Ian Hocking 11,000 words

Baker Street: A Natural Deduction Tool

by Ian Hocking

Abstract

Baker Street is an application for supporting proof production in propositional logic, primarily by students. The writing of proofs is seen as appropriate for computer support, given that it can be error prone and taxing. The app was developed in the modern C-family language Swift 5.2.5 for Apple Macintosh computers using an incremental software engineering approach. It runs natively on macOS 10.15 or above. For the original brief, the aims were (i) to validate and constructively comment on a student's proof and (ii) to guide the student towards a solution using hints. The finished app was published to the Mac App Store and its source code released on GitHub under an MIT Licence.

Author's Note

Thanks to Olaf Chitl for his cheerful and prompt support throughout the project. This dissertation uses first person throughout to avoid awkward phrasing. The referencing system is American Psychological Association (APA, 2020). Section titles in the text are hyperlinked. Baker Street can be downloaded on the Mac App Store and its source code is available on GitHub.

Table of Contents

1 Introduction and Problem Statement

1.1 Context

This project was originally selected from a list provided by project supervisors. The original proposal can be read here: Appendix: 8.1 Original Brief. The proposal indicates that Java should be used. However, it was agreed early on that I would create a native macOS application (for a discussion of this, see Section 4.2 Swift).

Key terms in this introduction, such as 'proof', will be defined in Chapter 2 Literature Review.

1.2 Problem Area

Several proof methods are taught in module CO884, as well as elsewhere. A proof method typically involves pencil and paper. It can be labour intensive, slow, error prone, and difficult to manage. A computer-assisted method would keep the experience of proof-production simple while providing support in key ways. Current computer solutions have steep learning curves, suboptimal user interfaces, or address multiple domains beyond the scope of the brief. Baker Street avoids this by taking each formula in a user-supplied proof and checking it for correctness. If the proof is correct, the Baker Street indicates that this is the case. If not, contextualised errors guide a student towards a solution.

1.3 Users

Baker Street is primarily aimed at students. There is some overlap between student use and teacher/researcher use. For instance, both might wish to export the proof in different formats. However, the student is the target user, so requirements like simplicity and rich (i.e. decomposable) errors will be prioritised (see 6.3.4.2 User Testing).

1.4 Methodology

The software engineering approach taken in this dissertation is incremental. Because only one developer was involved, there was little need for the overhead of formal Agile methods, but these were adopted informally where necessary (see Sections 4.4.1 Software Engineering Approach, 6.3.2 Planning vs. Doing and 6.3.4 Testing).

1.5 Motivation

This project was chosen for several reasons. I enjoyed writing natural deduction proofs in CO884 but often found them difficult; upon seeing a correct solution, my mistakes were often oversights or simple transcription errors that could be reduced by computer support. Secondly, while Java was suggested as the language, the Java UI experience on macOS has some room for improvement, as do similar web tools for natural deduction. A native macOS app would perform the task appropriately, and allow me to use the modern C-family language Swift, as well as macOS APIs.

1.6 Structure of this Dissertation

The dissertation will begin with Chapter 2 Literature Review, focusing on the background work I completed prior to development. I'll then turn to my feature and user interface decisions in Chapter 3 Design. In Chapter 4 Implementation, I will look at the process of development itself, including major increments and some chosen points of interest, including features, architecture and tests. I will then describe the app in Chapter 5 Walkthrough. I will evaluate in the app's performance against the brief in Chapter 6 Evaluation, while providing some more reflective commentary on the development process as a whole. I will end with Chapter 7 Conclusion and Future Work.

2 Literature Review

2.1 Overview

This chapter surveys some literature relating to natural deduction and existing programs designed to support proof production.

2.2 Natural Deduction Proofs

Natural deduction is a formal proof method that uses rules to establish the correctness of a theorem in logic. By theorem we mean any general proposition, and by logic we mean the formal study of argument (Nissanke, 1999; Logic, n.d.).

Natural deduction is widely taught across a range of disciplines, not just computer science. As a system, it relies on *assumptions* and so-called *rules of inference* that allow conclusions given a particular prior context. The context comprises theorems and formulae. A formula is an expression in a logical language; to an approximation, the language might have values (e.g. *true* or *false*) and connectives (e.g. *and*) to show how the values are related. Placeholders like p can mean 'all values of p'. Any formula combines the connectives and values in a particular manner. When the formula is evaluated with values substituted for placeholders, we are left with one value.

If our theorem proposes that when we have (1) p and q (this is the formula $p \wedge q$) we must then have (2) q and p (formula $q \wedge p$), we can try to prove this using assumptions and rules of inference.

Figure 1 shows this being tested. The top line is the main theorem (where the syntactic *turnstile* symbol, $\vdash$, means that one formula entails the other). Line 1 is an assumption. We can claim this is correct because it is claimed in the theorem. Lines 2, 3 are derivable from Line 1 by removing 'AND' using the inference rule 'AND Elimination'. Now we know these are also correct. Given q, p in Lines 2, 3, we are now in a position to apply another inference rule, 'AND Introduction', to supply our connective 'AND', deriving Line 4. Because we have made no mistakes in our application of the inference rules, our proof must be correct. Thus the theorem is

correct, and we have proved it using natural deduction.

$$p \wedge q \vdash q \wedge p$$

1	$p \wedge q$	ass 0
2	q	$\wedge$-E 1
3	p	$\wedge$-E 1
4	$q \wedge p$	$\wedge$-I 2, 3

Figure 1: A Natural Deduction Proof

2.3 Why Students Find This Challenging

For students, manual errors can be hard to identify, and will make the proof fail. They must know all inference rules; each rule can have multiple antecedents. They must understand what they can and cannot assume. Further, while the inference rules are deterministic, their combinations are many. To take the analogy of chess, an expert might see an imminent checkmate when a novice does not, despite the novice knowing how all chess pieces may move. And, like chess, the natural deduction process is creative. Expert natural deduction solvers will often work a proof from both directions; that is, they will take a guess about what the final inference might need to be, while continuing from the beginning, to see if they can make the proof 'meet in the middle'. This helps them avoid the blind alleys that can make the process frustrating for beginners.

