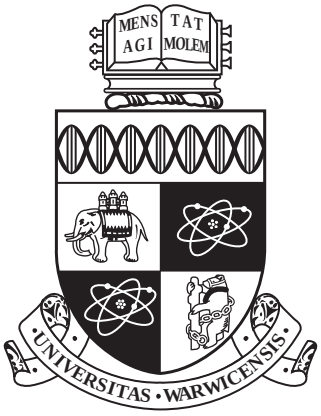

An Android Application for Building and Checking Proofs using First Order Logic

by

Rhys Davis

Supervised by Ranko Lazic

Department of Computer Science

University of Warwick

April 2013

Abstract

The purpose of this project was to create an application that will facilitate the learning of a particular language of symbolic logic, known as first order logic and the system of natural deduction that goes with it. An Android application was created which: allows the user to experiment with the formulation of sentences of first order logic, evaluates theses sentences and assists in the construction of proofs using natural deduction. This report details the research carried out, the design and the development the Android application.

Acknowledgements

I would like to thank my supervisor Dr Ranko Lazic, firstly, for the teaching and materials of CS242 Formal Specification and Verification on which this project is based and secondly for the support and feedback given throughout this project. I would also like to thank Dr Matthew Leeke who created the LaTeXtemplate which was used as a starting point for this report.

Abbreviations

FOL	First-order logic
TDD	Test driven development
ZF	Zermelo-Fraenkel
ZFC	Zermelo-Fraenkel set theory with the axiom of choice

Notations

r_2	Has some meaning

Contents

Abstract **ii**

Acknowledgements **iii**

Abbreviations **iv**

Notations **v**

List of Figures **viii**

List of Tables **ix**

0.1 Author's assessment of the project 1

1 Introduction **3**

1.1 Motivation 3

1.2 Background 4

1.3 FOL explained 8

1.3.1 Syntax 8

1.3.2 Deductive System 11

1.3.3 Rules 12

2 Project management **17**

2.1 Test Driven Development 17

2.1.1 Process 17

2.1.2 Advantages of Test Driven Development 18

2.1.3 Tools used 19

2.2 Prioritisation 20

2.3 Project management tools 21

2.3.1 Trello 21

2.3.2 Git 21

2.4 Justification of the choosing the Android platform 22

2.4.1 Choosing a mobile solution 22

2.4.2 Choosing a native solution . . . 23
2.4.3 Choosing the Android platform . . . 23

3 Formula checker **25**
3.1 User interface . . . 25
3.2 Parser . . . 26
3.2.1 Predictive parsing . . . 27
3.2.2 Operator precedence . . . 28
3.2.3 Eliminating left recursion . . . 28
3.2.4 Resolving FIRST/FOLLOW conflicts . . . 29
3.2.5 LL(1) grammar . . . 31
3.2.6 Implementation . . . 31
3.2.7 Abstract Syntax Tree . . . 32
3.3 Bound and free variables . . . 34

4 Proof Builder **36**
4.1 User interface . . . 36
4.1.1 Proof thingy . . . 36
4.1.2 Button bar . . . 37
4.1.3 Justifying a line . . . 39
4.2 Rules of Inference . . . 39
4.2.1 API . . . 40
4.3 Proof structure . . . 41
4.3.1 Class **Proof** . . . 42

5 Chapter Title **45**

List of Figures

2.1 Flowchart of TDD process 19
2.2 Smartphone ownership by age 22
2.3 Mobile operating system by market share 24

3.1 User interface for formula checker 27
3.2 Compilation process for FOL parser 33

4.1 User interface for proof builder 37
4.2 User interface for justifying a line 39

List of Tables

0.1 Author's assessment of the project

What is the technical contribution of this project?

This project involved a substantial software development undertaking in which a mobile phone application for the Android operating system was developed from scratch. Prominent challenges of this included creating a user friendly user interface and the suitable use and creation of data structures to store and manipulate data. Additional technical challenges include creating a parser for the language of FOL using the parser generator JavaCC and coding the rules of inference in the Java programming language.

Why should this contribution be considered either relevant or important to Computer Science / Discrete Mathematics?

The contribution is relevant to both Computer Science and Discrete Mathematics because the main function of this project function is as a teaching tool for Logic, which is an area of particular interest by researchers in both fields. In addition to this, the mobile application development aspect is an increasingly popular field in computer science.

How can others make use of the work in this project?

Teachers of FOL will be able to make use of this tool by using it as a way to get students to participate and attempt proofs individually. The feedback from the tool will help guide the student and correct mistakes allowing the teacher to focus on helping students with the creative aspect of a proof. The opportunity to work independently on creating proofs at their own pace will help students to gain confidence in constructing proofs. Due to the modularity of the code written parts of the system can be re-used in other systems that may expand on the work achieved in this project.

Why should this project be considered an achievement?

The projects objectives were all successfully completed. The end result is fast and makes efficient use of memory. Feedback from users was positive indicating that the system is both easy to use and fit for purpose. Extensive testing means that the system is robust and handles incorrect input in an appropriate manner.

What are the weaknesses of this project?

While the system achieved it's objectives the scope of the project is not particularly wide. To increase the usefulness of this project additional features would need to be added. As the project stands it is not possible to set goals for the student to reach and has no component that teaches about the use of models and environments in FOL.

Chapter 1

Introduction

1.1 Motivation

In my first year at university I was introduced to the notion of a formal language of logic, I found the idea of being able to communicate ideas and formulate arguments in a formal and rigorous manner very appealing. Learning to understand what the symbols meant, how they fitted together and reasoning within the language is not, however, a trivial task. The aim is to produce a tool that would ease the transition of the user as they attempt to familiarise themselves with a language which is very different from any natural languages they may have experience with.

As with learning any other language, participation and practice are key factors in the learning process. That is why I set out to create an application that will give the user freedom to explore the language and deductive system themselves, while also providing feedback that will help guide them in this activity.

1.2 Background

Logic can be used to refer to both the use of valid reasoning and as a distinctly different entity, the study of valid reasoning. It is the latter of these that will be the main focus of our attention. As such, the use of logic will subsequently be used to not simply mean the use of valid argument, but rather a reflection upon the principles of validity. While the use of valid reasoning has been attributed to early humans and even to non-human animals [14], the earliest known systematic study of formal logic is attributed to Aristotle in the 4th century BC. [12, p. 300]

The works of Aristotle, later grouped into in a collection known as the Organon, drew from sources such as the works of Aristotles teacher Plato, Zeno of Elea and geometrical proofs. [7]

One of Aristotle's most influential pieces was that of the syllogism which Aristotle defined as "a discourse in which, certain things being stated, something other than what is stated follows of necessity from their being so." [1, p. 4] Aristotle generally restricted a syllogism to three parts: the major premise, the minor premise and the conclusion. The premises were restricted to the form it where it contains a subject and a predicate and must either affirm or deny the predicate of the subject. [13] -explain terms

It is believed that the Pythagorean schools were the first to have attempted a systematic study of the demonstrative sciences and Aristotle used some of the ideas from the deductive system used in geometrical proofs. [8, p. 3]

An important concept in Aristotle's deductive system was that of Reductio ad Impossibile, which demonstrates that an assumption is false by showing that the original assumption leads to a contradictory conclusion. Aristotle attributed this concept to Zeno of Elea but compares it to the use Reductio ad Absurdum as used by Euclid in the proof that the square root of 2 is irrational.[8, p. 8]

Reductio ad Absurdum demonstrates that an assumption is false by showing that the original assumption leads to drawing of false conclusions. While Reductio ad Impossibile is similar to Reductio ad Absurdum, the latter is stronger since it only requires that the conclusion is clearly false, not necessarily a contradiction. [8, p. 9]

The term is a part of speech representing something, but which is not true or false in its own right, such as "man" or "mortal". The proposition consists of two terms, in which one term (the "predicate") is "affirmed" or "denied" of the other (the "subject"), and which is capable of truth or falsity.

The Stoic school developed a parallel logical system to Aristotelian logic. This system was focused on prepositions rather than terms. The greatest

contributor of the Stoics were made by Chrysippus. Chrysippus seperated prepositions into two classes, these classes are the equivalent of the modern day atomic and compostie propositions. The Stoics became the first of the logicians to study conditional statements. These conditional statements included strict implication, relevant implication and the now well know material implication i.e. statements of the from p $\Rightarrow$ q or equivalently if p then q, where p and q are propositions. They discussed the truth values of these conditionals and other logical connectives. From these discussions it is possible to construct the modern day truth table representation of these connectives.

Despite the belief, at the time of creation, that these two rivals were incompatible; the two systems were actually complimentary and eventually merged into one logical system.

After this period few major developments were made in western logic for quite some time. Most of the work involved simplification and combination of previous resulting in greater clarity. An effect of this greater clarity was that certain some concepts such as relevance implication were rejected. [7]

In the 18th century Leibniz, inspired by mathematics, envisaged a symbolic logic which would enable a form of logical calculus. Such a system would eventually come to fruition, however, this system was developed independently of Leibniz's ideas despite his early predictions. [11]

In the early 19th century Bernard Bolzano made many innovative contributions to logic in respect to topics such as logical truth, analyticity and validity. Later in the century George Boole attempted to show that logic how logic can be expressed algebraically. The system devised by Boole, along with alterations and additions by his followers, became the Boolean algebra that is used today.

Concurrent to the development of the algebraic system of Boole, the school of Logicism emerged. One of it's proponents, Gottlob Frege, is responsible for some of the some of the most important concepts used in logic today. Frege reasoned that expressions of the form A is B are not the same as every A is B and as such needed to be treated in a separate manner. Furthermore, every A is B is not an atomic proposition but rather a composite proposition which asserts that if some object satisfies the function is A then we necessarily have that this object satisfies the function is B****().

