MSc Computer Science

School of Computer Science, University of Birmingham

2020

Automated proof-checking for natural deduction in propositional and first-order logic

Mark Robinson

2059210

Supervisor: Dr. Benedikt Ahrens

Abstract

Hand-writing proofs of formal logic is a slow and error-prone process. Following faulty lines of reasoning, failing to spot invalid derivations, and generally making mistakes can be frustrating, and leaves many students of the subject believing that it is either too difficult, or too boring. Yet, formal logic is vital to many fields, including philosophy, mathematics, and computer science. The aim of this project was to apply the principles of software engineering to design and build a system which would allow students of formal logic to practice natural deduction without the pain points. A full requirements engineering process was undertaken, a detailed design of system architecture was produced, and the final system was implemented and extensively tested. The final product consists of a language for interacting with the system, designed in close correspondence with the syntax of formal logic; a full GUI for receiving inputs from the user and outputting proofs written in Fitch-notation; an interpreter to convert input of the language into the relevant data structures; and an inference-checking system to automatically identify which of thirty inference rules of natural deduction were used to derive that input.

Acknowledgements

I would like to gratefully thank my supervisor, Dr. Benedikt Ahrens, for his vital input throughout the project. Likewise, to Dr. Ahmad Ibrahim for his helpful feedback on the project demonstration.

1. Introduction

1.1. Motivation & Problem Overview

When a 21st Century philosopher trained in the western analytic tradition picks up 'On Liberty,' one of the most famous works by esteemed 19th Century liberal philosopher John Stuart Mill, they are likely to wind up scratching their heads. "Liberty consists in doing what one desires," Mill writes. The philosopher reaches for a cigarette to mull this over, and then remembers that they recently quit smoking. They desire a cigarette. But they also desire not to desire it. What, if any, is their true desire? Does this mean, by Mill's definition, that they are unfree?

'Logic' as an academic subject matter deals with arguments; sets of premises together with a conclusion. But the English language, rife with ambiguity, imprecision, and loaded with meaning from our nonverbal cues (Matsumoto, 2016) is not especially good at getting the point across. If language was a more precise tool, then the philosopher could get straight down to the business of arguing about the truth of Mill's claim, rather than arguing about what it means.

'Formal logic' is the name given to a class of formal languages which strip away the context and ambiguity of English language sentences to get down to their underlying structure. As a tool, it is commonly used in philosophical academia to analyse the validity of arguments in precise terms. More broadly, it has been used to study the foundations of mathematics (Ewald, 2019), formalize mathematical axioms (Kreitz and Pucella, 2005), disambiguate meaning in philosophy and mathematics (Shapiro and Kouri Kissel, 2018), and even to represent knowledge within computers in research on artificial intelligence (Nebel, 2000).

Going beyond its specific uses, formal logic is a common feature on undergraduate courses in the above fields, and is something which many aspiring philosophers, mathematicians and computer scientists will be faced with (Shapiro and Kouri Kissel, 2018)). Writing proofs in formal logic is an inevitable part of this, and like most mathematical skills, is best learned by engaging with and practicing the material (Magnus and Button, 2018) (Chubb, 2018).

But this can easily go wrong. Proofs can be long and intricate. Writing them out is an error-prone process. The student of logic may make a bad inference on some line of the proof and not realise until they had based several more inferences on that one. Alternately, they may follow a specific line of reasoning and realise that it will get them nowhere. Or, given all the symbols involved, they might simply make a mistake. All of this can lead to rework, a loss of motivation, and a sense that studying formal logic is 'too hard' and that whatever Mill truly meant will always be an enigma.

The goal of this project was to simplify the process of practicing with formal logic. The end result was a desktop application allowing the user to practice their formal logic skills – specifically, their skills in Fitch-style natural deduction for propositional and predicate logic – in an environment which avoids the hazards of handwriting proofs, whilst also skirting around the complexity and general usability issues of the proof-checking systems one can access today.

1.2. Methodology

As this was a software engineering project, it made sense to follow a model software engineering process. The proposed system had a very clear core functionality: receive input from the user, process it to check its inner logic, and output a proof. Therefore, as this core function was unlikely to change, the linear and plan-driven waterfall model of software engineering was chosen to begin with.

It was not possible to know exactly how much would be achievable in the project timeframe. Therefore, this linear structure was supplemented by incremental development of the core functionality. A module would be designed implemented based on the requirements specification, subjected to preliminary testing, and then refactored to add extra functionality / improve performance. This flexibility gave the best of both worlds: a structured approach with the benefit of being able to improve the designs and not run the risk of over-promising and under-delivering.

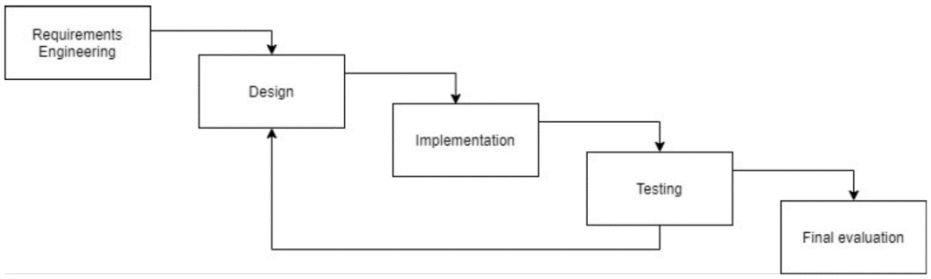


Figure 1: Model of the software engineering process

The three key activities of requirements engineering were the formulating of project objectives, heuristic evaluation of related systems, and production of personas and scenarios. The four key design areas for the system were the input language, the interpreter, the proof-checking algorithm, and the GUI. All of those areas will be presented in this report, followed by the test strategy, main results and evaluation.

To begin with, the reader will be presented with a primer in formal logic to aid them in their understanding of the rest of the concepts underlying the system.

2. Formal Logic

To understand this project, it is necessary to understand some key notions from the field of formal logic, all of which will be presented in this section.

Formal logic is a broad field, used in lots of different subject areas and with many different conventions. The conventions for formal logic followed for in this project – including vocabulary, grammar, syntax and proof systems – are those described in the textbook forallX:Cambridge (Magnus and Button, 2018). The first reason for this is the author's familiarity with the material; this was the textbook studied by the author during their first formal logic course. The second reason is that the typical user of the system is likely to be the typical user of the textbook: a student taking a first course in logic.

The first key notions are arguments, propositions and deductive validity. Formal logic deals with collections of premises, together with a conclusion. This is what we call an *argument*. But not any sentence can be part of an argument in logic. One cannot have for a conclusion a question or a grunt of pain – or at least, not in this logic system. An argument is made up of *propositions*: expressions to which a truth value can be assigned. An argument can be considered *deductively valid* if it is impossible for all the premises to be true, and the conclusion be false. In this report, 'valid' and 'deductively valid' are used interchangeably.

Arguments can be valid for different reasons. For example, the argument 'Kripke is a homo sapiens, therefore Kripke is a human' is valid because of the meaning of 'homo sapiens' and 'human.' The system of formal logic presented here is a tool for evaluating the validity of arguments based on their form. Consider the following example:

Syllogism: If Mark is a man, then Mark must die. Mark is a man. Therefore, Mark must die.

This argument is clearly valid, but its validity does not hinge on the specific English-language meaning of 'Mark,' 'man' or 'mortal.' Instead, it hinges on the key terms 'if' and 'then.' It is exactly this kind of underlying logical structure that formal logic tries to uncover.

Two languages of formal logic will now be discussed to elaborate on these ideas further.

2.1. Propositional Logic

Propositional logic is centred on two key notions: the atomic proposition, and logical connectives.

An atomic proposition is the smallest unit of a proposition which can itself have a truth value. In the above example, 'Mark is a man' and 'Mark is mortal' are both atomic propositions. 'If Mark is a man, then Mark is mortal' is a compound proposition; multiple atomic propositions, joined by a logical connective. In this case, the logical connective is 'if… then,' known formally as the conditional.

The version of propositional logic treated as canon for this project recognises five logical connectives: four binary, and one unary. With these connectives, one can begin to recursively build sentence of propositional logic. This can be done for each connective as follows (Magnus and Button, 2018):

Where A is an atomic or compound proposition:

Negation, symbolised as '¬A', paraphrased as 'It is not the case that A'

Where both A and B are atomic or compound propositions:

Conjunction, symbolised as 'A ∧ B', paraphrased as 'Both A and B'

Disjunction, symbolised as 'A ∨ B', paraphrased as 'Either A or B'

Conditional, symbolised as 'A → B', paraphrased as 'If A, then B'.

Biconditional, symbolised as 'A ↔ B', paraphrased as 'A if and only if B'.

As a convention, we also allow logical absurdity as an atomic proposition of propositional logic, symbolised as '⊥'. This is the canonical contradiction, which can be interpreted as shorthand for 'A ∧ ¬A'.

2.2. Predicate Logic

Propositional logic is a useful tool to help a student grasp the basics of formal logic, but its expressive power is limited. Predicate logic, also known as first-order logic, can prove much more interesting results.

Predicate logic can be treated as an extension of propositional logic, though for our purposes they will be treated as different languages. The two main differences are the replacing of atomic propositions with atomic formulae, and the introduction of quantifiers.

An atomic formula can take one of three forms. It may be a combination of predicate and term(s). The predicate is denoted by a capital letter and is best thought of as some property held by its terms. A term is denoted by a lowercase letter. For example, the atomic formula 'Axby' contains a three-place predicate and three terms. An atomic formula could also be an identity relation between two terms, such as 'a=c'. Or it could be absurdity, as in propositional logic.

The introduction of quantifiers is necessary because of the introduction of terms. A term can either be a variable or a constant. By convention, constants are denoted with lowercase letters from a-r and variables are denoted with lowercase letters from s-z. If it is a variable, then it must be 'bound' by a quantifier. Every quantifier must bind exactly one variable.

Where A is some proposition containing the variable x, the two quantifiers can be understood as follows (Magnus and Button, 2018):

> Universal quantifier, symbolised as '∀xA', paraphrased as 'For everything that is x, A is true'.
>
> Existential quantifier, symbolised as '∃xA', paraphrased as 'There exists at least one thing that is x of which A is true'.

2.3. Natural Deduction

The strongest way to assess the veracity of an argument is through the provision of a proof (Avigad, De Moura and Kong, 2020). The proof consists of the premises and conclusion of the argument, along with all subsidiary steps involved in establishing the conclusion based on the premises. A language of formal logic does not only come with a vocabulary, syntax and semantics, but also a proof system.

Natural deduction is the process of using self-evident inference rules to derive propositions from a set of premises. This demonstrates that an argument is deductively valid: because you inferred the conclusion from the premises using obvious rules, it is impossible for all of the premises to be true and the conclusion false. The basic rules of inference used in the natural deductive systems for propositional and predicate logic are introduction rules, and elimination rules.

Introduction rules tell us, for each symbol of the logic, what is required to infer a new sentence of the logic taking that symbol as its main logical operator (i.e. as the symbol with the broadest scope). Conversely, the elimination rules for each symbol tell us what we may infer given a sentence with that symbol as its main logical operator.

The style of natural deduction utilised in this project is Fitch-style natural deduction, characterised by each line of the proof containing a line number, a proposition, and the inference rule used to derive that line. Assumptions are represented by indents, the depth of which correspond to the depth of that assumption within the active assumptions of the proof. Two Fitch-style proofs are presented below.

1	(M → D)	Premise
2	M	Premise
3	D	→E 1,2

1	(Mm → Dm)	Premise
2	Mm	Premise
3	Dm	→E 1,2

Figure 2: Fitch-style proof of the syllogism in propositional and predicate logic (output by the final system)

On the left is a proof of the earlier syllogism concerning Mark's mortality in propositional logic. On the right is its counterpart in predicate logic. Note that 'M' and 'D' represent the propositions 'Mark is a man' and 'Mark must die' in propositional logic. In predicate logic, they represent predicates, while m represents the constant, Mark. Thus, the atoms of that prof can be read as 'Mark has the property of 'being a man'' and 'Mark has the property of 'must die.'' We could just as easily substitute m for any other human being without needing to create a whole new atom, as we would in propositional logic.

3. Problem Analysis & Requirements Engineering

To transform the problem into a set of functional system requirements, three key activities took place. First, the problem was sub-analysed into a broad set of objectives. These were categorised into three tiers, corresponding to how important it was to try to satisfy them in the final product. To clarify exactly what a solution to each sub-problem would entail, a detailed review of related technologies was carried out, and personas were created with accompanying scenarios.

3.1. Broad Objectives

As stated, the problem was subdivided into concrete, measurable goals and organised into three tiers of importance. Essential goals describe the minimum viable product. Conditional goals describe extra features which should feasibly be completed in the timeframe barring exceptional circumstances, such as an extended period of illness. The optional goals existed in case the project went much more quickly than expected. It became apparent relatively early on that the optional goals would be beyond the scope of this project, and so they are not reflected in the final requirements specification.

3.1.1. Essential Goals

There were seven essential goals which were to be satisfied to produce a minimum viable product:

i. The system should provide some coding language which allows the user to interact with it to build proofs, should contain a module to interpret this input, and should provide some mechanism for interfacing with the user.
ii. The system should recognise premises, derivations, assumptions, and conclusions as different types of sentence which need to be treated differently in the proof and provide some means for the user to label them as such.
iii. The system should provide a means of closing assumptions.
iv. Any input should be checked to ensure that it is a well-formed formula.
v. For an invalid derivation, the system should display an error message declaring it invalid. For a valid derivation, the system should automatically identify the inference rule used and add a reference to that rule to the proof.
vi. The system must output proofs in Fitch style.
vii. All objectives must be satisfied for both propositional and predicate logic.

3.1.2. Conditional Goals

The system had four conditional goals which should feasibly be completed, barring exceptional circumstances:

i. The system should be represented to the user using a graphical user interface.
ii. The user can save proofs to their own computer, begin new proofs, and load old proofs.
iii. Either buttons or drag-and-drop should be included to insert the logical connectives.
iv. The system should have buttons allowing the user to apply additional logical 'derived rules, such as modus tollens and DeMorgan's rules.

This last objective was eventually updated so that the system would automatically infer derived rules just as it does introduction and elimination rules.

3.1.3. Optional Goals

The system had optional goals to be satisfied only if time permitted.

i. If a user successfully builds a deductively valid proof, the system will search for a more elegant solution (i.e. one using less lines) and return it if one is found.
ii. If a user gets stuck on a proof, an auto-complete function will allow the system to complete the proof for them, deleting any redundant lines in their work so far.
iii. If there are patterns of inference which the user regularly uses, they should be able to save their own shortcut rules to represent this pattern of inference.
iv. The system should allow proof-finding for some systems of modal logic.

v. The system should allow the user to choose between classical and intuitionistic as the underlying logic.
vi. The system should allow for different proof systems (e.g. also incorporating tree proofs).

3.2. Review of Related Work

Once the goals of the project were mapped out, two key activities were carried out in the process of translating them into a full requirements specification. First was the review of related technologies, and second was the creation of personas and scenarios. In this section, technologies related to the proposed system are presented and evaluated. The aim of this was to understand the strengths and drawbacks of existent proof assistants, and to use these as guidance for structuring the requirements of the final system.

3.2.1. Introduction to Proof Assistants

Proof assistants exist to aid the development of formal proofs. They vary by the features they offer. On one end of the spectrum, interactive theorem checkers work by checking the veracity of a proof input by the user. This can be distinguished from automated theorem provers wherein the system will construct a proof based only on the input theorem and some parameters which ensure it finishes executing in a reasonable amount of time (Setzer, 2007) (Harrison, 2000) (Avigad, De Moura and Kong, 2020). Other systems sit in the middle of the spectrum and automate proof development to varying degrees.

Such systems vary significantly in their complexity. Many proof assistants are used to prove properties of computer programs, and many programming languages are designed based on a specific type system (Pierce and Pennsylvania, 2002, p.9). Furthermore, it is useful to keep track of the type of each mathematical object when writing mathematical proofs (Avigad, De Moura and Kong, 2020). For these reasons, interactive theorem provers based in type theory currently dominate the field. Such systems require a high level of technical expertise to use.

An overview of some of these more complex systems is given for the sake of putting this project into context. But their functionality is out of the scope of this project. The specification of this system corresponds to propositional and first order logic; these type-theoretic proof checkers draw on concepts from higher order logics – extensions of first-order logic with greater expressive power (Väänänen, 2020) (Barthe, n.d.).

The scope of this project lies within interactive proof checking, but among a class of systems with a different focus: pedagogy. Most of the tools which specifically use Fitch-style notation or similar are learning tools, allowing the user to develop and practice their powers of natural deduction.

The result of this project is one such tool. It aims to provide a higher level of usability – the quality of the user's experience – than the available alternatives (Usability.gov, 2019). It is a simpler alternative to hand-drawing proofs, and is easier to learn than an advanced type-theoretic theorem prover.

Finally, note that LATEX packages exist to help a logician typeset Fitch-style proofs in LATEX (the most prominent being fitch.sty by Johan Klüwer, 2003 (Smith, 2018). These are excluded from this project. Although they satisfy the requirement of helping with the writing up of formal proofs, they offer no proof checking functionality.

3.2.2. Evaluation Method: Heuristic Evaluation – Nielsen's Heuristics

As usability is the key focus, it made sense to evaluate the related technologies through the lens of usability. One technique for evaluating the usability of a system is to use a heuristic evaluation (Wong,

2020). A heuristic evaluation takes several 'rules of thumb' metrics of usability and uses these as a framework to evaluate the system.

Nielson's heuristics are ten such principles for assessing the usability of a system. They have been chosen for this project because they provide an easy-to-use framework which allows for many evaluations to be performed relatively quickly. Each of the ten heuristics will now be presented, summarised, and labelled for future reference (Nielsen, 2017):

Visibility of system. The system should provide relevant information about its status, such as what mode it is in, as well as giving appropriate feedback in reasonable time.

Match between system & real world. The system should align itself with the user's concepts of the underlying theory.

User control & freedom. The system should guide the user but be flexible to their needs. For example, giving them control over inputs but providing undo and exit functions if mistakes are made.

Consistency and standards. The system should adhere to any conventions for a system of its type and be consistent in its features, language, and behaviour.

Error prevention. The system should be designed to minimise the chance of errors and display appropriate error messages should they occur.

Recognition rather than recall. The system should minimise reliance on the user's memory, such as by offering menus rather than commands. It should also provide easily accessible instructions.

Flexibility and efficiency of use. Usage of the system should be as efficient as possible. For example, keyboard shortcuts can speed up interaction for advanced users.

Aesthetic and minimalist design. The system's interface should only contain relevant and frequently used information.

Helps users recognise, diagnose and recover from error. Error messages should clearly state the problem in plain language, and recommend a solution.

Help and documentation. Any formal documentation should be easy to search, understand and use.

3.2.3. Heuristic Evaluation of Current formal proof-checkers

In the following subsections, eight formal proof checkers are presented and discussed. They are briefly described, and the conclusions drawn from their heuristic evaluations are presented. The heuristic evaluations themselves can be repetitive, and so have been moved from the body of the text to Appendix B for reference, and as evidence of the work carried out. Reviews and demonstrations of the proof checkers have been relied upon to compensate for the author's lack of understanding where necessary. This is clearly stated in the relevant subsections.

3.2.3.1. Natural Deduction Proof Editor and Checker

Available online at: https://proofs.openlogicproject.org/ (Klement, n.d.)

This system offers an interactive GUI allowing the user to build Fitch-style proofs on a line by line basis. The user enters the premises along with the conclusion that they wish to derive. A basic proof is constructed by the system, adding each line after the user inputs it. Buttons are available to add a new line, delete a line, add a new subproof (referred to in this project as an 'assumption'), add a new line to

the parent proof below a subproof, and to add a new subproof to a parent subproof which already contains a subproof.

Once a line (or series of lines) has been input and constructed into a proof by the system, the user can "check proof" to determine if the argument that they have entered is valid. The system provides one of four rough categories of response: informing the user that they have made a syntax error, an invalid derivation, a valid derivation but have not yet reached the conclusion, or a valid derivation and they have reached the conclusion.

Overall, the Natural Deduction Proof Editor and Checker was evaluated to be a useful tool for practicing Fitch-style deduction, mirroring the formal logic conventions exactly. It also helpfully diagnosed the specific line of proof on which an error occurred. However, interacting with the system via repeatedly clicking buttons became tedious. As did manually typing out the inference rules, which must be formatted very specifically. These results encouraged the keyboard-centric approach to proof-checking taken in this project, and further entrenched the need for the system to automatically infer the inference rule. Furthermore, the cluttered user interface of the Natural Deduction Proof Editor and Checker motivated an approach centred around concealing all unnecessary guidance using the system until it was explicitly sought out.

3.2.3.2. Tree Proof Generator

Available online at: https://www.umsu.de/trees/ (Schwartz, 2020)

Wolfgang Schwartz's Tree Proof Generator has a different focus to this project. To interact with the system, the user inputs premises and a conclusion into a text field (or just a singular formula if they want the system to check if an interpretation exists which satisfies that formula). They then press 'Run' to automatically generate a tree proof – a different proof system Fitch-style deduction. From there, the user can either export the resultant proof as a png file, return to the start page, or change the input and run again. If the input is invalid, a countermodel is presented under which the given input is false. While this proof-finding system operates in a different area to the focus of this project, some useful insights were gleaned from experimenting with the interface.

Firstly, the system was exceptionally easy to start using with minimal practice. The Natural Deduction Proof Editor and Checker had multiple shortcuts for each logical connective, which could be confusing. The Tree Proof Generator had one shortcut for each symbol and provided buttons to input logic symbols into the text area. These buttons were displayed above the input field so novice users could easily get started. As the user becomes more experienced, the buttons become superfluous as the user can also write input by using only the keyboard shortcuts. This made the system both simpler and more efficient to interact with. The biggest limitation of the system for its purpose was that it did not support identity for predicate logic, limiting its practical applications.

3.2.3.3. FitchJS

Available online at: https://mrieppel.github.io/fitchjs/ (Rieppel, 2015)

FitchJS is the second of two proof assistants catering specifically to Fitch-style natural deduction featured in this research. It takes as input a set of premises, and a conclusion, written using keyboard shortcuts for the logical connectives. It creates a proof based on these. From there, the user can enter one formula at a time into a text field, selecting the rule used to derive that formula from a drop-down menu. An interesting quirk of this is that the user must apply some rule making use of an assumption to close that assumption. If a formula is not well-formed, a warning is displayed in a box below the proof. If a

derivation is invalid, the relevant error message is displayed in the same place. If the conclusion is reached, a message is displayed notifying the user and offering to export the proof.

The appending of the formula on a line by line basis via entering formulae into a text bar above the proof was much less frustrating to use than the button system of the Natural Deduction Proof Editor & Checker. Especially useful was the function for selecting the rule from a drop-down menu, and that the rules on offer depicted the keyboard shortcut for the logical symbol, rather than the symbol itself. This accelerated the pace of both individual cases of learning to use the system. The clean user interface design was also very impressive.

However, the system would not be chosen for intensive use; entering the proof line by line is a slow process, slowed down further by only being able to add and delete lines from the end of the proof. A means of 'importing' a whole proof from a text field is presented, but the exactitudes of writing out these proofs is not easy to learn. And, like the Natural Deduction Proof Editor & Checker, the user must still pause in their process to explicitly tell the system which inference rule to check.

3.2.3.4. Gateway to Logic's Logic Calculator

Available online at: https://www.erpelstolz.at/cgi-bin/cgi-form?key=0000624d. (Gottschall, 2019).

Rather than using Fitch-style deduction for proof checking, The Logic Calculator Proof Checker applies a similar natural deductive system known as System L (also known as Lemmon's Calculus). The proof is laid out line by line, with each line containing a list of assumptions drawn on by that line, a line number, a logical formula, and an inference rule (Steinberg, 2010). The input for the proof checker can be any number of these lines typed into a text box. From here, the user can press a button to either "reset" or "execute" the proof. On execution, either the proof is displayed error-free, or it is displayed with error messages diagnosing where the invalid derivations are, or a notification that the proof could not be processed due to diagnosed syntax errors is displayed.

Overall, this system's clean layout and utilitarian approach to proof writing was very impressive and allowed for proofs to be written quickly and efficiently. The user was given complete control over writing the input, and helpful guidance on doing so was easily accessible. The inputs were easy to learn, using keyboard shortcuts which closely resemble logic symbols. Recompiling the entire input text with each execution meant that a proof could easily be edited in the middle. While some buttons might have sped up a novice user's interactions with the system, the system's reliance on a text area did make for quick user interaction.

3.2.3.5. Lean

Demonstration of Lean: (Morrison, 2020). Lean available online at: https://leanprover.github.io/, (Avigad, De Moura and Roesch, 2020).

The Lean theorem prover is both an interactive theorem checker and prover, in that it gives the user the option to use automated theorem proving where desired whilst still providing theorem checking as its base functionality. The language used to interact with the system, also known as Lean, is based on dependent type theory, and allows the user to utilise concepts from propositional and first order logic to build proofs of far greater complexity than the scope of this project covers. Owing to the technical expertise required to use Lean, the evaluation of this system is based not only on the author's experiments, but also on the Lean manual and on a demonstration of the system.

The system takes as input the mathematical assertion to be proved, and a proof written in the Lean language. The screen is split into two areas: on the left is the input area, on the right is the system's

response. As the user types the proof, it recompiles in real time. An error message is displayed if a piece of input is syntactically incorrect, or if it is correct but the goal has not been proved. It also offers some automatic theorem proving techniques, such as by using backwards reasoning to fill in gaps in the proof.

Whilst evaluating a system as complex as Lean was always going to be a challenge, some useful lessons were learned. The keywords used to explicitly represent common functions of handwritten proofs to the system inspired the set of keywords implemented in the input language used to interact with the system produced during this project, as did the representation of logic symbols by their names. Similarly, the clean split screen aesthetic was drawn on for the GUI design of the final system. A significant takeaway from the Lean prover is consideration of the learning curve of the system. A coding language was developed to allow the user to interact with the system produced during this project. A constant consideration for that language, arising from studying Lean, was to ensure that it remained as intuitive to a non-specialist as possible. More precisely, that even without a manual, the coding language and GUI should suffice to allow the user to correctly guess at the syntax of the language.

3.2.3.6. Other Interactive Proof Assistants: Coq, HOL-Light, AGDA

First demonstration of Coq: (Institute for Advanced Study, 2016). Second demonstration of Coq: (Jeffrey, 2020). Browser version, jsCoq, available online at https://jscoq.github.io/.

Attempts were made to evaluate several other proof assistants flagged as being of interest during the creation of the project proposal document. Specifically: Coq, HOL-Light and AGDA. Upon inspection, these systems proved to be very complex and, as mentioned, offer much functionality which is beyond the scope of this project. Attempts were made to provide full heuristic evaluations, but these turned out very similar to the evaluation of Lean. Therefore, only a summary of each system is provided below.

Coq is an interactive theorem prover for the development of mathematical proofs (Paulin-Mohring, 2011) and the name of the language used to interact with the system (coq.inria.fr, 2019). The system checks input, allows the user to manually construct the proof, and implements "tactics" to build parts of the proof which the user does not wish to do by hand. The system takes text as input, beginning with a declaration of the thing to be proven. The system outputs the relevant subgoal of the proof, any relevant error messages, or a notification that the proof is complete. Moving the cursor to a specific line of the input area displays the output up to and including that line. As with the other proof assistants discussed so far, Coq represents logical connectives using a shorthand notation which resembles the connective. For example, the conjunction symbol is "\/" while the existential quantifier is "exists".

HOL-Light is a proof assistant for classical higher order logic, which the user can interact with using Objective CAML (Harrison, 2009) and is part of a wider HOL family of theorem provers (Harrison, 2000). AGDA is a programming language and proof assistant offering similar functionality to Coq, but with a focus on functional programming (Jeffrey, 2020) (Bove, Dybjer and Norell, 2009). Watching demonstrations of these systems revealed little which would be useful to this project which was not revealed in the detailed analysis of Lean and the shorter analysis of Coq.

3.2.4. Conclusions on Related Work

To usefully interpret the above evaluations, a discussion of the key issues within each heuristic and the resultant design decisions will now be given.

Visibility of system. In general, the 'simple' proof assistants inspected offer a limited enough range of functionality (i.e. a simple conversion between input and output) to not require updates on the status of the system.

Match between system & real world. All the proof checkers represented symbols of formal logic with shortcuts that closely corresponded to the formal logic itself. Furthermore, they adhered specifically to the conventions of the logic system used. The final system should adhere to the specific conventions of Fitch-style deduction, and avoid taking liberties with the formatting of outputs for the sake of simplifying the code.

User control & freedom. The amount of control that the user has over their input was a recurring theme. High control through text-entry made for fast and efficient use, but also increased the learning curve of the system and made it more prone to user error. Buttons had the opposite effect; easier to input, but slowing down the system. This tension informed two key design choices of the final system. First was that all the user's input was to be put into one input field to give them as much control as possible. Second was that buttons were to be provided depicting each formal symbol which, when pressed, would insert the system's representation of that symbol into the input field.

Consistency and standards. The system should not only adhere to the proof system of choice but should also ensure consistency in the inputs on offer to the user. For example, offering keyboard shortcut for a given logic connective was often less confusing than being given several, and having the system refer to that symbol by the shortcut rather than the symbol itself in any areas concerning user input helped to reinforce the user's knowledge of how they should interact with the system.

Error prevention. In general, error prevention was less important than providing appropriate error messages which diagnosed the problem. This gave the user greater freedom to experiment with their inputs, but a side effect was that learning how to input into that system could be difficult. The drop-down menu of FitchJS helped to alleviate the tedium of learning the convention of typing introduction rules (such as with the Natural Deduction Proof Editor and Checker). More error prevention would be achieved by removing the need to manually type in the inference rule, and have the system infer it automatically. This was done in the final system.

Recognition rather than recall. Guidance as to how to input into the system is necessary. Buttons can be helpful, but also slow and cumbersome. An ideal middle ground is to provide buttons to help novice users which, once pressed, insert the keyboard shortcut into the input area, whilst allowing experienced users to type straight into the input area.

Flexibility and efficiency of use. The previous point applies here. Also, even though it used a different proof system, Gateway to Logic's Logic Calculator seemed to be the easiest system to interact with. This is because it gave the user full control to type the entire proof in one go instead of needing to input it line-by-line. This is reflected in the final system by taking as input a text file containing all input code.

Aesthetic and minimalist design. Information on how to use the system should be easily accessibly but should not clutter the page. The most aesthetically pleasing systems displayed only the input area and the output, with options to view other information separately.

Helps users recognise, diagnose and recover from error. Error messages were generally very specific across all the systems, identifying the exact nature of the issue, its exact location and, where possible, a suggestion for a fix.

Help and documentation. The better help and documentation within the above systems was easily accessed and concise. In general, we can split it into two categories: information about the logic of choice, and about how the system represents that logic. For example, different Fitch-style proof systems can use different introduction and elimination rules for the logical symbols. Displaying which ones are compatible with the system, as the Natural Deduction Proof Editor & Checker did, is necessary to help

the user understand how to use the system. Both types of documentation should be provided in as much detail as is necessary for a user unfamiliar with the system to understand its features and logic.

These points provided a framework through which the user's interaction with the proposed system could be conceptualised, and consequently provided key considerations considered when creating the system's requirements specification.

3.3. User Analysis

The second step in the requirements engineering process was to understand typical users of the system.

Three categories of users were identified based on their level of experience with the system and with formal logic, and are presented with a summary of how the system should cater to them:

i. *Low familiarity with formal logic and low familiarity with the system*. This system is a pedagogical tool and is intended to help students of formal logic to practice natural deduction in as simple a manner as possible. So, the final product must be very intuitive to use.
ii. *High familiarity with formal logic and low familiarity with the system*. The system can also be used to write more powerful proofs and should be easy for a new user to master when they are already familiar with the underlying logic.
iii. *High familiarity with formal logic and high familiarity with the system*. Experienced users of both the system and the underlying logic should be able to create proofs as quickly and efficiently as possible.

Section 3.2 highlights that different levels of experience with the system will dictate the most efficient way for the user to interact with the system. To this end, the system should represent formal logic clearly enough that a novice learning formal logic for the first time would not struggle to understand. Ensuring that this is the case will ensure that category (ii) is catered for as well as category (i). But these features to simplify the representation come at the cost of efficiency. Therefore, different functionality should cater to category (iii).

Typical users from categories (i) and (iii) will now be modelled using by creating 'personas.' These are characterisations of those users which present their motives, goals, and pain points. Through building a more detailed picture of the user, it becomes possible to take the general concepts of good and bad design decisions identified in 3.1 and translate these into more concrete requirements (Muldoon, 2018).