2.4 Computers Can Help

There is good evidence to show that computers are helpful in the learning of structured, symbolic disciplines such as mathematics and logic (Hahn & Bussell, 2012). For instance, a qualitative study by Kaur, Koval and Chaney (2017) indicated that the use of iPads fostered conceptual understanding of numbers, order or

operations, expressions, as well as multiplication and division skills. One of the reasons is the reduction in working memory load (i.e. the mental information that must be maintained moment-to-moment in order to complete a task; sometimes called 'cognitive load').

Many papers exist on teaching natural deduction. Van Ditmarsch (1998), for instance, suggests several issues that need to be addressed when comparing how natural deduction can be supported by computers: the version of natural deduction, proof visualisation forms (as trees or sequences), forward (assumption driven) and backward reasoning (goal driven), proof debugging facilities, hints, and proof checking. Hints can be *global*, i.e. non-contextualised information about how to apply an inference rule, *tactical*, e.g. 'try to use inference rule X', or *strategic*, i.e. any help based on a proof plan (informed by a proof solution). Van Ditmarsch makes the point that we cannot be sure how much hints actually improve student learning. Such hints may not be correct—that is, they could lead the student up the garden path—or they might make the student lazy by reducing the depth of understanding needed. He notes that (as of 1998) tactical and strategic help is rare in computer-assisted natural deduction.

2.5 Other Natural Deduction Programs

Thus, just as a chess computer can help a novice improve their game, a program should be able to support students learning natural deduction. One tool that has been developed is HAND, the Helpful Assistant for Natural Deduction (an MSc project application made available via the CO884 Moodle). This tool is an excellent one in the context of the brief, particularly for confirming a proof that a student has produced offline. However, it does have some drawbacks. Its interface requires many user decisions and these are often *modal* (i.e. they halt the main program until the user interacts with an element; McConnell, 2004). Students can enter proof assertions in any order, but this is discouraged. Files are saved in a HAND-specific format, which makes portability a problem. HAND is, however, clear with its criticism of the student's proof. In other words, it is helpful in regards to pointing out what a user has done wrong. However, it is less useful in providing positive feedback, such indicating what a next helpful step would be to take in the proof. My systematic analysis of the HAND user interaction paradigm is included in Appendix: 8.3 HAND

- in depth. Other natural deductions programs work at a general level, requiring a substantial amount of coaching to begin using them (e.g. Lévy, 2019; Natural Deduction Proof Checker and Editor, n.d.; Nayuki, 2018; Lemonde-Labrecque, 2020; Schwarz, 2020), often give abstruse or cryptic feedback, and tend have a non-native user experience (i.e. compared to native apps they tend to be slower and less responsive, may require a browser or virtual machine to run, lack native device functionality, have offline storage and security issues, or have fewer interactive elements and less intuitive interfaces). One solution would be to provide a native application experience with tailored errors.

2.6 Concluding Comments

In this chapter, I have outlined natural deduction proofs, emphasising that they comprise well-connected assumptions and inferences that can be checked by computer. I also considered why students find them challenging. To support students, the application should 'get out of the way', i.e. limit interruption, and present the proof in a context that reduces working memory constraints. There are several natural deduction programs available, including the home-grown HAND, but they tend towards interrupting, complex or otherwise sub-optimal user experiences.

To address these criticisms, Baker Street will emphasise simplicity, free interaction, and provide targeted, contextual help.

3 Design

3.1 Overview

This chapter will outline the design considerations I derived from the literature review and brief, the functional and nonfunctional requirements that stem from them, and the full list of features.

3.2 Design Considerations

When elaborating the requirements, my first principle was *simplicity*. This addresses the sometimes overwhelming experience of finding a proof (see Section 2.3 Why Students Find This Challenging). This led to several immediate decisions:

- having the student write their proof in a lightweight markdown language (rather than the app supplying an extra interface element to enter glyphs for logical operators, etc.) (cf. Gruber, 2004)
- using a side panel to provided tailored hints
- using integer line numbers (rather than floating point line numbers that encode subproof/scope level, e.g. 1.2.1)

Simplicity suggested partitioning streams of information so that specific areas of the interface were associated with a given information type (e.g. part of the window would only be used for line numbers). It also suggested providing incorrect proof feedback in a simple, more general way at first, which could be expanded upon demand.

The second principle was the metaphor of a physical working space, stemming from the brief's suggestion that 'the user experience should be similar to writing a proof on paper'. This suggested keeping the application's main window focused on the proof itself; other 'documentation' sources would be available as satellites (i.e. descriptions of inference rules, the markdown language used for entering the proof, and definitions of key terms).