An implication of this distinction is the resolution of the problem of multiple generality. The problem of multiple generality is a short coming in traditional logic where, since sentences with more than one quantifier cannot be accurately represented, there are certain intuitively valid inferences which can not be made. The notion of scope of quantifiers (Quantifiers may be applied to specific parts of a sentence as well as the whole sentence) allows such

sentences to be expressed non ambiguously and the previously unachievable inferences can be made in this system.

Despite Frege attempts at a mathematically rigorous formal language, it was shown by Bertrand Russell to be inconsistent. Two of the most important of the proposed resolutions were Zermelo set theory and another was published by Bertrand Russell and Alfred North Whitehead in the Principia Mathematica.

Zermelo set theory was the first axiomatic set theory. It was this theory along with contributions by Abraham Fraenkel that eventually led to Zermelo-Fraenkel set theory which is now the standard form of axiomatic set theory. [3]

Bertrand Russell and Alfred North Whitehead's proposed solution attempted to resolve Russell's paradox by defining a hierarchy of types and subsequently assigning each entity a type. Aside from introducing many important notions, such as type theory, arguably it's most important contribution was the work that it inspired. Principia Mathematica, using it's more accessible notation, showed both the expressiveness of predicate logic and the deductive power of the new language while highlighting links with traditional logic. Ultimately Principia Mathematica stimulated a great deal of research within logic and related fields. [6]

Shortly after Leopold Löwenheim classified a particular subset of the logical system, now known as First-order Logic and demonstrated that for any sentence in First-order logic if it satisfiable then it is satisable in a countable domain. Work by Thoralf Skolem expanded on Löwenheim's research and went on to generalise results resulitng in the LöwenheimSkolem theorem. In 1923 during his research Skolem gave the earliest first-order axiomisation of ZF set theory.[7, p. 6]

The LöwenheimSkolem theorem led directly to the very important Gödel completeness theorem in 1929 which showed that a given sentence of FOL is deducible in a deductive system of the language if and only if it is logically valid. Gödel, in his Incompleteness thoerems, would go on to show that axiomatic systems of arithmetic contain a sentence which cannot be shown to be either true or false. [7]

By the 1930's the modifications to Zermelo-Fraenkel and the type theoretical system meant that the two opposing systems had became similar in many aspects. The rise of the acceptance of FOL, helped by the properties such as completeness and compactness, meant it became the norm to express the Zermelo-Fraenkel set theory in FOL. While Zermelo-Fraenkel set theory can be represented in FOL, type theory is a higher order logic. Due to the similarities but simpler of nature of the former it was the FOL based Zermelo-Fraenkel set theory that became the more prevalent.[5, p. 478]

The work of Alfred Tarski was another influential logician whose work had wide reaching significance, with his prestigious accounts of the concepts of truth and logical consequence perhaps being his most famous.

Aristotle's students Theophrastus and Eudemus continued Aristotle's work and made improvements to Aristotelian logic.

1.3 FOL explained

1.3.1 Syntax

Constants

Constants are analogous to names, however, each constant in FOL must refer to precisely one object. The reverse is not true, since for each object there may be 0 or more constants that refer to that particular object.

Functions

While constants allow us to refer to single objects, functions allow us to refer to a single object by relating it to 0 or more objects. The number of arguments a function takes is known as the arity of the function. The arity of each function is fixed so it cannot take differing numbers of arguments.

For instance a function may be: motherOf(kate) which takes a single argument (arity of the function is 1), in this case kate, and returns a single object which has the relationship of being the mother of kate. By extension to talk about kate's grandmother you can apply this function twice: motherOf(motherOf(Kate))

Note that since a function must refer to precisely one object it would be harder to define a grandmotherOf(x) function because any person can have two grandmothers.

If the arity of a function is 0 then the function is simply just a constant.

Terms

Terms also represent objects and are defined recursively as a variable or a function of 0 or more terms. In Bakus Naur Form this term is defined as:

$$t := x \mid f(t_1, .., t_n)$$

where x is a variable and f is a function of arity $n >= 0$

Predicate

Each predicate takes a fixed number $n >= 0$ arguments, where n is said to be the arity of the predicate. Let us consider three cases:

A predicate of arity 0 is a proposition which can represent some statement which can either be true or false.

A predicate of arity 1 represents a property about that particular object where the truth value of the predicate applied to this object represents

whether or not this object has this property. For example the predicate: Male(x) takes an arguement x and is true if and only if x is a male.

A predicate of arity $n >= 1$ represents some relationship between these n objects. Similarly truth value of the this predicate depends on whether the relationship between n objects holds or not.

Identity

While identity can be included as a predicate such as Equality(x,y), here it shall be included as a logical operator. The infix identity operator will be used as such: x=y means that the members x,y of the domain of discourse represent the same object.

Logical Connectives

FOL has a number of binary operators which can be used to connect two sentences together to create a single compound sentence. As well as the binary connective there is also the negation symbol which is a unary connective which, despite being a connective, is only applied to a single sentence. Each of these connectives are truth functional connectives, that is to say that the truth value of a sentence made from logical connectives relies only upon the the truth value of its constituent parts.

Conjunction: $\wedge$

Analogous to the English word 'and'. The sentence P$\wedge$Q is true if and only if P is true and Q is true.

Disjunction: $\vee$

Analogous to the English word 'or'. The sentence P $\vee$ Q is true if and only if P is true or Q is true. Note that this is an inclusive or, when both P and Q are true then P V Q is true.

Negation: $\neg$

Analogous to English word 'not'. The sentence $\neg$P is true if and only if P is false.

Material Implication: $\Rightarrow$

/* Material implication is a connective which is not intuitive to most people when they first come to learn logic.*/

Material implication can be thought of as analogous to the English words 'only if' although the meaning of this is far from intuitive to many people when they first come to study logic. An alternative and perhaps more intuitive way of reading P $\Rightarrow$ Q as English is 'if P then Q'. The sentence P $\Rightarrow$ Q is true if and only if P is false or Q is true. Hence it is false only in the case when P is true and Q is false and true in all three of the remaining cases.

Material Biconditional: $\Leftrightarrow$

Material biconditional is analogous to the English words 'if and only if'. The sentence P $\Leftrightarrow$ Q is true if and only if P and Q have the same truth value. Another way of looking at this symbol is by thinking it is true if and only if P $\Rightarrow$ Q is true and Q $\Rightarrow$ P is true.

Quantifiers

Quantifiers allows us to express sentences where a quantity of objects satisfy a formula. A quantifier alongside a variable is applied to a formula and all free occurrences of this variable within the formula are bound to this quantifier. A variable is free if and only if there is no quantifier to which the variable is already bound by. A quantifier may be only be applied to part of a sentence of FOL, the part of the sentence that the quantifier is applied to is known as the scope of the quantifier. Only variables within the scope of a quantifier can be bound and variables outside remain unaffected.

Multiple quantifiers can be used within a formula. Using multiple quantifiers and using them in combination with other operators permits the expression of sentences where a specific number of objects are specified such as $\exists$x $\exists$y (P(x) $\wedge$ P(y) $\wedge\neg$ (x=y)). This sentence expresses that there are at least two different objects that satisfy the predicate P.

Universal Quantifier: $\forall$

The universal quantifier refers to all objects within the domain of discourse. When we use use English words such as 'all' and 'every' then we would use the universal quantifier to represent this in FOL. When reading FOL it is common $\forall$x as 'for all x'. A sentence using a quantifier such as $\forall$x P(x) is true if and only if P(x) is true for every single object in the domain of discourse.

The universal quantifier is often used in conjunction with material implication. Using universal quantification along with material implication allows us to express for all objects if the object satisfies predicate P then the same object will satisfy predicate Q. For example ∀x (Human(x) ⇒ Mortal(x)) expresses that anything that is human is also mortal.

Existential Quantification: ∃

The existential quantifier refers to at least one object in the domain of discourse. The phrases in English that are can be captured most closely by the existential quantifier are 'at least one' and 'something'. The FOL symbols ∃x is usually read as 'there exists x'. A sentence such as ∃x P(x) is true if and only if there is at least object in the domain of discourse where P(x) is true.

The existential quantifier is often used together with the conjunction symbol. The combinations of conjunctions and existential quantifiers allow us to express sentences such as there exists an object x which satisfies predicate P and predicate Q. An example of this is: ∃ x (Large(x) ∧ Green(x)) which expresses there is some object which is both large and green.

Well Formed Formulas

A Well Formed Formula(WFF) is also defined recursively, using what we have so far a formula ϕ is defined:

$$\phi := Predicate \mid t_1 = t_2 \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \forall x\phi \mid \exists x\phi$$

As can be seen the base formula is either a predicate or an identity and all composite formulas are made by applying either a quantifier to a WFF , a negation to a WFF, or a binary logical connective two WFFs.

1.3.2 Deductive System

A deductive system is a purely syntactical way to demonstrate that one formula logically follows from another. Due to deductive systems being syntactical, sound arguments are correct irrespective of interpretation. There are many deductive systems but the one that shall be used in this project is the system of natural deduction.

The system of natural deduction was made as an attempt to replicate the natural way of thinking both in everyday situations and in informal proofs. It is defined it terms of rules of inference where from a set of formulas given

as the premises a conclusion can be derived by a finite number of applications of these rules.