Included with each persona is a scenario for how that user might interact with the system, and how the system should respond. A scenario is a natural language model of an interaction between the user and the system (Arms, 2014). Finally, a set of scenarios for how the system should handle different errors are presented. These scenarios are key to bridging the gap between the work in section 2.1, the personas, and the final requirements specification.

3.3.1: Persona 1: A Novice User

The purpose of this persona is to gain an understanding of the specific needs of a novice user of the system practicing natural deduction for the first time.

Name: Micaela Solis

Tagline: University Student studying formal logic as part of a busy course in philosophy.

Background:

- *Age:* 18
- *Role:* Philosophy student
- *Field of study:* Philosophy (formal logic)
- *Level of computer expertise:* experienced, but not with programming

Main points:

- Computer literate but very little background in computer science
- Is very busy with other work, so needs a small learning curve
- Is unfamiliar with formal languages

Goals:

- Write out formal proofs for her logic assignments
- Be able to use the system for multiple proofs
- Learn more about formal logic

Frustrations & Pain points:

- She dislikes having to memorise large numbers of commands
- Sometimes she makes mistakes in handwritten proofs and has to start again after noticing them later on in her proof.
- She dislikes using new technologies which take a long time to learn.

Narrative:

Micaela is an undergraduate studying Philosophy at a top University. She will sit an exam in Formal Logic at the end of the academic year. To help her to prepare, she is given worksheets throughout the year to practice formalisation of natural language sentences, and natural deduction. However, she has never studied formal logic before. She is not always confident in her work and gets frustrated having to wait for her fortnightly logic classes before finding out if she is correct. She would like a much quicker and easier way of writing out formal proofs than cranking them out by hand. She is trying out the proof system for the very first time.

A scenario was constructed to model the normal flow of events in a typical interaction between a student of formal logic with little technical computing background and the system. Assume that the user already has the system installed on their computer.

Scenario: Normal flow of execution for a novice user of both the system and formal logic.

1. Micaela opens the program. It features a text area, shortcut buttons for inserting symbols and keywords of formal logic, and a display area. Towards the top of the screen, there is an indicator that the logic of choice is currently propositional logic.
2. Micaela presses the "Premise" button.
3. The system inserts the code keyword for a new premise into the input area.
4. Micaela presses the shortcut button for logical conjunction.
5. The system inserts the code for logical conjunction into the text area.
6. Micaela leaves the mouse hovering over the conjunction symbol for a moment.
7. The system displays an overview of how to represent that symbol in code as a tooltip.
8. Micaela presses enter.
9. The system moves the cursor in the text field to a new line and warns that the previous line is not a well-formed formula of formal logic.
10. Micaela clears the text area.
11. The warning disappears.

12. Micaela presses the "Premise" button, types A, presses the conjunction button, and types B.
13. The system displays the input sentence in the display field as a sentence of propositional logic correctly formatted for a Fitch-style proof.
14. Micaela clears the text field.
15. The system ceases to display the line of proof.
16. Micaela looks at her logic worksheet, and enters two well-formed premises of propositional logic and a conclusion. She presses enter.
17. The premises appear in the proof.
18. She presses the "Derive" button.
19. The relevant keyword appears in the input area.
20. Micaela types an invalid derivation.
21. Instead of adding the new line to the proof, the system displays an error message declaring that that it is invalid to infer this proposition.
22. Micaela replaces the invalid derivation with a valid one.
23. The system displays the proposition in the proof and displays the line(s) and rule(s) from which it was derived.
24. This process repeats until Micaela types a valid derivation which matches the conclusion she has previously entered.
25. The system adds the line to the proof and displays a message that the conclusion has been successfully reached.

3.3.2. Persona 2: An experienced User

The purpose of this persona is to gain an understanding of the specific needs of an intermediate user of the system attempting to write out several proofs quickly.

Name: Wes Wrigley

Tagline: University lecturer teaching formal logic to students.

Background:

- *Age:* 32
- *Role:* Logic teacher
- *Field of study:* Mathematical Logic
- *Level of computer expertise:* High

Main points:

- Computer literate and experienced with online theorem-provers.
- Wants to save time and energy and reduce his chances of error when writing solutions to the logic problem sheets that he issues to students.
- Finds natural deduction to be a trivial process.

Goals:

- To produce many logic proofs quickly.
- To export these proofs into PDF files.
- To make changes to proofs that he has already created.

Frustrations & Pain Points:

- He finds the effort of formatting Fitch-style deductive proofs in LATEX very frustrating.

- He finds the writing out of proofs to be very tedious, so the system needs to allow him to work very quickly.
- He needs to make sure that the solutions he provides to students' logic problems actually are valid proofs.

Narrative:

Wes is a teacher of Formal Logic to undergraduates at a good university. His students are frequently tasked with using Fitch-style natural deduction to build proofs to establish a given conclusion from a given set of premises. Part of his role involves writing model answers to these logic problems. He needs to do this for both propositional and first-order logic. He has attempted to solve the problems by hand and use LATEX to format them, but finds this frustrating. He is so experienced with natural deduction now that this level of problem is a trivial exercise for him. However, he does sometimes make mistakes and provides invalid model answers – much to his chagrin. He has been experimenting with the system for several months now, and has found that it is very useful for him to work through many proofs quickly.

The purpose of the accompanying scenario is to illustrate the use of the system from the perspective of an experienced user of both the system and of formal logic. Assume for this scenario that the user already has the system installed on their laptop as a desktop application and has already successfully written and saved multiple full proofs in the past.

Scenario: Normal flow of execution for an experienced user of both the system and formal logic.

1. Wes begins to type code into the user input area. He only needs to type the proposition that he is inferring and specify what type of proposition it is (e.g. premise, assumption, etc).
2. The system automatically infers all other information (such as line numbers and inference rules) and displays them on the proof, allowing for fast typing of the proof.
3. Wes writes an invalid inference on one line of the proof by typing 'A' instead of 'B'.
4. The system displays an error message notifying Wes of the incorrect proposition.
5. Wes continues to type out the rest of the proof without rectifying the mistake.
6. The system displays more appropriate error messages, and only displays the proof up to the most recent valid inference.
7. Wes rectifies the error, compiles the proof, and presses a button to print the proof.
8. The system displays the correct proof and opens a dialogue box allowing Wes to either print the proof to a printer or to a PDF.
9. Wes saves the proof to PDF, navigates to where the PDF is saved, and views it.
10. The proof is displayed in the PDF as it appeared in the output area.
11. Wes presses the load button.
12. The system displays a file chooser, allowing Wes to open any plain text file. Wes opens a proof he saved earlier.
13. The input area now displays the contents of the text file chosen via the file chooser.
14. Wes presses enter.
15. The system compiles the new code into a proof.
16. Wes exports this in the same way as earlier.
17. Wes exits the system and navigates to one of the text files storing a proof. When he opens it, it contains only the code used to generate that proof.

3.3.3: Further Scenarios:

It is useful to consider not just normal flows of execution, but also scenarios in which mistakes are made. The scenarios below present some common mistakes which might be made when interacting with the system, exemplified by some particularly error-prone user of the system, named Tor.

Assume that the user's competency with the system varies a lot based on the weather. Also assume that Tor already has the system installed on their laptop as a desktop application and has opened the system to start a new proof.

Error in flow of execution: Syntax Errors

1. Tor writes what would be a grammatically correct sentence of propositional logic, except for that it is missing all brackets.
2. The system displays an error message stating that this is not a well-formed formula because brackets are missing.
3. Tor adds the brackets.
4. The message disappears and the system adds the line to the proof in the display area.
5. Tor writes a keyword which the system does not recognise.
6. The system displays an error message and enumerates the permitted keywords of the language.
7. Tor deletes the erroneous line.
8. The error message disappears.
9. Tor writes a sentence of the language with many whitespace characters between terms.
10. The system interprets them as it would if there are only one whitespace character.
11. Tor writes a sentence which would be grammatically correct except for that it includes a character not recognised by the system.
12. The system displays that this is not a well-formed formula of the language.

Error in flow of execution: Logic Errors

1. Tor opens a new proof, and switches to predicate logic.
2. The system changes the display to that of predicate logic.
3. Tor attempts to write a sentence of propositional logic.
4. The system displays an error message declaring that this is not a sentence of the language.
5. Tor deletes the sentence.
6. The system ceases displaying the error message.

Error in flow of execution: Other Errors

1. Tor realises that she has forgotten to include a line of the proof earlier on in the proof. She scrolls up to the line preceding the line she wants to add, and presses enter.
2. In the user input section, a new line is opened in that place. Nothing changes in the proof displayed, as the system ignores empty lines.
3. Tor types a proposition into the newly opened blank line, and presses enter.
4. The system displays the updated proof with the new proposition on the correct line, and all other line numbers and references updated.

3.4. Creating the Requirements Specification

At this stage in the project, the objectives had been laid out, related technologies had been researched to gain insight into key design decisions for the proposed system, and personas and scenarios were put together to better understand how the interaction between user and system should play out. The next step was to reconcile this information into the requirements specification.

As there are no business requirements involved here, the relevant types of requirement to be identified were user and system requirements. The user requirements should describe a high-level overview of the requirements of the system in a way accessible to the user. The system requirements should provide a lower-level description of how the system should provide the functionality described in the user requirements. Note that this is without going into detail on the implementation (ELGABRY, 2016).

Owing to time constraints, the process of explicitly enumerating user requirements was skipped. This ran a real risk of missing some significant usability considerations in the final product. However, all understanding of the system's required usability was gleaned from the work carried out earlier in section 3, and this information was referenced extensively whilst developing the system requirements (see below). Furthermore, the testing approach for the final system also evaluated the system's usability. Therefore, this trade-off seemed worthwhile to be able to progress with the project on schedule.

The algorithm for developing the system requirements was as follows:

Take the set of project objectives and split them up into implementable requirements. Next, compare these requirements against the personas and user stories to ensure they captured the usability considerations highlighted by that work. Refine the requirements to reflect the findings of the research. Compare the updated requirements against the review of related work. Refine the requirements to reflect the findings of the review of related work. Then compare the requirements against the original project objectives and update them if there was discrepancy. Repeat this process with the updated requirements from the point of comparing them against personas and user stories. After the repeat, carefully edit the requirements document, frequently comparing back against the research.

The final software requirements specification has been attached as Appendix C. I gratefully acknowledge (Wiegers, 2002) for providing the template used to create the requirements specification.

Some changes were made to the requirements specification during the project. Most of these were minor, but one key change should be noted. Once it became apparent that the optional objectives would not be achievable in the timeframe, they were removed from the requirements specification for the sake of clarity. The final specification represents the essential and conditional goals of the project.

4. Input Language Design

The final system can be split into three key modules: an interpreting system for the input language, a proof-checking system to validate that interpreted input, and a GUI for receiving input and producing output. All input is written in a language specifically designed for that purpose. Understanding this input language is crucial to understanding the design of the rest of the system. In this section, the design process for the input language is presented and discussed. The full language specification can be found in Appendix D.

4.1. Aims & Outcome

The main aim when designing the input language was to mirror the formal logic syntax for each symbol as closely as possible. This was a recurrent theme in the review of related work and is captured in the requirements specification.

In general, a formal language consists of a vocabulary, a syntax, and a semantics. The vocabulary enumerates the terms of the language. The syntax (or grammar) dictates rules for joining them together meaningfully. The semantics assigns some type of meaning to the words and sentences of the language (Siegfried, n.d.). Therefore, these are the three things which the language specification must provide.

In practice, two input languages were implemented. One for propositional logic, called PROPLOG, and another for predicate logic, called PREDLOG. The overall name given to refer to both languages together is simply an amalgamation of these names: PRODLOG. This is also the name given to the overall system. Throughout the rest of this document, either language will be referred to by their name, or as the 'input language(s),' or some variation thereof.

4.2. Keywords & High-Level Syntax

As specified in the requirements, the system should recognise four different classes of proposition: premises, assumptions, derivations, and conclusions. It should also recognise a command to close an assumption. There is no intuitive heuristic which the system could use to accurately infer what class of proposition a new line of input is without explicitly stating it. For example, how could it tell the difference between a new premise and an invalid derivation? Therefore, the user must specify the type themselves. These type-specifying terms are referred to as keywords.

A user-focused approach was taken to representing the keywords, and the language as a whole. The central idea was to ask what a user would say if they were presented with a proof and asked to read out the lines of that proof so that someone else could recreate it. For example, a user might say "Derive A and B from line 5," "Assume that for all x, A x"or "Close the assumption."

This approach suggested two things. First were five candidate keywords matching up to the five type-specifying commands: '#ASSUME,' '#DERIVE,' '#PREMISE,' '#CONCLUDE,' and '#ENDASSUME' – the latter was chosen because it is much shorter than '#CLOSEASSUMPTION.' The '#' was initially included to signify to the interpreter that a keyword was being used. This was not strictly necessary in the final implementation, but has been left in. Had the project been slightly longer, the code would be refactored to remove this and make the language more intuitive.

Second was that this approach suggested a structure to the language. Namely, a very primitive type system, such that every line of input should be of the format '*type content*' or, more precisely, '*keyword proposition*' (except for the '#ENDASSUME' command, which should not be followed by a proposition). The proposition should then be assigned that type when stored as a line of proof.

4.3. Propositions

The next design choice was to select a representation of propositions of formal logic. Originally, the plan was to allow multiple keyboard shortcuts for building compound proposition. The review of related work suggested that this was a poor idea (see 3.2.3.). To avoid confusing the reader by presenting too many options, the decision was made to simplify this to one for each.

A choice had to be made between sticking with the phonetic approach taken in 4.2, or using shortcuts which physically resemble the logical connectives. Non-alphanumeric characters may take longer to type, as the user is generally less familiar with them than with the alphanumeric symbols of the keyboard. Furthermore, differences of convention may cause issue. A user may be used to typing '->' to represent the conditional in one system and '>' in another. To allow for maximal efficiency and minimal confusion when typing input, it was decided that the code for each connective would be chosen using the same approach as in 4.2. Therefore, conjunction became 'AND,' universal quantification became 'FORALL,' an atomic proposition such as A is just 'A,' etc.

4.4. Bracketing Conventions

The next step was to decide on bracketing conventions. The chosen algorithm for the interpreter uses the locations of brackets to help identify syntactically correct sentences. Therefore, it was important to select a bracketing convention which would facilitate this.

Initially, the chosen convention attempted to mirror that of the underlying logic. For example, the convention for writing "it is not the case that A" in propositional logic would be to write '¬A', and so 'NOT A' was the original PROPLOG representation. However, this caused problems down the line when updating the interpreter to work for predicate logic, as the system failed to catch poor inputs which exploited those conventions.

Therefore, the bracketing conventions were made stricter. Now, the same proposition would be expressed as NOT(A). Similarly, though "for all x, Ax" would be represented symbolically as '∀xAx,' it is represented in PREDLOG as 'FORALL(x)(A(x)).'

4.5. Syntax Design

Once the vocabulary had been decided on, a syntax was needed to allow complex propositions to be constructed. The bracketing conventions are a subset of this.

The chosen syntax for constructing propositions and sentences borrows heavily from the definition of a sentence of propositional and predicate logic given in P.B. Magnus' and Tim Button's forAllX:Cambridge (2018) textbook. The reason for this is that the textbook specifies exactly how to recursively generate well-formed formulas of propositional and predicate logic. Assuming that the user is already familiar with the syntax of propositional and predicate logic, this design choice means that they need to learn almost no new information to understand the syntax of PRODLOG. Furthermore, for someone new to logic, practicing writing in PRODLOG will be very similar to practicing writing sentences of formal logic by hand. This is a step closer towards the goal of a user-friendly application to help learners of logic practice their skills.

4.6. Evaluating the Input Language

The strengths of the language are that it satisfies the requirements specification, features proxy terms for the symbols of formal logic which have a close correspondence with those symbols, and defines sentence using methods and terms familiar to the user. That latter point was why the language specification is presented the way it is, instead of adhering to a standardised programming language specification.

Furthermore, because it defines sentences in exactly the same way as the underlying formal logic defines sentences, the two languages have exactly the same amount of expressive power. The user should find that there is no expression of first-order logic that they cannot represent to the system using PREDLOG. This is a key system requirement.

However, there are two areas in which the language would have been improved, given more time. First is that the bracketing conventions can quickly become difficult to manage. For example, the sentence '∀x∀y∀z(Ax ∧ (Ay∧ Az))' becomes 'FORALL(x)(FORALL(y)(FORALL(z)(A(x) AND (A(y) AND A(z))))).' Without the relaxed bracketing conventions introduced in the forAllX textbook, the sentence of first order logic would not be much less complex: '∀x(∀y(∀z(Ax ∧ (Ay∧ Az)))).' But the point is that when hand-writing formal logic, it is at least possible to relax those bracketing conventions. Ideally, the same thing would be doable here. Furthermore, allowing square brackets as well as round brackets could also help the user to better manage this issue.

The second point is as much about the implementation of the whole system as about the specifics of the language. While the decision to limit the representation of formal symbols to one word per symbol does make for a less confusing language, giving the user the option to define how they wish to represent the logical connectives might be preferable still. For example, a user might prefer to type '&' rather than 'AND.' With more time, allowing the user to select their own input terms for each formal symbol would have been an excellent feature to implement.

The reason that those changes could not be implemented in the timeframe was that they would require significant modification of the system's interpreter. It was deemed more important to have a functional version of the interpreter working with the current, slightly imperfect language specification (see Appendix D) and progress with the rest of the project than to spend more time implementing a more advanced interpreter and possibly not finishing the rest of the work.

5. Back-End Design & Implementation

In this section, the key decisions regarding the design and implementation of the back-end system are presented. First, the high-level architecture is presented to provide an understanding of which components make up the system and how they relate to one another. To get closer to a model of the system's implementation, the low-level class structure is presented and discussed. Each key component is then discussed in detail in terms of both the relevant data structures and algorithms.

5.1. Software Architecture: High-Level View

A software architecture is a "set of structures needed to reason about the system, which comprise software elements, relations among them and properties of both" (Bass, 2015). The purpose of this section is to understand which components were used to implement the system, why they were chosen, how they work, and how they relate to one another.

To frame the problem of designing the architecture, an architectural style was chosen. This is a generic 'pattern' of architecture which can be adapted for specific uses. The first step to deciding on a style was to understand the architecture of the system from the user's perspective – from a 'use case view' (tutorialspoint.com, 2019). From a user's perspective, the input is a line of PRODLOG, then a process happens to which the user is not privy, and finally the user observes either a syntax error, a logical error, or an updated proof with a notification of whether the conclusion has been reached.

The second step was to understand the architecture from a 'logical view,' which helped to conceptualise the key components that needed to be in place to support the user view, as well as how they should interact with one another. Inputs would be received through the user interface and passed to the 'black box' by a controller. Within the black box, a lexer would convert the input into tokens (or throw an exception if the input was not in the PRODLOG vocabulary) and a parser would convert the tokens into an abstract syntax tree (or throw an exception if the syntax of PRODLOG was violated). The tree would be converted into a proposition data structure and a proof line data structure, which would be tested by the inference checker, generating the output.

To derive both perspectives, the system requirements and research sections were closely inspected, alongside further research conducted into the typical components of a programming language interpreter. To reconcile these two perspectives, a combination of two architectural styles was chosen for the system. As is discussed in depth in the later section on GUI design, a model-view-controller (MVC) style was chosen to separate out the complexity of managing the GUI from that of managing the back-end, allowing each to be developed independently and to maintain the 'black box' perspective of the user

view. In this style, input is passed through the view – the GUI – to a controller, which passes it to the appropriate part of the back-end system – the model.

Because the data will take a roughly linear path from the controller through the back-end system, a batch-sequential (BS) architecture style was chosen to represent the system *inside* of the black box. This style is characterised by a linear flow of execution and the complete processing of the data by a module before passing it to the next module (tutorialspoint.com, 2019a).

While this approach limited the possibility of making efficiency gains by adding concurrency to the system, it allowed for a clear programming logic, and for each component of the system to be developed as its own separate and independently testable module. Overall, this had the benefit of helping to manage the complexity of the system, making the implementation significantly more manageable and allowing for more modules to be added if additional data processing was required.

The high-level flow of execution representing the merging of these two architectural styles can be modelled diagrammatically as follows:

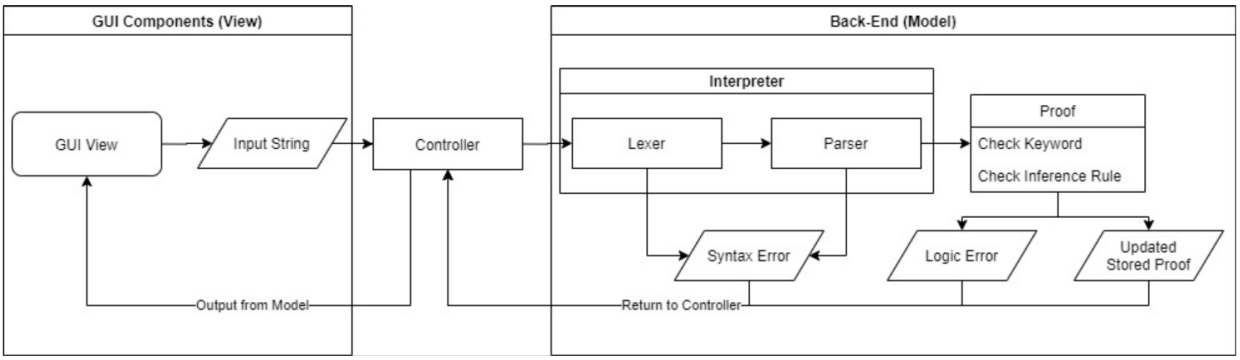


Figure 3: MVC / BS Architectural Style for PRODLOG system

As the diagram shows, the system is highly modularised with a linear flow of execution. This model offers a hopeful initial representation of the system requirements as a set of components and relations. The next was to translate this into a more detailed class structure which could be used as the basis of implementation.

5.2. Software Architecture: Low-Level View

A set of elements constituting the system has been presented, and the relations between them have been gestured to. These concepts will now be made more precise. To move closer towards implementation, a class structure diagram was constructed, demonstrating the classes required to represent those

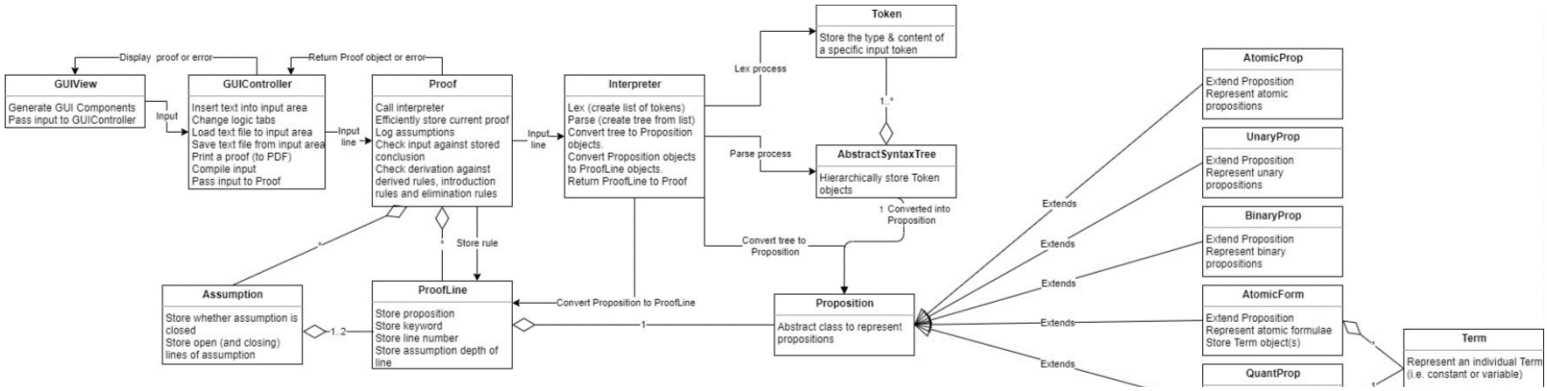


Figure 4: Class structure of PRODLOG system

components, their general functions, and the precise relations between them (Visual-paradigm.com, 2019).

When attempting to model the linear flow of data through the system in more detail, an issue was flagged: two different classes would have to communicate with the GUIController class; the Interpreter and the Proof. As this introduced more scope for error, it was decided that only one class should interface with the controller so that any errors in input and output could be more easily traced. Passing the input from the Proof to the Interpreter made more sense than the Proof passing the output back to the Interpreter to return it to the controller. So, the Proof class was chosen for this purpose.

The next subsection will be dedicated to detailed considerations for the design and implementation of the classes identified above, beginning with the Interpreter.

5.3. Interpreter

The input language has been described, and the class structure has been presented. The interpreter takes that language and converts it into the relevant data structures as indicated above: first into Token objects, then into an AbstractSyntaxTree, then into a Proposition, then into a ProofLine. To be able to successfully understand the interpreter, it is necessary to understand the data structures to which it converts the input. Therefore, the Proposition and ProofLine classes are presented before an in-depth discussion of how the interpreter was designed and implemented.

5.3.1. Proposition Superclass & Subclasses

Formal logic runs on propositions, so these needed to be represented to the system. The system needs to perform operations on propositions, and not all operations will be common to all types. For example, a binary proposition will need a method to get both its left and right operands, while an atomic proposition will require neither.

This motivated the design of the Proposition superclass; an abstract class with methods common to all Proposition subclasses. Proposition itself stores the type of the Proposition (i.e. the code term for the connective), the Unicode representation of its main logical operator (if there is one), and a unique code to identify that Proposition (discussed in more depth later). Each logical connective can then be represented as an extension of Proposition with added methods tailored to the specific needs of that Proposition subclass.

To limit the number of types in the system, general classes were created to be reused. For example, BinaryProp can be used for any Proposition with a binary connective as its main logical operator, and QuantProp is similar for any Proposition with a quantifier as its main logical operator.

The Proposition classes were designed to be recursive data structures. At their base, they have either an atomic proposition or an atomic formula. Layers of non-atomic Proposition objects are connected hierarchically to compose a specific compound Proposition object. Inspiration for this came from two sources. The first was realising that constituent propositions need to be extracted from compound propositions in formal logic, and this would be much simpler to implement as a hierarchical tree of Proposition objects than, for example, using the interpreter to extract the proposition.

The second source of inspiration came from inspecting the recursive definition of a sentence in propositional and predicate logic. For example, if A is a sentence of propositional logic, then ¬A is a sentence of propositional logic. If A and B are sentences of propositional logic, then A Λ B is a sentence of propositional logic, etc. (P.D. Magnus and Tim Button, 2018). The result is that compound propositions of

formal logic can themselves be viewed as trees, with their main logical operator as the root and atomic propositions as the leaves. It seemed logical to implement the Proposition classes the same way.

Slightly more complexity was introduced when designing the Proposition classes of predicate logic. As mentioned earlier, the basic unit of predicate logic is an atomic formula taking one of three forms: absurdity, a relation between a predicate and one or more terms, or an identity relation between two terms. Similarly, propositions whose main logical operator is a quantifier not only store the proposition ranged over by that quantifier, but also the term ranged over by that quantifier.

To solve this problem of storing terms, a Term class was designed to represent an individual term. Each QuantProp stores not only a Proposition, but also one Term object. Each AtomicForm stores a list of Term objects and can be one of three types. If the type is absurdity, the list is empty. If the type is identity, the list has two elements, with the first being the lefthand side of the identity statement, and the second being the righthand side of the identity statement. If the type is predicate, then the list must contain at least one element, but may contain an indefinite number of elements.

All these classes inherit and contain basic functions such as getters and setters. Especially useful were the getters and setters for constituent propositions of compound propositions, as these are depended on a lot in the proof-checking system. The Proposition subclasses also implement a generateKey method, which returns a String containing a unique key for that proposition. The significance of this is discussed in the design of the Proof class.

Every Proposition implements the abstract DeepClone method, which uses the immutability of String objects in Java to create a clone of the object with the exact same state, but a different reference in memory. The significance of this is discussed further on in the implementation of the formal rules for predicate logic.

5.3.2. The ProofLine class

The Proposition class alone does not store sufficient information to be recorded in a proof. For example, if some Proof class contained a list of its constituent Proposition objects, it would need to separately store the line number of each Proposition, as well as the rule used to derive that Proposition. Instead, the ProofLine class was created, which stores the line number, keyword, Proposition, key (using the Proposition's generateKey() method, which is discussed in more depth later), assumption depth and derivation rule. The main idea behind this class is that the ProofLine should contain all information needed for the proof system to begin checking the inference rules.

To make the class more usable, a few extra methods were added. For example, take the following two lines of input code: '#DERIVE (A AND B)\n #PREMISE C,' where '\n' is a newline. The introduction rule for the conjunction might reference lines 2 and 3 of the proof. But if a new premise is added to the start of the proof in the line of code after the derivation – and this is permitted in PRODLOG to save the user from having to scroll back to the start of the proof if they want to add another premise – then those line references will be off by 1. As a result, a changeRuleNumbers method was implemented, which takes a Boolean for a parameter. If the Boolean is true, then all line references containing numbers in the rule String are incremented by 1. If it is false, they are decremented by 1. A method changeLineNumber works the same way for the line number.

The ProofLine class also contains a print() method, which is discussed in more detail during the later section on implementing the Proof.

5.3.3. Elements of an Interpreter

Research into building an interpreter began by investigating how programming languages are implemented. Within an interpreted programming language, the interpreter generally consists of a lexer, a parser and an action tree (Wold, 2017). The purpose of a lexer is to tokenise each individual element of the language and identify what kind of token it is (E.g. connective, keyword, symbol, etc), filtering out or otherwise noting if something is not part of the language's vocabulary. The parser then converts these tokens into an abstract syntax tree (AST). This is a hierarchical structure based on the order of operations. The action tree adds more context to the AST such as recording what each function returns (Wold, 2017). The AST could be converted directly to a ProofLine in PRODLOG, so this step was skipped. The end goal of the interpreter is to complete the transformation from String of user input to ProofLine data structure.

5.3.4. Designing the Interpreter

5.3.4.1. The Lexer

As stated earlier, the lexer – or lexical analyser – should take as input a string of characters and output a collection of tokens. A token, in this context, is defined as some recording of the lexeme (the element of the vocabulary) with extra information. To this extent, a Token class was implemented. The only information required was the lexeme itself, its type, and an index to help keep track of the tokens.

There are two steps to designing a lexer. The first is to design a procedure for the system to discern individual lexemes from the input String. The second is to tokenise those lexemes.

The simplest way to carry out this first step is to iterate through the string, and split it based on pre-defined breakpoints (Khandelwal, 2019). In PRODLOG, there are two key breakpoints: spaces, and brackets. This is why the bracketing conventions in the language specification have to be so precise.

The method lex in the Interpreter class does exactly that. It takes a line of user input as a parameter, iterates through it character by character, and analyses each character using two switch blocks.

The first switch block checks for commenting. To enhance the system's usability, both inline ('//') and multiline (open: '/*' close: '*') are permitted. Anything contained within a comment is completely ignored by the interpreter, as it is not used by the rest of the system.

Until a breakpoint is reached, each character is appended to a StringBuilder. Once a breakpoint is reached, the contents of the StringBuilder are emptied and passed to the lexHelper method for tokenisation. To keep track of the tokens, an ArrayList of Token objects is used. An ArrayList was chosen because a dynamically-sized data structure was required where the only important operation was keeping order. If the breakpoint is an open bracket or a close bracket, then this is also tokenised and added to the ArrayList.

To check if the lexeme is in the vocabulary of the language, its type is identified from a list. If the type is successfully identified, then a new Token object is created, the Token is added to the list, and the records of token indexes is incremented and returned. Note that the interpreter has a field variable storing whether propositional or predicate logic is to be used. Terms specific to each logic will be flagged as an error if the interpreter is set to the wrong logic.

From comparison with the list, the type could be as follows:

- Keyword;
- Binary connective;
- Unary connective;

- Atom (if the lexeme is absurdity);
- Atom (if the logic is set to PROPLOG);
- Quantifier (if the logic is set to PREDLOG).

This is sufficient for PROPLOG, but not for PREDLOG. For example, the breakpoint approach cannot handle a string of terms 'abc' in the atomic formula P(abc). Furthermore, an identity statement such as 'a=b' does not contain any spaces. If the logic is set to PROPLOG and no match is found in the above list, an IllegalArgumentException indicating a syntax error is thrown, and the cause of the error is identified in the error message. If the same situation arises and the logic is set to PREDLOG, the analyzeLexeme method is called to check for these special cases.