3.2.1 Functional Requirements

In Tables 1 and 2 below, we can see a list of initial functional requirements (i.e. as they were determined at the beginning of development). Although this section is not intended to show the development process in full, I've indicated where the final app deviates significantly from these original requirements. I've avoided some repetition by not giving full examples of the features here; a fuller tour of the app is given in Chapter 5 Walkthrough.

Requirement	**Note**
Simple interface	Do not constrain the user; adapt to them
Formula validation	Incorrect form pointed out quietly
Proof validation	Incorrect form pointed out quietly
Proof formatting	Automatic numbering and indentation
Markdown language	e.g. p AND q *meaning* p ∧ q

Table 1: Core Functional Requirements

Simple interface

The interface itself should be 'bare bones', become more complex only when necessary, and make good use of macOS app conventions.

Formula validation

In order for a proof to be correct, its formulae must be well formed. The app should check these and feed back using syntax highlighting, which will be absent when a formula is not well-formed. (A later update included explicit formula syntax warnings to avoid disadvantaging users for whom colour is less useful).

Proof validation

The overall proof needs to be validated, and this validity communicated to the user.

Use a markdown language

The symbols used in logic are challenging to enter on a standard keyboard. To make reading and writing proofs simple, a simple markdown[1] was chosen with simple English or symbols representing connectives and the turnstile: e.g. `p AND q`, `p OR p`, `NOT p`, `->`, `<->`, `p AND q |- q AND p`. The final markdown set can be viewed in Appendix: 8.7 Baker Street Markdown Reference.

The requirements in the table below are secondary, and refer to behaviours that are less critical to the main brief, but still desirable.

Requirement	Note
Preferences	Font size
Solution hints	Local (i.e. step) or global (i.e. whole proof)
Export proof	To LaTeX, Markdown, graphic
Transform proof	Intro truth tables, tree diagrams

Table 2: Secondary Functional Requirements

Preferences

The user should be able to set the font size, the font, toggle syntax highlighting, and the degree of 'hint strength' (i.e. the amount of detail about how their errors). (The final version of the app focused on changing the font size only.)

Solution hints

These will indicate to the user what a useful next step might be. They can either be more local (i.e. suggested next step) or more global (i.e. hint at the strategy they should be working towards).

1. Technically, this is a *markup* language, but I wanted to connote a lightweight annotation system similar to Markdown (Gruber, 2004), which might be more familiar as a term.

Export proof

Useful to both students and teachers using the app, this would simply save the proof in John Gruber's Markdown format, PDF, or some form of graphic. (The final version was implemented text export only.)

Transform proof

Display the proof, or its formulae, or both, as trees or as truth tables. (This feature was not implemented, though the app does create tree and truth table representations for its internal use.)

3.2.2 Nonfunctional Requirements

The nonfunctional requirements illustrate higher level features. They are *usability*, *stability*, *responsiveness*, *reliability*, *fault tolerance* and *extensibility.*

Usability

In keeping with the brief's statement 'The user experience should be similar to writing a proof on paper', the app should not, for instance, constrain the user with step-wise decisions that inhibit their interactions (e.g. separating the act of writing an inference rule from entering its line number). The user should be able to start typing and the app adapt to the user by pointing out where he or she might be going wrong. The app itself should not require a manual to understand; a brief tutorial document should be sufficient. It should otherwise behave like a document-based app.

Stability

The metaphors used throughout the app should be consistent. For example, a proof preview should use the same colour scheme as an example proof in the documentation satellite windows (e.g. Rules Overview; see Section 5.8 Floating Document Windows).

Responsiveness

There should be no obvious delays in user interface interactions. This stays consistent with the Apple Human Interface Guidelines, which require that any non-UI code take place using a concurrent thread (Apple Human Interface Guidelines, n.d.).

Reliability

The app should rarely crash or behave unexpectedly. While it is not possible to guarantee this, undesirable behaviour can be minimised by ensuring the app interacts with macOS in the manner prescribed by its APIs, not working against them. The data files should have longevity; plain text is a good candidate format for this.

Extensibility

The architecture of the app should permit subsequent developers to add code straightforwardly. The use of Swift and macOS frameworks will mean that reworking the app into other operating systems that support similar frameworks, e.g. iOS, will be straightforward. The source code will conform, as much as possible, to a standard style guide (Ray Wenderlich Style Guide, n.d.) and be available on Github (bakerStreet GitHub Repository, 2020) under a permissive MIT Licence, allowing copying with attribution.

4 Implementation

4.1 Overview

In this chapter, I will outline my implementation process. This includes commentary on Swift, Cocoa, the software engineering methodology itself, and the major increments of development—including the features and architecture of these increments.

4.2 Swift

The original brief for this project suggested using the language Java (see Appendix: 8.1 Original Brief). Java is a mature, C-like, cross-platform, Object-Oriented Programming language (Oracle Java, n.d.). With the Java Virtual Machine implemented across many platforms, the language has been a success story on server, desktop and mobile platforms. However, while Graphical User Interface (GUI) libraries such as Swing provide excellent functionality, they often lead to applications that are not first-class user experiences on platforms such as macOS, which have a strong tradition of native frameworks based on robust, opinionated and platform-specific design (e.g. the Cocoa application programming interface; Isted, 2010).