Natural deduction is both a sound and complete deductive system. In simplified terms soundness means that from any proof in the deductive system there is no possible interpretation where the premises are all true and the conclusion is false. Complete means that for any situation where a statement is a logical consequence of a group of premises then there exists a proof in the deductive system of this statement from these premises.

1.3.3 Rules

Conjunction Introduction

Conjunction introduction takes two premises and concludes the conjunction of the premises.

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \wedge intro$$

Conjunction Elimination

The rule for Conjunction Elimination is split into two parts Conjunction Elimination 1 and Conjunction Elimination 2. Both of these rules take one sentence as a premise, this sentence being a conjunction of two sentences. Conjunction Elimination 1 concludes the first part of the conjunction while Conjunction Elimination 2 concludes the second part of the conjunction.

$$\frac{\phi \wedge \psi}{\phi} \wedge elim_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge elim_2$$

Disjunction Introduction

Disjunction Elimination also consists of two parts. Both take a single sentence as a premise and concludes a disjunction of both the premise and an arbitrary sentence. The first part puts the premise as the first item in the disjunction while the second puts the premise as the second part of the disjunction.

$$\frac{\phi}{\phi \vee \psi} \vee intro_1 \qquad \frac{\psi}{\phi \vee \psi} \vee intro_2$$

Disjunction Elimination

Disjunction elimination take three premises: a disjunction and two subproofs. Each subproof start with one of the elements of the disjunction and finish with the same conclusion, say χ. Since we have a disjunction and from assuimg either of the constituents of the disjunction we can can conclude χ then we can can conclude χ.

$$\frac{\phi \vee \psi \qquad \boxed{\begin{array}{c}\phi \\ \vdots \\ \chi\end{array}} \qquad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}}{\chi} \vee elim$$

Implies Introduction

Starting from a subproof which we assume one senctence and can conclude another means that if we have the first we can conlude the second. Hence we can conclude that the first sentence implies the second.

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \chi\end{array}}}{\phi \Rightarrow \psi} \Rightarrow intro$$

Implies Elimination

From $\phi \Rightarrow \psi$ and ϕ, clearly by combining these two we can conclude ψ. Also know as Modus tollens.

$$\frac{\phi \quad \phi \Rightarrow \psi}{\psi} \Rightarrow elim$$

Negation Introduction

If by assuming a certain sentence we can show that a false conclusion follows then we can conclude the negation of the sentence. Also known as Reductio ad absurdum.

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \bot\end{array}}}{\neg\phi} \neg intro$$

Negation Elimination

If we have both a sentence and its negation then this is a contradiction and we may conclude $\bot$.

$$\frac{\phi \quad \neg\phi}{\bot}\ \neg elim$$

Double Negation Introduction

Negating a sentence twice has no effect so from we may add two negations symbols to the start of any sentence.

$$\frac{\neg\neg\phi}{\phi}\ \neg\neg intro$$

Double Negation Elimination

Similarly we may remove two negation symbols from the start of a sentence since it has no overall effect.

$$\frac{\phi}{\neg\neg\phi}\ \neg\neg elim$$

Bottom Elimination

Form a false premise any conclusion is valid.

$$\frac{\bot}{\phi}\ \bot elim$$

Copy

A simple rule whose premise and conclusion are the same. Can be useful in proofs to use at the end of a subproof to conclude something which has previously been shown.

$$\frac{\phi}{\phi}\ copy$$

Equals Introduction

We may naturally conclude that any object is equal to itself from no premises at all.

$$\frac{}{t = t} = intro$$

Equals Elimination

If we have to equal objects with different names then we may replace any occurrences of one name with the other without changing the overall meaning of the sentence.

$$\frac{t1 = t2 \quad \phi[t_1/x]}{\phi[t_2/x]} = elim$$

Universal Quantifier Introduction

If we introduce a previously unseen variable and can deduce a sentence which contains this variable, then by the fact this this variable is arbitrary, we may conclude that this sentence holds when for all objects in the domain of discourse.

$$\frac{\boxed{\begin{array}{c} x_0 \\ \vdots \\ \phi[x_0/x] \end{array}}}{\forall x \phi} \forall intro$$

Universal Quantifier Elimination

If we have a universally quantified statement we may replace all occurrences of the bound variable by any term.

$$\frac{\forall x \phi}{\phi[t/x]} \forall elim$$

Existential Quantifier Introduction

From a sentence that holds for some term then we my conclude that there exists an object where this sentence holds.

$$\frac{\phi[t/x]}{\exists x \phi} \exists intro$$

Existential Quantifier Elimination

The rule of elimination for the existential quantifier requires two premises. Firstly, a statement which is existentially quantified and secondly, a subproof.

This rule has the reasoning that if we know that something does exist, we can give a temporary name to this object, from here we can use this temporary name to reason about it. If we can deduce another statement not including this name then we can conclude this statement. This reasoning is valid as long as the name is completely arbitrary and does not occur elsewhere within the proof.

$$\frac{\exists x\phi \qquad \boxed{\begin{array}{cc} x_0 & \phi[x_0/x] \\ & \vdots \\ & \chi \end{array}}}{\chi}\ \exists elim$$

Chapter 2

Project management

The project was approached using agile methodologies as opposed to the traditional waterfall approach. A main feature of the the approach used in this project was that of test driven development.

2.1 Test Driven Development

2.1.1 Process

The stages of this process are as follows:

Write a test

Firstly a list of tasks needs to be created. This list is made by analysing the requirements of the system and creating a specific task for each feature of the system.

If there is already a list of tasks then choose one and write a test that can only be passed when the system can carry out the feature as described by the task.

Run the test suite

The entire test suite should be run at this stage with the desired result of all tests passing except for the newly composed test. The new test should fail since the functionality it is testing should have been implemented at this point. If the test passes then either the functionality was added before the test created which is not as intended in the TDD process or it may mean the test is faulty. The test should be reviewed in this case.

If all tests except the new one passes then we can be sure at least that the new test requires some change to the code. This helps reduce the risk of writing tests that pass trivially, which can lead to falsely believing that code is providing functionality that it is not.

Code

In this stage code is written with the aim of passing the test only. No extensions or further functionality should be provided at this point. If the developer becomes aware of the need for expansion then this should be added as a new task. This helps ensure test coverage for all the code.

Run test suite again

The whole test suite is ran again with the aim of all tests passing. If any tests fail then return to the previous step and modify the code. This is repeated until all tests pass.

When all tests are passing then return to stage 1 and choose the next task if any remain.

2.1.2 Advantages of Test Driven Development

- One of the advantages of writing tests before writing code is that you are testing that the code meets the specification. When tests are created after code is written then there is a chance that the tests will only check that the code functions correctly but not that it meets all the requirements. In the case of TDD great care must be taken with writing the tests so that they accurately reflect the specification and requirements of the system; providing this is the case you can have confidence that the code meets these requirements.

- Due to nature of only writing code to pass tests then this inherently leads to good code coverage since all sections of code are covered by at least one test.

- Another advantage of TTD is that the passing tests clearly show which tasks have or have not yet been completed. This gives a very good way of to accurately monitor progress throughout development.

- Finally, as tests are wrote a test suite is built up which is run whenever code is changed. This means that unexpected affects of a code change are picked up straight away. This can save a lot of time in debugging as

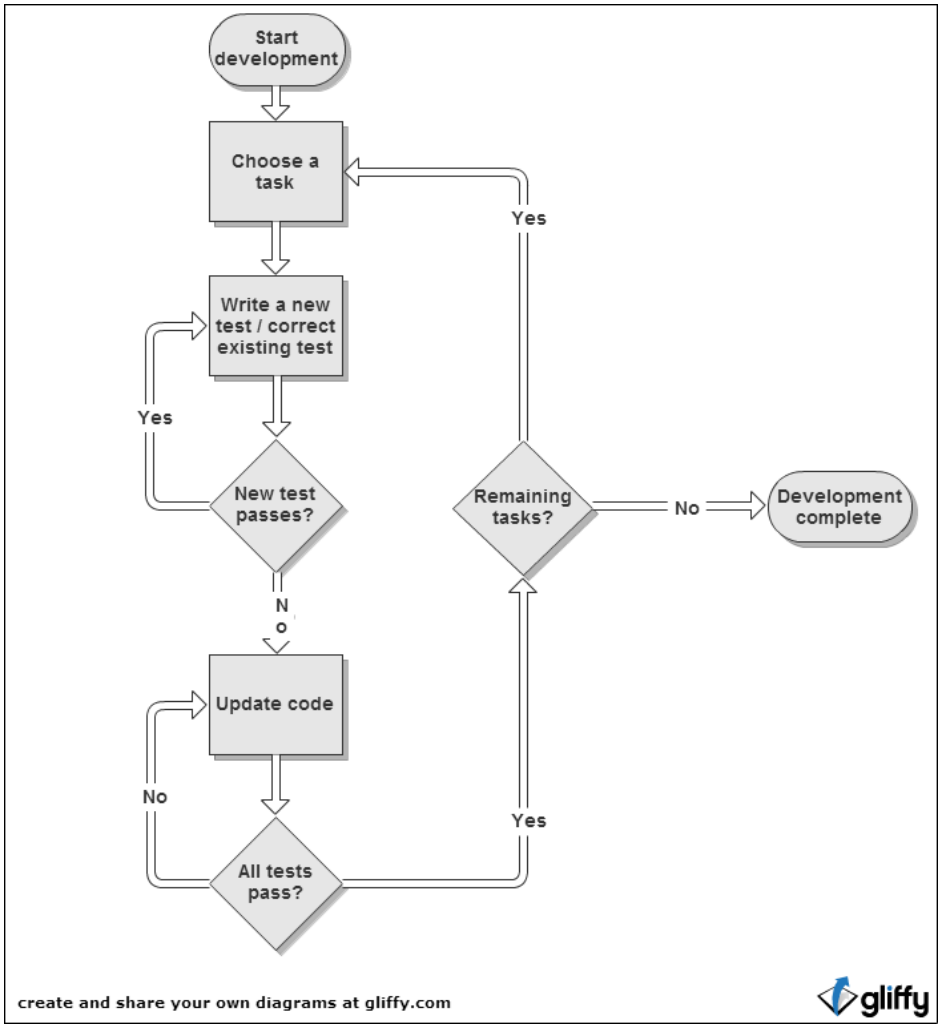

Figure 2.1: Flowchart of TDD process

if it was not picked up straight away it can become very time consuming to trace the source of errors especially if they cause error in other areas which mask the original source of the error.