This method takes the String lexeme as input, and returns a two-dimensional String[][] array. It iterates through the lexeme and tries to identify if each component is a predicate, an identity symbol, or a term. If it is, then the type is recorded as the first element of a new String array, the substring of input is recorded as the second element, and the array is added to the 2D array to be returned. Note that as trailing numbers are allowed at the end of terms and predicates, these are also accounted for. This process repeats until one of two things happens. The first is that all tokens are successfully identified, and the 2D array is returned to the lexHelper. The second is that an error is found – a term which is not in the vocabulary of the language. In this case, a different 2D array is returned, which contains only one array. This array contains the String "EMPTY" twice.

The lexHelper method will then check the array returned by analyzeLexeme. If it contains the array containing EMPTY twice, then an exception is thrown with an informative error message, because the lexeme is not in the vocabulary of the language. Otherwise. The returned array is iterated through, and a new Token object is created for each element and added to the list of Tokens.

One of two things will now happen in the system's execution. Either an exception will have been thrown, in which case it will terminate and the error message will be passed up to the GUI controller for the user to see where they went wrong. Or, this process will repeat for the next lexeme passed to the lexHelper. Once all lexemes have been iterated through, the complete ArrayList of Token objects is returned to the calling method – in this case, the Proof class. With the Token list created, the Proof class can now pass the result to the Parser.

5.3.4.3. The Parser

The parser imposes structure on the tokens. This means that in the same way that the lexer found vocabulary errors, the parser will identify any grammatical errors. The aim of the parser is to produce an Abstract Syntax Tree, from which Proposition data structures can easily be created.

5.3.4.3.1. The Abstract Syntax Tree

Before going any further, the implementation of the AbstractSyntaxTree class should briefly be discussed. Each AST stores a root of type Token, a left subtree, a right subtree, a Boolean variable to determine if the tree is empty, an ArrayList of Token objects with type term, and a Boolean declaring whether or not that tree has terms. Depending on the constructor of the tree, one or more of these values may be initialised to null.

There are four such constructors. The first takes as parameter a root Token, a left subtree, and a right subtree. This allows for the creation of ASTs representing binary propositions. The second takes only a root and a left subtree, for handling unary connectives, keywords, and quantifiers. If it is a quantifier, its array of term Token objects will contain exactly one element. The right subtree is initialised as an empty tree. The third takes only a root – this can deal with atomic formulae or the '#ENDASSUME' keyword. If

it is of type predicate, then the term array will contain at least one element, but may contain an indefinite number. If it is of type identity, the term array will contain two elements. Both of its subtrees are empty. Finally, there is a constructor which takes no parameters at all. In this case, the Boolean isEmpty is set to true, hasTerms is set to false, and all other fields are initialised to null.

To illustrate this, figure 5 depicts an AST for the input line, "#DERIVE ((A(a) AND B(bc)) OR D(d))":

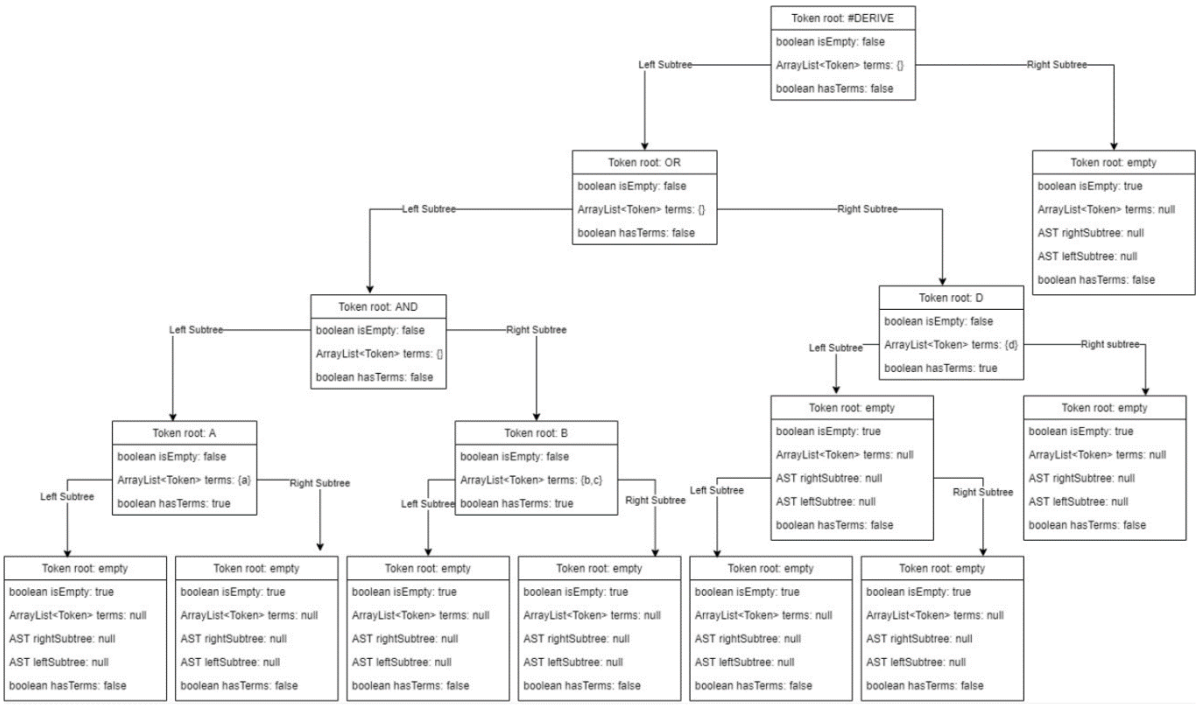


Figure 5: Graph Depicting an AbstractSyntaxTree for the input "#DERIVE ((A(a) AND B(bc)) OR D(d))"

It should be clear that this is an appropriate data structure to use as a hierarchy for organising Token objects when considering the recursive definition of sentences in PRODLOG. The correspondence between subtrees and Proposition objects contained within other Proposition objects meant that once the tree has been constructed, a relatively simple algorithm can convert it into a ProofLine object.

5.3.4.3.2. The Parser Proper

With the intended output identified, the design and implementation of the parser can now be discussed. The parsing strategy used is top-down left-to-right recursive descent parsing. This means that the algorithm tries to find the most general token first (the keyword, followed by the main logical operator of the outermost compound proposition, etc.) by reading through the token list from left to right (Tomassetti, 2017), and recursively working through the non-terminal elements of the input (i.e. anything which is not an AtomicProp or AtomicForm) (GeeksforGeeks, 2019). The main motivation behind this choice is that it allowed for relative ease of implementation without much of a performance cost, given the simplicity of PRODLOG's grammar.

There are two key principles driving this parser. The first is that simple propositions should be directly solvable (i.e. easily converted into a tree). The second is that the main logical operator of compound propositions can be identified by counting brackets (P.D. Magnus and Tim Button, 2018). Hence the bracketing conventions of the input language had to be implemented so strictly. At the highest conceptual level, the parser is scanning the list of Token objects from left to right, identifying the main logical operator, creating a new AST with that main logical operator as the root, and then setting its left

and right subtrees as the trees returned by performing a recursive call on each sub-list on either side of the main logical operator (minus the outermost brackets).

More forensically, the parse method takes as input the full ArrayList of Token objects. Since any grammatical sentence of the language must begin with a keyword, the parse method attempts to construct an AST with the keyword at the root. If the first Token of the array is not a keyword, an exception containing an appropriate error message is thrown up to the user interface.

The parse method is only applied to the keyword at the start of the Token array. Its subtree, and all subsequent subtrees, are generated via the parserHelper method. If parse successfully constructs a tree with a keyword at the root, it constructs a sub-list containing all elements of the list of Token objects except for the keyword. It then passes this list to the parserHelper. The parserHelper checks the type of the first element of the sub-list and behaves differently dependent on the result.

Several types are converted into an AST, either directly or via recursion, with relative ease. If the type is atomic, then the problem is directly solvable, and a new AST with the atom as the root is returned. If it is a unary connective, then a new AST can be returned with the connective as the root, a sub-list can be produced with the unary connective removed, and the parserHelper can be recursively called on all of the elements of its sub-list. The result is set as the left subtree of the unary connective tree. The predicateHelper method is called to directly convert a predicate-term formula into a tree. This method first checks that the syntax of the predicate is correct, ignoring the terms. Then it goes into the terms and adds them to the term Token array of the returned tree. The quantifierHelper processes quantified propositions in a similar manner to the unary proposition, but also stores the one variable attached to the quantifier. The identityHelper checks that the first element is of type term, the second is identity, and the third is type term, before returning a tree with identity as the head.

The ‘relative ease’ of the previous operations is relative to the situation where the first element of the list passed to parserHelper is a Token of type open bracket. To handle that case, a recursive strategy has been carefully designed and implemented. The openBracketHelper method initialises a bracket counter to 1 and begins iterating through the Token array from the first element after the open bracket. Because all other input types have been accounted for by the parser helper, the system is searching for a binary connective. Every time an open bracket is encountered, the bracket counter is incremented. Every time a close bracket is encountered, the counter is decremented. When a binary connective is reached, it is the main logical operator of that sentence if and only if the bracket counter is 1 at that point. If that is the case, then we can split the Token list into two sub-lists. One containing the lefthand Token objects (minus the first opening bracket), and one containing the righthand Token objects (minus the last closing bracket). A new AST is created with the binary connective at the root. The left and right subtree are formed by calling the parserHelper on each sub-list.

In theory, all types of Token object can be solved using one of these methods in a grammatical sentence of PRODLOG. Therefore, the parser is guaranteed to either produce an AST, or throw an exception carrying an error message which can be passed back to the user.

5.3.4.4. The toProp and toLine methods

The user input has been converted into a list of Token objects, each of which contains a lexeme corresponding to the vocabulary of the language. These Token objects have been organised hierarchically in an abstract syntax tree, using the AbstractSyntaxTree class, and their organisation is syntactically correct for the language. The final step for the Interpreter is to transform this input into the ProofLine class discussed earlier.

5.3.4.4.1. toProp

The Proof class receives the AST from the Interpreter, and calls the toLine method of the Interpreter on the tree, passing it extra information stored in the Proof such as the line number and the current assumption depth. Initially, the String "Invalid inference" is passed instead of a rule, as the rule has not yet been identified. The toLine method immediately calls the toProp method on the left subtree of the AST. That is, since the root of the AST is always a keyword, the 'proposition' portion of the AST is passed to the toProp method.

The toProp method recursively traverses the tree from the root to its leaves in preorder (i.e. visiting the root, then the left subtree, then the right subtree). For each root that it visits, it creates a new Proposition object using the type of the root Token, and stores the lexeme of the root Token as the proposition String. If it is a compound Proposition (i.e. BinaryProp, UnaryProp or QuantProp) then the Propostion(s) it stores will be set as the Proposition(s) returned by applying the same procedure to its subtree(s). When an AST containing a list of term-type Token objects is created, the helper method getTermArray is called on the tree. This simply goes through the list of term-type Token objects and creates a corresponding list of Term objects, which it returns. This list of Term objects is then stored as the Proposition object's list of Term objects. The traversal will terminate when all leaf nodes of the AST have been visited and converted into Proposition objects.

The Proposition portion of the AST depicted in Figure 5 above can be visually represented as follows:

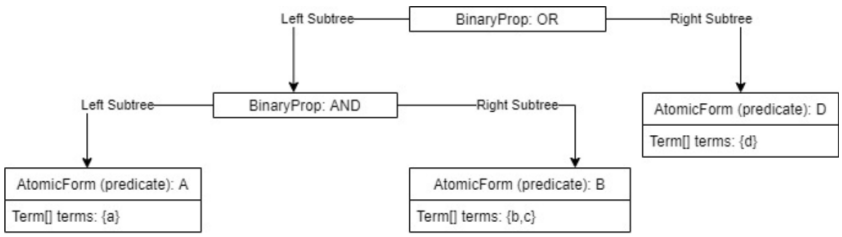


Figure 6: Graph Depicting the input "((A(a) AND B(bc)) OR D(d))" as a Proposition data structure

5.3.4.4.2. toLine

The final step for the interpreter is to convert the Proposition output by toProp to a ProofLine. This is simple for PROPLOG. A ProofLine attaching the information passed to the toLine method is sufficient. The method is more complex for PREDLOG. In first order logic, for a sentence containing terms to be grammatical, it must contain no free variables. That is, every variable which occurs within the sentence must be bound by a quantifier. Ensuring this condition is met is the purpose of the propChecker and propCheckerHelper methods.

The propChecker method takes as input the Proposition to be checked for free variables and returns a Boolean. If any free variables are found, false is returned. Otherwise, true is returned. The method works by checking the type of the Proposition, and either returning true, calling the propChecker on the Proposition(s) stored within that Proposition, or calling the propCheckerHelper on the Proposition stored within that Proposition along with its Term array.

True is returned if the type is Atomic, as there are no terms to speak of. The propChecker is initially called on the Proposition(s) stored within a BinaryProp or UnaryProp. This is because until we encounter a quantifier, predicate or identity statement, there are no terms to speak of. If the type is quantifier, the term is checked to ensure that it is a variable (as constants should not be bound by quantifiers). If it is, then both the Proposition stored within the QuantProp and an array containing its Term are passed to

the propCheckerHelper. If the type is predicate or identity, then the whole AtomicForm is passed to the propCheckerHelper.

The final method to be discussed in the Interpreter class is the propCheckerHelper method itself, which takes as parameters a Proposition and a Term array. Like the propChecker, propCheckerHelper returns false if a free variable is found, and true otherwise. The method works by recursively calling itself on the sub-Propositions of its parameter Proposition. Every time a quantifier is encountered, the variable bound by that quantifier is added to the Term array passed in the recursive call. The algorithm recursively works through the entire Proposition in this way until it reaches an AtomicForm. If the AtomicForm is an atom, then true is returned (as this is absurdity, which stores no terms). If a predicate or identity statement is reached, then the term array of the predicate or identity statement is iterated through. Every time a variable is found in this array, it is compared against the list of variables passed to the propCheckerHelper in the recursive call. That is because this list contains all the variables which are in the scope of a quantifier at that part of the Proposition. If no match is found for a variable, then false is returned, as a free variable has been identified. If this happens, then an Exception is thrown containing an appropriate error message. Otherwise, true is returned.

If the propChecker returns true, then the Proposition has been validated as a grammatical sentence of predicate logic. This means that the toLine method can now add the Proposition to a new ProofLine.

5.3.4.5. Overview of Interpreter Structure

A detailed presentation of the design and implementation of the Interpreter has been given. It included a lot of moving parts, so a diagram describing the flow of execution through the Interpreter class is presented in figure 6 for ease of understanding. Observe how for each component, in keeping with the batch-sequential architecture style, all helper methods finish running and a data structure is completed before passing output to the next module. For the lexer, a complete ArrayList<Token> is output. For the parser, a complete AST is output. For toLine, a complete ProofLine is output. Also note the number of helper methods; conscious efforts have been made to keep the class as modular as possible, and to try to minimise the complexity of any one method.

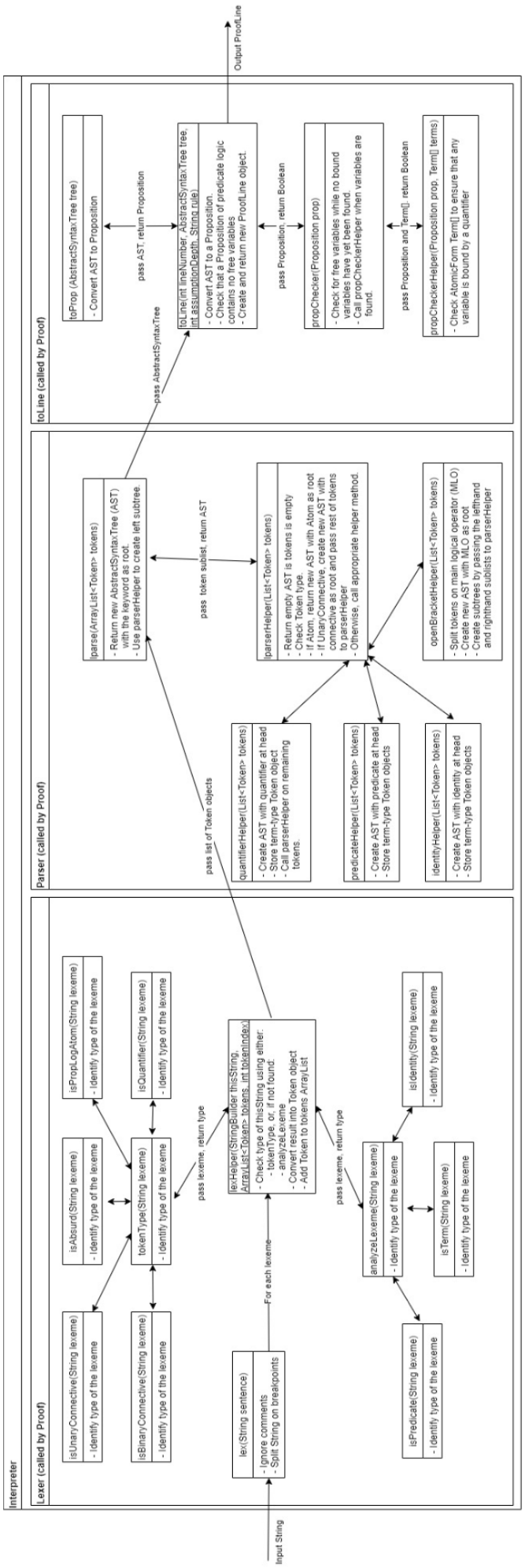


Figure 7: Complete depiction of the Interpreter system

5.3.5. Interpreter: Final Considerations & Justifications

The process of transforming a String of PRODLOG code into a ProofLine data structure has been presented and described in detail. Before moving on to discuss the proof-checking system, a few important points should be raised.

First is that the above overview is by no means comprehensive. Some implementation details are not included, including entire helper methods. Where details have been ommitted, it is usually because those details are not interesting – not because they are not useful.

Second is that the Interpreter as a class contains no constructor, and all its methods and class variables are static. This decision was taken because on the current implementation, there is no benefit to being able to store, say, the AbstractSyntaxTree for a line of proof interpreted four lines ago. All relevant information for the proof-checking system is stored in the ProofLine. There is no need to store Interpreter 'objects' recording any of the steps taken to create that output. Instead, the Interpreter simply offers a suite of functions which can be called upon as necessary.

This is a strength of the Interpreter from the point of simplicity. Fewer objects in memory mean fewer objects to manage. However, under a more complex implementation of the whole system, allowing Interpreter objects to be created and storing the steps used to derive each ProofLine could have lead to efficiency gains. If large amounts of input remain the same across multiple compilations, storing the outputs of the different Interpreter components in Interpreter objects could allow compilation to be skipped for those lines of input – or at least reduced. As the system is currently implemented, no such optimisation exists. Every time the input is entered, the system treats it as if it is seeing it for the first time. With more time to work on the project, this would be an area to explore in depth.

Third is that the design for the Interpreter took inspiration from different sources and was based in research, but was ultimately designed and implemented from scratch. A vast number of automatic lexer and parser generators are available online (Wikipedia Contributors, 2019). There were two reasons that these were not used.

First is the typical complaint that the grammar of PRODLOG was a work-in-progress, and that each change to the grammar would require re-running the generator and reconnecting the result to the system (Wold, 2017). This would waste time.

The second and more important reason was that the proof-checking algorithm designed for this system relies on the specific implementation of the many different classes comprising the system. Maintaining more control over the output of the interpreting system and the process used to get there provided greater flexibility to implement the proof checker as planned, without having to tailor it to the output of a procedurally-generated interpreter, which ultimately resulted in better implementation.

5.4. Proof-Checking

The requirements engineering process was presented, followed by a specification of the input language and a detailed examination of how the system converts that input language into ProofLine objects. The proof-checking system is the next significant component of the software. The purpose of this module is to take a ProofLine and follow a course of action based on its keyword. For everything but a derivation, this means performing some simple operations and then adding the line to a proof. For a derivation, this means something more complex: identifying the specific inference rule used to derive that line of the proof. This is where this system comes into its own relative to the other proof-checkers within the same

area. Automated identification of the inference rule for propositional and predicate logic was not encountered as a feature of any of the systems studied.

In this subsection, an overview of the proof-checking architecture is presented. Key design and implementation decisions for the Proof class are laid out. The proof-checking procedure itself is then presented, along with an overview of the method used to create its subroutines and some examples of particularly interesting proof-checking algorithms.

5.4.1. First Principles of Proof-Checking

The purpose of the Proof class is to identify the type of a ProofLine, and to store the ProofLine. Between these two processes, the system should check that any ProofLine with type derivation is a valid derivation, given the contents of the Proof so far. If it is, then the system should identify the specific inference rule used to derive it. If it is not, then an exception should be thrown containing an error message warning of an invalid derivation. The challenge was to efficiently represent those rules. Two sources were drawn on when designing the Proof class: the requirements specification, and the rules of formal logic that the proof-checking system represents.

As stated in the introduction to formal logic, every logical symbol has an introduction rule, which allows a proposition containing that symbol as its main logical operator to be validly asserted on a new line of proof. Similarly, every symbol has an elimination rule, allowing a proposition to be validly asserted on a new line of proof because another proposition in the proof has that symbol as its main logical operator. Furthermore, there are derived rules. These are 'shortcuts' which allow the logician to validly insert a line of a proof by referencing a pre-proven pattern of inference, instead of having to go by way of proving that pattern of inference directly using introduction and elimination rules.

The first principle of proof-checking employed in this system is that the inference rules of propositional and formal logic essentially describe search algorithms. Most inference rules reference another line of proof. Consider the following output from the final system:

1	A	Premise
2	B	Premise
3	(A ∧ B)	∧I 1,2

Figure 8: Instance of conjunction elimination

Line three uses the conjunction introduction rule. It can do so because the left conjunct of the new line is already contained within the proof, and the right conjunct of the new line is also contained within the proof.

We can describe a primitive algorithm to determine whether conjunction introduction is permitted. For a conjunction: First, search the proof for the left conjunct. If the left conjunct is present and is not contained in a closed assumption, search the proof for the right conjunct. If the right conjunct is also present and not contained in a closed assumption, then conjunction introduction is valid.

A similar process can be carried out for all introduction and elimination rules. For example, take one of the most complex elimination rules: existential elimination. The crux of the algorithm hinges on being able to substitute the variables in the quantified statement for a constant, remove the quantifier and search the proof for the result.

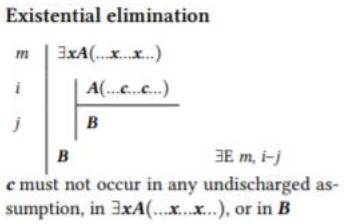


Figure 9: Existential Introduction. Diagram copied from (Magnus and Button, 2018)

An algorithm can be described based on the above rule. Assuming that a record has been kept of all constants which are not contained in undischarged assumptions, begin by iterating through each term in that record. Check that the term is not contained in the new line or the existential-bound proposition. If it is, continue to the next iteration. If it is not contained in the new line or existential-bound proposition, then replace all instances of the existential-bound term with that term in the existential-bound proposition and remove the quantifier or existential-bound term. Search the proof for the result of the substitution. If it is not found, continue to the next iteration and repeat the process. If it is found, check if it is the opening line of an assumption. If it is not, continue to the next iteration. If it is, check if the last line is the newest line of the assumption. If it is not, continue iterating. If it is, then the new line is an instance of existential introduction.

In this case, as in the case of all inference rules, being able to efficiently search the proof for a given proposition is critical. When hand-drawing simple proofs, this is easy; the author need only look. When deciding how to store a proof in a manner conducive to efficient search in a piece of software, more care must be taken. Working through the inference rules in this way also helps to build a clearer picture of which data structures need to be stored if those algorithms are going to be implemented in code.

As well as being efficient with regards to search, there are other considerations. The proof must be searchable by proposition to be efficient. But its order must also be recorded, as it is important to be able to iterate through the proof based on the line number. It should also be possible to search for a line of proof based on the line number, as this will be required by some inference rule implementations.

This need for efficient search by proposition and by line number lead to the first critical design decision: within the Proof class, the proof itself is represented using two data structures to keep track of all the constituent ProofLine objects. The first is an ArrayList<ProofLine> This guarantees that the lines are stored in order. Each line of the proof is indexed at its own line number minus one. Accessing a given ProofLine by its line number takes O(1) time, and iterating through the entire list in the worst case takes O(n) time.

As stated, the Proof should also be searchable by Proposition. Each Proposition implements the generateKey() method, and calls it in its constructor. The key of a Proposition is a String representation of that Proposition, but with more brackets than are used in the line of PRODLOG which generated that Proposition. For example, '(A AND (B OR C))' has the key '((A)AND((B)OR(C))).' The extra brackets were needed in the earlier stages of the language specification when its bracketing conventions were looser. This key is unique to the proposition. Therefore, a HashMap<String, Integer> is the second data structure used to store the ProofLines, where the String is the key and the integer is the line number. Searching for a given key in the proof takes O(1) time, and returns an integer which can be used to access a ProofLine in the ArrayList<ProofLine> in O(1) time.

To see why this implementation is so powerful, consider the above example of conjunction introduction. If the user inputs (A AND B), then the system needs to be able to check if the proof contains both A and B. Under this implementation, it can create a new AtomicProp A, and use its generateKey method to create a key. With this, it can efficiently search the HashMap and retrieve the line number of the A in the

Proof. Furthermore, it can efficiently access the ProofLine itself and check that the line is not in some closed assumption, using the assumption depth and some helper methods. It can follow the same procedure with B, and then it has confirmation that it is valid to introduce the conjunction of A and B, with the specific line numbers for reference.

Leveraging this implementation to efficiently search the proof is the cornerstone on which the further proof-checking design decisions were built.

5.4.2. Architecture of the Proof Class

When discussing the Interpreter, it was possible to work through the entire implementation. The Proof class is significantly more complex and consists of several thousand lines of code, so a different approach is needed. The full Proof class is represented in the following diagram (where the top compartment contains field variables and the bottom compartment contains methods):

Figure 10: UML Diagram of the Proof Class

The approach taken here will be to discuss the flow of execution through the Proof class. There are three phases to this. First is passing the input through the Interpreter and collecting the result; second is using the keyword of the resultant ProofLine to identify and execute the relevant function to store the line of the proof; third is that the Proof finishes running and the GUIController takes the updated Proof object and uses it to generate a visual representation of the Proof. Following a more detailed description of this process, the overall methodology used to derive each proof-checking subroutine will be presented, with suitable examples.

The entry point to the functionality of a Proof object is through its readInput method. This takes as input a String containing the user's input, and an integer representing whether the Interpreter should be set to PROPLOG or PREDLOG to process that line. The GUIController class feeds input to the Proof on a line-by-line basis. So, the String userInput will contain the material needed to create a single ProofLine object. Proof passes the input through the three components of the Interpreter, and then examines the keyword of the output line to decide how to handle it.

The requirements specify that premises, conclusions, assumptions, and derivations should be treated separately. As such, a switch block identifies the specific keyword and different helper methods carry out the relevant function. There is no theoretical limit on the number of premises a proof may have, so a

premise is simply added to the proof. Note that from hereon out, 'added to the proof' shall be taken to mean that the relevant data structure has been added to the ArrayList and HashMap which store the ProofLine objects, and relevant admin around doing so (such as incrementing line numbers) has been taken care of. For conclusions, it is a simple enough process to replace a previously-stored conclusion with another, ignore a new one if the new one matches a stored one, or to store a first conclusion. Adding a new assumption is also straightforward: any proposition of the language can be assumed at any time. A method endAssumeHelper helps with the admin of closing assumptions. It does a bit more work, but that will not be focused on here.

For any of those keywords, the Proof will finish the storage admin, print the Proof to the command line, and then the readInput method will terminate. The GUIController will use the updated Proof to generate a Fitch-style proof for the GUI.

Most of the Proof class concerns the '#DERIVE' keyword. That is not to say that just because handling these other keywords is a relatively simple process compared to derivation, it is not complex in its own way. Rather, with a class as complex as Proof and a limited page count, prioritisation is necessary. When the user derives a new line of proof, the system attempts to identify which inference rule was used to derive that line, and that process is far more interesting than the others.

The ProofLine is first passed to the checkDerive method. This method works by identifying the main logical operator of the line's Proposition, identifying the conditions which must be present in the proof for that operator to be introduced, and then checking whether those conditions are satisfied. If they are, then a String containing the reference for the rule – the text which displays at the rightmost edge of a Fitch-style proof – is returned. Thus, the first step to checking a derivation is to identify whether the rule is just the introduction of the new line's main logical operator.

If the introduction rule cannot be satisfied, then the ProofLine is passed to the eliminationRules method. This method is slower than the checkDerive method. It works by iterating through the entire Proof, line by line. First it attempts to apply the derived rules to the current line of the proof, given the output. This is because the derived rules are generally simpler than the elimination rules, and therefore quicker to execute. If a derived rule can be used to infer the new line of input, then that rule is returned. If no derived rule can be identified, then the algorithm checks, for the current line in the iteration, whether the new line could be validly asserted if that line was eliminated and, if so, whether the contents of the proof so far are sufficient to eliminate that line. If so, then the rule is returned. Otherwise, the String "1" is returned.

The checkDerive method will therefore return either a String containing "1", or a String containing the rule used to derive the line. If the former is the case, then an exception is thrown and the error message for an invalid derivation is passed up to the user. If the latter is the case, then a helper method adds the line to the proof.

In the simplest terms possible, the flow of execution through the proof-checker can be summarised as follows:

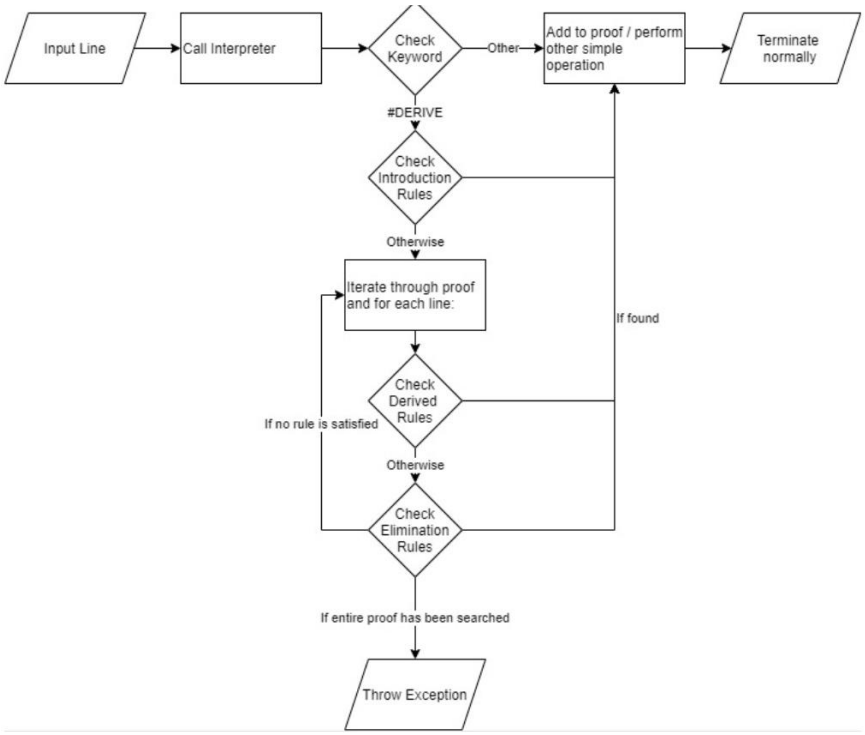


Figure 11: Diagram Representing the Flow of Execution through the Proof-Checker

5.4.3. Designing the Inference Rules

The mechanism driving the proof-checker has been presented. Now, the specific method used to design the inference rules will be discussed.

First, a list was compiled of all the rules which should be included in the system. There are all the introduction, elimination and derived rules highlighted in the FORALLX textbook which describes the systems of propositional and predicate logic used here. This list is included in the requirements specification.

The next step was to examine each rule, examine the data structures currently present in the Proof, and work out how to map between the two of them. This process involved describing each inference rule of formal logic in natural language, and then refining it until it matched the system implementation. If more data structures were required to do so, then more data structures were introduced.

Four examples of this process and the resultant algorithms are now presented. These examples are typical of the process of implementing the inference rules.

5.4.3.1. Implementing Conditional Introduction

The rule for conditional introduction is as follows:

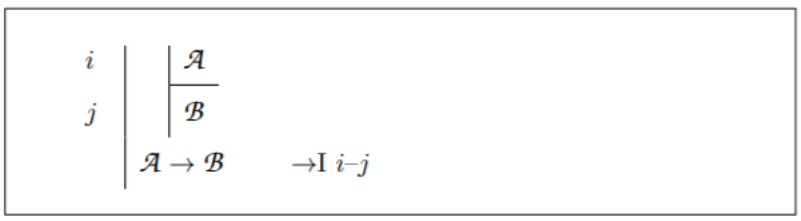


Figure 12: Conditional Introduction. Diagram copied from (P.D. Magnus and Tim Button, 2018)

That is, if we assume some proposition, and derive some proposition based on that assumption, then we may infer that the first proposition implies the second at an assumption depth one shallower than that of the closed assumption.