For several years, Apple has promoted Objective-C as its native development language (Mathias & Gallagher, 2017). Developed in the 1980s, this is a strict superset of C with Object-Oriented Programming (OOP) enhancements. Objective-C has been superseded by Swift (now at version 5.2.5) for several reasons (Swift, n.d.). First, Objective-C was popularised prior to the emergence of scripting languages like PHP and Python, whose syntax is more streamlined with regards OOP. Many view Objective-C syntax as parochial and relatively difficult to work with (Mathias & Gallagher, 2017). Second, Objective-C has safety issues, i.e. it explicitly permits operations that could lead to runtime errors.

In 2014, Apple introduced Swift. It maintains the C/Objective-C advantage of speed, is a general-purpose language, has a cleaner, more expressive syntax, and is

considerably safer thanks to compile time checks.

Thus, while Java has the advantage of portability, Swift allows the creation of a native Mac app that, for its users, should provide a better learning experience.

4.3 Cocoa

MacOS has rich software frameworks for creating apps (Matthias & Gallagher, 2017). I decided to use Cocoa, which includes Foundation Kit, Application Kit, and Core Data frameworks, and is a modern successor to the the NeXTSTEP and OpenStep software frameworks. Today, Cocoa has a solid, rich and mature basis compared to the alternative, UIKit.

4.4 Methodology

4.4.1 Software Engineering Approach

Overall, I used an *incremental* software development methodology, where functioning prototypes are produced in series (McConnell, 2004). In contrast to the Waterfall method, and others that emphasise upfront planning, this was better adapted to the uncertainty of completion times for the features. This uncertainty stemmed from my inexperience. At the beginning of the project, I had never used neither Swift nor Cocoa. It turned out to be the case, for instance, that my initial attempts at certain behaviours in the app code were reasonable but not idiomatic Swift. The iterative approach, therefore, was a sensible precaution against superfluous planning detail.

When coding, I employed several principles such as code re-use/DRY ('Don't Repeat Yourself'), inheritance (for classes, not Swift structs, which are value types), Swift protocols (see Section Proof: The Key Data Structure), and KISS ('Keep It Simple, Stupid'). I also adopted the Model-View-Controller approach because this is most compatible with Apple APIs (Isted, 2010); I explain this in more detail in Section 4.5.5 Architecture); looking back, several wrong turns could be traced to not fully adhering to these principles (see Section 6 Evaluation). Finally, I avoided third

party frameworks to (i) reduce dependency risk and (ii) enhance my learning by producing my own solutions.

My testing strategy was straightforward. I created positive and negative tests for public functions of key classes and structs, particularly those involved in formula processing and proof validation (McConnell, 2004). New tests were created to capture performance on failing scenarios (e.g. correct general proof statements with no left hand side were seen as incorrect initially), helping to prevent regressions. I provided as many exemplars as possible; for instance, Baker Street succeeds on all proofs used for coursework assessment in 2020, as well as all examples given in teaching materials. The tests integrate fully into Xcode (e.g. can be run separately and analysed). While Xcode provides a code coverage metric, I was less interested in this, given my emphasis on public functions and high level tests (e.g. providing an entire proof, which by necessity mobilised most of the non user-interface codebase). Testing the user interface was much harder. Though frameworks exist to achieve this, I chose not to use them on the grounds of expediency, preferring manual testing.

With regards the user interface, I worked from the dictum supposedly spoken by Picasso: 'Good artists copy. Great artists steal.' The design of Baker Street steals from several best-in-class Mac apps, elements of which can be seen in Appendix: 8.2 Initial Design Notes. I also kept as closely as possible to the spirit of Apple's Human Interface Guidelines (Apple Human Interface Guidelines, n.d.): design is fundamentally about serving human beings; actions should have predictable consequences; apps should be stable, clear, streamlined and have simple workflows; feedback should be clear, immediate and understandable; visibility improves usability; disclosure should be progressive (i.e. start with the simple and progress to the complex); the app should be clear on what actions the interface affords.

4.4.2 Supervisor Support

Since the initial planning stage, there were regular feedback meetings with my supervisor. These tended to discuss high-level aspects of architecture and how a user might interact with the program.

4.4.3 Daily Log

I kept a daily log. This was completed at the end of the work day, and occasionally at the beginning if I wanted to set particular reminders or goals. On Monday mornings, I reviewed the previous week's log entries. The complete log is available in Appendix: Log.

4.4.4 Documentation

Some class and state diagrams were used, though not extensively, and mostly at the beginning of the project. The refactoring process meant these became out of date quickly. For much of the time, I diagrammed small parts of the application on paper when focusing on their development. The Agile approach sometimes uses a table of features that are derived from user stories (i.e. goal-rationale pairs such as, 'As a student, I want to be able to export my proof so that I can present it as an answer') (McConnell, 2004). These features can be cross-tabulated against their business worth as a way of prioritising development. I chose not to do this; my aim was to produce at least the core functional requirements (see Section 3.2.1 Functional Requirements), and my Agile 'sprint' was the summer (McConnell, 2004). However, I did use an informal 'importance' metric on several occasions. See, for instance, Section Blind Alleys.

4.4.5 Source Control

I used Git from mid-way through the project. Some source code was contained in external, sandpit-like 'Swift Playgrounds' at the time; see Section Proof: The Key Data Structure. Once all code was placed in a single Xcode project at the mid-way point, it was placed under source control in a repository whose history can be viewed on GitHub (bakerStreet GitHub Repository, 2020). My default was *master*, and though I worked on the *develop* branch most of the time; other branches addressed important features and bugs. They used camel-case names like *adviceViewPadding*. Each commit used a short title with further details if necessary. Branches were merged into *develop* initially and then into *master*.