2.1.3 Tools used

To assist the in the process of TDD I utilised the tools of JUnit and Robotium. JUnit is a testing framework for the Java programming language which assists in the creation and running of automated unit tests. Robotium is a testing framework which facilitates the creation of automated system tests and acceptance tests for the Android platform.

Using a combination of these tools I was able to create a multitude of tests which test individual components and also the interaction between these components. Robotium enabled me to create automated acceptance which could carry out tasks from start to finish such as building an entire proof, simulating the use of the graphical user interface by activating button press and touch screen events. Since Robotium simulates using the user interface I was able to test that not only did the code function correctly but also that users would be able to actually access and use the functions that are offered. Without such automated testing of the graphical features, testing could have become a very time consuming process if it were to be repeated after all code changes as these automatic tests were.

2.2 Prioritisation

Tasks were chosen using the agile principle of structuring and completing tasks in such a way that gives highest value whilst attempting to keep the time frame of delivery small. Use of this principle has a number of advantages. One of these advantages is the modularity of the code developed. Since code is split into tasks with each adding a specific a function then code is less dependent on others parts of the system. This reduces complexity, makes testing easier and makes code more reusable.

Another way this principle was adhered to was by developing code in such a way that at key points the application was operational, but only implementing a subset of the desired functional requirements. An example of this was the design choice of implementing all the necessary requirements so that the system could be fully operational for the language of Propositional Logic before extending it to FOL. The advantages of this are threefold:

1. In a business situation a product can be delivered to end users at a much earlier date, with non critical functionality being added incrementally.

2. This working application can be used as a prototype. It can be used by people not involved with the development to gain important feedback at an earlier stage than would normally be possible. The feedback may uncover some changes that would improve the user experience, again due to the product being in development it is easier to make changes at this point than it would have been after the project had been fully completed.

3. In the worse case scenario, should unexpected problems arise at a later point which prevents completion of the system then there is still a

working application. Despite not meeting all the desired requirements this is certainly better than an application that has attempted to meet all desired requirements but is not actually in a usable state.

2.3 Project management tools

2.3.1 Trello

Following the initial design phase I made use of a web-based project management tool to help facilitate the development process. The tool I used was Trello which follows the Kanban paradigm for software development, a well known agile methodology. Kanban enabled me to visualise the workflow of the project using coloured cards and lists which represented tasks and stages in the workflow respectively. In addition to the tasks, I also created a separate list for bugs. This enabled me to document bugs as soon as they were found without distracting too much from the current task at hand. Bugs were sorted by priority so that the most important issues would be dealt with first.

For each objective I would start by breaking it into smaller tasks. These tasks were then put onto a to-do list on Trello. As each task was then picked for development it would be moved along the various lists each one representing a stage in the TDD process described earlier. In addition to the stages described above each task would enter a penultimate stage of further testing; previous tests showed that this task is functioning as required so tests in this stage focused on unusual input intended to cause errors. Finally, when this testing is complete and any discovered errors fixed, the task can be moved into the completed list.

2.3.2 Git

Git is a source code management system. One of the benefits of using Git was that all code and documentation could quickly and easily be uploaded to a remote repository. Since it did not take much time I would upload the latest version after every time a new test was created or passed. This remote repository was used as a backup, so should any data be lost or become inaccessible it could easily be recovered from the most recent version.

The version control aspect of this system also meant that previous versions can be restored. In cases where a bug is accidentally introduced, especially during refactoring, it may be quicker to restore to a previous version rather than spend time debugging the program.

2.4 Justification of the choosing the Android platform

In this section I will discuss the thought process and reasons for choosing to build an application for the Android platform. The reasoning here shall be split into three main choices.

2.4.1 Choosing a mobile solution

One of the first major design choices in this project was to implement the solution for use on a mobile device. Most people find the best way to learn something new is by doing. My vision for this project was to create a tool that would facilitate interactive learning in a classroom environment. My aim is that the teacher will be able to set an exercise such as asking the students to prove a statement from the given premises. Each student will then be able to tackle the problem at their own pace receiving feedback from the application as they proceed. I decided the best way to achieve these goals was to design a mobile solution.

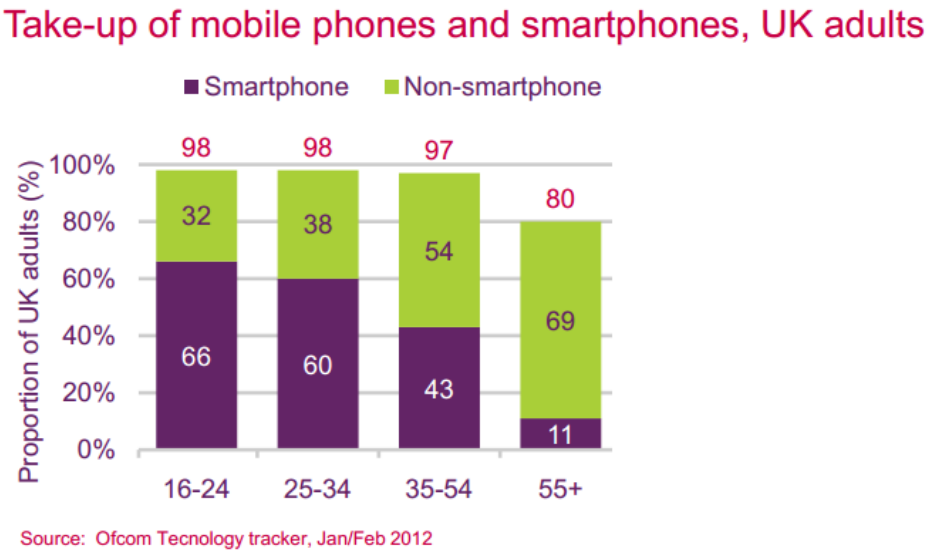

Figure 2.2: Smartphone ownership by age

The graph shows that two thirds of people, who are the typical university student age, own smartphones. Furthermore, a fifth of these own a tablet computer. [10] The ownership of mobile technology such as smartphones and tablet computers has been dramatically increasing and it appears that this

trend will continue. This increasing prevalence of mobile computing devices in our lives contributed to the choice of developing a mobile solution.

2.4.2 Choosing a native solution

The next big choice was to decide whether the implementation should be a mobile web application or a native application. A native application is an application that is wrote to work on a specific platform, whereas a mobile web application can be accessed by multiple platforms using the internet. Of course both solutions have their advantages and disadvantages. The progress being made using HTML5 has meant the distinction between these alternatives have become increasingly vague, even during the time of this project. However, here are some of my resons for the choice I made:

One advantage of the native application is that they are generally faster than their web app counterparts.

After the initial download of a native app, access to the internet is no longer necessary. This may be important in classroom situation where internet connection is not available.

Another advantage of using the native app approach is that they are easy and quick to access. At a click of an icon a user can almost instantly start to use a native app.

Finally, native application development is targeted specifically for mobile devices with smaller screens. This makes it easier to create a mobile device interface which utilise features such as touch screen.

2.4.3 Choosing the Android platform

The choice of Android platform was made based mainly on two reasons:

Firstly, as a developer I was already vaguely familiar with Android development and Java , which is the primary language used in Android development. This familiarity and knowledge allowed me to be asses more accurately, compared to other platforms, as to what would and would not be possible.

The second and more crucial reason, is the dominance that Android currently holds over the mobile computing market. Figure 2.3 shows the operating systems of mobile devices.

Android has a clear dominance holding almost 70% of the market share. Since people are most likely to own a device that runs on Android, this was the platform I decided to develop the application for.

Figure 2.3: Mobile operating system by market share

Chapter 3

Formula checker

The first development section of the project to be completed was the formula checker. The formula checker takes as input input a sentence of predicate logic and checks whether this sentence is a well formed formula.

3.1 User interface

I started with the design of the user interface. I specifically decided to design the user interface first as, due to the nature of the project as a learning tool, the user experience was a priority. By designing the user interface first the underlying representation can then be designed to reflect the interface rather than the other way around. This allowed me to create an intuitive and user friendly interface which gives the features that need to be implement.

One of the first problems that I had to overcome was that many of the symbols of FOL are not available on the standard input keyboard, so it was necessary to provide the user with a different method of input. The user interface created is similar to that of a calculator with a display, a button for each of the logical operators and a backspace button.

In addition to these buttons there is a also a button for the user to add a predicate. Upon pressing the button the user is presented with a dialog box and they can use the soft (on-screen) keyboard input to name a predicate and specify it's arguments. Once the user has done this they can click the OK button to confirm. The predicate's name and arguments will be validated at this point. If validation fails then an error message is displayed explaining why the name/ arguments given are not acceptable and the user is then given the opportunity to amend the input. When the validation is passed this predicate will be added to the list of predicates. Since predicates are often used more than once during a proof, the list of predicates provides

a convenient way for users to input previously used predicates by simply selecting the desired predicate from the list.