To begin to convert this into code, a procedure is specified:

i. Get the antecedent and consequent
ii. Check that the proof contains the antecedent and consequent
iii. Check if the antecedent and consequent are the opening and closing lines of an assumption.

The first approach taken to solving this problem was to use the assumptionDepth field alone. The algorithm became:

i. Get the antecedent and consequent from the input proposition.
ii. Check that the proof contains both antecedent and consequent.
iii. Search for the antecedent in the Proof. If it is found (otherwise, return):
    i. Check that the line before it in the proof has an assumption depth one less than that of the line containing the antecedent (otherwise, return).
    ii. Scan through the proof from that line until a line containing the consequent is found, checking the assumption depths of each line on the way. If they change, then return.
    iii. Once the consequent is found, check that the next line of the proof has an assumption depth lower than that of the consequent (otherwise, return).
    iv. Return the line numbers of the antecedent and consequent.

However, while the general approach to converting between formal logic and algorithm worked well, this algorithm suffered myriad issues. For example, take the below proof, output by the final system:

1	|	A	Assumption
2	|	(A ∨ B)	∨I 1
3	|	C	Assumption
4	|	(C ∨ B)	∨I 3

Figure 13: Countermodel to First Algorithmic Attempt to Represent Conditional Introduction

By this algorithm, which assumes that if consecutive lines have the same assumption depth then they are part of the same assumption, conditional introduction will allow the user to infer (A IFTHEN (C OR B)). Furthermore, since this would be proven with no premises, it would be classed as what is known as a 'theorem' of the logic – a tautological truth. The result is a massive disconnect between the system and the logic that it represents. This is clearly a disaster.

Problems like this were frequent at the start of the project as the system took shape and became rarer as the system became more intricate and precise. To handle this particular issue, the Assumption class was introduced. This stores a Boolean for determining whether an Assumption object is 'closed,' and one or two ProofLines corresponding to the opening and closing line of the assumption (depending on whether or not the proof has been closed). For efficient search, each Assumption object is stored in a HashMap where the key is simply the key of the ProofLine opening the assumption.

The readInput method calls the relevant methods to help with the admin of creating and updating Assumption objects when the '#ASSUME' and '#ENDASSUME' keywords are used.

With this guarantee in place, one might try to update the conditional algorithm:

i. Get the keys of the antecedent and consequent propositions.
ii. Check if the keys are present in the proof (otherwise, return).
iii. Check if the HashMap of assumptions contains the key of the antecedent (otherwise, return). If so:
    a. Check if the assumption is closed (otherwise, return). If so:
        i. Check if the closing line's key matches the key of the consequent (otherwise, return).

However, this highlights another issue. Storing each line of the Proof in a HashMap based on its key does not allow for the same proposition to be included in the proof multiple times. They would have the same key, and by the Java implementation of HashMap, the ProofLine stored first would be lost. Take the following example:

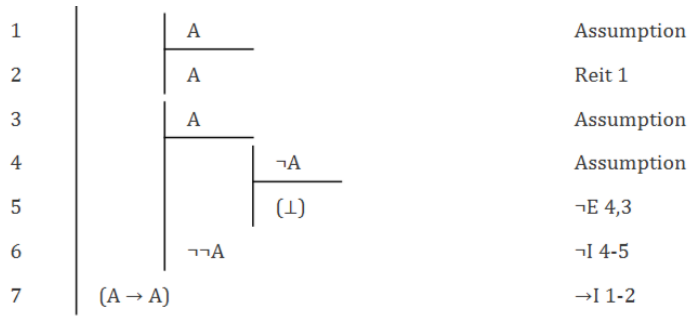


Figure 14: Proof Demonstrating the Importance of Indexing Duplicate Keys

If we were attempting to derive (A IFTHEN A) – line 7 of the proof – then the above algorithm would search for A, find the most recent instance of A included in the Proof, see that the assumption is not closed by the consequent of the conditional, and return unsuccessfully. In spite of the fact that lines 1-2 suffice to derive the sought Proposition.

To get around this problem, the method addLineHelper, which takes care of the admin of adding a new line to the proof, calls a method lineKeyHelper. This method appends a number to a key if it is already contained in the Proof. The first time A is stored, its key will be (A). The second time, its key will be (A)1. The third time, (A)2, etc.

With the added capability of being able to handle repeated lines comes further complexity. The algorithm must be updated again:

i. Get key of the antecedent and consequent.
ii. Check if those keys 'bookend' an assumption (i.e. the antecedent opens the assumption and the consequent closes it) by:
    a. Checking that both keys are contained in the proof (otherwise, return)
    b. Initialise an index counter to 1. Append 1 to both keys. Search for each new key in the proof.
        i. If the indexed key is found for the antecedent, add it to the list of prospective antecedent keys.
        ii. If the indexed key is found for the consequent, add it to the list of prospective consequent keys.

iii. Increment the index counter.
iv. Repeat steps i-iv until neither indexed key is found any longer.

c. Iterate through the prospective antecedent keys. For each, check if an assumption is stored using that key in the assumptions HashTable and that said assumption is closed if it exists (otherwise, return). If so:
    i. Iterate through the prospective consequent keys. If the endLine of the assumption has a key matching the a consequent key, then the pair of keys successfully 'bookend' the assumption.
    ii. Return the line numbers of the antecedent and consequent.

This algorithm is exactly what was implemented. The methodology was a case of moving incrementally towards code with each revision of the algorithm. This process got easier with each algorithm decided upon. For example, multiple derivation rules utilise the helper method called to check if two lines bookend an assumption. Widening the set of necessary data structures to be included in the Proof with the earlier algorithms lead to having more options to work with when designing the later algorithms.

This example provides an illustrative insight into how all of the algorithms for designing the inference rules were derived: trying to design the algorithm in the simplest possible way, thinking through the flaws, adding extra functionality to resolve those issues, and redesigning the algorithm.

5.4.3.2. Implementing Disjunction Elimination

The next rule implementation to be discussed is also a rule of Propositional Logic. The disjunction elimination rule is presented in forallx:Cambridge as follows:

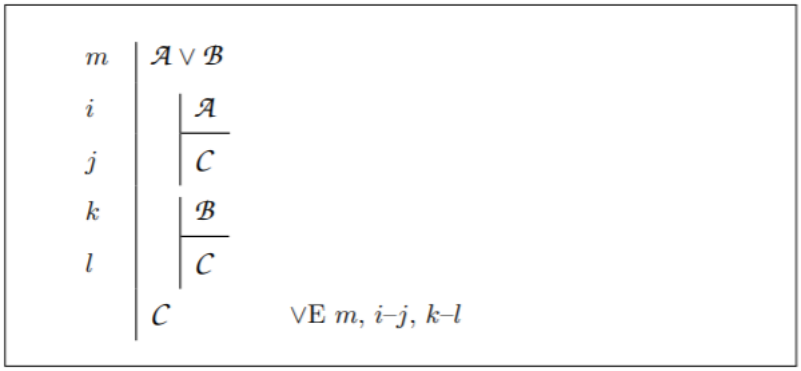


Figure 15: Disjunction Elimination. Diagram copied from (P.D. Magnus and Tim Button, 2018)

In simple terms, this means that if both disjuncts of a disjunction entail the same thing, then because by the nature of a disjunction at least one of those two disjuncts must be true; anything that is entailed by both of those disjuncts must also be true.

The introduction rules were designed before the elimination rules, as they were generally simpler. This meant that when it came time to design this elimination rule, the process was much easier than in the previous subsection because the relevant data structures were already in place. The algorithm was then just:

i. Get the keys of both disjuncts of the disjunction.
ii. Get the key of the input line.
iii. Check if the left disjunct and the input line keys bookend an assumption (otherwise, return).
    a. If so: store the line numbers of the opening and closing lines of the assumption.
iv. Check if the right disjunct and the input line keys bookend an assumption (otherwise, return).

a. If so: store the line numbers of the opening and closing lines of the assumption.

v. If both pairs of line numbers have successfully been found, construct and return the String representing the rule.

Lines (iiia) and (iva) can easily be implemented using arrays. Lines (iii) and (iv) can easily be implemented using the bookendAssumption method designed for conditional introduction. And (v) can easily be implemented by typing up the rule and inserting the stored line references. A relatively complex function can therefore be implemented very simply.

This illustrates an important point: for the most part, designing the algorithms got easier with the more algorithms that were designed. Both because of experience with that way of thinking, and also because of the larger number of data structures and functions available to work with.

5.4.3.3. Implementing Universal Introduction

Representing the inference rules for predicate logic was much more complex than for propositional logic. This was due to multiple factors. First, the rules require the availability of extra data structures and functions which were not needed for any of the rules of propositional logic. Second was that designing those functions was often very technical. And third was that the rules themselves are just more complex, containing more constraints.

The introduction rule for the universal quantifier is defined in forAllX as follows:

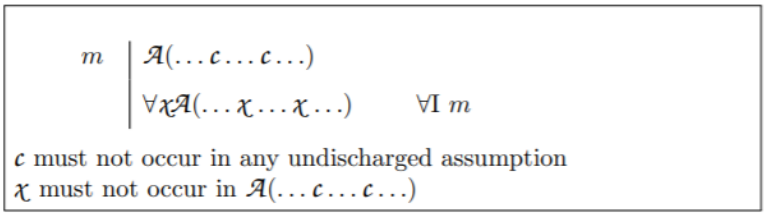


Figure 16: Universal Introduction. Diagram copied from (P.D. Magnus and Tim Button, 2018)

This problem was tackled using the same iterative approach as in 5.4.3.1. To begin with, broad steps were defined:

i. Get the proposition ranged over by the universal quantifier.
ii. Replace all instances of the bound variable with each constant in the proof which is not stored in an undischarged assumption.
iii. Search for the key of the resultant Proposition in the lineKeys HashMap.

Step (i) is easily done using the getProp() method of the QuantProp class.

Step 2 highlights some key points: first is that all the constants in the proof needed to be recorded. The only alternative to creating and storing a record of them was to search through every line of the proof every time a constant was sought out, which is much less efficient. So, a function needed to be added which would scan any line for constants before it could be added to the proof. Those constants were stored in a HashSet<Term>. A HashSet was used to avoid duplicates terms and so make some slight efficiency gains when iterating through the Set.

Second, a method needed to be implemented which would determine whether a given constant was contained in an undischarged assumption. This was implemented using an ArrayList<HashSet<Term>> which contained a HashSet<Term> containing all the constants at a given assumptionDepth. When a new assumption was opened, its constants were added to a new HashSet, which was added to the back

ArrayList. This was easily implemented as the lines were being scanned for constants anyway. When an assumption was closed, the last HashSet was removed from the ArrayList.

Because of step (ii), we have a list of all the constants in the proof, and we know which of those are contained in undischarged assumptions. Step (iii) requires us to substitute all instances of that variable in the proposition with each of the constants in the proof that do not appear in an undischarged assumption, and search for the result.

This requirement resulted in the replaceAllTermInstances method, which was critical to the success of the introduction and elimination rules for predicate logic. The method itself is simple enough: work through a copy of the proposition recursively from the top-down until an AtomicFormula is reached, and if that AtomicFormula contains the variable to be replaced, replace it with the constant. Once this has been done for all occurrences of the variable, search for the resultant key in lineKeys.

The problem that arose from this was that it would only work for the first Proposition substituted. Even though a copy of the Proposition was being made, substituting the copy resulted in changes to the original. So, the original Proposition was lost, and the substitution would fail on the next iteration.

After some research, it turned out that this is due to how Java passes references. Though Java does pass object references by value, the value passed for object references is a pointer to the object in memory (Nero, 2020). So, even though the substitution was purportedly being performed on a copy, it was being performed on the original object.

Java does provide a 'Cloneable' interface which allows the user to create deep copies of objects, but it large amounts of code would need to be refactored to implement it (Gurgul, 2018) and the cloning technique itself is generally viewed as being quite flawed (Ruzicka, 2017). However, taking inspiration from the idea of a deep copy, an abstract method deepClone was built in Proposition.

The deepClone implementation developed for this project relies on the immutability of String objects in Java. That is, once a String is created, its reference in memory cannot be changed. Though the Proposition classes add many useful functions, at their basic building block – AtomicProp or AtomicForm objects – the content of the Proposition is recorded as a String. Therefore, the deepClone implementation for a Proposition subclass works by recursively calling the deepClone method of lower-down Proposition objects until the basic unit is reached, where a new atom is built around a new String.

This deepClone solution worked and provided the base for introduction and elimination in predicate logic.

With this established, the algorithm for universal introduction could be made more precise:

i. Get the Proposition ranged over by the universal quantifier.
ii. Get the variable bound by that quantifier.
iii. Begin iterating through the set of constants in the proof.
    i. Check if the constant occurs in any of the sets of constants contained in undischarged assumption. If it does, continue.
    ii. If it does not, deepClone the proposition.
    iii. Call replaceAllTermInstances on the deepClone to replace all instances of the variable with the current constant.
    iv. Get the new proposition's key.
    v. Use searchForKey helper method to search the proof for the given key, checking if the key is occurring in a closed assumption, and also searching for the key with indexes attached.

vi. If the key is found, return the line number of the key.
vii. Otherwise, repeat (iiia-g) with the next constant in the iteration.

Again, this algorithm is much more actionable.

5.4.3.4. Implementing Identity Elimination

The final example of an inference-checking algorithm is identity elimination. This example was chosen because it was especially tricky to solve.

Identity elimination can be represented as two symmetric rules (diagrams taken from Magnus and Button, 2018):

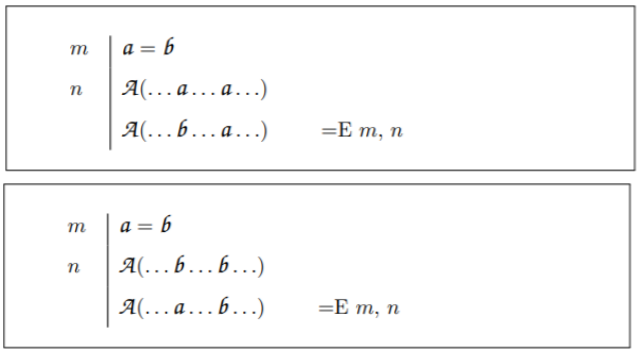


Figure 17: Identity elimination rules. Diagram copied from (Magnus and Button, 2018)

This algorithm was complex and difficult to formulate. So, the problem had to be broken down into steps before it could be transformed into an algorithm. First, as it was not immediately clear how to describe the search criterion of this rule, the target of the search was formulated precisely.

Two criteria of the proposition sought by the identity elimination rule were identified. First, the proposition should be identical in almost every way to the proposition stored in the new line of the proof – the only permitted differences were in their term arrays. Second, the term stored in each proposition at the index where the terms differ must both be operands in the identity statement we seek to eliminate.

This highlighted an important first step for the algorithm: if the new line of proof contains neither operand of the identity statement, then it cannot possibly have been derived by eliminating that identity statement. Thus, the algorithm should first make this check, and return "EMPTY" – the code indicating the elimination rule has not been satisfied – to the calling method if it fails. This allows for the intensive functions of the identity elimination to be skipped.

The next step was to identify if a line of the proof exists which is identical to the new line in every regard except for the elements of its term array. Searching by key would have been pointless here, as the combination of terms stored in the key was unknown and could become intractably complex to calculate with brute force as the number of terms in the proposition increased. Therefore, the only option was to iterate through the proof line-by-line and compare the new line with each line under inspection.

To do this, the algorithm required a function allowing two propositions to have their structures recursively compared. This was the identityEliminationHelper method. It takes for parameters the current proposition in the iteration through the proof, the proposition stored in the new line of the proof, and the identity statement we seek to eliminate. Their type is compared. If they are a complex proposition and have matching types (and matching variables in the case of a quantified proposition), then the method is recursively called on their sub-proposition(s). If they have different types, the method returns false. If they are atomic formulae, then the method inspects them.

If both formulae are absurdity, true can be returned, as there are no terms to speak of. If they are identity statements or predicate-term formulae, the term arrays should be iterated through, and the terms compared. The term at each position in each term array is checked to work out whether it is an operand of the identity statement. Note that they do not need to be different operands to pass this check. If they are not in the identity statement, then it is checked whether they are the same term. If they are not, false is returned, because the difference in terms cannot be explained by the identity statement we seek to eliminate. If the iteration completes without violating one of these conditions, then given that we know our new line of proof contains at least one term from the identity statement, the line being inspected must be the line which has had its constants substituted using identity elimination to derive the new line.

i. Iterate through new line of proof to check if it contains either term contained in the identity statement. If it does not, return "EMPTY."
ii. Iterate through the proof, comparing the proposition stored in each line with the new proposition:
    a. If they have the same main logical operator, compare the main logical operators of their sub propositions. Continue this process until an AtomicForm is reached (or if the Proposition was originally an AtomicForm, skip to next step). If they do not match, continue iterating.
    b. Check if the AtomicForm objects contain the same non-term components. If not, continue iterating.
    c. Iterate through both term arrays, checking if the terms stored at the same position are both stored in the identity statement.
        i. If are not, check if they are the same term. If so, continue iterating through the terms. If not, continue iterating through lines of proof.
        ii. If so, continue iterating through the terms.
        iii. If the end of the term array is reached, return true.
iii. If the lines of the proof are exhausted, return "EMPTY."

5.4.4. Closing Remarks on the Back-End Architecture

The system implements thirty inference rules. The small selection of four presented here hopefully demonstrate critical aspects of how the proof-checking algorithm works, how its subroutines were designed, and some of the key challenges that were faced during that process. As discussed earlier, inputs are converted into ProofLine objects via the Interpreter, and the process of validating the assertion of each line of input and adding it to the Proof is handled by the proof-checker. Implementing these inference rules so that the system can automatically identify the rule used to infer a new line is what demarcates this system from the related systems evaluated in section 3.

Before the strategy used to test the back-end functionality is discussed, the design process for the Graphical User Interface will be presented. The reasons for this are twofold. First, it allows all the design and implementation topics to be covered together. Second is that as the GUI controls the input to a Proof and displays output based on the resultant Proof, and therefore interacting with the GUI was a very important part of the testing strategy.

6. User Interface

Because this project placed such a heavy emphasis on the usability of the system, a proper design process for the user interface was crucial. This was an iterative process. The first prototype of the GUI layout was created based on the review of related work and on the initial project objectives specified in the project proposal. This prototype was evaluated using Nielson's heuristics – the same technique used to analyse the related systems in section 2 – to ensure consistency in the standards used to evaluate usability across

the project. A second prototype was built based on this feedback and re-evaluated to ensure that it satisfied the requirements specification.

The tool used to create this prototype was MockFlow (https://www.mockflow.com/), a tool containing a library of User Interface elements which can be combined to easily iterate through interface designs (Crozdesk Limited, n.d.). This tool was chosen for its simple drag-and-drop layout function, and the ease with which prototypes can be edited.

6.1. First GUI Prototype

The purpose of this prototype was to make a first attempt to convert the project goals into a graphical user interface design, considering also the points highlighted in the review of related work.

At this point in the design phase, the code used to interact with the system was nascent. For example, the permitted shortcuts displayed in the UI include typing "<->" to represent the biconditional, when the final choice was made to only allow the word "IFF" to represent the biconditional.

The user interacts with the system by inputting premises into the premises bar, separated by a comma. They can optionally add a conclusion which, if derived as the last line of the proof, will result in a message displaying. The user can also type the proof into the input area, which has line numbers included in the same way that IDEs often do. Tabbing to "First-Order" will add extra buttons to the display, representing the symbols of first order logic. The prototype can be seen below.

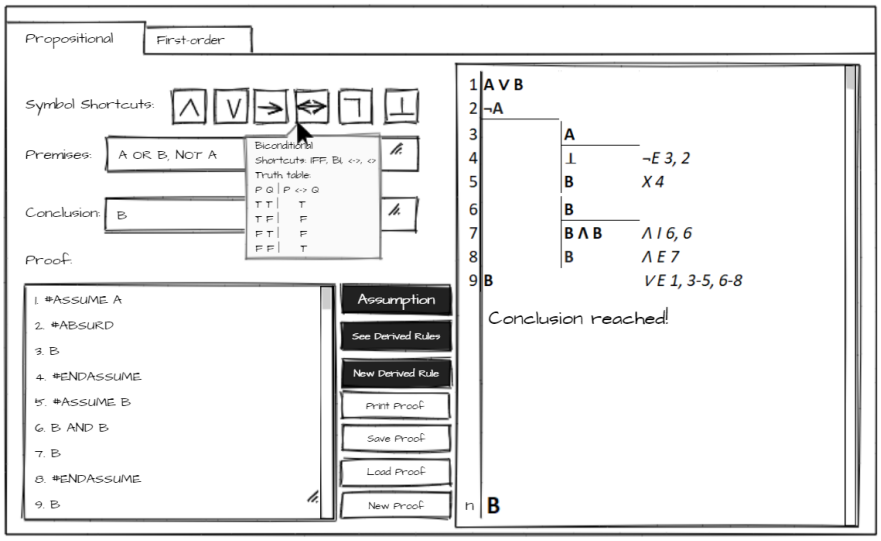


Figure 18: First Prototype of PRODLOG GUI

The prototype was evaluated using the same set of heuristics used to evaluate the related systems (Nielson, 2017). This evaluation has been included in Appendix E. Overall, the prototype was a good first attempt to satisfy the project objectives but would have benefitted from a closer reading of the review of related work.

6.2. Second GUI Prototype

The purpose of this prototype was to use the evaluation of the first-generation prototype to develop a prototype more closely resembling the final user interface. At this stage in the project, the requirements specification was close to finalised, and the back-end architecture was largely designed. It was therefore possible to build something resembling the final system, and which satisfied the GUI display

requirements detailed in the requirements specification. This list was used as a checklist when adding components to the prototype.

The prototype can be seen below. Note that the premise and conclusion bar were removed to simplify the interface, and to simplify saving and loading to text files. Buttons have been added allowing the user to input each keyword. Furthermore, notifications are displayed underneath the screen in the input area. Again, assume that tabbing to first-order logic adds symbols of first-order logic to the screen. The prototype was evaluated using the same set of heuristics used to evaluate the related systems. This evaluation can also be found in Appendix E (Nielson, 2017).

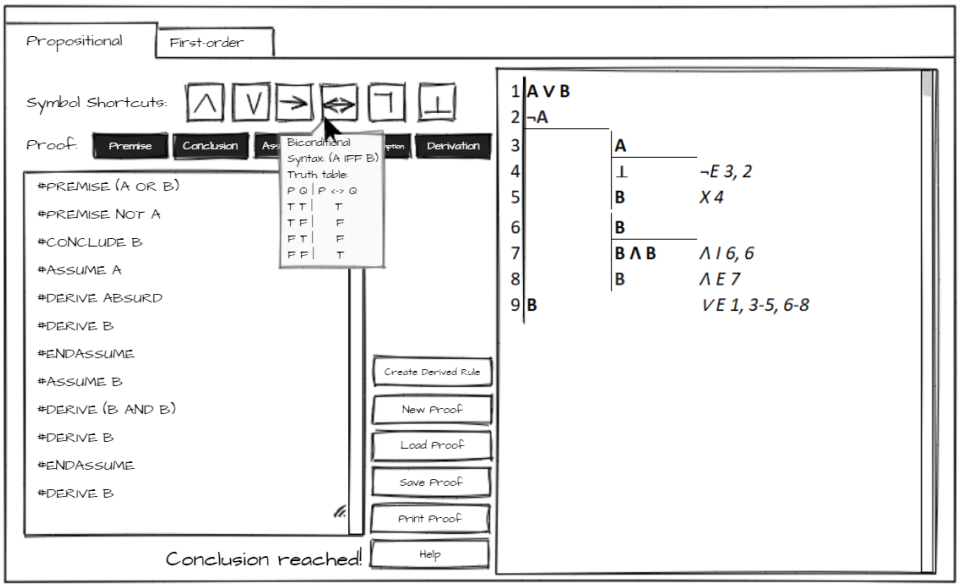


Figure 19: Second Prototype of PRODLOG GUI

Overall, this pared-down interface design builds effectively on the points raised in the previous design, and is a strong move towards a more flexible, clean and efficient interface. An interesting feature was the "Create derived rule" button, which would have allowed the user to save their own derived rule. This functionality was eventually dropped from the system due to time constraints. The "New proof" button was removed to clean up the GUI, and the "Help" button was replaced with more detailed syntax information in the tooltips and a short manual.

In the next subsection, considerations of how to convert this interface into code were taken.

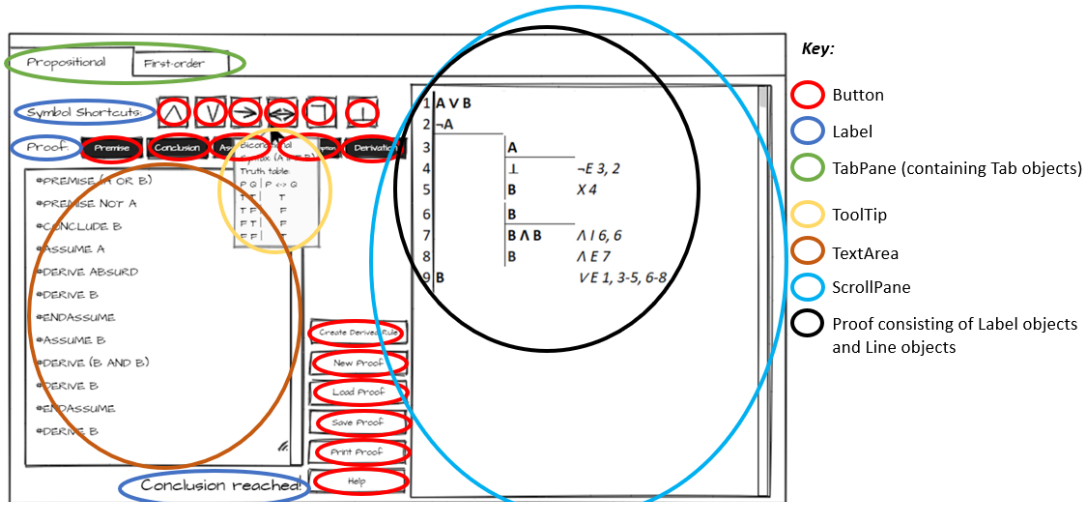


Figure 20: Second GUI Prototype Annotated with JavaFX Components

6.3. Prototype Design & Implementation in JavaFX

The next step in the design process was to research options for representing the elements of the second-generation prototype in JavaFX. JavaFX was chosen for two reasons. The first was due to the author's familiarity with it. The second was that it is also highly portable, and therefore able to run on many different machines. Drawing on the author's knowledge of JavaFX and information from Oracle (Oracle.com,. 2019), an annotation of the second prototype was produced highlighting which JavaFX elements to use (Figure 20, prev. page).

The next choice was deciding how to implement the GUI. As discussed in the earlier section of software architecture, one common design choice taken when building user interfaces is separate out the model, view and controller into different classes (CodeProject:rj45, 2008). Instead of implementing the main Proof class as a JavaFX file also containing everything needed to run the user interface, a view class was created with the single purpose of displaying the elements of the user interface.

This division was planned at a very early stage, allowing for the development of the back-end system – or 'model' – completely independently of the user interface. Another benefit of this approach, aside from reducing the complexity of the system via increasing its modularity, was that should the implementation of the base functionality of the system have proven too time-consuming or complex, the command-line prototypes constructed to test the back-end in its early stages would still function properly without any interface at all. This would have allowed the minimum viable product to be delivered in that scenario.

To interface between the view and the model, the GUIController class was implemented. The purpose of this class was to supply all event handlers for the view, to pass input to the back-end, and to handle the output from the back-end (i.e. to either display an error message or draw a proof).

Instead of hard-coding the GUI, the Java SceneBuilder tool allows the user to use drag-and-drop to construct their user interface (Oracle.com, 2020). The output is an FXML file representing the GUI view, to which event handlers can be added in a controller class to connect the interface with the model.

The author already had some experience with Java SceneBuilder from building a user interface prototype for a separate project and decided not to use it here based on that experience. There were two main reasons for this. The first is that the author found that being sufficiently precise about the hierarchy of elements when using SceneBuilder required that hierarchy to be worked out in detail in advance. Otherwise, elements could easily end up in the wrong family of nodes. The second reason was that since this hierarchy of elements needed to be worked out anyway, it made more sense to implement them by hand and tailor that implementation to the controller and back-end functionality. For this same reason, Java was used rather than FXML.

After some refinements, the final Scene graph of elements was as follows:

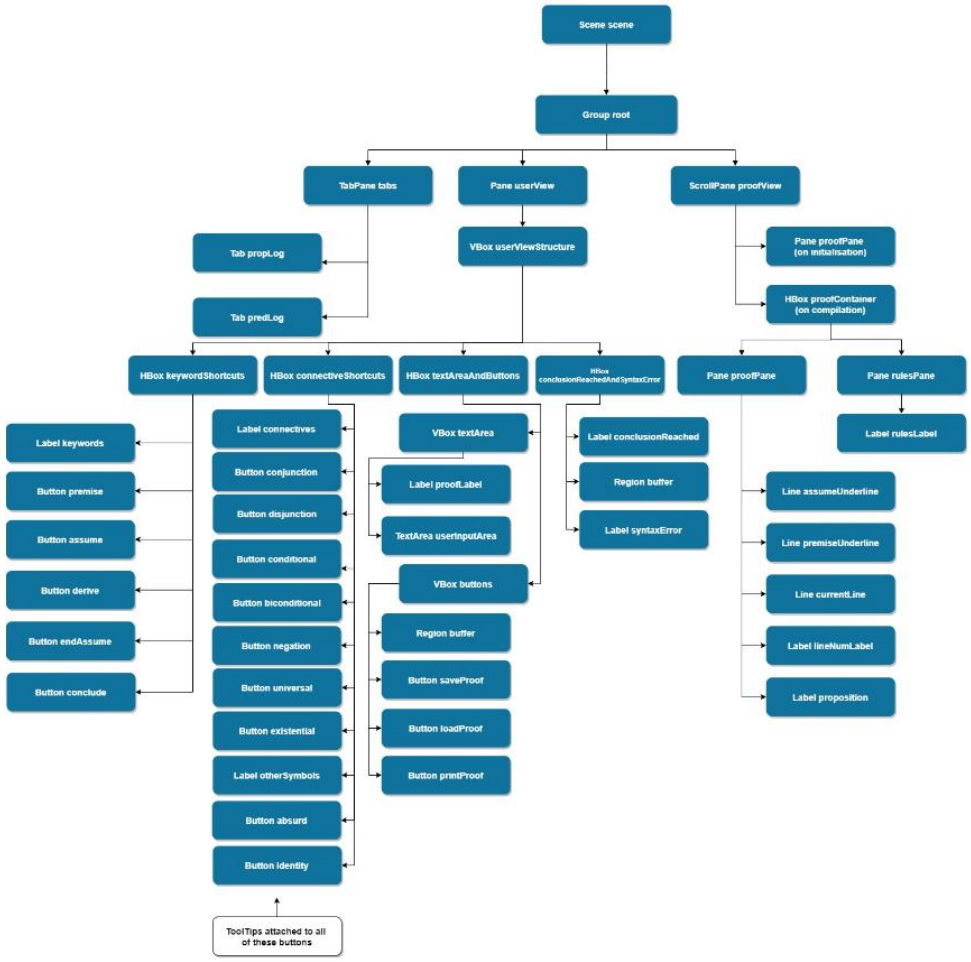


Figure 21: Scene Graph of JavaFX Elements

Note that many layout panes are included. The reasoning behind this was to minimise the error involved in the programmer attempting to manually position each element.

Within the code, modularity is kept by having separate methods generating each area of the screen. Typically, a different method is called to generate each pane. For example, createUserView creates the userView pane and calls createUserViewStructure to generate the relevant HBox, which calls four more methods to generate the four HBoxes below it in the Scene graph. Breaking each method down like this meant that the code was a lot more manageable.