4.4.6 Planning

This section shows the planning that took place at the beginning of the project. Both the Timeline (Table 3) and Risk Analysis (Table 4) were useful exercises. The former gave me a sense of what was possible with the timeframe. Generally speaking, I worked on the app as quickly as possible, staying ahead of deadline for some elements (e.g. the interface prototype, which was completely quickly), but behind on others. For instance, though I met the 'formula validation' goal prior to 12 June, I continued to fix bugs and optimise this feature up to the last week of the project.

Week		Interface	Functional Requirement Met	Dissertation Goal Met
1	29/05/2020			
2	05/06/2020			Requirements Analysis
3	12/06/2020		Formula Validation (basic)	Risk Assessment
4	19/06/2020			Literature Review
5	26/06/2020	Prototype	Formula Validation	Early Deliverable (contents TBC)
6	03/07/2020			
7	10/07/2020		Proof Validation	
8	17/07/2020			Design
9	24/07/2020		Preferences	
10	31/07/2020			Implementation
11	07/08/2020			
12	14/08/2020		Solution Hints	
13	21/08/2020		Export Proof	Analysis
14	28/08/2020	Final	Transform Proof	Final
15	04/09/2020			

Table 3: Timeline

The Risk Analysis was particularly useful in underscoring the steps I would need to take to ensure quality in the code and the finished user experience. An outcome such as 'Proof Validation feature incomplete' was unacceptable, and for this reason validation was prioritised. Some elements such as 'Proof validation is poorly

implemented' did indeed come to pass (see Section 6.3.7 Optimisation); I did remedy this somewhat by pushing the calculations to a background thread, but the problem was due to the manner of implementation.

Label	Probability	Loss (1-5)	Exposure (P * L)	Mitigation
Complexity of code leads to poor quality app	0.2	3	0.6	Diagrams to maintain a 'map' of the app (e.g. classes, structs and functions). Constant refactoring as part of the incremental approach. Avoid 'hacks'. Use design patterns as intended by APIs; don't fight them. Comment app appropriately.
Interface refactoring takes too long	0.2	3	0.6	Develop a solid understanding of how the Cocoa framework works, particularly the design patterns (e.g. delegate) preferred for inter-object communication. This could slow down development
Formula and/or Proof Validation is poorly implemented	0.3	2	0.6	If the validation takes too long, it could slow down the main thread. This is unlikely (given that formulae are short), but could be solved by making the process concurrent
Data loss	0.1	5	0.5	Local and remote backups. Main project itself will use source control.
Proof Validation feature incomplete	0.1	5	0.5	This is a key feature and needs to happen. I know that it's technically possible, so I need to make sure it's done by prioritising it.
Formula Validation feature incomplete	0.1	4	0.4	This is a key feature and would have severe consequences if it doesn't work. However, I know it's possible and I should be able to do it. This needs to be prioritised.
Solution Hints feature incomplete	0.2	2	0.4	A feature that would be nice to have. However, not straightforward; should be a lower priority than the core features
Preferences feature incomplete	0.3	1	0.3	A 'bonus feature' that would be nice to have. Straightforward but a lower priority than the core features

Export Proof feature incomplete	0.3	1	0.3	A 'bonus feature' that should be fairly easy to implement. However, I'll de-prioritise it.
Transform Proof feature incomplete	0.3	1	0.3	A 'bonus feature', probably the least 'core' of all the feature. Still, it would be nice to have and should be a 'bolt on' given the modular nature of the app
Illness	0.2	1	0.2	There needs to be enough slack in the schedule that a short period of illness doesn't affect things
Requirements Analysis too vague	0.1	2	0.2	They'll never be perfect but there's a risk that I'm overlooking something. Careful adherence to the iterative and incremental approach should help here; try to look for things that I don't 'know'.
Syntax highlighting incomplete	0.2	1	0.2	As a 'bonus feature', this wouldn't be too much of a problem. I could achieve a similar effect simply by introducing a little 'tick' indicating that the formula is well formed

Table 4: Risk Analysis

4.5 Major Increments

4.5.1 Overview

In this section, I will outline the development process by comparing five increments from initial user interface prototype to final app. I've broken this down to user features (i.e. discrete things the app can do for a user), the user interface, app architecture (i.e. major classes and structs), tests, and bugs.

4.5.2 Final App Statistics

Baker Street contains 48 Swift files (CLOC, 2020). There are 8759 code lines and 1875 comment lines.

4.5.3 Features

User Feature	May	June	July	August	Final
Proof validation	No	Yes	Yes	Yes	Yes
Syntax highlighting	No	No	Yes	Yes	Yes
Tailored errors	No	No	No	Yes	Yes
Comments in proof	No	No	Yes	Yes	Yes
Operator/ rule insertion	No	Yes	Yes	Yes	Yes
Proof preview	No	No	No	Yes	Yes
Proof export and copy	No	No	No	No	Yes
Zoom	No	No	No	Yes	Yes
Dark mode	Yes	No	No	No	Yes
Example proofs	No	No	No	No	Yes
Inference rules overview	No	No	No	No	Yes

Inference rules in detail	No	No	Yes	Yes	Yes
Definitions	No	No	Yes	Yes	Yes

Table 5: Increments and User Features

❖

In Table 5, we can see, first of all, the expected trend of later increments having more features. The 'Proof validation' feature was implemented early in the process because it was critical; without this feature, the app itself would not be viable. 'Syntax highlighting'—also important but somewhat less so than proof validation—appears in the increment from the following month, July. 'Proof preview' was not a core feature and was implemented relatively late, in August. The 'Dark mode' feature was fully implemented in the first version of app because its interface was so simple (i.e. it had few custom elements, such as colours, that needed to be switched to alternates when the system-wide dark mode was activated). I was not satisfied with dark mode behaviour until the final week, which is when I succeeded in reliably (i) detecting dark mode and (ii) triggering a wide-ranging redraw of all windows to make them honour the mode.