The process of naming predicates is purposely separated from entering the logical symbols of FOL. This feature was chosen to limit the number of input methods available to the user at any one point. This simplifies and enhances the usability of the application.

A token representing an operator or a predicate will be generated each time a user touches the various operator buttons or adds a predicate to the input sentence. These tokens are stored within a list and can be removed by using the backspace button at the appropriate position. When the user chooses to submit the sentence these tokens will be sent to the parser for evaluation.

3.2 Parser

explain CFG.

Parsers may be separated into one of two categories: top-down and bottom. Top-down parsers work by making predictions about which of the productions to use at each stage, starting with the most general and ending with the most specific i.e. terminal productions. If the next token in the stream contradicts the prediction then the parser must backtrack until a production is found that is not contradicted by the tokens read. If no such production can be found then the input is not part of the language recognised by the parser.

Bottom-up parsers take the opposite approach, reading the input and collecting them into higher level tokens as per the productions. At the end all tokens should have been collected into the single start symbol. If this is not the case then the parser will backtrack and combine tokens in different ways. If there is no way of combining the tokens so that it ends as a single start symbol then the input is not recognised by the parser.

Top-down parsers are generally less complex and easier to debug than their bottom-up counterparts. Also, if designed well, they are as least as fast as an equivalent bottom-up parser in most cases. [4] However, top-down parsing has the problem that if a grammar is left recursive then this can cause an infinite loop, meaning parsing will never finish. Furthermore, backtracking in bottom-down parsing is slow, time complexity can be exponential. Based on the analysis of both approaches it was decided that the top-down approach would be best, provided the problems caused by left recursion and backtracking could be overcome.

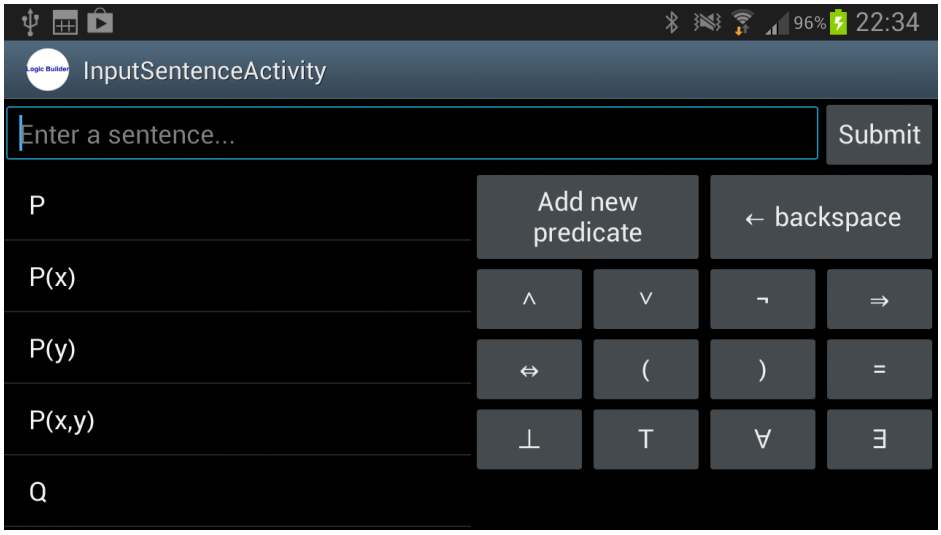

Figure 3.1: User interface for formula checker

3.2.1 Predictive parsing

Both disadvantages of top-down parsing mentioned in the previous section can be avoided by creating a predicative parser. This is only possible if the grammar can be transformed into a grammar that belongs to the class LL(k), for some finite integer k. An LL grammar is one which is can be parsed from left to right constructing leftmost derivations. LL(k) is the class of LL grammars which can deduce the production to use looking only at the next k tokens of input.

It is possible to define the a property that is both necessary and sufficient for a grammar to be in LL(1). Some definitions are needed first:

Definition. *The set FIRST(α) is the set of terminal symbols that can appear as the first word in some string derived from α.*

Definition. *The set FOLLOW(α) is the set of words that can occur immediately after α in a sentence.*

Definition.

$$First^{+}(A \to \beta) = \begin{cases} FIRST(\beta) & \text{if } \epsilon \notin FIRST(\beta) \\ FIRST(\beta) \cup FOLLOW(A) & \text{otherwise} \end{cases}$$

A grammar can be shown to be in LL(1) iff for all productions of the form A $\to \beta_1 \mid \ldots \mid \beta_n$ we have:

$$First^{+}(A \to \beta_i) \cap First^{+}(A \to \beta_j) = \emptyset \quad \forall\, 1 \leq i, j \leq n,\ i \neq j \tag{3.1}$$

Definitions and further details about the theory of parsing can be found in Engineering a Compiler by Cooper and Torczon. [4]

As defined earlier, the structure of a well formed formula is:

$$\phi := Predicate \mid t_1 = t_2 \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \forall x \phi \mid \exists x \phi$$

3.2.2 Operator precedence

A decision was made to allow the user to omit unnecessary brackets by using the precedence rules: $\neg$ binds more tightly than $\wedge$ and $\vee$, and the latter two bind more tightly than $\Rightarrow$ and $\Leftrightarrow$. Also all binary operators are taken to be right associative so P $\diamond_1$ Q $\diamond_2$, where $\diamond_1$ and $\diamond_2$ are binary operators from the same precedence level, should be interpreted as (P $\diamond_1$ Q) $\diamond_2$ R not P $\diamond_1$ (Q $\diamond_1$ R).

To express the rules of hierarchy a context-free grammar was created in which non-terminals were added so that reflects the precedence hierarchy of the operators. The idea is simply that each sub-formula of an expression cannot contain an operator of lower precedence unless this sub-formula is surrounded by brackets.

$$\begin{array}{lcl} \phi_1 & \rightarrow & \phi_1 \Rightarrow \phi_1 \quad | \quad \phi_1 \Leftrightarrow \phi_1 \quad | \quad \phi_2 \\ \phi_2 & \rightarrow & \phi_2 \wedge \phi_2 \quad | \quad \phi_2 \vee \phi_2 \quad | \quad \phi_3 \\ \phi_3 & \rightarrow & Predicate \quad | \quad (\phi_1) \quad | \quad \neg \phi_3 \quad | \quad \forall Var \, \phi_3 \\ & & | \quad \exists Var \, \phi_3 \quad | \quad Var = Var \end{array} \tag{3.2}$$

3.2.3 Eliminating left recursion

Grammar 3.2 does not meet property 3.1 for a number of reasons. This particular grammar is both left and right recursive and so is ambiguous. Ambiguous in this sense means that a string can have more than one distinct parse trees. As mentioned, left-recursive grammars can cause top-down parsers to enter infinite loops.

Left recursion is also a specific case of a FIRST/FIRST conflict. The reason why is fairly simple. Take any left-recursive production, there is at least one alternative production or the recursion is endless. Clearly, since the left-recursive production has the symbol of the left hand side as the first symbol on the right hand side then it's FIRST set is the union of the FIRST set of all alternative productions. Since the left-recursive production shares

elements of it's FIRST set with alternative productions then it is not possible to know which production to use by only looking at the next token of input.

Many FIRST/FIRST conflicts can be solved using left factoring which is simply the process of taking a non-terminal with multiple productions, factoring out shared prefixes of the alternate productions and inserting a new non-terminal which has the suffixes as it's alternatives. This process does not always produce an LL(k) grammar since not all grammars have an equivalent LL(k) grammar.

The problem of left recursion can be solved since there exists an algorithm to convert a left recursive grammar into a weakly equivalent right-recursive grammar. Here, weakly equivalent, means that the language they generate are the same. This does not, however, mean that they have the same derivation trees.

Let $a_1, a_2, ..., a_n$ and $b_1, b_2, , b_m$ be sequences of terminals and non-terminals where b_1, b_2, b_m do not start with the non-terminal A. Then for a production of the form:

$$A \quad \rightarrow \quad Aa_1 \mid Aa_2 \mid \ldots \mid Aa_n \mid b_1 \mid b_2 \mid \ldots \mid b_m$$

We can change this to:

$$\begin{aligned} A &\rightarrow b_1A' \mid b_2A' \mid \ldots \mid b_mA' \\ A' &\rightarrow a_1A' \mid a_2A' \mid \ldots \mid a_nA' \mid \end{aligned}$$

The result of eliminating left-recursion from grammar 3.2 is:

$$\begin{array}{lll} \phi_1 & \rightarrow & \phi_2\ \phi_1' \\ \phi_1' & \rightarrow & \Rightarrow \phi_1\ \phi_1' \quad \mid \quad \Leftrightarrow \phi_1\ \phi_1' \quad \mid \quad \epsilon \\ \phi_2 & \rightarrow & \phi_3\ \phi_2' \\ \phi_2' & \rightarrow & \wedge\ \phi_2\ \phi_2' \quad \mid \quad \vee\ \phi_2\ \phi_2' \quad \mid \quad \epsilon \\ \phi_3 & \rightarrow & Predicate \quad \mid \quad (\ \phi_1\ ) \quad \mid \quad \neg\phi_3 \quad \mid \quad \forall\, Var\ \phi_3 \\ & & \mid \quad \exists\, Var\ \phi_3 \quad \mid \quad Var = Var \end{array} \tag{3.3}$$

3.2.4 Resolving FIRST/FOLLOW conflicts

Grammar 3.3 now has what can be referred to as a FIRST/FOLLOW conflict. FIRST(ϕ_1) = $\{\epsilon, \Rightarrow, \Leftrightarrow\}$ and FOLLOW(ϕ_1) = $\{\ \Rightarrow, \Leftrightarrow\}$. This creates a conflict on input $\Rightarrow$ or $\Leftrightarrow$ because ϕ_1 could be matched by one of these or it could be matched with the empty string and the operator could then be

matched the following production. A similar FIRST/FOLLOW conflict is seen for ϕ_2.