Once the view was implemented (and some neutral colours added), this was the resultant interface:

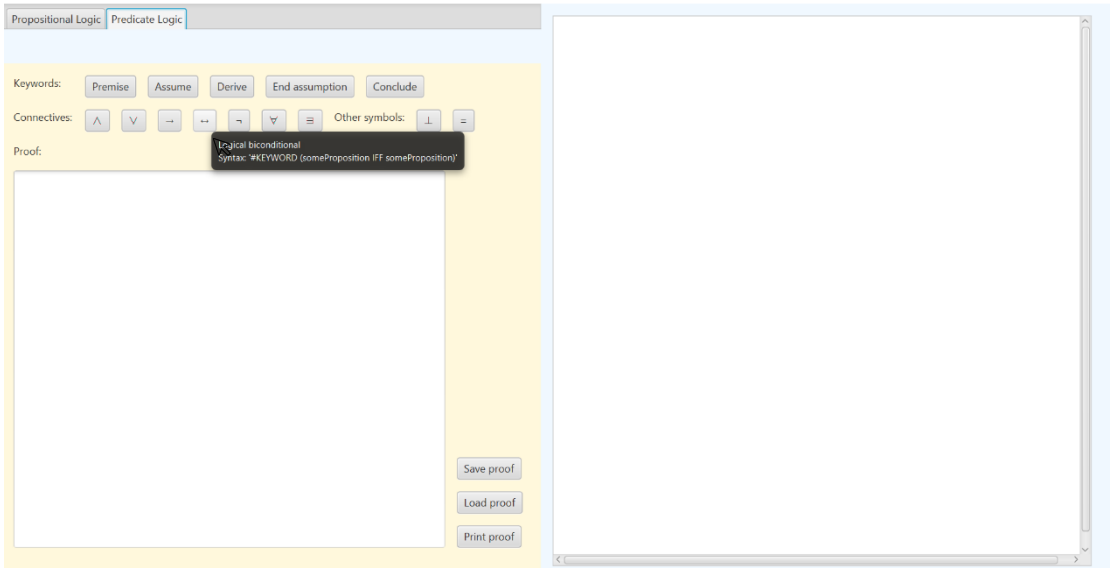


Figure 22: Final GUI Interface

The next step was to build the controller to connect this functionally useless GUI to the model.

6.4. The Controller & Key Algorithms

As mentioned earlier, the controller connects the user's interaction with the view to the back-end model (CodeProject:rj45, 2008). The class GUIController contains methods returning EventHandlers which could be attached to each of the interactive components in the user interface. The implementation of most of these can be seen in the source code, highlighted by comments. However, two key functions of the GUIController and GUIView are worth highlighting here.

The first is the static method compileLines(), which returns an EventHandler. This EventHandler is attached to the TextArea into which the user inputs their proof and handles the event of the enter key being pressed. When that happens, a new Proof object is created. The input String is copied from the input area, and split into an array of Strings on a newline character. The resultant array is iterated through via a for loop. This for loop is encased in a try-catch block.

For each line of the array, the Proof object's readInput method is called. If an exception is thrown – i.e. if there is a problem with the input syntax, or if an invalid derivation is found – then the error message is caught in the catch block, assigned as the contents of the error Label underneath the text area, and the Label is set to visible. If no error is found, the Proof is passed to the drawProof method of the GUIView class, and the result is assigned as the contents of the proofView HBox. This is the mechanism via which the controller interacts with the back-end.

The second is the drawProof method. This was ultimately included in the GUIView class because its sole purpose is to display elements on the screen. It takes as input a Proof object and returns a HBox containing a Pane proofPane and a Pane rulesPane. This HBox allowed the rules to always display in line with one another, and to never overlap with the proof itself, all without having to manually calculate the distances between the body of the proof and the rules. To fill the proofPane with line numbers, lines and propositions, the algorithm iterates through the ProofLine objects of the input Proof.

Whilst deciding how to draw each feature of the proof was a challenge in its own right, the most significant challenge was managing the drawing of the lines for open and closed assumptions.

To do this, an ArrayList of Line objects was used to store all of the vertical lines in the proof. Some of those needed to have their end y-coordinate updated. For example, the first line needed to be updated with every compilation so that it reaches from the top of the proof to the bottom, and the line delineating a particular assumption would need to be extended with each subsequent line before that assumption was closed. However, not all of them did; once an assumption is closed, the line identifying that assumption could be ignored.

To manage this, a stack containing the indexes of the lines of the ArrayList which needed to be updated was maintained. With each compilation, the stack would be iterated through in descending order, incrementing the y-coordinate of the endpoint of the line stored at the index of the array returned by the stack iterator. This stack was implemented using the Deque class, because there is no guarantee that the Stack.iterator() in Java will iterate in order (GeeksforGeeks, 2018). The Deque is a stack-queue hybrid with an iterator() method which can be set to work in either ascending or descending order (GeeksforGeeks2018a).

To decide whether or not to push or pop from the Deque with the adding of a new line to the proof, the assumption depth of the new line was compared against the assumption depth stored in the previous ProofLine object. There were four possible cases to consider. If the keyword of the new line was not "#ASSUME" and the assumption depths matched, then the lines stored at all the indexes recorded in the deque would be incremented, because this indicates that there is no change to the assumptions. If the keyword is not "#ASSUME" and the assumption depths do not match, then the deque was popped from the top so that only the current assumption depth's worth of indexes were left in the stack. In this situation, at least one assumption has been closed, so one or more lines no longer need to be kept track of. If the keyword does equal "#ASSUME" and the new assumption depth is greater than the previous assumption depth, then all lines have their y-value incremented, and a new line is added to the list and deque, starting at that assumption depth. If the keyword does equal "#ASSUME" and the assumption depth is less than or equal to the previous assumption depth, then we pop from the stack so that there are (current assumption depth – 1) indexes left on the stack, and this new line is added to the list and stack. This is because at least one assumption has been closed before opening this new one, and those lines should no longer be tracked.

The result can be seen below:

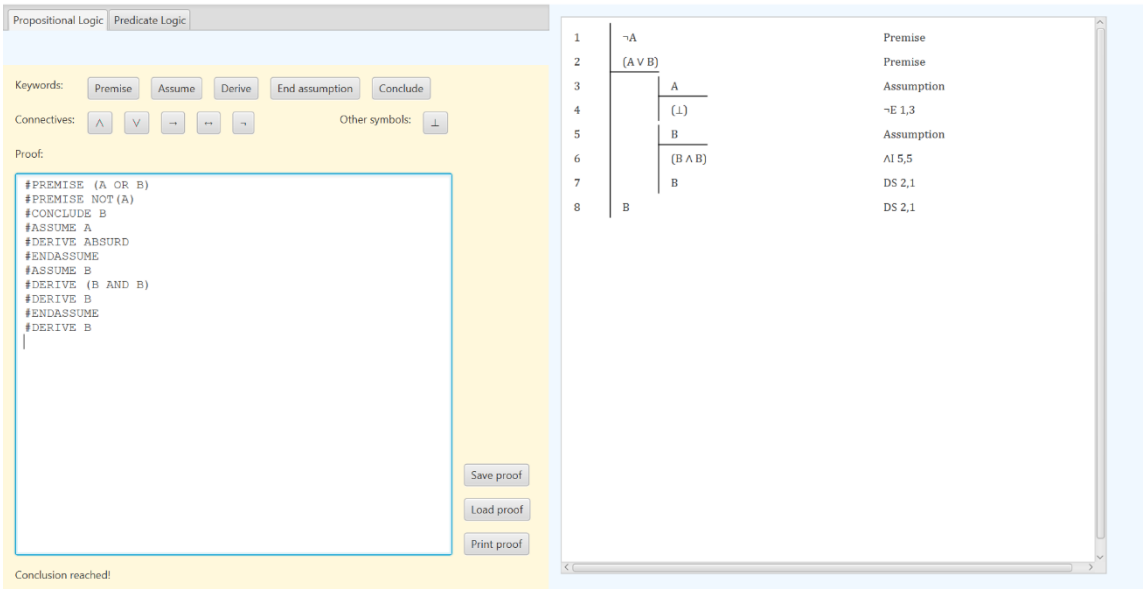


Figure 23: Final GUI Prototype Displaying a Proof

Note that this proof is the same one used in Figure 19. The difference in the rules listed to derive each line occur because of the back-end implementation. The proof is of the disjunctive syllogism rule. This rule states that if we have a disjunction statement and the negation of one disjunct, then the other disjunct must be true. As this rule is already implemented within the system, the system recognises this rule and uses it as the reference to derive lines 7 and 8.

7. Testing Strategy

With all relevant areas of design and implementation covered for the back-end and GUI, the next stage in the software development process was testing. A mixture of formal and informal testing methods were applied for the purposes of this project. Informal testing consisted of printing the values stored at different points in different branches of the program to the command line, visually inspecting that the output is expected, and moving on. Formal testing consists of a mixture of recorded black box and white box testing methods. Owing to time constraints, how expansively each module was tested ended up being a calculated choice. More important modules and functions received more attention.

The tests were written with the requirements specification in mind. Each test recorded contains a summary of the feature being tested. For the JUnit test cases, this is necessarily tailored more to the specifics of the implementation than the requirements document. This is because the data structures need to work reliably for the rest of the system to function. Furthermore, the test cases for the interactive proof-checking system relate specifically to the requirements specification. Tests were generated by hand, owing to the complexity of learning to use automated testing being deemed too great in the short timeframe of the project (Smartbear.com, 2019). The comprehensive set of over 340 formal test cases is included as Appendix F this report.

A mixture of white-box and black-box testing was used. The former makes use of the detailed internal structure of the tested class, while the latter focuses more in the inputs and outputs (CodeFirst, 2020). In the next subsections, each testing approach is presented and discussed.

7.1. White-Box Testing of Proposition, AtomicProp and AtomicForm

A white box test plan was designed for the most important data structures: the Proposition class, AtomicProp, and AtomicForm class. Proposition was chosen because it is the superclass of most of the data structures in the system, so any failures will be inherited by its subclasses. AtomicProp and AtomicForm were chosen because they are the basic units of Proposition data structures. Other Proposition classes may store terms and types, but aside from that they mostly just store different hierarchies of AtomicProp objects or AtomicForm objects. It was important, therefore, to ensure that these cornerstones of the system were implemented correctly.

The white box testing plan aimed for 100% path coverage in each method, and involved implementing 135 test cases, organised by method tested and covering all methods. The tests were written by studying the requirements specification and source code of each class to ensure that all necessary conditions were satisfied. However, even though the classes were studied closely, it is still possible that some elements were missed due to human error.

One aspect of the design of this system is that it is the interpreter's responsibility to ensure that only 'good' inputs are passed to each method in the creation of the system's Proposition objects. The reason for this was that implementing checks in each method of each class would have required adding significantly more code to each class. Making this the job of the interpreter requires significantly less

code, and the final product should be easier to test and debug as a result. A side effect is that while some bad inputs are screened for in these classes, undesirable inputs generally pass the JUnit tests.

However, the objects were able to correctly carry out their key functions on both 'good' and 'bad' inputs, so the impact of this is limited if the interpreter's screening mechanisms are well tested.

In many cases, the tests revealed interesting bugs in the system's implementation. For example, the generateKey() method of the AtomicProp class was outputting a String representation of the atomic proposition with no brackets. This would have meant that the system could not have distinguished between an AtomicProp storing proposition "A1", and an AtomicProp storing Proposition "A" which has the number 1 appended to its key because "A" is already stored somewhere in the proof. This was clearly a mistake, so the code was refactored to add more brackets into the output of generateKey.

7.2. Black-Box Testing of Other Back-End Classes

Thirty-four test cases were written, mostly for the purposes of testing the constructors of the object classes which would not be directly tested in the other two test plans. The purpose of these tests was simply to establish that if passed the correct inputs, the classes could be relied on to produce the correct outputs. This is of course indirectly tested in the other test plans, but knowing that these particular functions work correctly significantly reduces the chances that errors in the later black-box testing of the whole proof system would be caused by errors in how these classes were handling information.

7.3. Black-Box Testing of the GUI Components

The easiest way to test the GUIView and GUIController classes was to take a 'manual and model approach. 'Manual' because each component in the requirements specification is tested, and 'model' because the test does not end at checking that the element is there: all interactive elements are interacted with (Ghildiyal and Chandra, 2019).

Developing these test cases was a simple process, as the specification for what every GUI component should do is already provided in detail in the requirements specification. The hardest part was making sure nothing was missed from the list of cases.

Nevertheless, these tests yielded some interesting results. Because the author is well-versed in PRODLOG, most testing prior to that point had been done by typing the language into the input field. This meant that the conditional button being assigned the incorrect event handler went unnoticed until this testing.

7.4. Black-Box Testing of the Interpreter and Proof-Checker

At this stage, it has been established that the most important data structures behave exactly as expected down to the specifics of their methods. Furthermore, there is a general sense that the other, more peripheral data structures also behave as expected. The GUI is also behaving exactly as it should. All of this goes to build up confidence that when testing the interpreter and proof-checker via the GUI, any problems which arise will be cause by the interpreter and proof-checker, as opposed to some other part of the program. This cannot be guaranteed but can be asserted with a reasonable amount of confidence.

The original strategy for testing the interpreter and proof checker was to produce a boundary value analysis (BVA). BVA is based on the idea that extreme values tend to be the cause of errors, and so values on the edge of being 'bad' inputs are tested as well as a set of 'normal' test cases (Sharma, 2019). For example, one might close an assumption, open a new assumption at the same level immediately after, and try to reiterate something from the previous closed assumption. If the check for a closed assumption

ran solely on changes in assumption depth (which it did in earlier implementations), this reiteration would be incorrectly classed as valid.

Unfortunately, it was not possible to implement a full BVA. This was mostly due to time constraints; implementing other parts of the system took longer than expected. Nevertheless, the principle of BVA has been carried forwards into the testing strategy for the interpreter and proof checker.

These test cases work by inputting lines of PRODLOG into the GUI and observing the output (either a change to the proof, a message, or some combination). All 30 inference rules are individually tested for correct operation, with a mixture of boundary cases mixed in: bad inputs which closely resemble good inputs, and bad inferences which closely resemble good inferences.

7.5. Discussion of Testing Strategy

In the end, all test cases passed bar one – that printing an empty proof to PDF should output a blank PDF. In the implementation of the drawProof(Proof proof) method, the leftmost line of the proof is given an initial set of coordinates regardless of whether the proof contains lines. This means that it displays in the PDF of an empty proof as an almost-imperceptible black dot. This would have been fixed but was noticed at a very late staged.

The testing strategy itself was an appropriate way to manage the testing of the system. By the recursive nature of generating sentences of formal logic, proofs can become indefinitely long and complex. The testing approach guarantees that we can strongly rely on our data structures and our interface. Furthermore, all inference rules work well on small inputs. This should all go to build up some confidence that the system will continue to function well with larger inputs.

However, a fair criticism can be made here. As the proofs get longer, they store more information. Storing more information means more information to search through, perform functions on, and make mistakes with. Until the system is tested on longer and more complex proofs, it cannot be shown that bugs in implementation which do not show up in small-scale tests will not appear.

Furthermore, whilst all the classes are extensively tested indirectly through the provided test cases, several are not subjected to rigorous tests of their own. It is possible that some flaws in the implementation of these test classes might have slipped through, and only appear in longer proofs.

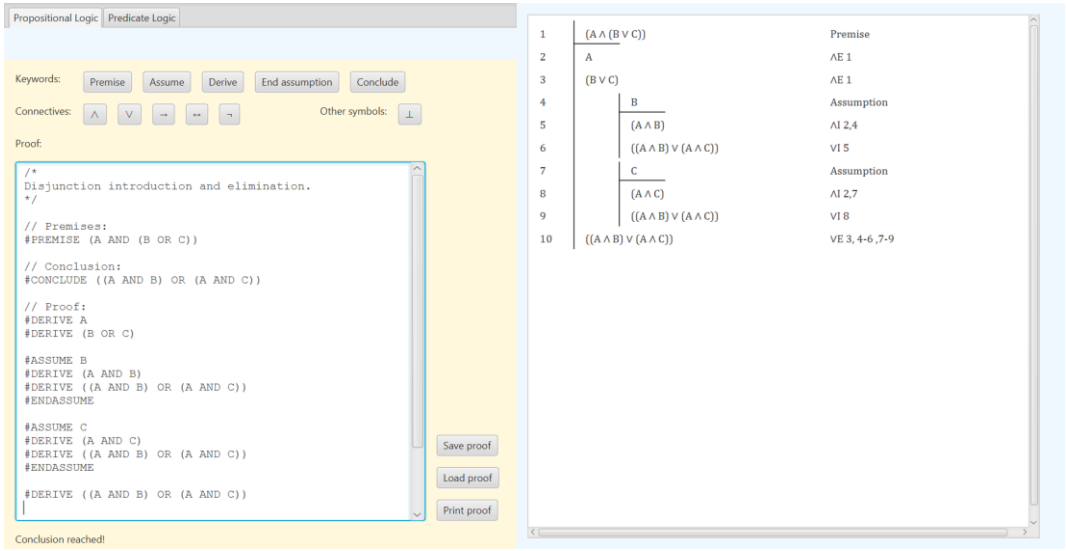
With more time, longer and more complex proofs would have been tested, the aforementioned classes would have been tested in more depth, and the dot-displaying flaw in the drawProof method would be rectified. Making these changes and still passing the tests would allow us to be extremely confident in the system's reliability.

8. Results & Evaluation

8.1. Overview of Main Achievements

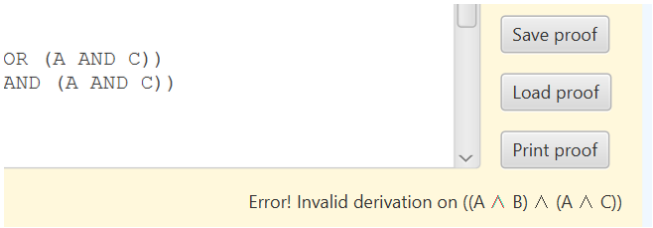
The final product provides an input language through which the user can interact with the system. It provides a full GUI for the user to input into the system and displays output from the system in the form of a Fitch-style proof and/or error message. It implements an interpreter to transform the user's input into the relevant data structures. And, most significantly, it implements thirty rules of natural deduction to identify the rule by which the user derived that input. To that extent, it fulfils all the essential and conditional requirements specified in the original project proposal.

The system can be used to build proofs in propositional logic:

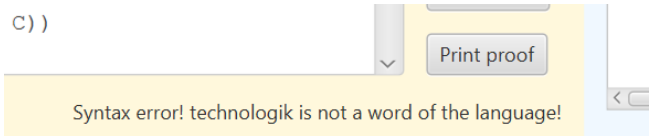


Note that the system notifies the user that the conclusion has been reached.

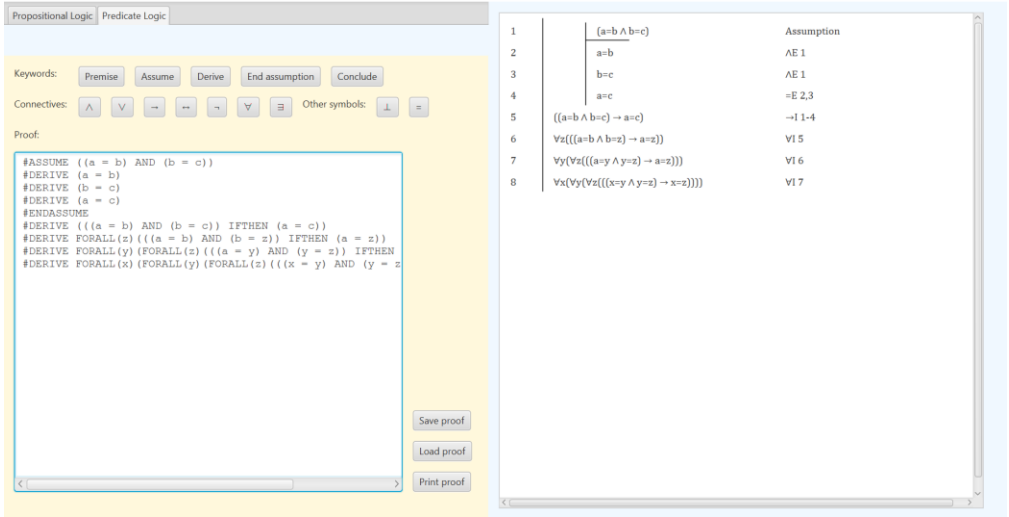
If we try to make an invalid derivation, the system will catch it:



As it will catch inputs which do not class as sentences of the input language:



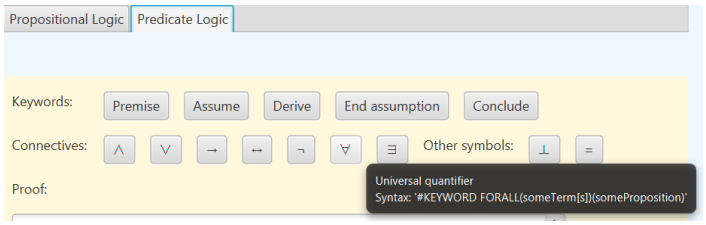
The system also allows the user to build proofs of predicate logic:



The above proof is a nontrivial result, demonstrating that the transitive property of identity is a theorem of first-order logic.

8.2. Usability

The core tension of the design process was balancing usability for novice users against allowing efficient use for experienced users. The comprise that was struck was to allow the user complete control over their inputs via a text area, whilst also providing guidance on the input via a suite of buttons on the GUI which would input into the text area and display syntax help in the form of tooltips. This helps to satisfy one of the core objectives of being of use to both novice and experienced practitioners of formal logic.



With more time, this would have been taken a step further. Several extra usability features came to mind over the course of the project. For example, letting the user set their own keyboard shortcuts for the logical connectives. Also, allowing the user to select the convention for the introduction and elimination rules that they want to work with. The set used for this project are a subset of the rules that are out there. More coverage would mean a system which is more useful to more people.

Finally, one point which frustrated me as a user was not being able to open more than one proof at once. The big blue-grey strip above the buttons would be a perfect place to install another TabPane where multiple open proofs can be tabbed between. This could remove the frustration which comes with continually having to save and load proofs from and to the file chooser.

8.3. Reliability

When considering the reliability of the system, it is useful to do so from the perspective of the user classes defined earlier in this paper. Recall that we can delineate users as novice and experienced users of both formal logic and this system.

A novice student of logic is likely to find the system to be extremely reliable. It has been tested extensively over its lifespan on thousands of inputs, many of which were literally solving problems from the practice sections of logic textbooks. The system is likely to work very reliably for the user who is simply using the system to practice formal logic and solve those kinds of simple puzzle.

But they are not the only user class. When this system was conceived of, its primary purpose was as a pedagogical tool. But the intention was always to provide the capability to handle more intensive usage. Real work is being done in first order logic, and the system offers a quick and convenient way to do that work efficiently. It is where this intensive work begins that confidence in the reliability of the system begins to wane.

There are two perspectives one might take on this. The first is to remain optimistic and say that we can be confident that the system has a strong foundation, based on the wide base of testing that was done. With this strong foundation alone, we cannot be certain that it will perform well when the input sizes get large, but we have good reason to think so.

However, there is good reason to be more cautious than this. As the input proofs gets larger, more information is stored. The record of constants in undischarged assumptions grows larger; more propositions are reiterated and stored again at a key with an index attached; more assumptions need to be recorded and managed. There are plenty of opportunities for things to go wrong, and the ways in

which they go wrong may only come to light after the same inference rule has been applied five or six times – which might require a hundred lines of proof if other inferences are being made as well.

The proof-checking system is inherently conservative. It is much more likely to fail by failing to admit a valid derivation than by admitting an invalid derivation. This is because the system works by testing inference rules against the prior contents of the proof. This information is so specific that it is unlikely to accidentally let anything pass. However, while failing to admit a valid derivation sounds better than its inverse, this is not really the case. If there are inferences made using the inference rules implemented within the system and these are flagged as invalid, then something is wrong with the implementation. There is a strong possibility that this could happen with proofs over a certain size. More work needs to be done to ensure that the system is ready to handle large inputs.

8.4. Comparison to specification

The system aligns very closely to the requirements specification, not really deviating significantly. This is mostly due to the very specific nature of the system. Most of the work was in designing the data structures and the proof-checking algorithms. The requirements specification can only describe these in so much detail before going beyond its scope and discussing the elimination. But this very specific system would struggle to function properly without any of the things that it can describe.

One area where the system did not meet its specification was in the optional requirements. As will be discussed in the next subsection, several external constraints put a lot of pressure on the project period, and so achieving all of the conditional objectives was a struggle. However, many of these pertained to interactive theorem proving, which operates in a different field to the scope of this project. In the future, an excellent experiment would be to combine techniques from that field with this system.

8.5. Other Evaluation

Evaluation has been carried out continuously across the course of this project. Most sections end in an evaluation section discussing key areas where things work well and do not work well. In this section, a few final points will be presented which did not really fit in elsewhere.

First of all is the interpreter. There are several aspects of the interpreter's implementation which could increase the system's usability given substantial refactoring. Primary among these is that it is currently implemented as a class with no constructor and full of static methods. The reasoning being that there would be no benefit to storing any outputs other than ProofLine objects after an input is run through the system. However, significant efficiency gains could be made if the interpreter did store this information, and checked new lines of input against input that it has already received. If it has received a line of code before, saving time by not having to recompile it could lead to serious efficiency gains in the long run.

Along a similar vein, the Proof class would also benefit from significant refactoring. I did not anticipate how complex it would become at the start, and so all formal rules are implemented internally. The end result is a class which is several thousand lines of code long. I am convinced that with more time I could find a more appropriate way to do this.

Finally is the presentation of the system. Presenting a collection of class files is not ideal. I would be keen to investigate methods of running my project code on a website in the near future.

8.6. Project Management

Overall, the project was well-managed. Not in the sense of sticking to the original plan, but in the sense of adapting rapidly to change in order to still deliver the product.

A change in approach was required relatively early on. The initial plan of following a solely waterfall model of software development fell through as it became apparent that the design, implementation and testing would be best carried out in a feedback loop.

I began by building a simple command line prototype of how the system should work for PROPLOG. This developed into an interactive command line prototype, and a similar basic prototype was made for PREDLOG. This allowed for the back-end to be tested independently of the GUI, and also provided a means of delivering the minimum viable product for the system.

The project timeframe was accelerated significantly towards the end. I lost a lot of work time due to developments in my personal life, and when this was combined with much more difficulty implementing the PREDLOG system than anticipated, I found that I was struggling to keep up. The formal testing especially had to be carried out in an accelerated time frame, along with the report-writing.

In the future, the main thing that I would do differently would be to embrace the iterative development approach even more. I still tried to keep to a plan-driven approach for this project and I think that sticking rigidly to a plan at the start meant that implementation started slightly later than was ideal.

9. Summary & Conclusions

The aim of this project was to help students of formal logic practice their skills of Fitch-style natural deduction by providing an interactive proof-checking environment which is quick to learn, intuitive to use, and which protects the user from making invalid derivations and easily allows them to edit or remove sections of their proof at any point. Specifically, the system was designed to remedy the problems with hand-written proofs whilst improving on the issues with existent proof-checkers which are discussed in this paper. This system is narrower in scope than some of those proof-checkers, but it is intended to be excellent for its specific purpose. It is also more powerful than other pedagogical proof-checkers, as the system does not just convert user input to proofs, but goes a step further to add new information to that input by inferring the inference rule used to derive that input. All of this was carried out with a focus on usability in mind to create a highly intuitive and useful tool.

10. Bibliography & Source Code References

10.1. Bibliography

Arias, E.J.G. and Itzhaky, S. (2018). *jsCoq – Use Coq in Your Browser*. [online] jscoq.github.io. Available at: https://jscoq.github.io/

Arms, W. (2014). *CS 5150 Software Engineering Scenarios and Use Cases*. [online] Available at: https://www.cs.cornell.edu/courses/cs5150/2014fa/slides/D2-use-cases.pdf

Avigad, J., De Moura, L. and Kong, S. (2020). *Theorem Proving in Lean Release 3.18.4*. [online] Available at: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf

Avigad, J., De Moura, L. and Roesch, J. (2020). *Programming in Lean*. [online] Available at: https://leanprover.github.io/programming_in_lean/programming_in_lean.pdf.

Barthe, G. (n.d.). *Introduction to Dependent Type Theory and Higher-Order Logic*. [online] Available at: https://www.asc.ohio-state.edu/pollard.4/type/readings/barthe-dtt-hol.pdf

bauerandrej (2011). *A first proof with Coq (Frobenius rule)*. *YouTube*. Available at: https://www.youtube.com/watch?v=z861PoZPGqk

Bove, A., Dybjer, P. and Norell, U. (2009). *A Brief Overview of Agda - A Functional Language with Dependent Types*. [online] Available at: https://wiki.portal.chalmers.se/agda/Main/Othertutorials?action=download&upname=AgdaOverview2009.pdf.

Chubb, M. (2018). *The role of "practice" in mathematics class*. [online] Thinking Mathematically. Available at: https://buildingmathematicians.wordpress.com/2018/09/05/the-role-of-practice-in-mathematics-class/#:~:text=When%20practice%20allows%20students%20to

CodeFirst. (2020). *The Difference Between Black Box And White Box Testing*. [online] Available at: https://www.codefirst.co.uk/blog/difference-black-white-box-testing/

coq.inria.fr. (2019). *Core language — Coq 8.12.0 documentation*. [online] Available at: https://coq.inria.fr/distrib/current/refman/language/core/index.html

ELGABRY, O. (2016). *Requirements Engineering — Requirements Specification (Part 3)*. [online] Medium. Available at: https://medium.com/omarelgabrys-blog/requirements-engineering-elicitation-analysis-part-5-2dd9cffafae8.

Ewald, W. (2019). *The Emergence of First-Order Logic*. Spring 2019 ed. [online] Stanford Encyclopedia of Philosophy. Available at: https://plato.stanford.edu/entries/logic-firstorder-emergence/

GeeksforGeeks. (2018a). *Deque descendingIterator() method in Java*. [online] Available at: https://www.geeksforgeeks.org/deque-descendingiterator-method-in-java/

GeeksforGeeks. (2018b). *Stack iterator() method in Java with Example*. [online] Available at: https://www.geeksforgeeks.org/stack-iterator-method-in-java-with-example/

GeeksforGeeks. (2019). *Recursive Descent Parser*. [online] Available at: https://www.geeksforgeeks.org/recursive-descent-parser/

Ghildiyal, S. and Chandra, P. (2019). *GUI Testing Tutorial: User Interface (UI) TestCases with Examples*. [online] Guru99.com. Available at: https://www.guru99.com/gui-testing.html.

Gottschall, C. (2019). *Proof checker of Christian Gottschall's Gateway to Logic*. [online] www.erpelstolz.at. Available at: https://www.erpelstolz.at/gateway/formular-uk-beweis.html

Gurgul, A. (2018). *How to Make a Deep Copy of an Object in Java*. [online] Baeldung. Available at: https://www.baeldung.com/java-deep-copy

Hadden, S.G. and Feinstein, J.L. (1989). Symposium: Expert systems. Introduction to expert systems. *Journal of Policy Analysis and Management*, 8(2), pp.182–187.

Harrison, J. (2000). *The HOL Light manual (1.1)*. [online] Available at: https://www.cl.cam.ac.uk/~jrh13/hol-light/manual-1.1.pdf

Harrison, J. (2009). HOL Light: An Overview. *Lecture Notes in Computer Science*, pp.60–66.

Harrison, J., Urban, J., Wiedijk, F. and Paulson, L. (2014). *HISTORY OF INTERACTIVE THEOREM PROVING*. [online] *cl.cam.ac.uk*. Available at: https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf

Institute for Advanced Study (2016). *Introduction to the Coq Proof Assistant - Andrew Appel*. *YouTube*. Available at: https://www.youtube.com/watch?v=3WBUHEVr56c

Jeffrey, A. (2020). *Alan Jeffrey*. *YouTube*. Available at: https://www.youtube.com/watch?v=8WFMK0hv8bE

Khandelwal, C. (2019). *How I Wrote A Lexer*. [online] Medium. Available at: https://medium.com/young-coder/how-i-wrote-a-lexer-39f4f79d2980

Klement, K.C. (n.d.). *Fitch-style proof editor and checker*. [online] proofs.openlogicproject.org. Available at: https://proofs.openlogicproject.org/

Kreitz, C. and Pucella, R. (2005). *Encoding Mathematics in First-Order Logic*. [online] Available at: https://www.cs.cornell.edu/courses/cs486/2005sp/lect19.pdf

Ltd, C. (n.d.). *MockFlow | Software Reviews & Alternatives*. [online] crozdesk.com. Available at: https://crozdesk.com/it/mockups-prototyping-software/mockflow

Magnus, P. and Button, T. (2018). *forallx:Cambridge*. [online] Available at: http://www.homepages.ucl.ac.uk/~uctytbu/forallxcam.pdf

Matsumoto, D. (2016). Speaking of Psychology: Nonverbal communication speaks volumes. *https://www.apa.org*. [online] Feb. Available at: https://www.apa.org/research/action/speaking-of-psychology/nonverbal-communication.