4.5.4 User Interface

User Interface	**May**	**June**	**July**	**August**	**Final**
Open/save files	Yes	Yes	Yes	Yes	Yes
Use BKProof files	No	No	Yes	Yes	Yes
Adjustable splits	No	Yes	Yes	Yes	Yes

Proof status indicator	No	No	No	Yes	Yes
Menu and toolbar items	No	Yes	No	Yes	Yes
Redundant menus (e.g. Format) removed	No	No	No	No	Yes
Tutorial	No	No	No	No	Yes
Toolbars use bespoke icons	No	No	No	Yes	Yes
Autohide advice view	No	No	No	Yes	Yes
Use BK colour palette	No	No	No	No	Yes
Document windows closable via menu, toolbar and window	No	No	No	No	Yes

Table 6: Increments and User Interface

At a high level, the user interface was inspired by the classic three-pane-with-toolbar approach of apps like Soulver (see Appendix: 8.2 Initial Design Notes), and this was established in the initial May increment, as indicated in Table 6. Also

established very early on was the use of a markdown language to enter the proof. This was intended to solve the problem that logical connective glyphs (e.g. ∧) cannot be straightforwardly entered using a keyboard, and I was not satisfied by other approaches (such as a floating palette of selectable glyphs, or binding glyphs to the keyboard). The markdown solution, while not perfect, was designed to protect 'flow' during the construction of a proof (see Section What Works for more reflection on this).

'Open/save files' comes as standard thanks to the macOS API for a document-based app. An early decision was to use plain text files. For Baker Street, these come in two forms. The first is the OS-registered file type 'BKProof', which is plain text but has the extension '.BKProof'. Upon app install, macOS will understand that Baker Street should open these. The second is any plain text file, which can be dragged onto the Baker Street icon for it to open. Plain text has two advantages: (i) they continue to be accessible long after the lifetime of the app (cf. reliability in Section 3.2.2 Nonfunctional Requirements), and (ii) they are human readable.

The 'Proof status indicator' was a new feature I included late in August as a way of indicating to the user that their proof was (i) correct (green dot with 'Proof correct'), was (ii) being validated (amber dot with animated progress spinner), or was (iii) incorrect (red dot with 'Proof Incorrect'). Below, in Figure 2 you can see the status of a correct proof. This feature replaced a large green tick (similar to the one at Line 0) that had been used in the right hand 'advice' panel (see 5.2 The Three Panels) until this point; after some discussion with my supervisor, it was agreed that the advice panel was better served only in displaying errors. Putting the status indicator inside the footer, which spans all three panels, also provides a visual cue that it applies to the entire proof rather than a particular line. This figure also shows the decision to represent scope (a concept similar to scope in computer programming) using indentation; colour is also used for proofs with complex scopes (see Figure 3).

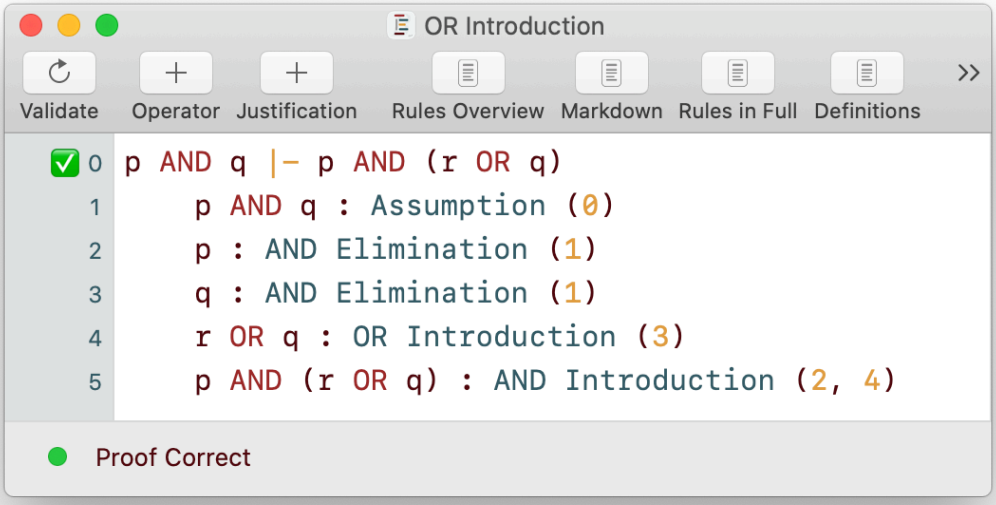


Figure 2: The Proof Status Indicator

'Document windows closable…' was an important interface component. First, this means that the floating document panels (see Section Floating Panels) can be opened and closed using the app's toolbar (see Section 5.5 Toolbar), and the relevant toolbar item will remain highlighted while the panel is open. This behaviour was relatively simple to code. However, it became clear during usability testing that the floating panels should be closable independently (i.e. using their own close buttons, or the common keyboard shortcut ⌘w). Making this interact with the toolbar highlight colour was challenging due to edge cases. Finally, I wanted a menu item to read, for example, 'Show Preview' when the floating preview window was closed and 'Hide Preview' when it was open. Coordinating between these behaviours was difficult, and achieved only in the final increment. The desired behaviour showing the updated menu, toolbar highlight and floating window can be seen in Figure 4.