This conflict may be resolved by changing each ϕ_1, on the RHS of the productions for ϕ_1', to the production that is one level higher in the precedence hierarchy i.e. ϕ_2. Similarly, change each ϕ_2, on the RHS of the productions for ϕ_2' to ϕ_3.

Grammar 3.4 shows the result after this conflict has been resolved:

$$\begin{array}{lcl}
\phi_1 & \rightarrow & \phi_2\ \phi_1' \\
\phi_1' & \rightarrow & \Rightarrow \phi_2\ \phi_1' \quad | \quad \Leftrightarrow \phi_2\ \phi_1' \quad | \quad \epsilon \\
\phi_2 & \rightarrow & \phi_3\ \phi_2' \\
\phi_2' & \rightarrow & \wedge\ \phi_3\ \phi_2' \quad | \quad \vee\ \phi_3\ \phi_2' \quad | \quad \epsilon \\
\phi_3 & \rightarrow & Predicate \quad | \quad (\ \phi_1\ ) \quad | \quad \neg\phi_3 \quad | \quad \forall\, Var\ \phi_3 \\
 & & \quad | \quad \exists\, Var\ \phi_3 \quad | \quad Var = Var
\end{array} \tag{3.4}$$

Another way to fix the conflict would have been removing each ϕ_1 from the RHS of ϕ_1'. Although slightly obscured by the left-factoring, the productions ϕ_1 and ϕ_2 would then become right-recursive; more specifically they would have indirect right recursion.

Consequently the operators $\Rightarrow, \Leftrightarrow, \wedge$ and $\vee$ would become right associative. So a sentence such as P $\vee$ Q $\vee$ R would be accepted by the parser and interpreted as P $\vee$ (Q $\vee$ R), the opposite of what is wanted.

If both of the changes that were mentioned were implemented the productions would then become non-recursive, meaning that a string such as P $\vee$ Q $\vee$ R would not be recognised by the grammar; brackets would need to be in appropriate places to be recognised by the grammar. This is certainly a usable grammar for FOL. Nevertheless, the aim was to create a more flexible grammar than would infer meaning of using left associativity of binary operators, so it is the alternative that shall be used.

The difference between the grammar 3.4 and the one mentioned above may be subtle but has quite a big effect. Strings of the form P $\diamond_1$ Q $\diamond_2$ R, where $\diamond_1$ and $\diamond_2$ are binary operators from the same precedence level, are now recognised by the grammar, in fact strings of arbitrary length will be accepted provided they keep this structure. No associativity is imposed on the operators at this stage, in the sense that P $\diamond_1$ Q $\diamond_2$ R is not equivalent to either P $\diamond_1$ (Q $\diamond_2$ R) or (P $\diamond_1$ Q) $\diamond_2$ R. This is, nonetheless still progress, as this can easily be corrected when the abstract syntax tree is constructed, which is discussed in section 3.2.7.

3.2.5 LL(1) grammar

It will now be shown that grammar 3.4 belongs to the class LL(1). The end of file will be represented by the symbol \$.

The non-terminal ϕ_1' has three alternative productions, the FIRST$^+$ set for each of the productions is given below:

$$\begin{aligned}
&\text{FIRST}^+(\phi_1' \rightarrow \ \Rightarrow \phi_2\phi_1') = \{\Rightarrow\} \\
&\text{FIRST}^+(\phi_1' \rightarrow \ \Leftrightarrow \phi_2\phi_1') = \{\Leftrightarrow\} \\
&\text{FIRST}^+(\phi_1' \rightarrow \ \epsilon) = \{\epsilon, \$,)\}
\end{aligned}$$

Each of these sets are disjoint, so the intersection of any two of these sets is the empty set. Similarly, ϕ_2' also has three alternative productions, the FIRST$^+$ set for these are:

$$\begin{aligned}
&\text{FIRST}^+(\phi_2' \rightarrow \ \vee\phi_3\phi_2') = \{\vee\} \\
&\text{FIRST}^+(\phi_2' \rightarrow \ \wedge\phi_3\phi_2') = \{\wedge\} \\
&\text{FIRST}^+(\phi_2' \rightarrow \ \epsilon) = \{\epsilon, \$, \Rightarrow, \Leftrightarrow,)\}
\end{aligned}$$

Again, the intersection between any pair of these is the empty set. Finally, the only remaining non-terminal with more than one RHS is ϕ_3, which has 5 alternate productions. The FIRST$^+$ set is given for each:

$$\begin{aligned}
&\text{FIRST}^+(\phi_3 \rightarrow \text{Predicate}) = \{\text{Predicate}\} \\
&\text{FIRST}^+(\phi_3 \rightarrow (\phi_1)) = \{(\} \\
&\text{FIRST}^+(\phi_3 \rightarrow \neg\phi_3) = \{\neg\} \\
&\text{FIRST}^+(\phi_3 \rightarrow \forall\text{Var}\phi_3) = \{\forall\} \\
&\text{FIRST}^+(\phi_3 \rightarrow \exists\text{Var}\phi_3) = \{\exists\}
\end{aligned}$$

We can see that these 5 sets are all disjoint. So there is no overlap between any FIRST$^+$ sets that have the same LHS. Therefore, grammar 3.4 satisfies property 3.1 and so we can conclude that this grammar belongs to the class LL(1).

3.2.6 Implementation

The next choice that had to be made was how the parser itself was to be implemented. The parser could be manually coded or a parser generator could be used. Parser generators take formal description of a language such as a grammar and generates a parser for that language. Many parser generators exist, so to compare the the different options I looked mainly at the following

areas:

Parser class The classes of parsers than can be generated e.g. LL(1)/ LALR(1).

Output language The language of the source code of the parser that is generated.

Input notation The notation that is used to describe the language.

The class of the parsers than can be generated is clearly vital. This is not too much of a constraint since it is known that the grammar is in the class LL(1) which is a subclass of many of the other grammar classes such as LR(1).

For the needs of this project the output language another important feature since it needs to be deployable on the Android platform. The Android platform runs on Dalvik byte-code but code is generally wrote in Java source code which is compiled to Java byte-code and then translated to Dalvik Virtual Machine compatible byte-code automatically. Of course, other languages could be used but this would introduce an extra translation stage between languages which is an unnecessary complication which adds to the complexity and development time.

Finally, the input notation does not have a strict criteria but should parser generators be equal in other areas then the parser generator with the least complex input notation shall be used.

A manually coded parser could be created in Java that recognises the constructed grammar but this option would require the greatest development time.

After comparing the multitude of parser generators available I opted to use the parser generator JavaCC. JavaCC takes a description of an LL(k) grammar and produces Java source code of a recursive descent parser for that grammar. The description of the grammar is given using Extended Backus-Naur Form. This is similar to the form used so far in this report to describe the different grammars. This simple input notation is a clear benefit of using JavaCC.

Since grammar 3.4 belongs to the class LL(1), JavaCC can generate a predictive parser that has a run time complexity that is linear in terms of the input size.

3.2.7 Abstract Syntax Tree

In conjunction with JavaCC, JJTree a JavaCC pre-processor is used. JJTree allows the user to add annotations alongside productions in the grammar

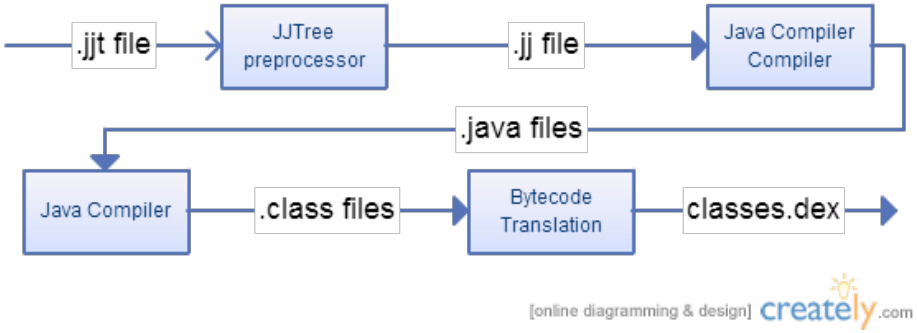

Figure 3.2: Compilation process for FOL parser

which JJTree uses to insert code to the parser that will generate an abstract syntax tree during parsing.

The AST is generated by creating nodes for terminals first, these will be the leaves of the tree. A tree is built bottom-up with child nodes being assigned as each node is created. The last node to be created will be the root node of the tree. Since each node can have at most two children the resulting AST will be a binary tree.

The following table explains the effect each of the terminals has on the AST.