Morrison, S. (2020). *Interactive theorem proving demo: infinitely many primes*. *YouTube*. Available at: https://www.youtube.com/watch?v=ArGLTAjak3g

Muldoon, N. (2018). *Customer Personas: How to Write Them and Why You Need Them In Agile Software Development*. [online] Medium. Available at: https://blog.easyagile.com/customer-personas-how-to-write-them-and-why-you-need-them-in-agile-software-development-3873f3525975

Nebel, B. (2000). *Logics for Knowledge and Representation*. [online] ResearchGate. Available at: https://www.researchgate.net/publication/229067802_Logics_for_Knowledge_Representation

Nero, R.D. (2020). *Does Java pass by reference or pass by value?* [online] InfoWorld. Available at: https://www.infoworld.com/article/3512039/does-java-pass-by-reference-or-pass-by-value.html#:~:text=Java%20always%20passes%20parameter%20variables

Nielsen, J. (2017). *10 Heuristics for User Interface Design: Article by Jakob Nielsen*. [online] Nielsen Norman Group. Available at: https://www.nngroup.com/articles/ten-usability-heuristics/.

Oracle.com. (2019). *JavaFX*. [online] Available at: https://www.oracle.com/java/technologies/javase/javafx-overview.html.

Oracle.com. (2020). *JavaFX Scene Builder Information*. [online] Available at: https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html.

Paulin-Mohring, C. (2011). *Introduction to the Coq proof-assistant for practical software verification*. [online] Available at: https://www.lri.fr/~paulin/LASER/course-notes.pdf

Pierce, B.C. and Pennsylvania), B.C. (Professor P., University of (2002). *Types and Programming Languages*. [online] *Google Books*, MIT Press, p.9. Available at: https://books.google.co.uk/books?hl=en&lr=&id=ti6zoAC9Ph8C&oi=fnd&pg=PR13&dq=%22Types+and+Programming+Languages%22&ots=ECKdtHl1-D&sig=tZ7etBjsYEd6_jCsHmN3cCItvjA&redir_esc=y#v=onepage&q=%22Types%20and%20Programming%20Languages%22&f=false

Rieppel, M. (2015). *Fitch Proof Constructor*. [online] mrieppel.github.io. Available at: https://mrieppel.github.io/fitchjs/

rj45 (2008). *Simple Example of MVC (Model View Controller) Design Pattern for Abstraction*. [online] Code Project. Available at: https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design

Ruzicka, V. (2017). *Java Cloning Problems*. [online] Vojtech Ruzicka's Programming Blog. Available at: https://www.vojtechruzicka.com/java-cloning-problems/

Schwartz, W. (2020). *Tree Proof Generator*. [online] www.umsu.de. Available at: https://www.umsu.de/trees/

Setzer, C. (2007). *CS_336/CS_M36 (Part 2)/CS_M46 Interactive Theorem Proving Dr. Anton Setzer Lent Term 2008 CS 336/CS M36 (part 2)/CS M46 Interactive Theorem Proving, Lent Term 2008, Sect. 0*. [online] Swansea University Computer Science Department. Available at: http://www.cs.swan.ac.uk/~csetzer/lectures/intertheo/07/intertheodraft0.pdf

Shapiro, S. and Kouri Kissel, T. (2018). *Classical Logic*. Spring 2018 ed. [online] Stanford Encyclopedia of Philosophy. Available at: https://plato.stanford.edu/entries/logic-classical/.

Sharma, L. (2019). *What is Boundary Value Analysis (BVA) of Black Box Testing Technique?* [online] TOOLSQA. Available at: https://toolsqa.com/software-testing/istqb/boundary-value-analysis/

Siegfried, R.M. (n.d.). *CSC 270 - Syntax and Semantics in Scheme (Racket)*. [online] home.adelphi.edu. Available at: https://home.adelphi.edu/~siegfried/cs270/270rl6.html#:~:text=The%20vocabulary%20is%20the%20collection

Smartbear.com. (2019). *What is Automated Testing | Software Testing Basics*. [online] Available at: https://smartbear.com/learn/automated-testing/what-is-automated-testing/.

Smith, P. (2018). *3. Natural deduction and sequent proofs*. [online] Logic Matters. Available at: http://www.logicmatters.net/latex-for-logicians/nd/

Steinberg, A. (2010). *ND.sty Lemmon-style natural deduction proofs*. [online] Available at: https://www.logicmatters.net/resources/pdfs/nd-manual2a.pdf

Thomas Hales (2018). *A Review of the Lean Theorem Prover*. [online] Jigger Wit. Available at: https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/

Tomassetti, G. (2017). *A Guide To Parsing: Algorithms And Terminology*. [online] Federico Tomassetti - Software Architect. Available at: https://tomassetti.me/guide-parsing-algorithms-terminology/

tutorialspoint.com (2019a). *Data Flow Architecture - Tutorialspoint*. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/software_architecture_design/data_flow_architecture.htm#:~:text=Batch%20Sequential

tutorialspoint.com (2019b). *UML - Use Case Diagrams*. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/uml/uml_use_case_diagram.htm.

Usability.gov. (2019). *Usability Evaluation Basics | Usability.gov*. [online] Available at: https://www.usability.gov/what-and-why/usability-evaluation.html.

Väänänen, J. (2020). *Second-order and Higher-order Logic*. Fall 2020 ed. [online] Stanford Encyclopedia of Philosophy. Available at: https://plato.stanford.edu/entries/logic-higher-order/#HighOrdeLogiVisVisTypeTheo

Visual-paradigm.com. (2019). *UML Class Diagram Tutorial*. [online] Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/.

Wiegers, K.E. (2002). *Software Requirements Specification Template*. [online] exinfm.com. Available at: https://exinfm.com/training/M2C3/.

Wikipedia Contributors (2019). *Comparison of parser generators*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Comparison_of_parser_generators.

Wold, W.W. (2017). *The Programming Language Pipeline*. [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/the-programming-language-pipeline-91d3f449c919/

Wong, E. (2020). *Heuristic Evaluation: How to Conduct a Heuristic Evaluation*. [online] The Interaction Design Foundation. Available at: https://www.interaction-design.org/literature/article/heuristic-evaluation-how-to-conduct-a-heuristic-evaluation#:~:text=Pros%20of%20Heuristic%20Evaluation&text=Heuristic%20evaluation%20does%20not%20carry

10.2. Source Code References

Code snippets from other classes and methods were gratefully borrowed from:

Class AbstractSyntaxTree drew heavy influence from Uday Reddy for his 2018 Tree class, which heavily influenced the design of this tree.

Class GUIController; method appendToCaretTextArea: https://alvinalexander.com/source-code/javafx-textarea-how-insert-text-cursor-caret-position/ for appending to cursor position.

Class GUIController; method compileLines: https://stackoverflow.com/questions/13880638/how-do-i-pick-up-the-enter-key-being-pressed-in-javafx2 for detecting enter button being pressed in TextArea.

Class GUIController; method saveProofCommands: https://www.genuinecoder.com/save-files-javafx-filechooser/ for saving to a text file.

Class GUIController; method loadProofCommands: https://www.genuinecoder.com/save-files-javafx-filechooser/ for using fileChooser.

Class GUIController; method loadProofCommands: http://tutorials.jenkov.com/java-io/filereader.html for reading from a file.

Class GUIController; method printToPDF: https://ijavayou.wordpress.com/2016/02/08/javafx-easy-way-to-save-scenesnodes-as-pdf/ for printing contents of a JavaFX element to PDF.

Class GUIView; initialisation of PRIMARYSCREENBOUNDS field: // https://docs.oracle.com/javafx/2/api/javafx/stage/Screen.html for setting the height and width to the current screen sizs.

Class GUIView; method createLogicTabs: https://www.geeksforgeeks.org/javafx-tabpane-class/#:~:text=Add%20event%20handler%20to%20the,select()%20function. for creating the logic tabs.

Class GUIView; method createConnectiveShortcuts: https://stackoverflow.com/questions/37542955/how-to-set-a-tooltip-on-a-javafx-button/37543167 for adding ToolTips to the connective buttons.

Class Prood; method notInClosedAssumption: https://stackoverflow.com/questions/1066589/iterate-through-a-hashmap for iterating through a HashMap.

11. Appendices

Appendix A: Source Code

The source code for this system can be found at:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2019/mxr917

Three directories can be found at that location. The directory named "FinalSubmission-MXR917" contains the complete and final source code for the project. Within that directory are all the class files which constitute the system. It also contains a PDF titled "Quick-Start User Guide," which contains a guide on how to interact with the GUI for the novice user. The two directories named "Prototype1(PROPLOG)" and "Prototype2(PROPLOG,PREDLOG,GUI)" can be ignored, as they are backups of the source code from earlier on in the project period.

The entry point to the programme is GUIView.java. To run this file, the following external JAR files need to be added to the program's module path:

- javafx-swt.jar
- javafx.base.jar
- javafx.controls.jar
- javafx.fxml.jar
- javafx.graphics.jar
- javafx.media.jar
- javafx.swing.jar
- javafx.web.jar

The JDK 11 library was used as the specific JRE for this project, owing to difficulties in setting up JavaFX with Eclipse (the author's IDE of choice). It is strongly recommended that the code be run with JDK11.

The following VM arguments need to be added to the run configurations:

- --module-path (location of JavaFX JAR files in memory)
- --add-modules=javafx.base
- --add-modules=javafx.controls
- --add-modules=javafx.fxml
- --add-modules=javafx.graphics
- --add-modules=javafx.media
- --add-modules=javafx.swing
- --add-modules=javafx.web

To run the JUnit test classes, the JUnit 5 library should be configured in the class path.

Furthermore, to ensure correct outputs, run with a UTF-8 character set.

If in doubt, open and run the code in Eclipse.

Appendix B. Requirements Specification Document

Other appendices may include project proposal, Software Requirements Specification, selection of experimental data, schedules, testing strategy, risk management plans, glossary, manual.

Appendix B: Heuristic Evaluatios of Related Systems

As stated in 3.2, Nielson's heuristics were applied to several related technologies. They can become repetitive to read, and so were ommitted from the body of the text, but have been written up and included here for reference and as evidence of the work done.

Heuristic Evaluation of 3.2.2.1: Natural Deduction Proof Editor and Checker

Available online at: https://proofs.openlogicproject.org/ (Klement, n.d.)

Visibility of system. System provides no information about its current status, but the functionality of the system is simple enough that this would likely not be expected. It is clear to the user that the system is waiting for the user to check the proof. Feedback after checking the proof is quick and useful.

Match between system & real world. System closely follows the same logic textbooks used as the basis for the system of formal logic discussed in this process. To that extent, it captures the user's concepts closely. System accepts as input the specific logical connectives, or various easily recognisable proxies.

User control & freedom. Deleting a line can be easily done with a button. While the system should provide guidance for the user, it should also be flexible to their needs, such as by providing 'undo' functions or offering an 'emergency exit' to exit the system's current state if things go wrong.

Consistency and standards. The system offers a diversity of different symbols for each logical connective, but arguably this is not a consistency issue as it gives the user more freedom to interact with the system in a manner of their choosing.

Error prevention. Because the system is a tool for users to practice writing Fitch-style proofs, the user is not protected from making errors in the writing of proofs until after they have sent their proof for checking.

Recognition rather than recall. All terms of formal logic must be entered manually, but a table of the relevant keyboard shortcuts is provided.

Flexibility and efficiency of use. Number of different keyboard shortcuts for different connectives of formal logic help to speed up the inputting process. Manually adding lines using buttons can become tedious. User must explicitly declare the rule they used to derive the new line of the proof. This slows the process down significantly.

Aesthetic and minimalist design. System displays all relevant formal rules on the homepage, alongside all instructions for use. The result is very cluttered.

Helps users recognise, diagnose, and recover from error. The error messages displayed accurately described the nature of the error (i.e. a syntax error or an invalid derivation) with a clear reference to the line of the proof on which it occurs. "Start Over" clears all lines except the premises and allows the user to begin from scratch.

Help and documentation. All keyboard shortcuts are provided in a table. Some ambiguity around this, e.g. the table displays a symbol closely resembling the alphabetical 'V' when in fact it is the formal symbol for disjunction, and using 'V' instead results in a syntax error.

Heuristic Evaluation of 3.2.2.2: Tree Proof Generator

Available online at: https://www.umsu.de/trees/ (Schwartz, 2020)

Visibility of system. When running, the system displays each level of the tree subsequently as that level is generated. In doing this, the system makes it implicitly clear to the user that it is processing the proof.

Match between system & real world. The system closely adheres to the syntax of formal logic with its input language, using 'lookalike' symbols (such as <> to represent the diamond of modal logics) to represent the logical connectives, and generally only offering one such symbol per connective. The resultant proof is a semantic tableau reminiscent of any one might hand-draw when practicing drawing tree proofs.

User control & freedom. The system offers the user the freedom to use several different logics, including propositional, first-order, and some normal modal logics (which are beyond the scope of this project). As the system only collects a small amount of input from the user, it does not afford the user much control. However, it could be argued that the user does not require much control in this setting. If there are multiple solutions to a problem, the user can only use the one automatically derived by the generator. The user cannot cancel the proof generating process once it has begun.

Consistency and standards. The system displays inputs and outputs as well-formed formulas of formal logic, and adheres to the conventions of displaying tree proofs. There is little internal disparity in the system's own syntax. Pressing buttons to automatically enter logical connectives inserts the actual symbol rather than the provided shortcut. This should not be confusing to anyone.

Error prevention. Syntax errors are caught when the user runs the system, rather than when they are writing input.

Recognition rather than recall. The system offers both buttons for inexperienced users and keyboard shortcuts for more experienced users. Multiple bracketing conventions are supported for ease of use.

Flexibility and efficiency of use. Keyboard shortcuts speed up input for more experienced users.

Aesthetic and minimalist design. Information provided on the home screen comprises of a summary on how to enter formulae, how to enter full arguments, and which logic the system supports. This is displayed underneath the user input area, making for a cleaner aesthetic. The display is still somewhat cluttered, but with a utilitarian feel.

Helps users recognise, diagnose, and recover from error. Formulas which are not well-formed are caught during runtime. The system identifies the specific cause of the error, such as the token at fault. The user can then edit their input accordingly.

Help and documentation. All information for usage is displayed on the homepage of the system. However, it is concise, and displayed underneath the area where the user interfaces with the system.

Heuristic Evaluation of 3.2.2.3: FitchJS

Available online at: https://mrieppel.github.io/fitchjs/ (Rieppel, 2015)

Visibility of system. As the system's functionality is very specific – simply checking a formula and rule, and adding them to the proof if they are valid derivations – there are no significant points during which the system needs to notify the user of its status.

Match between system & real world. The Fitch-style proof matches the formatting that the user would expect. The keyboard shortcuts for the logical connectives closely resemble the symbols themselves.

User control & freedom. The user has the option to delete the bottom line of the proof. If needs be, they could delete the entire proof this way. Being unable to remove lines within the proof could prove an issue (e.g. if the user follows a certain line of reasoning the proof, realises a different line would better reach the conclusion, and thus renders those lines superfluous). The user also has the power to "Import" the proof if they would rather type it by hand instead of entering it using the buttons. Instructions are provided into the box regarding how to do this, but these are very unclear. Examples can be found on a link on the "Reference" help page which reveal that the user must import the proof by manually drawing it using "|" and "_" symbols to represent the vertical and horizontal lines. This is a longwinded and tedious process, as the formatting must be exactly correct for the system to be able to make sense of the input.

Consistency and standards. The system is consistent in its representation of logic to the user. All aspects of the system dealing with user input take as input keyboard shortcuts representing the logical connectives, including the drop-down menu of logic rules.

Error prevention. Malformed formulas are not added to the proof, instead receiving an error message. An invalid derivation encourages a prompt to help the user understand their error. For example, attempting to use conjunction elimination on a line which does not contain a conjunction prompts an error message that the formula on the line for which elimination is attempted must be a conjunction.

Recognition rather than recall. The "Reference" tab at the top of the page lets the user see all relevant keyboard shortcuts. Tabbing back and forth may be less efficient than if the system was altered slightly to include button shortcuts.

Flexibility and efficiency of use. The keyboard shortcuts make the system quick and easy to use, as does the derivation rule using the drop-down menu. The system is much less 'clunky' than the Natural Deduction Proof Editor & Checker.

Aesthetic and minimalist design. The user interface is very clean. Three tabs allow the user to either construct a proof, import or export a proof, or see the logic rules permitted and keyboard shortcuts used. This separation into tabs makes each individual page very clean.

Helps users recognise, diagnose and recover from error. As mentioned, invalid derivations or malformed formulas are clearly identified with a helpful prompt as to how to resolve the issue.

Help and documentation. The "Reference" tab provides all required information to use the proof builder. This information is categorised into drop-down menus for efficient use. As mentioned earlier, the help for importing a proof is inadequate to get started – the user must separately tab to the help section and view example proofs.

Heuristic Evaluation of 3.2.2.4: Gateway to Logic's Logic Calculator

Available online at: https://www.erpelstolz.at/cgi-bin/cgi-form?key=0000624d. (Gottschall, 2019).

Visibility of system. The system is very quick to check the proof, but like the other proof checkers, its functionality is specific enough that a status update is not necessary.

Match between system & real world. It is relatively easy to type L-style proofs compared to Fitch-style, owing to the lack of vertical and horizontal lines to draw. The system can therefore take as input a proof which looks exactly like the hand-drawn proof that they would create. The selection of keyboard shortcuts representing formal proofs again closely represent the symbols of formal logic.

User control & freedom. The user has full control over the input in this case. While the system should provide guidance for the user, it should also be flexible to their needs, such as by providing 'undo' functions or offering an 'emergency exit' to exit the system's current state if things go wrong.

Consistency and standards. The system adheres to both the conventions of its natural deductive system, and to its own internal conventions for representing logic formulas. The shortcuts resemble the logical connectives, and only one shortcut is provided per connective.

Error prevention. No error prevention is provided within the user input area; the user is given total control over input. A "Help with language" link takes the user to a page which can help them to understand the syntax of the system.

Recognition rather than recall. As no buttons are provided, the system relies totally on the recall of the user. However, the aforementioned help link allows the user to easily check the rules governing acceptable input.

Flexibility and efficiency of use. Using the system is fast and efficient. Giving the user total control over the inputs mean that writing out a full proof is a quick process. The system also ignores blank space between elements, and so the user does not need to be concerned with laying out the proof exactly in line as they would when hand-drawing an L-style proof.

Aesthetic and minimalist design. Most help documentation is provided in a separate page. On the main proof-checking page, there is only links to other areas of the site, a note on representing quantifiers, the input area, and the two buttons. Once the execute function is applied, the system only the proof and the relevant messages and links to other parts of the page are displayed.

Helps users recognise, diagnose and recover from error. The system is specific in helping the user identify a problem, ranging from identifying any malformed tokens, to identifying a wrong list of assumptions and indicating what the correct list should be. The system is very helpful to the user in that sense.

Help and documentation. The user guide is concise, but conveys all necessary information to begin using the system.

Heuristic Evaluation of 3.2.2.5: Lean

Lean available online at: https://leanprover.github.io/, (Avigad, De Moura and Roesch, 2020).

Demonstration of Lean used (Morrison, 2020).

Visibility of system. The system responds to each new line that is input as that line is being written. For example, if the user pauses hallway through writing a sentence, the system will display an error message.

Match between system & real world. The system attempts to map between the input language and the mathematical interpretation. For example, "intro N" is shorthand for "N is among the natural numbers." Keywords such as "let" and "assume" operated as similar shorthand which mirror the language used in mathematical proofs. A natural language correspondence is provided where possible; shorthand is provided instead when it is expedient to do so. Within propositional logic, the text editor shortcuts match the name of the logical connective. For example, "\iff" represents the biconditional. However, no diamond symbols are allowed, even these are frequently used in many different areas of mathematics (Thomas Hales, 2018).

User control & freedom. The system provides a keyword 'sorry' which stands as a proxy for some subproof, allowing the user to engage with the high-level structure of the proof before going into the details. The user must adhere closely to the syntax of the Lean programming language in order to interact with the system successfully, but can use any syntactically correct term of the language that they wish. It is easy for the user to edit or remove any part of the proof as they can just modify that part of the input field.

Consistency and standards. The input language used to interact with Lean is large, and covers many concepts and terms with which the user is unfamiliar. It is therefore difficult to comment on this.

Error prevention. The user has full control over the input as the input area is just a text editor. No error prevention is provided.

Recognition rather than recall. The system relies on extensive knowledge of the programming language it takes as input. However, the Lean Theorem Prover is clearly not intended for novice users, so this makes sense.

Flexibility and efficiency of use. The user is only allowed to use the keyboard to interact with the system, so the efficiency of use will be closely connected with the user's knowledge of the system's input language. The user also has the freedom to use "Tactics" in proofs, which allow the system to use some technique to prove some part of the proof. These include tactics pre-installed in the system, and tactics written by the user. This can make the proof harder for a reader to interpret, but also saves the reader a lot of work.

Aesthetic and minimalist design. The interface is very clean, containing a few buttons opening specific functions (such as debugging and exploring files). Otherwise, the screen is split down the middle between the text editor in which the user inputs code, and the display area where the system displays any messages (such as error messages, or a mathematical representation of a certain lemma).

Helps users recognise, diagnose and recover from error. Error messages clearly diagnose problems.

Help and documentation. The online Lean manual (Avigad, De Moura and Kong, 2020) is a document with over 170 pages, covering everything from the syntax of the Lean language to the theory in which it is based. It is as searchable as any PDF. The browser version of Lean also offers an interactive textbook at https://leanprover.github.io/programming_in_lean/#02_Programming_Basics.html.

Software Requirements Specification

for

PRODLOG

Automated Proof-Checker for Propositional and Predicate Logic

Prepared by Mark Robinson

Table of Contents

Introduction

Purpose

PRODLOG is the name given to the automated proof-checker for propositional and predicate logic discussed within this document. It is also the name given to the language used to interact with that system. The purpose of this document is to specify the set of functional (and nonfunctional) requirements for that system.

Intended Audience and Reading Suggestions

This document is primarily intended for project management and quality purposes by the author and developer of the system. It may also be of use for the project's supervisor and marker.

References

The systems of propositional and predicate logic used as the basis of this system are those specified in the 2018 update of the textbook forallx:Cambridge by P.D. Magnus Tim Button:

> Magnus, P. and Button, T. (2018). *forallx:Cambridge*. [online] Available at: http://www.homepages.ucl.ac.uk/~uctytbu/forallxcam.pdf.

Furthermore, grateful acknowledgement is given to W.E. Wiegers for creating the template requirements specification adapted into this document:

> Wiegers, K.E. (2002). *Software Requirements Specification Template*. [online] exinfm.com. Available at: https://exinfm.com/training/M2C3/.

Overall Description

Product Perspective

The system is a self-contained product with the primary purpose of allowing the user to produce proofs of propositional and first-order logic using Fitch-style notation.

Product Features

The system consists of a graphical user interface through which the user and system interact. The user communicates with the system via the PROPLOG and PREDLOG languages. These are special-purpose languages designed for this project. They represent propositional and predicate logic respectively, and are collectively referred to as PRODLOG (or, less formally, 'the input language'). The system contains an interpreter, which converts this input into the relevant data structures, and a proof-checking system which stores those data structures and checks the validity of logical inferences. The output is a deductive proof adhering to Fitch-style notation.

User Classes and Characteristics

Three primary user classes are recognized, distinguished on the lines of their familiarity with formal logic, and with the system:

i. *Low familiarity with formal logic and low familiarity with the system*. This system is a pedagogical tool and is intended to help students of formal logic to practice natural deduction in as easy a manner as possible. This means that the implementation must be highly intuitive to use.
ii. *High familiarity with formal logic and low familiarity with the system*. The system can also be used to write more powerful proofs and should be easy for a new user to master when they are already familiar with the underlying logic.
iii. *High familiarity with formal logic and high familiarity with the system*. Experienced users of both the system and the underlying logic should be able to create proofs as quickly and efficiently as possible.

Operating Environment

The software should be able to run on any desktop operating system running the Java Software Development Kit with added JavaFX libraries.

User Documentation

The PRODLOG Quick-Start User Guide covers the basic information required for the user to begin interacting with the system. A brief scan of forallx:Cambridge (Magnus and Button, 2018) is recommended for users unfamiliar with the specific conventions of formal logic represented by this system.

System Features

The functional requirements of seven key features of the system are presented. For the input language, data structure used to store propositions, and the data structure used to store proofs, they are relatively brief. More detailed are the requirements for the interpreter, inference checker, and the components and functionality of the GUI. By the end of this section, the entire system is fully specified in terms of functional system requirements.

Input Language

3.1.1 Description and Priority

The input language is of high priority, as it is the means by which the user will interact with the back end of the system. It is difficult to describe functional requirements for such a language without crossing over into a description of non-functional requirements and implementation, but an attempt to do so has nevertheless been made.

3.1.2 Functional Requirements

3.1.2.1. A full language specification document should be produced to describe the input language.
3.1.2.2. The input language specification should describe a vocabulary, grammar, and semantics.
3.1.2.3. The vocabulary should include a term to specify whether a line of proof is a premise, assumption, conclusion, derivation, or command to close an assumption.

3.1.2.4. The syntax should require the user to specify the type of each line of input (as either a premise, assumption, conclusion, derivation, or command to close an assumption).
3.1.2.5. The semantics should require that the specification of any type other than a command to close an assumption should be followed by a proposition.
3.1.2.6. The vocabulary should have a term corresponding to each non-alphanumeric symbol of formal logic (conjunction, disjunction, conditional, biconditional, negation, absurdity, universal quantifier, existential quantifier, identity symbol, open bracket, close bracket).
3.1.2.7. The semantics of each term of the vocabulary corresponding to a term of formal logic should be the same as for that term of formal logic.
3.1.2.8. The vocabulary must allow for the creation of atomic formulae/propositions.
3.1.2.9. The vocabulary must allow for the creation of compound propositions.
3.1.2.10. It must be possible to formulate in the input language any proposition which can also be formulated in propositional and predicate logic.
3.1.2.11. It must not be possible to formulate in the input language any proposition which cannot also be formulated in propositional and predicate logic.

Data Structures: Proposition

3.2.1 Description and Priority

The data structure used to store propositions is critical to the success of the system, and as such is of high priority. It has a relatively small number of requirements compared to the other components of the system, but they are important to specify.

3.2.2 Functional Requirements

3.2.2.1. The proposition data structure should store the type of the proposition (i.e. atomic proposition, atomic formula, conjunction, disjunction, conditional, biconditional, universally quantified, existentially quantified).
3.2.2.2. The proposition data structures representing compound propositions should store the proposition(s) ranged over by the proposition's main logical operator.
3.2.2.3. The process described in 3.2.3.2 should be a recursive process, with a base case of an atomic formula or atomic proposition.
3.2.2.4. The proposition data structure should generate a key which is unique to that proposition but will be common among all instances of that proposition.

Data Structure: Proof

3.3.1 Description and Priority

There are some necessary requirements for how proofs are stored within the system. These are small in number but are of high priority.

3.3.2 Functional Requirements

3.3.2.1. The proof data structure should store a proof as a collection of lines.
3.3.2.2. The proof data structure should store, for each line, the number of that line.
3.3.2.3. The proof data structure should store, for each line, the key generated by that line's proposition as specified in 3.2.3.4.
3.3.2.4. The proof data structure should store, for each line, the number of assumptions deep that line is.
3.3.2.5. The proof data structure should store, for each line, the rule used to derive that line (including if the line was a premise, assumption, or conclusion).

3.3.2.6. The proof data structure should store, for each line, the proposition data structure to which that line refers.
3.3.2.7. The proof data structure should store up to one conclusion.

Interpreter

3.4.1 Description and Priority

The purpose of the interpreter is to convert each line of the user's input into a data structure representing a line of a proof as specified in 3.3.2. It consists of three modules: a lexer, for breaking the input down into labelled tokens; a parser for arranging those tokens into an abstract syntax tree; and a function to convert the tree into a proposition data structure and line of proof.

3.4.2 Functional Requirements

3.4.2.1. The interpreter should take as input a line of input in PRODLOG corresponding to a line of a Fitch-style proof.
3.4.2.2. The interpreter should contain a lexing system.
3.4.2.3. The lexer should convert the input into a list of tokens.
3.4.2.4. Each token should be labelled with what type of term of the vocabulary it is.
3.4.2.5. If a token is not a part of the vocabulary of the language, the system should throw an exception containing an appropriate error message.
3.4.2.6. The interpreter should contain a parsing system.
3.4.2.7. The parsing system should take as input the list of tokens output by the lexer.
3.4.2.8. The parser should use the type attached to each token in the list to convert the list into a hierarchical abstract syntax tree, in accordance with the syntax of the input language.
3.4.2.8. If any part of the term list violates the syntax of the language, the system should throw an exception containing an appropriate error message.
3.4.2.9. The interpreter should contain a function to convert an abstract syntax tree into a line of proof.
3.4.2.10. The function described in 3.4.2.9 should take as input an abstract syntax tree.
3.4.2.11. The function described in 3.4.2.9 should turn the input into a proposition data structure as described in 3.2.
3.4.2.12. The function described in 3.4.2.9 should add to the proposition output by 3.4.2.11 to create a line of proof as described in 3.3.2.
3.4.2.13. The interpreter should output the line of proof output in 3.4.2.12.

Inference Checker

3.5.1 Description and Priority

The purpose of the inference checker is to check if the lines currently stored in the proof data structure, in conjunction with the algorithmic representation of the rules of inference, are sufficient to validly infer the line of proof output by the interpreter. If they are, then the system should execute successfully and output a representation of the proof to the command line. The controller of the GUI should then output the final proof to the GUI (if implemented). If they are not, then an exception specifying an invalid derivation should be thrown. The inference checker is of high priority. It is necessary to create a minimum viable version of the system.

3.5.2 Functional Requirements

3.5.2.1. The inference checker should check the type of the line output by the interpreter.
3.5.2.2. If the line is a premise, the system should add it to the start of the stored proof.
3.5.2.3. If the line is an assumption, the system should add it to the proof.

3.5.2.4. If the line is an assumption, the system should add it to the proof's record of open assumptions.

3.5.2.4. If the line is an instruction to close an assumption and the proof contains at least one open assumption, then the outermost assumption should be closed.

3.5.2.5. If the line is an instruction to close an assumption and the proof does not contain at least one open assumption, then the command should be ignored.

3.5.2.6. If the line is a conclusion then the system should check if the proof already stores a conclusion.

3.5.2.7. If the proof does not already store a conclusion, then the system should store the new line as the proof's conclusion.

3.5.2.8. If the proof already stores an assumption, the new line should be compared against that conclusion.

3.5.2.9. If the new line contains the same proposition as the stored conclusion, the new line should be ignored.

3.5.2.10. If the new line does not contain the same proposition as the stored conclusion, then the new line should be stored as the proof's conclusion, replacing the old line.

3.5.2.11. If the line is a derivation and the system's logic is set to propositional logic, then the system should check if the lines of proof stored so far are sufficient to validly infer that new line using any one of the rules in the following list. Note that the specific form of these rules is that specified in forallx:Cambridge (Magnus and Button, 2018).

- Conjunction Introduction
- Disjunction Introduction
- Negation Introduction
- Conditional Introduction
- Biconditional Introduction
- Conjunction Elimination
- Disjunction Elimination
- Negation Elimination
- Conditional Elimination
- Biconditional Elimination
- Explosion (Ex Falso Quodlibet)
- Tertium Non Datur (Law of Excluded Middle)
- Reiteration
- Double Negation Elimination
- Disjunctive Syllogism
- Modus Tollens
- De Morgan Rules (all four versions)

3.5.2.12. If the line is a derivation and the system's logic is set to predicate logic, then the system should check if the lines of the proof stored so far are sufficient to validly infer that new line using any one of the rules in the list specified by 3.1.2.11, or any one of the rules from the following list. Note that the specific form of these rules is that specified in forallx:Cambridge (Magnus and Button, 2018).

- Universal Quantifier Introduction
- Existential Quantifier Introduction
- Identity Introduction
- Universal Quantifier Elimination
- Existential Quantifier Introduction
- Identity Elimination
- Conversion of Quantifiers (all four versions)

3.5.2.13. If the line is a derivation and satisfies an inference rule, then the line should be updated to store the inference rule used to derive it.

3.5.2.14. If the line is a derivation and satisfies an inference rule, then the line should be added to the end of the proof.

3.5.2.15. If the line is a derivation and satisfies an inference rule and the proof stores a conclusion, the proposition stored in that line should be compared against the proposition stored in the conclusion of the proof.

3.5.2.16. If the line is a derivation and satisfies an inference rule and the proof stores a conclusion and the proposition stored in the line matches the

proposition stored in the conclusion of the proof, then a notification that the conclusion has been reached should be output.

3.5.2.17. If line is a derivation and satisfies an inference rule and the line has been added to the proof, then the complete proof should be visually represented on the command line.