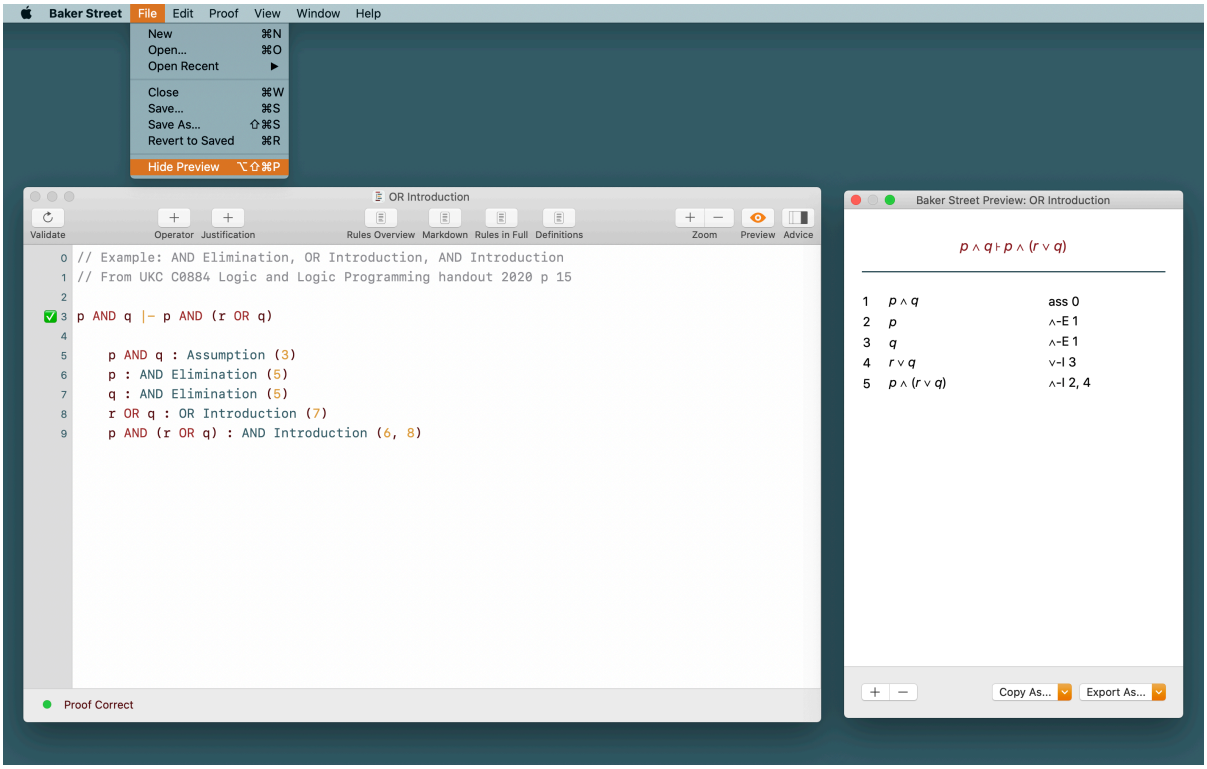


Figure 4: Baker Street with Preview Window open, Menu text changed, and Toolbar item highlighted

4.5.5 Architecture

Table 7 shows the introduction of major classes and structs. Full descriptions for these are provided in Section 8.8 Architecture Elements. Baker Street follows the Model-View-Controller approach (Apple Human Interface Guidelines, n.d.; Isted, 2010), in which the user interacts with a view, which is monitored by controller code, which then manipulates data (a model), which then updates what the user sees (the view). The distinction is not always straightforward ('Preferences', for instance, holds global enumerated types, functions and constants related to appearance). In the table, it is clear that relatively important and difficult elements were tackled earlier. The Formula Parser, for instance, is a struct (a Swift value type) that was developed outside the app in a Swift Playground (see Section 4.4.1 Software Engineering Approach) for rapid iteration before being implanted, in immature form, into the June increment, where tests were added; the job of this struct is to take the kind of string that a user might enter, for instance `p AND q`, handle lexer/parser operations,

and return a custom Formula type. A later element, the *Semantic Permuter*, is another value type whose job is to take a Formula type and produce a truth table.

App Architecture Element	May	June	July	August	Final
Proof Controller	No	Yes	Yes	Yes	Yes
Exported Proof	No	No	No	Yes	Yes
Inference Controller	No	No	Yes	Yes	Yes
DocumentView Controller	No	No	No	Yes	Yes
PreviewView Controller	No	No	No	Yes	Yes
ImageView Controller	No	No	No	No	Yes
Preferences	No	Yes	Yes	Yes	Yes
Syntax Styler	No	No	Yes	Yes	Yes
Advice Styler	No	No	Yes	Yes	Yes
Formula	No	Yes	Yes	Yes	Yes
Semantic Permuter	No	No	No	No	Yes
AdviceView Controller	No	No	Yes	Yes	Yes
Examples	No	No	Yes	Yes	Yes

Table 7: Increments and App Architecture

4.5.6 Tests

Tests	May	June	July	August	Final

Precedence Operator	No	Yes	Yes	Yes	Yes
Tree	No	Yes	Yes	Yes	Yes
Lexer	No	Yes	Yes	Yes	Yes
RpnMake	No	Yes	Yes	Yes	Yes
Formula	No	Yes	Yes	Yes	Yes
Proof (simple)	No	Yes	Yes	Yes	Yes
Proof (complex)	No	No	No	Yes	Yes

Table 8: Increments and Tests

Table 8 shows the introduction of tests across the increments. A more informative table is Table 9, which breaks down the tests into positive (expecting success), negative (expecting failure) and uses harness (requires struct/object set up) (see McConnell, 2004).