Predicate	Each time the parser encounters a Predicate a node is created representing this terminal and this node is then added to a stack.
$\neg$	Node creation is delayed until the next symbol has been fully expanded. Once this has happened the node on the top of the stack is popped and added as a child to the $\neg$-node which is then itself added to the stack
$\wedge, \vee, \Rightarrow, \Leftrightarrow, =$	As above a node will only be created after the following non terminal has been fully expanded; the two nodes at the top of the stack are then popped and assigned as the children of this node (The grammar guarantees that there will be at least two nodes on the stack at this point). This node is then added to the stack. The process of taking two nodes as soon as the following non-terminal has been fully expanded gives the operators $\wedge, \vee, \Rightarrow$ and $\Leftrightarrow$ the left associativity that is desired. The $=$ operator does not become left associative as the grammar purposely does not allow it.
$\forall, \exists$	A similar approach is taken for $\forall$ and $\exists$. The difference here is that $\forall$ and $\exists$ are always followed by a variable. The value of the variable is stored in the same node as the Quantifier. After this the process is the same as for $\neg$, once the following symbol has been fully expanded, the node at the top of the stack is added as a child to this one before being added to the stack itself.
Variable	If the variable is following a quantifier then the value of the variable is stored with the quantifier. Otherwise a node is created and added to the stack.
(,)	No nodes are created for brackets. Brackets can alter the derivation tree of a string, the information conveyed by any brackets is stored implicitly by the structure of the tree constructed. This has the advantage that regardless of how many brackets the user uses when entering a sentence it will be stored in a consistent manner.

3.3 Bound and free variables

When a sentence is being parsed each of the AST nodes that are created for predicates and variables are linked to Java objects representing a predicate or a variable. This allows two different nodes of the same type to link to

the same object. This reflects the way that the same predicate or variable can appear multiple times in a sentence. This is not necessary for any other symbols since each operator has only one interpretation. From here on the words predicate or variable starting with a capital letter will refer to the Java object while the the lower case version will convey the FOL meaning of the word.

A Predicate has two parts: a name and a list of parameters. The name is stored as a string and each parameter is stored as a Variable.

When a sentence is being parsed, a map of strings to Variables is built. As each predicate is reached the name of each of it's parameters are looked up in the map. If the name is already a key in the map then this argument becomes a reference to the corresponding Variable. If there is no match then a new Variable is created and added to the map. This argument will then refer to this newly created Variable.

When a quantifier is reached a new Variable is created using the variable name given and it is associated with the quantifier.

At the end of parsing all variables in the scope of a quantifier that share the same name as the variable being quantified are bound. This is done as follows: For each predicate in the scope of quantifier, take each argument which has the same name as the quantified variable and set this argument as the Variable associated with the quantifier.

After parsing all variables are either free or bound. If they are free then they point to a Variable that is stored in the map, if they are bound then they point to a Variable associated with a quantifier. This serves as an implicit way of connecting similar Variables. This can be used by rules of inference to easily find all occurrences of a variable, which otherwise would have to traverse the whole tree of a sentence to find each occurrence.

Chapter 4

Proof Builder

4.1 User interface

Donald Norman's research on UI was used. Typical Android application conventions such were followed such as the use and look if dialog boxes. Also the dark background with light font is very common for this platform.

Perceived affordances as described by Norman were implemented in regards to touch screen buttons that users will press and lists that users can push up or down.

In accordance with the project procedure the first part of the proof builder to be designed was the user interface. The user interface was developed by taking simple user stories that described what a user would want to do and attempting to create a graphical interface for each task that would be intuitive to use.

4.1.1 Proof thingy

It was decided that displaying the actual proof was the most important part of the user interface and that this would be given the majority of the screen space. Since space is limited on mobile devices, it was decided that the application would display in landscape mode to enable most sentences of FOL to fit onto one line, enhancing the readability of the proof. The intent was that the proof would resemble a hand written proof of FOL as much as possible.

This proof section of the screen will start off empty and each line will be displayed as it is added to the proof.

In order from left to right each line displays the following:

- The line number of the proof.

- A boxed variable if one has been introduced in this line.
- The FOL sentence , this may be blank if a variable is introduced without an assumption.
- The justification of this line.

When more more lines are added than can fit on the screen then the list of lines becomes vertically scrollable. The user may scroll the list by simply using their finger to push the list up and down. This is an intuitive motion which all users who tested the product used without prompting.

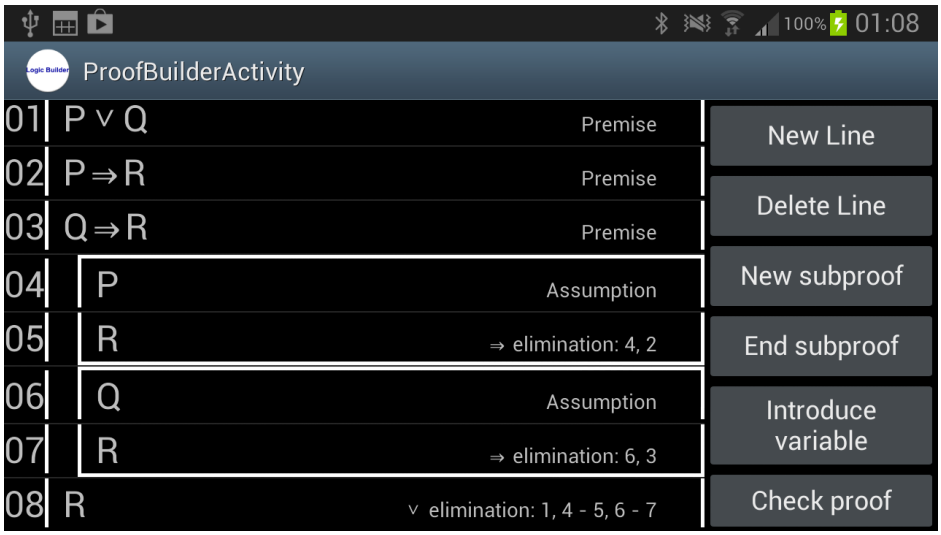

Figure 4.1: User interface for proof builder

4.1.2 Button bar

A section fixed to the right hand side displays a button bar which allows the user to carry out various tasks. The buttons make use of the proximity and similarity to give the user the sense that the buttons are a group of objects, rather than separate entities.

The buttons are listed below along with a description of how the graphical user interface reacts and changes for each.

New line

Selecting the new line button will activate the formula checker activity. This will appear as described in the previous chapter. After the user submits a sentence a new line is added to the end of the proof with the same indentation as the previous line and within any boxes that the previous line was inside of. The justification of the line will be set to 'No justification given'. This serves as a prompt to the user to add a justification,

Delete line

Pressing the delete line button will remove the most recent line of the proof.

New subproof

When the new subproof button is pressed the formula checker activity will be displayed. Upon submitting a sentence a new line will be added to the end of the proof which is indented more than the previous line and the top section of a box will surround it. The justification for this line is set to 'Assumption'.

End subproof

Similarly, pressing the option end subproof will present the user with the formula checker activity to enter a sentence. When a sentence has successfully been submitted the new line will be added at the end of the proof. Additionally the bottom part of the box will be formed and the whole subproof will be neatly enclosed within the box. As with the new line button the justification for this line is set to 'No justification given'.

Introduce variable

Choosing the introduce variable option will display a dialog box. The user will be able to use this dialog box to name a variable that is to be introduced to the proof. Should the user enter a variable name that is not valid such as an empty name then a message will be displayed informing them why the name cannot be accepted and they will then be given the opportunity to enter a different name.

A checkbox also appears within the dialog box. The user may mark the checkbox if they wish to make an assumption as well as introduce a variable. After an appropriate name has been chosen and the user chooses the OK button, the formula checker activity will be displayed if the user has marked the checkbox. Once the user has submitted or sentence, or if the checkbox was not marked, then line will be added to the end of the proof.

The variable will be displayed in a small box. If a sentence was submitted this will be displayed followed by the justification which will be set to the word 'Assumption'. Otherwise, the rest of the line will be left empty. Similar to starting subproof, the whole line is indented more than the previous and the top section of a box will surround it.

Check proof

Pressing check proof triggers the display of a dialog box. This dialog box will inform the user whether the proof entered so far is complete and valid. If not it will inform the user as to why with messages such as 'All subproofs must be closed to check proof' or 'Line x has not been justified', where x is the number of a line of which no justification has been specified. If the proof is complete and valid then the user will presented with a message that confirms that the

premises semantically entails the conclusion e.g for premises $\phi_1, \ldots, \phi_n$ and conclusion ψ the message would read 'Congratulations, you have successfully proved: $\phi_1, \ldots, \phi_n \models \psi$'

4.1.3 Justifying a line

To change the justification of a line in the proof the user simply needs to touch the line that they wish to change. This will bring up a dialog box. A drop-down list contains all the rules of inference. The display will change depending on which ROI is currently selected. The labels serve as clues to what form the premises take for this ROI.

A drop-down list is displayed next to the label for each premise which can be used to specify which line or subproof is to be used. The drop-down list will only contain single line numbers, if the premise requires just a sentence, or a block of line numbers, if this premise needs to be a subproof. Furthermore, only line numbers and whole subproofs that occur after the line being justified appear in the list.

When the user has specified ROI to use and the premises then the OK button must be pressed to continue. If there is a problem with the justification an error message is displayed which explains why it has not been accepted. This is explained further in section 4.3. If the justificationis accpeted then dialog box is removed and the justification of the chosen line is updated with the chosen ROI and the line numbers of the premises used.

As can be seen in figure 4.2 the proof is partially visible when justifying a line. The system was designed this way as when it was not visible users commented that they found it difficult to remember the lines numbers that were to be used as premises for the ROI. Having the proof visible even if partially greatly helped users to identify the correct line numbers that they needed.