3.5.2.18. If line is a derivation and satisfies an inference rule and the line has been added to the proof and the GUI has been implemented, then the controller of the GUI should handle outputting the proof to the GUI (see 3.7).

3.5.2.19. If the line is a derivation and does not satisfy any inference rule, then an exception should be thrown warning that the user's input is an invalid derivation, and it should not be added to the proof.

3.5.2.20. If no GUI is implemented, the proof-checking system should also provide a means for the user to interact with the system via the command-line.

3.5.2.21. In the case described in 3.4.2.19, the proof-checking system should pass the user's input to the interpreter.

Graphical User Interface: Components

3.6.1 Description and Priority

A rich GUI is the easiest way for the user to interact with the system and should allow for maximum usability by containing shortcuts for all components of the input language. It is of medium priority, as the specification for the proof checker describes a minimum viable product allowing input and output to and from the command line. The code generating the GUI's layout should be contained in a separate class to the rest of the code.

3.6.2 Functional Requirements

3.6.2.1. The display should be clearly split into two halves: an input area and an output area.

3.6.2.2. The input area should contain:

3.6.2.2.1, A button to switch between propositional and predicate logic.
3.6.2.2.2. A label indicating the location of the keyword shortcuts.
3.6.2.2.3. A button to insert a premise into the input field.
3.6.2.2.4. A button to insert a derivation into the input field.
3.6.2.2.5. A button to insert a assumption command into the input field.
3.6.2.2.6. A button to insert a conclusion command into the input field.
3.6.2.2.7. A label indicating the location of the logical symbol shortcuts.
3.6.2.2.8. A button to insert a conjunction into the input field.
3.6.2.2.9. A button to insert a disjunction into the input field.
3.6.2.2.10. A button to insert a conditional into the input field.
3.6.2.2.11. A button to insert a biconditional into the input field.
3.6.2.2.12. A button to insert a negation into the input field.
3.6.2.2.13. A button to insert a universal quantifier into the input field.
3.6.2.2.14. A button to insert an existential quantifier into the input field.
3.6.2.2.15. A button to insert logical absurdity into the input field.
3.6.2.2.16. A button to insert identity into the input field.
3.6.2.2.17. Hovering the mouse over any button denoting a logical symbol should display a tooltip demonstrating the syntax of using that symbol in the PRODLOG language.
3.6.2.2.18. A large, scrollable area into which text can be received, constituting the input field.
3.6.2.2.19. A button to save the contents of the input field.
3.6.2.2.20. A button to load text into the input field.
3.6.2.2.21. A button to print the contents of the output area.

3.6.2.3. The output area should include:

3.6.2.3.1. An area into which Fitch-style proofs are displayed (the requirements specifying the output proof are described in 3.7.2.23 and 3.7.2.24).
3.6.2.3.2. An area into which messages to the user are displayed.

Graphical User Interface: Functionality

3.7.1 Description and Priority

The event handlers for the components of the GUI are described here, at the same medium priority level as the GUI itself.

3.7.2 Functional Requirements

3.7.2.1. Switching from propositional to predicate logic should cause the predicate logic symbols to appear.
3.7.2.2. Switching from predicate to propositional logic should cause the predicate logic symbols to disappear.
3.7.2.3. Pressing the button to insert a premise should insert the PRODLOG code for a premise on a new line of the input area.
3.7.2.4. Pressing the button to insert a derivation should insert the PRODLOG code for a derivation on a new line of the input area.
3.7.2.5. Pressing the button to close an assumption should insert the PRODLOG code for closing an assumption on a new line of the input area.
3.7.2.6. Pressing the button to insert a conclusion should insert the PRODLOG code for a conclusion on a new line of the input area.
3.7.2.7. Pressing the button to insert a conjunction should insert the PRODLOG code for a conjunction at the user's cursor position in the input area.
3.7.2.8. Pressing the button to insert a disjunction should insert the PRODLOG code for a disjunction at the user's cursor position in the input area.
3.7.2.9. Pressing the button to insert a conditional should insert the PRODLOG code for a conditional at the user's cursor position in the input area.
3.7.2.10. Pressing the button to insert a biconditional should insert the PRODLOG code for a biconditional at the user's cursor position in the input area.
3.7.2.11. Pressing the button to insert a negation should insert the PRODLOG code for a negation at the user's cursor position in the input area.
3.7.2.12. Pressing the button to insert a universal quantifier should insert the PRODLOG code for a universal quantifier at the user's cursor position in the input area.
3.7.2.13. Pressing the button to insert an existential quantifier should insert the PRODLOG code for an existential quantifier at the user's cursor position in the input area.
3.7.2.14. Pressing the button to insert logical absurdity should insert the PRODLOG code for logical absurdity at the user's cursor position in the input area.
3.7.2.15. Pressing the button to insert an identity symbol should insert the PRODLOG code for identity at the user's cursor position in the input area.
3.7.2.16. Pressing the button to save a proof should open a file chooser allowing the user to save the contents of the input field to a text file.
3.7.2.17. Pressing the button to load a proof should open a file chooser allowing the user to load a text file into the input area.
3.7.2.18. The file chooser for loading a proof into the input area should prevent the loading of any file type of file other than a text file into the input area.
3.7.2.19. Pressing the button to print a proof should open a window allowing the user to select options for printing the proof, including printing to PDF.
3.7.2.20. Pressing the enter key while the cursor is in the input area should pass the contents of the input area to the interpreter.
3.7.2.21. If the back-end system produces any exceptions (such as those described in 3.4.2.8 and 3.5.2.19), these should be caught by the controller and the error message displayed in the output area.
3.7.2.22. If a derivation matches the stored conclusion of a proof (a specified in 3.5.2.16), a notification that the conclusion reached should be displayed in the output area for as long as that derivation is the last line of the proof.
3.7.2.23. After the proof has been updated following a press of the enter key in the input area; if the proof contains no inference errors, the proof should be displayed in the output area, adhering to Fitch-notation.

3.7.2.24. If the proof contains at least one inference error, all lines of proof before the error (if any exist) should be displayed in the output area, adhering to Fitch-notation.

Other Nonfunctional Requirements

Performance Requirements

Identifying inference rules will require a lot of searching within the proof data structure, so designing an efficient technique to do this will be paramount.

Software Quality Attributes

The system should be able to handle any and all inputs passed to it by the user. Any malformed inputs should be identified, and an error message should be displayed. All invalid derivations should be caught. The inference rule identified for a valid derivation should always be identified correctly. No valid derivation should be missed and declared invalid. The system should also be highly modular for maintainability and testability. It should be capable of handling both simple and complex proofs. Finally, the user interface should be intuitive to use, offering as close a match to the user's concepts of formal logic as possible.

Appendix D: Specification for the Input Language

Grateful acknowledgement is given to P.B. Magnus and Tim Button for their textbook forallx:Cambridge (2018). Many of the concepts below, such as the recursive definition of a proposition / formula / sentence, are derived from that work.

D.1. Vocabulary, Syntax, and Semantics of PROPLOG

The vocabulary of PROPLOG can be split into three broad categories. First is the vocabulary and semantics for formulating propositions:

Type of Proposition	Word of Vocabulary	Represents
Binary	AND	Conjunction
	OR	Disjunction
	IFTHEN	Conditional
	IFF	Biconditional
Unary	NOT	Negation
Atomic	One capitalised alphabetic letter, with or without trailing numbers	Atomic proposition
	ABSURD	Logical absurdity

The "represents" column describes the concept of propositional logic represented by the word of PROPLOG. They are grouped by type based on whether they join two sentences of PROPLOG (binary), range over one sentence of PROPLOG (unary), or are an atomic sentence of the language (atomic). The significance of this will soon be described when considering the syntax of the language.

The second category of the PROPLOG vocabulary is keywords:

Word of Vocabulary	Represents
#PREMISE	Assert a premise
#CONCLUDE	Assert a conclusion
#ASSUME	Assert an assumption
#ENDASSUME	Close the outermost assumption
#DERIVE	Assert a new derivation

Again, the concept from propositional logic represented by each word is included in the "Represents category."

The third is symbols:

Symbol	Represents
(	Open bracket
)	Close bracket

The syntax of PROPLOG is simple enough and builds on this separation of the vocabulary into categories. We first draw a distinction between propositions and sentences of the language. A sentence of the language can be defined as a valid input of the system. A proposition of the language can be defined as something which, if paired with a keyword, would constitute a sentence of the language.

Therefore, we can recursively define propositions of the language as follows:

If ***A*** is an atomic proposition, then ***A*** is a proposition of PROPLOG.

If ***B*** is a proposition of PROPLOG and ***B*** *is not* enclosed in brackets, then NOT(***B***) is a proposition of PROPLOG.

If ***B*** is a proposition of PROPLOG and ***B*** *is* enclosed in brackets, then NOT***B*** is a proposition of PROPLOG.

If ***C*** and ***D*** are both propositions of PROPLOG, then:

- (***C*** AND ***D***)
- (***C*** OR ***D***)
- (***C*** IFTHEN ***D***)
- (***C*** IFF ***D***)

Are all propositions of PROPLOG.

Nothing else is a proposition of PROPLOG.

Based on this definition of a proposition, we can define a sentence of PROPLOG as follows:

If ***E*** is a proposition of PROPLOG, then:

- #PREMISE ***E***
- #CONCLUDE ***E***
- #ASSUME ***E***
- #DERIVE ***E***

Are all sentences of PROPLOG.

#ENDASSUME is a sentence of PROPLOG.

Nothing else is a sentence of PROPLOG.

D.2. Vocabulary, Syntax, and Semantics of PREDLOG

PREDLOG is to predicate logic what PROPLOG is to propositional logic. As discussed in the introduction to logic, quantifiers are introduced, and the basic unit of the language shifts from being an atomic proposition to an atomic formula.

PREDLOG, like PROPLOG, still contains propositions, keywords, and symbols. It also adds another category: terms. The vocabulary and semantics of PREDLOG is as follows:

Element of the vocabulary with type term:

Type of Term	Word of Vocabulary	Represents
Constant	One lowercase alphabetic letter from a-r inclusive, with or without trailing numbers	Constant
Variable	One lowercase alphabetic letter from s-z inclusive, with or without trailing numbers	Variable

Elements of the vocabulary with type symbol:

Symbol	Represents
(	Open bracket

)	Close bracket
=	Identity symbol

Elements of the vocabulary with type proposition:

Type of Proposition	Word of Vocabulary	Represents
Binary	AND	Conjunction
	OR	Disjunction
	IFTHEN	Conditional
	IFF	Biconditional
Unary	NOT	Negation
Atomic Formula	One capitalised alphabetic letter, with or without trailing numbers, followed by a parenthesised, nonempty set of terms	Predicate-Term(s)
	One term, followed by the identity symbol, followed by another term	Identity formula
	ABSURD	Logical absurdity
Quantifier	FORALL	Universal quantification
	FORSOME	Existential quantification

Note here that the one capitalised letter of the predicate-term combination is referred to as a predicate (e.g. the 'A' in 'A(gyz)'). A predicate alone is not a word of the language.

Finally, the keyword set (this is the same as for PROPLOG):

Word of Vocabulary	Represents
#PREMISE	Assert a premise
#CONCLUDE	Assert a conclusion
#ASSUME	Assert an assumption
#ENDASSUME	Close the outermost assumption
#DERIVE	Assert a new derivation

A slight change in terms is used to define a sentence of PREDLOG, since a variable must be bound by a quantifier. An occurrence of a variable is bound iff that variable falls within the scope of a quantifier, and is free otherwise. The notion of a proposition in PROPLOG is expanded by the notion of a formula in PREDLOG. A *sentence* is still defined as something which is a valid input of the system; a *proposition* is still defined as something which, if paired with a keyword, would constitute a sentence of the language; a *formula* is defined as an expression of the language which is either a proposition, or could be made into a proposition if ranged over by an appropriate quantifier.

Propositions of PREDLOG can now be defined recursively as any formula which contains no free variables and satisfies any of the following conditions:

If ***A*** is an atomic formula and contains no free variables, then ***A*** is a proposition of PREDLOG.

If ***B*** is a proposition of PREDLOG, contains no free variables, and *is not* enclosed in brackets, then NOT(***B***) is a proposition of PREDLOG.

If ***B*** is a proposition of PREDLOG, contains no free variables and *is* enclosed in brackets, then NOT***B*** is a proposition of PREDLOG.

If ***C*** and ***D*** are both propositions of PREDLOG and do not contain free variables, then:

- (***C*** AND ***D***)
- (***C*** OR ***D***)
- (***C*** IFTHEN ***D***)
- (***C*** IFF ***D***)

Are all propositions of PREDLOG.

If ***E*** is a formula of PREDLOG, ***x*** is a variable, ***E*** contains at least one occurrence of ***x***, neither 'FORALL(***x***)' nor '#FORSOME(***x***)' were used in the construction of ***E***, ***E*** contains no free variables, and *is not* enclosed in brackets then:

- FORALL(***x***)(***E***)
- FORSOME(***x***)(***E***)

Are both propositions of PREDLOG.

If ***E*** is a formula of PREDLOG, ***x*** is a variable, ***E*** contains at least one occurrence of ***x***, neither 'FORALL(***x***)' nor '#FORSOME(***x***)' were used in the construction of ***E***, ***E*** contains no free variables, and *is* enclosed in brackets then:

- FORALL(***x***)***E***
- FORSOME(***x***)***E***

Are both propositions of PREDLOG.

Nothing else is a proposition of PROPLOG.

A sentence of PREDLOG can then be defined in exactly the same way as a sentence of PROPLOG:

If ***E*** is a proposition of PREDLOG, then:

- #PREMISE ***E***
- #CONCLUDE ***E***
- #ASSUME ***E***
- #DERIVE ***E***

  Are all sentences of PREDLOG.

#ENDASSUME is a sentence of PREDLOG.

Nothing else is a sentence of PREDLOG.

Appendix E: Heuristic Evaluations of the GUI Prototypes

Heuristic Evaluation of First Prototype. Heuristics supplied by (Nielson, 2017)

Visibility of system – The derivation algorithm should be sufficiently fast to not require information on the system status, as was the case with the other systems evaluated. The system openly displays whether the user is using propositional or first-order logic.

Match between system & real world – At this stage, keyboard shortcuts for formal symbols which physically resembled those symbols were still being considered. Though this idea was scrapped in the final language specification, bridging the gap between input and logic via tooltip was a useful feature. Furthermore, proofs are represented in Fitch-style.

User control & freedom – The user has a lot of control in that everything that they write is editable. If they need to go back to an earlier line and change it, then this is permissible. The standard undo and redo keyboard shortcuts should be implemented. To help the user to structure their proofs, bars are included where the user can enter the premises and conclusion.

Consistency and standards – As mentioned in (2), the language specification at the time would have allowed for multiple keyboard shortcuts to represent the same logical symbol. This is a problem from the point of consistency; whilst possibly increasing usability, this feature was found confusing in the evaluation of related technologies. Also problematic with this design is the inclusion of line numbers in the input area. This could be confusing to the user, because the "#ENDASSUME" keyword occupies a line in the input but does not occupy a line of its own in the proof. Line 1 of the input area is also not going to be the first line in the output area for any proof which has premises entered into the premises bar.

Error prevention – The system reduces the chance of syntax error by providing shortcut buttons to help the user enter their proof.

Recognition rather than recall – In keeping with the recommendations of the evaluation of related systems, buttons are feature to help the novice user to interact with the system, while the experienced user can bypass those by typing directly into the box. However, no buttons are featured here to specify the keywords representing premises, derivations, conclusions, assumptions and closing assumptions. This was a poor choice.

Flexibility and efficiency of use – The user is given full flexibility to type the proof as they see fit without needing to enter it on a line-by-line basis and wait for each line to be evaluated before they continue. For example, if they make a mistake on some line, then they can keep typing into the area without rectifying that mistake, and return to the error when ready. However, if the user wishes to use a derived rule, they must press a button to do so. This could significantly slow down usage, as it seems to defeat the point of the system being able to infer the relevant inference rule.

Aesthetic and minimalist design. Information on how to use the interaction language can be accessed by hovering over the symbol, instead of cluttering up the interface. However, the premises and conclusion bar seem to take an undue amount of space on the screen. They also complicated the importing and exporting of proofs from text files.

Helps users recognise, diagnose and recover from error. Error messages are not displayed here, but the assumption is that the error message would be displayed in the same place as the "Conclusion reached!" notification, printing whatever detailed error message is passed up from the back end.

Help and documentation. Help on how to use the syntax of the language is provided by hovering over the buttons. However, a more comprehensive help document should be made available to the user so that they can see which derivation rules are available.

Heuristic Evaluation of Second Prototype. Heuristics supplied by (Nielson, 2017)

Visibility of system. Again, the status of the system need not be displayed. Which logic is in use is clearly displayed by the tab at the top.

Match between system & real world. The proofs accurately represent Fitch-style proofs, and the language specification has been cleaned up. Furthermore, the system no longer displays the conclusion on line n, as this is not a convention of the proof system, and therefore could cause frustration when the user finds it contained in an exported proof.

User control & freedom. Because the text fields to separately input premises and conclusions have been removed, the user now has complete control over their input within the confines of the convention of the input language. The user no longer needs to move the cursor to a different text box to declare a new premise or a conclusion; they need only type it as any point in the proof. More buttons have been added to help the novice user get started with the system. The trade-off is some degree of usability for novice users.

Consistency and standards. To fix the consistency point, only one keyboard representation of each logical symbol is permitted in the interaction language, and therefore only the syntax of using that code is represented when hovering over a symbol. Line numbers have been removed to handle the discrepancy between line numbers in the input area and in the proof.

Error prevention. More precise guidance about the syntax of the interaction language is displayed when the user hovers over a button. No other error prevention has been added.

Recognition rather than recall. A full suite of relevant buttons has now been added to help a novice user with their input, whilst removing the premise and conclusion entry fields should speed up the usage process for more advanced users.

Flexibility and efficiency of use. The option to use a derived rule has been removed, reflecting that the backend system should automatically be able to infer which derived rule has bene used. No other changes have been made on this front.

Aesthetic and minimalist design. Removing the premise and conclusion bars has decluttered the screen significantly. Extra buttons were added in a way which does not over-complicate the aesthetic.

Helps users recognise, diagnose and recover from error. No change.

Help and documentation. A “Help” button was added to further aid the user by allowing them to view full information about the system (including the interaction language and proof rules).

Appendix F: Test Cases

Num	*Class*	*Approach*	*Property Tested*	*Input*	*Expected Output*	*Result*
Testing AtomicForm, Atomicprop and Proposition classes using JUnit						
JU1	AtomicFormTest	JUnit	Testing generateKey method for predicate-term formula	AtomicForm("A1", {x, y}).generateKey()	((A1)(x)(y))	Pass
JU2	AtomicFormTest	JUnit	Testing generateKey method for predicate-term formula with empty predicate	AtomicForm("", {a, b}).generateKey()	(()(a)(b))	Pass
JU3	AtomicFormTest	JUnit	Testing generateKey method for predicate-term fomrula with empty predicate & empty terms	AtomicForm("", new Term[] ={}).generateKey()	(())	Pass
JU4	AtomicFormTest	JUnit	Testing generateKey method for identity statement between constants	AtomicForm("=", {ab}).generateKey()	((a)=(b))	Pass
JU5	AtomicFormTest	JUnit	Testing generateKey method for ideneity statement between variables	AtomicForm("=", {x, y}).generateKey()	((x)=(y))	Pass
JU6	AtomicFormTest	JUnit	Testing generateKey method for absurd formula passed a normal input on creation	AtomicForm("ABSURD").generateKey()	(ABSURD)	Pass

JU7	AtomicFormTest	JUnit	Testing generateKey method for formula passed a weird input on creation	AtomicForm("CURMUDGEON").generateKey()	(ABSURD)		Pass
JU8	AtomicFormTest	JUnit	Testing getTypeCode method for a predicate	AtomicForm("A1", {x, y}).getTypeCode()		1	Pass
JU9	AtomicFormTest	JUnit	Testing getTypeCode method for a different predicate	AtomicForm("", {a, b}).getTypeCode()		1	Pass
JU10	AtomicFormTest	JUnit	Testing getTypeCode method for a different predicate	AtomicForm("", new Term[] ={}).getTypeCode()		1	Pass
JU11	AtomicFormTest	JUnit	Testing getTypeCode method for a identity	AtomicForm("=", {ab}).getTypeCode()		2	Pass
JU12	AtomicFormTest	JUnit	Testing getTypeCode method for a identity	AtomicForm("=", {x, y}).getTypeCode()		2	Pass
JU13	AtomicFormTest	JUnit	Testing getTypeCode for absurdity	AtomicForm("ABSURD").getTypeCode()		3	Pass
JU14	AtomicFormTest	JUnit	Testing getTypeCode for absurdity	AtomicForm("CURMUDGEON").getTypeCode()		3	Pass
JU15	AtomicFormTest	JUnit	Testing setTypeCode with a bad type code	AtomicForm("A1", {x, y}).setTypeCode(123456789)	Exception thrown.		Pass
JU16	AtomicFormTest	JUnit	Testing setTypeCode with a bad type code	AtomicForm("A1", {x, y}).setTypeCode(0)	Exception thrown.		Pass
JU17	AtomicFormTest	JUnit	Testing setTypeCode with a good type code	AtomicForm("A1", {x, y}).setTypeCode(2)		2	Pass

JU18	AtomicFormTest	JUnit	Testing getPred method for normal predicate	AtomicForm("A1", {x, y}).getPred()	"A1"	Pass
JU19	AtomicFormTest	JUnit	Testing getPred method for empty predicate with terms	AtomicForm("", {a, b}).getPred()	""	Pass
JU20	AtomicFormTest	JUnit	Testing getPred method for empty predicate with no terms	AtomicForm("", new Term[] ={}).getTypeCode()	""	Pass
JU21	AtomicFormTest	JUnit	Testing getPred method for identity	AtomicForm("=", {ab}).getPred()	Null	Pass
JU22	AtomicFormTest	JUnit	Testing getPred method for identity	AtomicForm("=", {x, y}).getPred()	Null	Pass
JU23	AtomicFormTest	JUnit	Testing getPred method for absurdity	AtomicForm("ABSURD").getPred()	Null	Pass
JU24	AtomicFormTest	JUnit	Testing getPred method for absurdity	AtomicForm("CURMUDGEON").getPred()	Null	Pass
JU25	AtomicFormTest	JUnit	Testing setPred by providing an empty predicate	AtomicForm("A1", {x, y}).setPred("")	""	Pass
JU26	AtomicFormTest	JUnit	Testing setPred method by providing a two-word predicate	AtomicForm("A1", {x, y}).setPred("Hello World")	"Hello World"	Pass
JU27	AtomicFormTest	JUnit	Testing setPred method by passing null	AtomicForm("A1", {x, y}).setPred("Null")	Null	Pass
JU28	AtomicFormTest	JUnit	Testing getTerms for a variable array	AtomicForm("A1", {x, y}).getTerms()	{x, y}	Pass
JU29	AtomicFormTest	JUnit	Testing getTerms for a constan array	AtomicForm("", {a, b}).getTerms()	{a, b}	Pass
JU30	AtomicFormTest	JUnit	Testing getTerms for an empty array	AtomicForm("", new Term[] ={}).getTerms()	{}	Pass

JU31	AtomicFormTest	JUnit	Testing setTerms for an array containing multiple variables and constants	AtomicForm("A1", {x, y}).setTerms({a, b, x, y})	{a, b, x, y}	Pass	
JU32	AtomicFormTest	JUnit	Testing setTerms for an empty array	AtomicForm("A1", {x, y}).setTerms({})	{}	Pass	
JU33	AtomicFormTest	JUnit	Testing setTerms for an array containing one term and one constant	AtomicForm("A1", {x, y}).setTerms({a, x})	{a, x}	Pass	
JU34	AtomicFormTest	JUnit	Testing setTerms for null	AtomicForm("A1", {x, y}).setTerms(null)	null	Pass	
JU35	AtomicFormTest	JUnit	Testing getProp for a normal predicate-term formula	AtomicForm("A1", {x, y}).getProp()	"A1xy"	Pass	
JU36	AtomicFormTest	JUnit	Testing getProp for a predicate-term formula with an empty predicate	AtomicForm("", {a, b}).getProp()	"ab"	Pass	
JU37	AtomicFormTest	JUnit	Testing getProp for a predicate-term formula with an empty predicate and empty terms	AtomicForm("", new Term[] ={}).getProp()	""	Pass	
JU38	AtomicFormTest	JUnit	Testing getProp for an identity statement between constants	AtomicForm("=", {ab}).getProp()	"a=b"	Pass	
JU39	AtomicFormTest	JUnit	Testing getProp for an identity statement between variables	AtomicForm("=", {x, y}).getProp()	"x=y"	Pass	
JU40	AtomicFormTest	JUnit	Testing getProp for absurdity	AtomicForm("ABSURD").getProp()	"ABSURD"	Pass	

JU41	AtomicFormTest	JUnit	Testing getProp for absurdity with a strange input	AtomicForm("CURMUDGEON").getProp()	"ABSURD"		Pass
JU42	AtomicFormTest	JUnit	Testing setProp to a long String	AtomicForm("A1", {x, y}).setProp("PROPOSITION")	"PROPOSITION"		Pass
JU43	AtomicFormTest	JUnit	Testing setprop to an empty String	AtomicForm("A1", {x, y}).setProp("")	""		Pass
JU44	AtomicFormTest	JUnit	Testing setProp for a String containing a mix of character types	AtomicForm("A1", {x, y}).setProp("A45!_")	"A45!_"		Pass
JU45	AtomicFormTest	JUnit	Testing netProp with null	AtomicForm("A1", {x, y}).setProp(null)	null		Pass
JU46	AtomicFormTest	JUnit	Testing deepClone produces a copy	originalAtomicForm.equals(AtomicForm.deepClone())		TRUE	Pass
JU47	AtomicFormTest	JUnit	Testing that changes to the copy do not lead to changes to the original	originalAtomicFormula.equals(deepClone.replaceTerm(new Term("n")))		FALSE	Pass
JU48	AtomicFormTest	JUnit	Testing toString for a normal predicate-term formula	AtomicForm("A1", {x, y}).toString()	"A1xy"		Pass
JU49	AtomicFormTest	JUnit	Testing toString for a predicate-term formula with an empty predicate	AtomicForm("", {a, b}).toString()	"ab"		Pass
JU50	AtomicFormTest	JUnit	Testing toString for a predicate-term formula with an empty predicate and empty terms	AtomicForm("", new Term[] ={}).toString()	""		Pass
JU51	AtomicFormTest	JUnit	Testing toString for an identity statement between constants	AtomicForm("=", {ab}).toString()	"a=b"		Pass

JU52	AtomicFormTest	JUnit	Testing toString for an identity statement between variables	AtomicForm("=", {x, y}).toString()	"x=y"	Pass
JU53	AtomicFormTest	JUnit	Testing toString for absurdity	AtomicForm("ABSURD").toString()	"(\u22A5)"	Pass
JU54	AtomicFormTest	JUnit	Testing toString for absurdity with a strange input	AtomicForm("CURMUDGEON").toString()	"(\u22A5)"	Pass
JU55	AtomicPropTest	JUnit	Testing constructor for a normal proposition	new AtomicProp("A")	Pass	Pass
JU56	AtomicPropTest	JUnit	Testing constructor for an indexed propositon	new AtomicProp("B33")	Pass	Pass
JU57	AtomicPropTest	JUnit	Testing constructor for a proposition passed an empty String	new AtomicProp("")	Pass	Pass
JU58	AtomicPropTest	JUnit	Testing constructor for a proposition passed a long name	new AtomicProp("ABCDEFGHIJ")	Pass	Pass
JU59	AtomicPropTest	JUnit	Testing generateKey for a normal proposition	AtomicProp("A").generateKey()	"(A)"	Pass
JU60	AtomicPropTest	JUnit	Testing generateKey for an indexed propositon	AtomicProp("B33").generateKey()	"(B33)"	Pass
JU61	AtomicPropTest	JUnit	Testing generateKey for a proposition passed an empty String	AtomicProp("").generateKey()	"()"	Pass
JU62	AtomicPropTest	JUnit	Testing generateKey for a proposition passed a long name	AtomicProp("ABCDEFGHIJ").generateKey()	"(ABCDEFGHIJ)"	Pass

JU63	AtomicPropTest	JUnit	Testing getKey for a normal proposition	AtomicProp("A").getKey()	"(A)"		Pass
JU64	AtomicPropTest	JUnit	Testing getKey for an indexed propositon	AtomicProp("B33").getKey()	"(B33)"		Pass
JU65	AtomicPropTest	JUnit	Testing getKey for a proposition passed an empty String	AtomicProp("").getKey()	"()"		Pass
JU66	AtomicPropTest	JUnit	Testing getKey for a proposition passed a long name	AtomicProp("ABCDEFGHIJ").getKey()	"(ABCDEFGHIJ)"		Pass
JU67	AtomicPropTest	JUnit	Testing setKey for an integer key	AtomicProp("A").setKey("1")	"1"		Pass
JU68	AtomicPropTest	JUnit	Testing setKey for a different integer key	AtomicProp("B33").setKey("2")	"2"		Pass
JU69	AtomicPropTest	JUnit	Testing setKey for a symbol	AtomicProp("!").setKey("!")	"!"		Pass
JU70	AtomicPropTest	JUnit	Testing setKey for a full word	AtomicProp("ABCDEFGHIJ").setKey("HELLO")	"HELLO"		Pass
JU71	AtomicPropTest	JUnit	Testing absurd check for an absurd propoistion.	AtomicProp("ABSURD").absurdCheck()		TRUE	Pass
JU72	AtomicPropTest	JUnit	Testing absurd check for a non-absurd proposition.	AtomicProp("A").absurdCheck()		FALSE	Pass
JU73	AtomicPropTest	JUnit	Testing the getUnicode method for an absurd proposition	AtomicProp("ABSURD").getUnicode()	"\U22a5"		Pass
JU74	AtomicPropTest	JUnit	Testing the getUnicode method for a non-absurd proposition	AtomicProp("A").getUnicode()	Null		Pass

JU75	AtomicPropTest	JUnit	Testing getProp for a normal proposition	AtomicProp("A").getProp()	"A"		Pass
JU76	AtomicPropTest	JUnit	Testing getProp for an indexed propositon	AtomicProp("B33").getProp()	"B33"		Pass
JU77	AtomicPropTest	JUnit	Testing getProp for a proposition passed an empty String	AtomicProp("").getProp()	""		Pass
JU78	AtomicPropTest	JUnit	Testing getProp for a proposition passed a long name	AtomicProp("ABCDEFGHIJ").getProp()	"ABCDEFGHIJ"		Pass
JU79	AtomicPropTest	JUnit	Testing setProp for a normal proposition	AtomicProp("A").setProp("B")	"B"		Pass
JU80	AtomicPropTest	JUnit	Testing setProp for an indexed propositon	AtomicProp("B33").setProp("A44")	"B33"		Pass
JU81	AtomicPropTest	JUnit	Testing setProp for a mix of letters and symbols	AtomicProp("").setProp("OH_NO")	"OH_NO"		Pass
JU82	AtomicPropTest	JUnit	Testing setProp for a proposition passed a long name	AtomicProp("ABCDEFGHIJ").setProp("QWERTYUIOP")	"QWERTYUIOP"		Pass
JU83	AtomicPropTest	JUnit	Testing deepClone produces a copy	AtomicProp.equals(AtomicProp.deepClone())		TRUE	Pass
JU84	AtomicPropTest	JUnit	Testing that changes to the copy do not lead to changes to the original	AtomicProp.equals(deepClone.setProp(new Term("B")))		TRUE	Pass
JU85	AtomicPropTest	JUnit	Testing toString for a normal proposition	AtomicProp("A").toString()	"A"		Pass
JU86	AtomicPropTest	JUnit	Testing toString for an indexed propositon	AtomicProp("B33").toString()	"B33"		Pass

