can get the event action performed on the view. Using this we can check if the touched position is within the bounds of any of the nodes drawn to the screen. If the position is within the bounds of a node we display applicable rules for that node by calling `setRules()` on the current activity (`BeginEquivalenceActivity`). Selecting a node also forces a redraw so that we can highlight the selected node.

9.6 Rule Application

9.6.1 Rule Display

The rules are displayed in a PopupMenu on screen either above or below the selected tree. The rules are generated by the `RuleEngine` as described in Section 6.1.

Calling `setRules()` adds the generated rules iteratively to the PopupMenu. If the rule requires user input (eg. $\top := \bot \rightarrow a$) a SubMenu is created with the variables that can be applied as shown in Listing 6. Each entry in the PopupMenu contains the String representing the rule and the rule key. The key is required so we know which rule to apply on touch events.

```
public void setRules(SparseArray<String> rules, Node selected,
        FormationTree selectedTree) {
    ...

    int key;
    for (int i = 0; i < rules.size(); ++i) {
        key = rules.keyAt(i);

        if (key < min_user_input_required) {
            rulesList.getMenu().add(Menu.NONE, key, Menu.NONE,
                    rules.get(key));
        } else {
            SubMenu sub = rulesList.getMenu().addSubMenu(Menu.NONE,
                    key, Menu.NONE, rules.get(key));
            SortedSet<String> vars = topTree.getVariables();
            vars.addAll(bottomTree.getVariables());

            for (String v : vars) {
                sub.add(Menu.NONE, key, Menu.NONE, rules.get(key)
                        + " using " + v);
            }
        }
    }
    rulesList.show();
}
```

Listing 6: Rules being added to the rulesList PopupMenu in setRules

9.6.2 Responding to PopupMenu Touch Events

Overriding `onMenuItemClick()` in `BeginEquivalenceActivity` allows us to determine which item was selected from the PopupMenu and and apply that rule to the selected node. We can determine which rule should be applied by taking the key of selected item and applying that rule from our BitSet of rules using the `RuleEngine` as described in Section 6.1.

Applying a rule to the selected node adds a new TextView to the respective list of equivalences (either top or bottom). Once the rule has been applied the application will check for completion and cycles (described in Section 9.8.1). If either are found a Toast (a simple popup for displaying feedback) will be displayed alerting the user.

We also override `onDimiss()` so that when a user dismisses the PopupMenu without selecting a rule we can also redraw the formation tree to deselect the selected node.

9.7 Difficulty

9.7.1 Length of generated equivalences

9.7.2 Number of variables/operators

9.7.3 Number of rules different

9.8 Help

9.8.1 Detecting cycles

Evaluation

10.1 Testing

10.1.1 JUnit tests

10.1.2 Survey - People who do/don't understand logic

10.2 Performance

Once the application has been built we will need to find a way to test that it has met the objectives we set out to achieve. The main objective is to create an intuitive Android application that helps students improve in solving equivalences; as such, we will survey a wide range of test subjects. These tests will be split into two groups: people who understand logic and equivalences well and people who are new to propositional logic.

We need to ask people who understand logic (e.g. lecturers) so we can confirm that it works as a teaching tool. People who are new to logic (e.g. first year students) are necessary so we can evaluate how effective it is as a learning tool. This can be carried out through asking the target groups to use the application and provide feedback.

Conclusions

11.1 Completed objectives

11.2 Comparisons to related work

Future Work

12.1 Improvements to completed objectives

12.2 Device support

12.2.1 Reduced app without trees for phones

12.3 More complex help

12.3.1 Calculating the optimal route

12.3.2 Suggested next move

12.3.3 Future dead end detection

12.4 First Order Logic

12.4.1 Extend grammar

12.4.2 Add and handle new rules

Bibliography

[1] Wikipedia, *Propositional Logic*. http://en.wikipedia.org/wiki/Propositional_calculus

[2] IEP, *Propositional Logic*. http://www.iep.utm.edu/prop-log

[3] Wikipedia, *Android (Operating System)*. http://en.wikipedia.org/wiki/Android_(operating_system)

[4] Harry McCracken, *Who's Winning, iOS or Android?*. http://techland.time.com/2013/04/16/ios-vs-android

[5] Colin Allen, Chris Menzel, *Logic Daemon*. http://logic.tamu.edu

[6] Imperial College London, *Pandora*. http://www.doc.ic.ac.uk/pandora/newpandora/index.html

[7] Enrico Franconi, *Propositional Logic Calculator*. http://www.inf.unibz.it/~franconi/teaching/propcalc

[8] C. Wetherell and A. Shannon, *Tidier Drawings of Trees*. IEEE Trans. Software Eng., vol SE-5, pp. 514-520. 1979.

[9] Edward M. Reingold and John S. Tilford, *Tidier Drawings of Trees*. IEEE Transactions on Software Engineering, vol SE-7, no. 2, March 1981.

[10] Kenneth J. Supowit and Edward M. Reingold *The Complexity of Drawing Trees Nicely* vol 18, issue. 4, pp 377-392, January 1983.

[11] K. Marriott, *NP-Completeness of Minimal Width Unordered Tree Layout*, Journal of Graph Algorithms and Applications, vol. 8, no. 3, pp. 295-312 (2004). http://www.emis.de/journals/JGAA/accepted/2004/MarriottStuckey2004.8.3.pdf

[12] Bill Mill, *Drawing Presentable Trees*. http://billmill.org/pymag-trees

[13] Buchheim C, Jünger M, Leipert S. *Improving Walker's Algorithm to Run in Linear Time* Graph Drawing Lecture Notes in Computer Science Volume 2528, 2002, pp 344-353

[14] Buchheim C, Jünger M, Leipert S. *Drawing rooted trees in linear time*. Software—Practice and Experience 2006; 36(6):651–665

[15] *Abego TreeLayout* https://code.google.com/p/treelayout/

User Guide

13.1 How to use the app