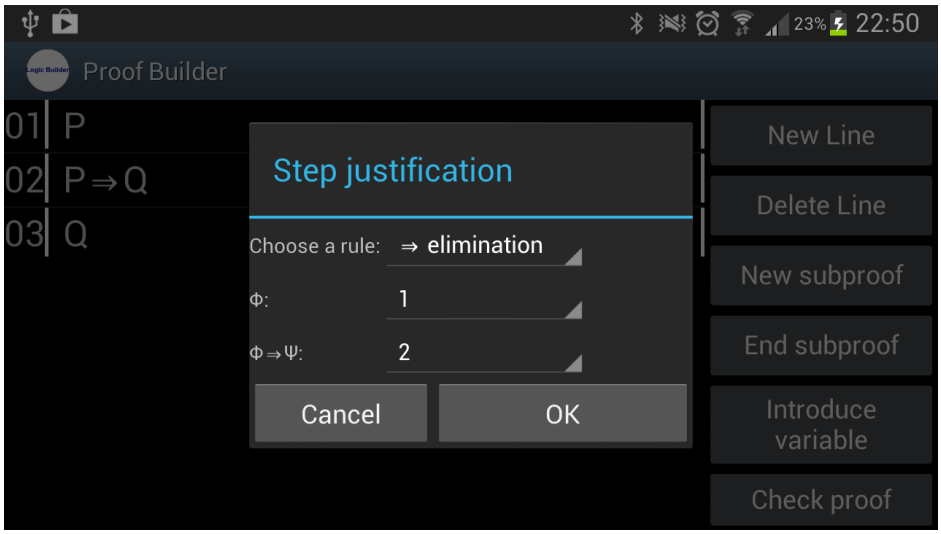

Figure 4.2: User interface for justifying a line

4.2 Rules of Inference

Rules of inference were programmed in the Java language. A method was created for every ROI. This method then takes as arguments the premises of each rule followed by the line that is being justified. The order of the premises that are passed into the method must be the same as defined by the ROI.

As mentioned in the previous section, sentences of FOL are stored in a tree data structure. This is utilised by the rules of inference since all data about about a sentence can be found from only the root node. If a premise is single statement then the root node of the statement should be passed into the method. If a premise is a sub-proof then a list of root nodes should be passed. This list may contain either the whole subproof or only the required parts such as the first and last line. The order of nodes must be the same as the order of the lines in the subproof.

Firstly, each method will check that the premises and the conclusion are of all of the correct form for this ROI. The correct form may be any WFF of FOL or it may be more restrictive such as it has to be a negation of a sentence. If any of the arguments do not match the expected form then an exception is thrown by the method. There are two types of exception that can be thrown: PremiseException and ConclusionException. Intuitively these exceptions relate to the premises or conclusion not being of the required form. Each exception contains a message explaining what the discrepancies between the given arguments and what would be expected.

In most cases, to check whether a sentence is of the correct form only requires looking at the operator with greatest scope. This is handled efficiently by the tree structure since only the root node has to be inspected. Furthermore, finding part of a sentence, such as the LHS or RHS of a binary operator, is also very simple with the tree data structure. This is done by taking the appropriate child node of the operator; this child will be the root node of the sub tree for the part of the sentence required.

Subsequently, the method will then evaluate the ROI on the given premises. If only one conclusion is possible, such as the case of $\wedge$ introduction, then this unique conclusion is determined and compared with the line that is being justified. If they are not syntactically equivalent then a ConlusionException is thrown. This exception contains an error message that details what would have been expected using the chosen ROI on the given arguments. If more than one conclusion is possible then the line that is being justified is checked to make sure that it is a possible conclusion using the chosen ROI and the given premises.

4.2.1 API

The ROI have been coded as a standalone component that may be used by other applications. As such an API has been created that explains how the ROI class can be used. Full technical documentation can be found in the appendix.

Documentation was generated using Javadocs. The API contains:

- A brief description to describe the purpose of the class.
- All publicly accessible fields of the class, along with a description.
- For each method the API may contain:
  - A description of the purpose of the method
  - The name, type and a description of each parameter
  - The type and description of what is returned by the method

Each rule that deduces a unique conclusion from a set of premises, has two similar and related methods. The methods differ in that one method requires a conclusion given as an argument and the other does not. The method which does not take a conclusion will return the unique conclusion that is deduced from the premises.

Below is an example of how the API represents such a case, using overloaded methods:

- *doubleNegationElimination*
  `public static SimpleNode` **doubleNegationElimination**`(`
  `logic.proof.builder.parser.SimpleNode` **p** `)`
  - **Parameters**
    - `p` - The root node of a sentence of FOL starting with two negations
  - **Returns** - The premise without the first two negations
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *doubleNegationElimination*
  `public static void` **doubleNegationElimination**`(`
  `logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `premise` - The root node of a sentence of FOL starting with two negations
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

4.3 Proof structure

When the application is started an Java Proof object is instantiated. This object stores data about the proof and is used t

The Proof object is a high level abstraction that hides the complexity of the actual proof structure. A proof consists of a tree of ProofSteps. Each ProofStep contains data about a particular line of proof. This data includes:

lineNumber The number of the line in the proof

rootNode Root node of the sentence tree

subproofs List of pointers to sub proofs following this line

next Pointer to next non subproof step

parent Pointer to previous step

introducedVariable The name of the introduced variable, null if n/a

level The number of nested subproofs that this line is inside of

justification The ROI used along with line numbers of the premises

A crucial part of Fitch style proofs is the use of subproofs and the notion of scope. Scope, in this context, is the notion that a line within a subproof cannot be used as a premise of a ROI for any line outside of the subproof. As well as this the premises used for a ROI must precede the line being justified. Since each ProofStep explicitly stores pointers to other steps and subproofs, the information which other lines or subproofs can be used as a premises to justify this line can be found quickly, without having to traverse the whole proof tree.

4.3.1 Class **Proof**

Stores all data necessary to construct a proof. Simple Methods are provided to manipulate the proof such as adding or deleting lines.

Declaration

public class Proof
extends java.lang.Object

Fields

- public List predicates
  - A list of all the named predicates in the proof. Used to populate the predicate list.

CONSTRUCTORS

- *Proof*
  `public` **Proof**`( )`
  - **Usage**
    - Default constructor. Constructs an empty proof

METHODS

- *addStepAsEndOfSubproof*
  `public ProofStep` **addStepAsEndOfSubproof**`( logic.proof.builder.parser.SimpleNod`
  node`, java.lang.String` **formula** `)`
  - **Usage**
    - Adds a new proofstep that is the last line of a subproof
  - **Parameters**
    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addStepAsNewLine*
  `public ProofStep` **addStepAsNewLine**`( logic.proof.builder.parser.SimpleNode`
  node`, java.lang.String` **formula** `)`
  - **Usage**
    - The default method to add a new proofstep to the proof
  - **Parameters**
    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addStepAsStartOfSubproof*
  `public ProofStep` **addStepAsStartOfSubproof**`( logic.proof.builder.parser.SimpleNo`
  node`, java.lang.String` **formula** `)`
  - **Usage**
    - Adds a new proofstep that is the start of a subproof
  - **Parameters**
    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addVar*
  `public ProofStep` **addVar**`( java.lang.String` **var** `)`

  - **Usage**
    * Add a new proofstep which introduces a boxed variable
  - **Parameters**
    * `var` - The name of the variable being introduced
  - **Returns** - Returns the proofstep that has been added

- *addVar*
  `public ProofStep` **addVar(** `java.lang.String` **introducedVariable,** `logic.proof.builder.parser.SimpleNode` **rootNode,** `java.lang.String` **formula** **)**

  - **Usage**
    * Add a new proofstep which introduces a boxed variable alongside an assumption
  - **Parameters**
    * `introducedVariable` - The name of the variable being introduced
    * `node` - Root node of the sentence
    * `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *getCurrentLevel*
  `public int` **getCurrentLevel(** **)**

  - **Usage**
    * Returns the number of subproofs currently open
  - **Returns** - the number of subproofs currently open

- *getLines*
  `public ArrayList` **getLines(** **)**

  - **Usage**
    * Returns the ordered list of proofsteps
  - **Returns** - the ordered list of proofsteps

- *removeStep*
  `public void` **removeStep(** **)**

  - **Usage**
    * Removes the most recent line from the proof

Chapter 5

Chapter Title

Bibliography

[1] Aristotle. *Prior Analytics*. Kessinger Publishing, 2004.

[2] D. Barker-Plummer, J. Barwise, J. Etchemendy, and A. Liu. *Language, Proof and Logic [With Software]*. CSLI Publications, 2011.

[3] D. Cantone, E. Omodeo, and A. Policriti. *Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets*. Monographs in Computer Science. Springer, 2001.

[4] K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2011.

[5] J. Ferreiros. *The Road to Modern Logic-An Interpretation*. The Bulletin of Symbolic Logic, Vol. 7, No. 4, 2001.

[6] A. D. Irvine. Principia mathematica. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2010 edition, 2010.

[7] P. King and S. Shapiro. Oxford Companion to Philosophy (Oxford University Press), 1995.

[8] W. Kneale and M. Kneale. *The Developments of Logic*. Oxford University Press, 1962.

[9] J. Koetsier. Android captured almost 70% global smartphone market share in 2012, apple just under 20%.

[10] Ofcom. Internet and web-based content.

[11] V. Peckhaus. Leibnizś influence on 19th century logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*.

[12] M. Salmon. *Introduction to Logic And Critical Thinking*. Thomson Wadsworth, 2007.

[13] R. Smith. Aristotleś logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2012 edition, 2012.

[14] K. Wolfgang and W. Köhler. *The Mentality of Apes*. Number v. 3 in International Library of Psychology. Routledge, 1999.