JU87	AtomicPropTest	JUnit	Testing toString for a proposition passed an empty String	AtomicProp("").toString()	""	Pass
JU88	AtomicPropTest	JUnit	Testing toString for a proposition passed a long name	AtomicProp("ABCDEFGHIJ").toString()	"ABCDEFGHIJ"	Pass
JU89	AtomicPropTest	JUnit	Testing toString for an absurd proposition	AtomicProp("ABSURD").toString()	(\u22A5)	Pass
JU90	Proposition	JUnit	Testing getUnicode for an atomic proposition	(A).getUnicode()	Null	Pass
JU91	Proposition	JUnit	Testing getUnicode for a predicate-term formula	(C(x)).getUnicode()	Null	Pass
JU92	Proposition	JUnit	Testing getUnicode for a conjunction	(A AND NOT(B).getUnicode()	"\u2227"	Pass
JU93	Proposition	JUnit	Testing getUnicode for a disjunction	(B OR C).getUnicode()	"\u2228""	Pass
JU94	Proposition	JUnit	Testing getUnicode for a conditional	(A IFTHEN NOT(B)).getUnicode()	"\u2192"	Pass
JU95	Proposition	JUnit	Testing getUnicode for a biconditional	(A IFF B).getUnicode()	"\u2194"	Pass
JU96	Proposition	JUnit	Testing getUnicode for a negation	NOT(B).getUnicode()	"\u00AC"	Pass
JU97	Proposition	JUnit	Testing getUnicode for a universally quantified proposition	FORALL(x)(A(x)).getUnicode()	"\u2200"	Pass
JU98	Proposition	JUnit	Testing getUnicode for an existentially quantified proposition	FORSOME(x)(B(x)).getUnicode()	"\u2203"	Pass

JU99	Proposition	JUnit	Testing getUnicode for an identity statement	(x=x).getUnicode()	"\u003D"	Pass
JU100	Proposition	JUnit	Testing getUnicode for an absurd AtomicProp	(ABSURD).getUnicode()	"\u22A5"	Pass
JU101	Proposition	JUnit	Testing getUnicode for an absurd AtomicForm	(ABSURD).getUnicode()	"\u22A5"	Pass
JU102	Proposition	JUnit	Testing setUnicode for a normal String	(A AND NOT(B).setUnicode("hello")	"hello"	Pass
JU103	Proposition	JUnit	Testing setUnicode for an empty String	(A AND NOT(B).setUnicode("")	""	Pass
JU104	Proposition	JUnit	Testing setUnicode for unicode	(A AND NOT(B).setUnicode("\u2227")	"\u2227"	Pass
JU105	Proposition	JUnit	Testing getType for an atomic proposition	(A).getType()	"ATOMIC"	Pass
JU106	Proposition	JUnit	Testing getType for a conjunction	(A AND NOT(B).getType()	"AND"	Pass
JU107	Proposition	JUnit	Testing getType for a disjunction	(B OR C).getType()	"OR"	Pass
JU108	Proposition	JUnit	Testing getType for a conditional	(A IFTHEN NOT(B)).getType()	"IFTHEN"	Pass
JU109	Proposition	JUnit	Testing getType for a biconditional	(A IFF B).getType()	"IFF"	Pass
JU110	Proposition	JUnit	Testing getType for a negation	NOT(B).getType()	"NOT"	Pass
JU111	Proposition	JUnit	Testing getType for a universally quantified proposition	FORALL(x)(A(x)).getType()	"FORALL"	Pass
JU112	Proposition	JUnit	Testing getType for an existentially quantified proposition	FORSOME(x)(B(x)).getType()	"FORSOME"	Pass

JU113	Proposition	JUnit	Testing getType for an identity statement	(x=x).getType()	"IDENTITY"	Pass
JU114	Proposition	JUnit	Testing getType for a predicate-term formula	(C(x)).getType()	"PREDICATE"	Pass
JU115	Proposition	JUnit	Testing getType for an absurd AtomicProp	(ABSURD).getType()	"ATOMIC"	Pass
JU116	Proposition	JUnit	Testing getType for an absurd AtomicForm	(ABSURD).getType()	"ATOMIC"	Pass
JU117	Proposition	JUnit	Testing setType for a normal type	(A IFF B).setType("AND")	"AND"	Pass
JU118	Proposition	JUnit	Testing setType for multiple words	(A IFF B).setType("BICONDITIONALS ARE FUN!")	"BICONDITIONALS ARE FUN!"	Pass
JU119	Proposition	JUnit	Testing setType for an empty String	(A IFF B).setType("")	"(A)"	Pass
JU120	Proposition	JUnit	Testing getKey & generateKey for an atomic proposition	(A).generateKey()	"(NOT(B))"	Pass
JU121	Proposition	JUnit	Testing getKey & generateKey for a negation	NOT(B).generateKey()	"((A)AND(NOT(B)))"	Pass
JU122	Proposition	JUnit	Testing getKey & generateKey for a conjunction	(A AND NOT(B).generateKey()	"((A)IFTHEN(NOT(B)))"	Pass
JU123	Proposition	JUnit	Testing getKey & generateKey for a conditional	(A IFTHEN NOT(B)).generateKey()	"((A)IFF(B))"	Pass
JU124	Proposition	JUnit	Testing getKey & generateKey for a biconditional	(A IFF B).generateKey()	"(((A)IFF(B))OR(C))"	Pass
JU125	Proposition	JUnit	Testing getKey& generateKey for a disjunction	(B OR C).generateKey()	"(NOT(NOT((A)AND(((A)IFF(B))OR(C)))))"	Pass

JU126	Proposition	JUnit	Testing getKey & generateKey for an identity statement	(x=x).generateKey()	"((x)=(x))"	Pass
JU127	Proposition	JUnit	Testing getKey & generateKey for a predicate-term formula	(C(x)).generateKey()	"((A)(y))"	Pass
JU128	Proposition	JUnit	Testing setKey method for an empty String	NOT(B).setKey("")	""	Pass
JU129	Proposition	JUnit	Testing setKey method for mix of letter, number and symbol	NOT(B).setKey("OHNO_1")	"OHNO_1"	Pass
JU130	Proposition	JUnit	Testing setKey for a mix of brackets and letter	NOT(B).setKey("((a)")	"((a)"	Pass
JU131	Proposition	JUnit	Testing setKey for a key with an index	A(y).setKey(A(y).getKey() + "1")	"1"	Pass
JU132	Proposition	JUnit	Testing equals method for a Proposition and itself.	(A AND NOT(B)).equals(A AND NOT(B))	TRUE	Pass
JU133	Proposition	JUnit	Testing equals method for a Proposition and a different Proposition	(A AND NOT(B)).equals(A AND (B OR C))	FALSE	Pass
JU134	Proposition	JUnit	Testing equals method for a Proposition and a copy of that Proposition.	(A AND NOT(B)).equals(new BinaryProp("AND", A, NOT(B)))	TRUE	Pass
JU135	Proposition	JUnit	Testing equals method for a Proposition and a copy of itself (not a deepClone) after changing the key.	(A AND NOT(B)).equals(new BinaryProp("AND", A, NOT(B).setKey("bananas"))	TRUE	Pass

Testing Back-end Classes by printing to the command line. The purpose of this set of tests was mostly to check that all constructors work as expected, and to test some basic functions. Ensuring that the correct input is passed to these constructors is the responsibility of the Interpret and Proof classes. Most of these are also tested indirectly in the JUnit testing classes, and all are indirectly tested in the testing the the GUI classes.						
CL1	AbstractSyntaxTree	Print to command line	Construct new tree	Create a new tree with Token root & two new trees for subtrees, each with Token for root	Root: ATOM A Left subtree: (Root: CONNECTIVE AND Left subtree: (Empty) Right subtree: (Empty)) Right subtree: (Root: ATOM B Left subtree: (Empty) Right subtree: (Empty))	Pass
CL2	Assumption	Print to command line	Create new open assumption	new Assumption(new ProofLine(6, "#ASSUME", notAIfThenBOrC, 1, ("\u2192I 4,5")))	Start Line: 6: #ASSUME (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 1 Rule: →I 4,5 Assumption closed: false	Pass
CL3	Assumption	Print to command line	Close assumption w/ line of different assumption depth	closeAssumption(new ProofLine(17, "#DERIVE", notAIfThenBOrC, 2, "Assumption");	Exception thrown. Print: Exception caught. Error message: Error! Cannot close an assumption with a line of a different depth	Pass
CL4	Assumption	Print to command line	Close assumption w/ line of same assumption depth	closeAssumtpion(new ProofLine(17, "#DERIVE", notAIfThenBOrC, 1, "Assumption");	Start Line: 6: #ASSUME (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 1 Rule: →I 4,5 End Line: 17: #DERIVE (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 1 Rule: Assumption Assumption closed: true	Pass
CL5	Assumption	Print to command line	Create new closed assumption	new Assumption(new ProofLine(6, "#ASSUME", notAIfThenBOrC, 1, ("\u2192I 4,5")), new ProofLine(17, "#DERIVE", notAIfThenBOrC, 1, "Assumption"))	Start Line: 6: #ASSUME (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 1 Rule: →I 4,5 End Line: 17: #DERIVE (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 1 Rule: Assumption Assumption closed: true	Pass
CL6	AtomicForm	Print to command line	Create new predicate-term formula w/ variables	new AtomicForm("F", {x,y,z})	Fxyz	Pass
CL7	AtomicForm	Print to comand line	Create new predicate-term formula w/ constants	new AtomicForm("A", {a,c})	Aac	Pass
CL8	AtomicForm	Print to command line	Create new predicate-term formula w/ one term	new AtomicForm("P", {b})	Pb	Pass
CL9	AtomicForm	Print to command line	Create new identity statement	new AtomicForm("=", a, c)	a=c	Pass

CL10	AtomicForm	Print to command line	Create new identity statement	new AtomicForm("=", c, b)	c=b	Pass
CL11	AtomicForm	Print to command line	Create new absurd AtomicForm	new AtomicForm("ABSURD")	(⊥)	Pass
x	AtomicForm	(also tested in JUNit)				
x	AtomicProp	(tested in JUnit)				
CL12	Binary Prop	Print to command line	Create conjunction	new BinaryProp("AND", A, NOT(A))	(B ∧ ¬A)	Pass
CL13	BinaryProp	Print to command line	Create binary prop between a predicate-term formula & an identity statement	new Binaryprop("AND", A1xy, a=b)	(A1xy ∧ a=b)	Pass
CL14	BinaryProp	Print to command line	Create binary prop between a complex proposition and absurdity	new BinaryProp("AND", test13Input, absurdity)	((A1xy ∧ a=b) ∧ (⊥))	Pass
CL15	BinaryProp	Print to command line	Create a conditional	new BinaryProp("IFTHEN", A, NOT(A))	(A → ¬A)	Pass
CL16	BinaryProp	Print to command line	Create a biconditional	new BinaryProp("IFF", A, B)	(A ↔ B)	Pass
CL17	BinaryProp	Print to command line	Create a complex disjunction	new BinaryProp("OR", test16Input, C)	((A ↔ B) ∨ C)	Pass
CL18	BinaryProp	Print to command line	Create a simple disjunction	new BinaryProp("OR", B, C)	(B ∨ C)	Pass
CL19	BinaryProp	Print to command line	Create a more complex conjunction	new BinaryProp("AND", A, test17Input)	(A ∧ ((A ↔ B) ∨ C))	Pass
x	GUIController	(Tested with GUI)				
x	GUIView	(Tested with GUI)				
x	Interpreter	(Tested with GUI)				

x	Proof	(Tested with GUI)				
CL20	ProofLine	Print to command line	Create a new ProofLine	new ProofLine(1, "#PREMISE", notAIfThenBOrC, 0, ("\u2192I 4,5"));	1: #PREMISE (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 0 Rule: →I 4,5	Pass
CL21	ProofLine	Print to command line	Increment all line references in its rule	changeRuleNumbers(true)	1: #PREMISE (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 0 Rule: →I 3,4	Pass
CL22	ProofLine	Print to command line	Decrement all line references in its rule	changeRuleNumbers(false)	1: #PREMISE (¬A → (B ∨ C)) KEY = ((NOT(A))IFTHEN((B)OR(C))) Assumption depth: 0 Rule: →I 4,5	Pass
CL23	Proofline	Print to command line	Print method with assumption depth 0	test20Input.print()	1 |(¬A → (B ∨ C)) →I 4,5	Pass
CL24	Proofline	Print to command line	Print method with assumption depth 1	new ProofLine(2, "#ASSUME", notAIfThenBOrC, 1, "Assumption").print()	2 | |(¬A → (B ∨ C)) Assumption	Pass
x	Proposition	(Tested in JUnit)				
CL25	QuantProp	Print to command line	Create a complex existential proposition containing a universal proposition	new QuantProp("FORSOME", x, (A(x) AND new QuantProp("FORALL", y, (A(y) IFTHEN x=y)))	∃x((Ax ∧ ∀y((Ay → x=y))))	Pass
CL26	QuntProp	Print to command line	Check key of test25	test25Input.generateKey()	((FORSOME(x)(((A)(x))AND((FORALL(y)(((A)(y))IFTHEN((x)=(y))))))	Pass
CL27	Term	Print to command Line	Create a new constant term	new Term("a")	Type: CONSTANT Term: a	Pass
CL28	Term	Print to command Line	Create a deepClone of that term	Change the value of the original term & print a comparison of the two	A: b aBackup: a	Pass
CL29	Term	Print to command Line	Create a new variable term	new Term("x")	Type: VARIABLE Term: x	Pass
CL30	Token	Print to command Line	Create a simple Token	new Token("ATOM", "A", 0)	Type: ATOM Lexeme: A index: 0	Pass
CL31	Token	Print to command Line	Create a simple Token	new Token("CONNECTIVE", "AND", 1)	Type: CONNECTIVE Lexeme: AND index: 1	Pass

CL32	Token	Print to command Line	Create a simple Token	new Token("ATOM","B", 2)	Type: ATOM Lexeme: B index: 2	Pass
CL33	UnaryProp	Print to command Line	Create a minorly complex UnaryProp	new UnaryProp("NOT", (A AND B))	¬(A ∧ B)	Pass
CL34	UnaryProp	Print to command Line	Create a moderately complex UnaryProp	new UnaryProp("NOT", (test33Input OR FORALL(x)(A(x))))	¬(¬(A ∧ B) ∨ ∀x(Ax))	Pass
Testing that once GUI has loaded, all components are present and perform the correct function.						
GUI1	GUIView&GUIController	Press on Tab	Changing logic tab from propositional logic to predicate logic makes the predicate logic symbols visible.	Predicate Logic Tab pressed	Predicate logic buttons appear	Pass
GUI2	GUIView&GUIController	Press on Tab	Changing logic tab from predicate logic to propositional logic makes the predicate logic symbols invisible.	Propositional Logic Tab pressed	Predicate logic buttons dissapear	Pass
GUI3	GUIView&GUIController	Press Button	Insert "#PREMISE" at the end of the input area.	Premise Button pressed	"#PREMISE" appended to end of input field.	Pass
GUI4	GUIView&GUIController	Press Button	Insert "#ASSUME" at the end of the input area.	Assume Button pressed	"#PREMISE" appended to end of input field.	Pass
GUI5	GUIView&GUIController	Press Button	Insert "#DERIVE" at the end of the input area.	Derive Button pressed	"#PREMISE" appended to end of input field.	Pass
GUI6	GUIView&GUIController	Press Button	Insert "#ENDASSUME" at the end of the input area.	End Assumption Button pressed	"#PREMISE" appended to end of input field.	Pass
GUI7	GUIView&GUIController	Press Button	Insert "#CONCLUDE"	Conclude Button pressed	"#PREMISE" appended to end of input field.	Pass

			text at the end of the input area.			
GUI8	GUIView&GUIController	Press Button	Insert " AND " at the cursor position	Conjunction Button pressed	" AND " inserted at the cursor position	Pass
GUI9	GUIView&GUIController	Press Button	Insert " OR " at the cursor position	Disjunction Button pressed	" OR " inserted at the cursor position	Pass
GUI10	GUIView&GUIController	Press Button	Insert " IFTHEN " at the cursor position	Conditional Button pressed	" IFTHEN " inserted at the cursor position	Pass
GUI11	GUIView&GUIController	Press Button	Insert " IFF " at the cursor position	Biconditional Button pressed	" IFF " inserted at the cursor position	Pass
GUI12	GUIView&GUIController	Press Button	Insert "NOT(" at the cursor position	Negation Button pressed	"NOT(" inserted at the cursor position	Pass
GUI13	GUIView&GUIController	Press Button	Insert "FORALL(" at the cursor position	Universal Quantifier Button pressed	"FORALL(" inserted at the cursor position	Pass
GUI14	GUIView&GUIController	Press Button	Insert "FORSOME(" at the cursor position	Existential Quantifier Button pressed	"FORSOME(" inserted at the cursor position	Pass
GUI15	GUIView&GUIController	Press Button	Insert "ABSURD" at the cursor position	Absurdity Button pressed	"ABSURD" inserted at the cursor position	Pass
GUI16	GUIView&GUIController	Press Button	Insert "=" at the cursor position	Identity Button pressed	"=" inserted at the cursor position	Pass
GUI17	GUIView&GUIController	Press Button	Save the contents of an empty input area	Save Proof pressed while input field is empty.	New empty text file created at the location specified in the FileChooser	Pass
GUI18	GUIView&GUIController	Press Button	Save the text of a nonempty input area	Save Proof pressed while input field is nonempty	New nonempty text file created at the location specified in the FileChooser	Pass
GUI19	GUIView&GUIController	Press Button	Load an empty text file to the input area	Load Proof pressed and an empty text file is selected	Empty text file loaded to the input area via FileChooser (input area now empty)	Pass
GUI20	GUIView&GUIController	Press Button	Load a nonempty file to the input area	Load Proof pressed and a nonempty text file is selected	Nonempty text file loaded to the input area via file chooser	Pass

GUI21	GUIView&GUIController	Press Button	Print an empty output area (to PDF)	Print Proof while the output area is empty	Empty PDF has been created.	Fail
GUI22	GUIView&GUIController	Press Button	Prin a (nonempty) empty file (to PDF)	Print Proof while the output area is nonempty	PDF displaying a simple proof created.	Pass
Testing Interpreter and Proof System via interaction with the GUI.						
Proof 1: PROPLOG	All	Input to & Output from GUI	Conjuntion introduction, conjunction elimination & reiteration in PROPLOG	#PREMISE (A)	Add to proof. Rule: premise	Pass
				#PREMISE (B)	Add to proof. Rule: premise	Pass
				#PREMISE (C AND D)	Add to proof. Rule: premise	Pass
				#DERIVE (A AND B)	Add to proof. Rule: conjunction introduction 1, 2	Pass
				#DERIVE A	Add to proof. Rule: reiteration 1	Pass
				#DERIVE B	Add to proof. Rule: reiteration 2	Pass
				#DERIVE C	Add to proof. Rule: conjunction elimination 3	Pass
				#DERIVE D	Add to proof. Rule: conjunction elimination 3	Pass
Proof 1: PREDLOG	All	Input to & Output from GUI	Conjuntion introduction, conjunction elimination & reiteration in PREDLOG.	Same inputs and outputs, but replacing A with A(a), B with b=e, C with C(cf) and D with D(d)		Pass
Proof 2: PROPLOG	All	Input to & Output from GUI	Disjunction introduction, disjunction elimination, blank lines, inline comments & multi-line comments in PROPLOG		Ignore. No output.	Pass
				/*	Ignore. No output.	Pass
				Disjunction introduction and elimination.	Ignore. No output.	Pass
				*/	Ignore. No output.	Pass
					Ignore. No output.	Pass
				// Premises:	Ignore. No output.	Pass
				#PREMISE (A AND (B OR C))	Add to proof. Rule: premise	Pass
					Ignore. No output.	Pass

				// Conclusion:	Ignore. No output.	Pass
				#CONCLUDE ((A AND B) OR (B AND C))	Store conclusion. No change to output.	Pass
					Ignore. No change to output.	Pass
				// Proof:	Ignore. No change to output.	Pass
				#DERIVE A	Add to proof. Rule: conjunction elimination 1	Pass
				#DERIVE (B OR C)	Add to proof. Rule: conjunction elimination 1	Pass
					Ignore. No change to output.	Pass
				#ASSUME B	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE (A AND B)	Add to proof, indented one place. Rule: conjunction introduction 2, 4	Pass
				#DERIVE ((A AND B) OR (A AND C))	Add to proof, indented one place. Rule: disjunction introduction 5.	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
					Ignore. No change to output.	Pass
				#ASSUME C	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE (A AND C)	Add to proof. Rule: conjunction introduction 2, 7	Pass
				#DERIVE ((A AND B) OR (A AND C))	Add to proof. Rule: disjunction introduction 8.	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
					Ignore. No change to output.	Pass
				#DERIVE ((A AND B) OR (A AND C))	Add to proof. Rule: disjunction elimination 3, 4-6, 7-9. Output that conclusion has been reached.	Pass
Proof 2: PREDLOG	All	Input to & Output from GUI	Disjunction introduction, disjunction elimination, blank lines, inline comments & multi-line comments in PREDLOG	Same inputs and outputs, but replacing A with A(a), B with b=e, C with C(cf) and D with D(d)		Pass
Proof 3: PROPLOG	All	Input to & Output from GUI	Conditional introduction, conditional elmination &	// Setting up premises	Ignore. No output.	Pass
				#PREMISE (P IFTHEN Q)	Add to proof. Rule: premise	Pass
				#PREMISE (Q IFTHEN R)	Add to top of proof. Rule: premise	Pass

			inline commenting in PROPLOG	// Setting up conclusion	Ignore. No change to output.	Pass
				#CONCLUDE (P IFTHEN R)	Store conclusion. No change to output.	Pass
				// Derivation	Ignore. No change to output.	Pass
				#ASSUME P	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE Q	Add to proof. Rule: conditional elimination 2,3	Pass
				#DERIVE R	Add to proof. Rule: conditional elimination 1, 4	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#DERIVE (P IFTHEN R)	Add to proof, no indent. Rule: conditional introduction. Output that conclision has been reached	Pass
Proof 3: PREDLOG	All	Input to & Output from GUI	Conditional introduction, conditional elmination & inline commenting in PREDLOG	Same inputs and outputs but replacing A with A(bcdef), B with b=c and C with c=d		Pass
Proof 4: PROPLOG	All	Input to & Output from GUI	Syntax error with wrong logic, biconditional introduction, biconditional elimination in PROPLOG	Set logic to PREDLOG	PREDLOG buttons become visible	Pass
				#PREMISE (S IFF T)	Syntax error message displayed.	Pass
				Set logic to PROPLOG	PREDLOG buttons become invisible	Pass
				#PREMISE (A IFF B)	Add to proof. Rule: premise	Pass
				#CONCLUDE (A IFF (B OR A))	Store conclusion. No change to output.	Pass
				#ASSUME A	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE B	Add to proof. Rule: biconditional elimination 1, 2	Pass
				#DERIVE (B OR A)	Add to proof. Rule: Disjunction introduction 3.	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#ASSUME (B OR A)	Add to proof, indented one place, Rule: assumption	Pass
				#ASSUME B	Add to proof, indented two places. Rule: assumption	Pass
				#DERIVE A	Add to proof. Biconditional elimination 1, 6	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass

				#ASSUME A	Add to proof, intended two place. Rule: assumption.	Pass
				#DERIVE A	Add to proof. Rule: reiteration 8	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#DERIVE A	Add to proof. Rule: disjunction elimination 5, 6-7, 8-9	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#DERIVE (A IFF (B OR A))	Add to proof. Rule: biconditional introduction 2-4, 5-10. Output that conclusion has been reached.	Pass
				Set logic to PREDLOG & compile	Syntax error message displayed. Proof disappears.	Pass
Proof 4: PREDLOG	All	Input to & Output from GUI	Biconditional introduction & biconditional elimination in PREDLOG	Same inputs and outputs but replacing A with a=b & B with b=c		Pass
Proof 5: PROPLOG	All	Input to & Output from GUI	Negation introduction & elimination in PROPLOG	#PREMISE D	Add to proof. Rule: premise	Pass
				#ASSUME NOT(D)	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE ABSURD	Add to proof. Rule: negation elimination 2, 1	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#DERIVE NOT(NOT(D))	Add to proof. Rule: negation introduction 2-3	Pass
Proof 5: PREDLOG	All	Input to & Output from GUI	Negation introduction & elimination in PREDLOG	Same inputs and outputs but replacing A with A(f)		Pass
Proof 6: PROPLOG	All	Input to & Output from GUI	Explosion in PROPLOG	#PREMISE B	Add to proof. Rule: premise	Pass
				#CONCLUDE (NOT(B) IFTHEN A)	Store conclusion. No change to output.	Pass
				#ASSUME NOT(B)	Add to proof, indented one place. Rule: assumption	Pass
				#DERIVE ABSURD	Add to proof. Rule: negation elimination 2, 1	Pass
				#DERIVE A	Add to proof. Rule: explosion 3.	Pass
				#ENDASSUME	Decrement recorded assumption depth. No change to output.	Pass
				#DERIVE (NOT(B) IFTHEN A)	Add to proof. Rule: conditional introduction 2-4. Output that conclusion has been reached.	Pass

Proof 6: PREDLOG	All	Input to & Output from GUI	Explosion in PREDLOG	Same inputs and outputs but replacing A with A(b) and B with B(a)		Pass
Proof 7: PROPLOG	All	Input to & Output from GUI	Tertium non datur in PROPLOG	#PREMISE (A IFTHEN (NOT A)) #ASSUME A #DERIVE (NOT(A)) #ENDASSUME #ASSUME NOT(A) #DERIVE NOT(A) #ENDASSUME #DERIVE NOT(A)	Add to proof. Rule: premise Add to proof, indented one place. Rule: assumption Add to proof. Rule: conditional elimination 1, 2 Decrement recorded assumption depth. No change to output. Add to proof. Indented one place. Rule: assumption Add to proof. Rule: reiteration 4 Decrement recorded assumption depth. No change to output. Add to proof. Rule: tertium non datur 4-5, 2-3	Pass Pass Pass Pass Pass Pass Pass Pass
Proof 7: PREDLOG	All	Input to and Output from GUI	Tertium non datur in PREDLOG	Same inputs and outputs but replacing A with a = a		Pass
Proof 8: PROPLOG	All	Input to and Output from GUI	Double negation elimination in PROPLOG	#PREMISE NOT(NOT(A)) #DERIVE A	Add to proof. Rule: premise Add to proof. Rule: double negation elimination 1	Pass Pass
Proof 8: PREDLOG	All	Input to and Output from GUI	Double negation elimination in PREDLOG	Same inputs and outputs but replacing A with A(abcdefghijklmnop)		Pass
Proof 9: PROPLOG	All	Input to and Output from GUI	Disjunctive syllogism in PROPLOG	#PREMISE (A OR (B AND C)) #PREMISE (D AND E) #PREMISE (E IFTHEN NOT(A)) #DERIVE E #DERIVE NOT(A) #DERIVE (B AND C)	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to top of proof. Rule: premise Add to proof. Rule: conjunction elmination 2 Add to proof. Rule: conditional elimination 1,4 Add to proof. Rule: disjunctive syllogism 3,5	Pass Pass Pass Pass Pass Pass
Proof 9: PREDLOG	All	Input to and Output from GUI	Disjunctive syllogism in PREDLOG	Same inputs and outputs but replacing A with a=b, B with b=c, C with c=d, D with d=e and E with e=f		Pass
Proof 10: PROPLOG	All	Input to and	Modus Tollens in PROPLOG	#PREMISE (A IFTHEN B) #PREMISE NOT(B)	Add to proof. Rule: premise Add to top of proof. Rule: premise	Pass Pass

		Output from GUI		#DERIVE NOT(A)	Add to proof. Rule: Modus Tollens 2, 1	Pass
Proof 10: PREDLOG	All	Input to and Output from GUI	Modus Tollens in PREDLOG	Same inputs and outputs but replacing A with A(f) and B with B(f)		Pass
Proof 11: PROPLOG	All	Input to and Output from GUI	DeMorgan's Rule One & DeMorgan's Rule Three in PROPLOG	#PREMISE NOT(A OR B) #PREMISE NOT(A AND B) #DERIVE(NOT(A) AND NOT(B)) #DERIVE ((NOT(A) OR NOT(B)))	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to proof. Rule: DeMorgan's on 2 Add to proof. Rule: DeMorgan's on 1	Pass Pass Pass Pass
Proof 11: PREDLOG	All	Input to and Output from GUI	DeMorgan's Rule One & DeMorgan's Rule Three in PREDLOG	Same inputs and outputs but replacing A with A(f) and B with B(f)		Pass
Proof 12: PROPLOG	All	Input to and Output from GUI	DeMorgan's Rule Two & DeMorgan's Rule Four in PROPLOG	#PREMISE(NOT(A) AND NOT(B)) #PREMISE((NOT(A) OR NOT(B))) #DERIVE NOT(A OR B) #DERIVE NOT(A AND B)	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to proof. Rule: DeMorgan's on 2 Add to proof. Rule: DeMorgan's on 1	Pass Pass Pass Pass
Proof 12: PREDLOG	All	Input to and Output from GUI	DeMorgan's Rule Two & DeMorgan's Rule Four in PREDLOG	Same inputs and outputs but replacing A with A(f) and B with B(f)		Pass
Proof 13: PREDLOG	All	Input to and Output from GUI	Universal Introduction & Universal Elimination in PREDLOG	#PREMISE FORALL(x)(A(x)) #DERIVE A(c) #DERIVE FORALL(y)(A(y))	Add to proof. Rule: premise. Add to proof. Rule: universal elimination 1 Add to proof. Rule: universal introduction 2	Pass Pass Pass
Proof 13: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong logic	Set logic to PROPLOG and recompile proof 13	Syntax error on FORALL	Pass
Proof 14: PREDLOG	All	Input to and Output from GUI	Existential Introduction in PREDLOG	#PREMISE R(aad) #DERIVE FORSOME(x)(R(aax)) #DERIVE FORSOME(x)(R(xxd)) #DERIVE FORSOME(x)(R(xad)) #DERIVE FORSOME(y)(FORSOME(x)(R(xyd)))	Add to proof. Rule: premise Add to proof. Rule: existential introduction 1 Add to proof. Rule: existential introduction 1 Add to proof. Rule: existential introduction 1 Add to proof. Rule: existential introduction 4	Pass Pass Pass Pass Pass

Proof 14: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong logic	Set logic to PROPLOG and recompile proof 14	Syntax error on aad	Pass
Proof 15: PREDLOG	All	Input to and Output from GUI	Existential Elimination in PREDLOG	#PREMISE FORALL(x)(F(x) IFTHEN G(x)) #PREMISE FORSOME(x)(F(x)) #ASSUME F(o) #DERIVE (F(o) IFTHEN G(o)) #DERIVE G(o) #DERIVE FORSOME(x)(G(x)) #ENDASSUME #DERIVE FORSOME(x)(G(x))	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to proof, indented one place. Rule: assumption Add to proof. Rule: univeral elimination 2 Add to proof. Rule: conditional elimination 4,3 Add to proof. Rule: existential introduction 5 Decrement recorded assumption depth. No change to output. Add to proof. Rule: existential elimination 1, 3-6	Pass Pass Pass Pass Pass Pass Pass Pass
Proof 15: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong logic	Set logic to PROPLOG and recompile proof 15	Syntax error on FORALL	Pass
Proof 16: PREDLOG	All	Input to and Output from GUI	Identity introduction in PREDLOG	#DERIVE a=a	Add to proof. Rule: identity introduction	Pass
Proof 16: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong language	Set logic to PROPLOG and recompile proof 16	Syntax error on a=a	Pass
Proof 17: PREDLOG	All	Input to and Output from GUI	Identity elimination with PREDLOG	#PREMISE((a = b) AND (b = c)) #DERIVE (a = b) #DERIVE (b = c) #DERIVE (a = c)	Add to proof. Rule: premise Add to proof. Rule: conjunction elimination 1 Add to proof. Rule: conjunction elimination 1 Add to proof. Rule: identity elimination 2, 3	Pass Pass Pass Pass
Proof 17: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong language	Set logic to PROPLOG and recompile proof 17	Syntax error on =	Pass
Proof 18: PREDLOG	All	Input to and Output from GUI	Conversion of Quantifiers One & Conversion of	#PREMISE FORALL(x)(NOT(A(x))) #PREMISE FORSOME(x)(NOT(A(x))) #DERIVE NOT(FORALL(x)(A(x)))	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to proof. Rule: conversion of quantifiers 1	Pass Pass Pass

			Quantifiers Three	#DERIVE NOT(FORSOME(x)(A(x)))	Add to proof. Rule: conversion of quantifiers 2	Pass
Proof 18: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong language	Set logic to PROPLOG and recompile proof 18	Syntax error on FORALL	Pass
Proof 19:	All	Input to and Output from GUI	Conversion of Quantifiers Two & Conversion of Quantifiers Four	#PREMISE NOT(FORSOME(x)(A(x))) #PREMISE NOT(FORALL(x)(A(x))) #DERIVE FORSOME(x)(NOT(A(x))) #DERIVE FORALL(x)(NOT(A(x)))	Add to proof. Rule: premise Add to top of proof. Rule: premise Add to proof. Rule: conversion of quantifiers 1 Add to proof. Rule: conversion of quantifiers 2	Pass Pass Pass Pass
Proof 19: PROPLOG	All	Input to and Output from GUI	Syntax error with wrong language	Set logic to PROPLOG and recompile proof 19		Pass