Tests	**Positive**	**Negative**	**Uses Harness**
Precedence Operator	3	3	No
Tree	4	1	Yes
Lexer	2	1	No
RpnMake	7	1	Yes
Formula (well formedness)	31	18	Yes
Formula (individual logical connective)	16	0	Yes

Proof	35	18	Yes
Total	**98**	**42**	

Table 9: Increments and Selected Bugs

Full descriptions for each element tested are provided in Appendix: 8.9 Tested Elements. The app has only 140 unit tests, which is not a large number considering the code base size (see Section 4.5.2 Final App Statistics). Negative tests are outnumbered by positive tests, partly because the set of positive tests for some of the elements tested is larger. Importantly, the Proof class was tested thoroughly with all the proofs I could find; note that the testing of a proof necessarily involves the testing of all other elements in Table 9 because these are required for validation.

4.5.7 Bugs

Table 10, below, represents a selection of bugs that took an unusual effort (greater than two days) to resolve. Some of these are referred to in Section Points of Interest and Chapter 6 Evaluation. To pick one, the July 'Proof validation very slow' issue was traced back to the algorithm generating the proof, which originally had a low linear complexity but worsened to logarithmic complexity[1] (Knuth, 1997) when I increased its workload to generate example proofs as part of the help text provided in the advice view. I investigated the complexity issue with Xcode's Instrument/Time Profiler, used to indicate which functions are alive for a given proportion of a time sample; this was overwhelmingly the 'help text' code. I solved the issue by including a flag that told the proof generating code to produce a 'light version' (without any help text) when requested, and this returned proof validation to linear complexity (see Appendix: Log, July 22nd).

1. The complexity was easier to fix than the first issue this created (a low-level 'bad exception' error that I traced to an infinite recursion).

June	July	August	Final
Unproven theorem disappears on validation	Proof validation very slow	Advice auto hide causes text to stretch	Zoom bugs
Comments disappear on validation		Proof export has incorrect numbering	

Table 10: Increments and Selected Bugs

5 Walkthrough

5.1 Overview

In this chapter, I will provide a brief overview of the final app, taking the perspective of a user creating a proof for the first time.

5.2 The Three Panels

Baker Street opens with an empty, untitled window shown in Figure 5.

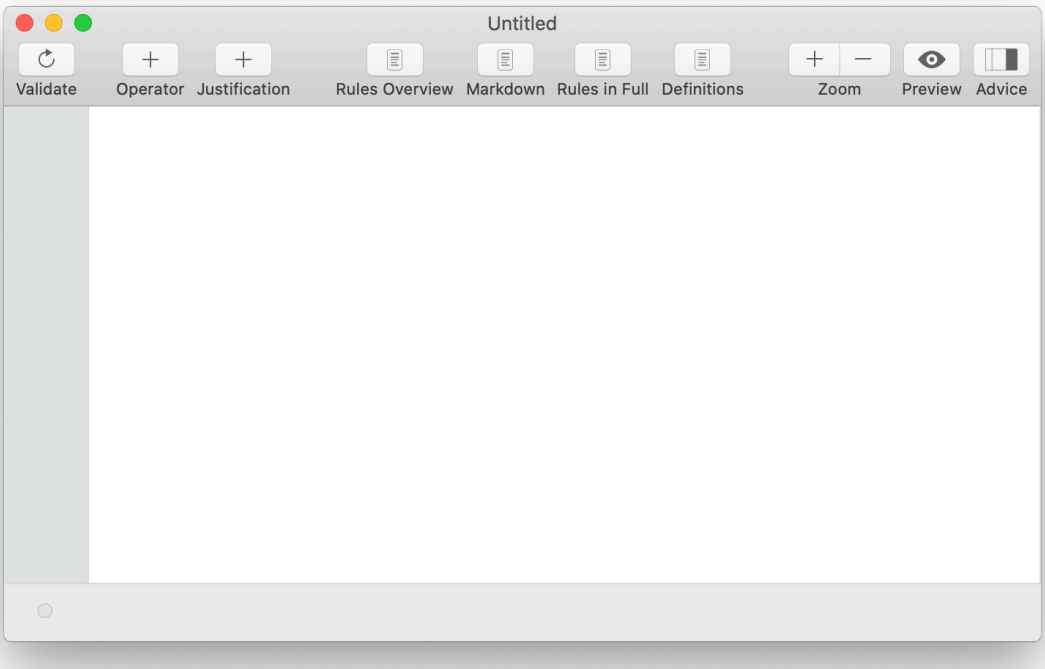


Figure 5: Empty Baker Street Window

The *Help* menu includes a tutorial that provides a quick introduction to its features. However, the user is free to start entering a proof immediately. Below, in

Figure 6, Baker Street is shown with a partially entered proof. The figure illustrates many of the user interface elements and can be compared with Figure 7, which is the same proof, but complete and correct.

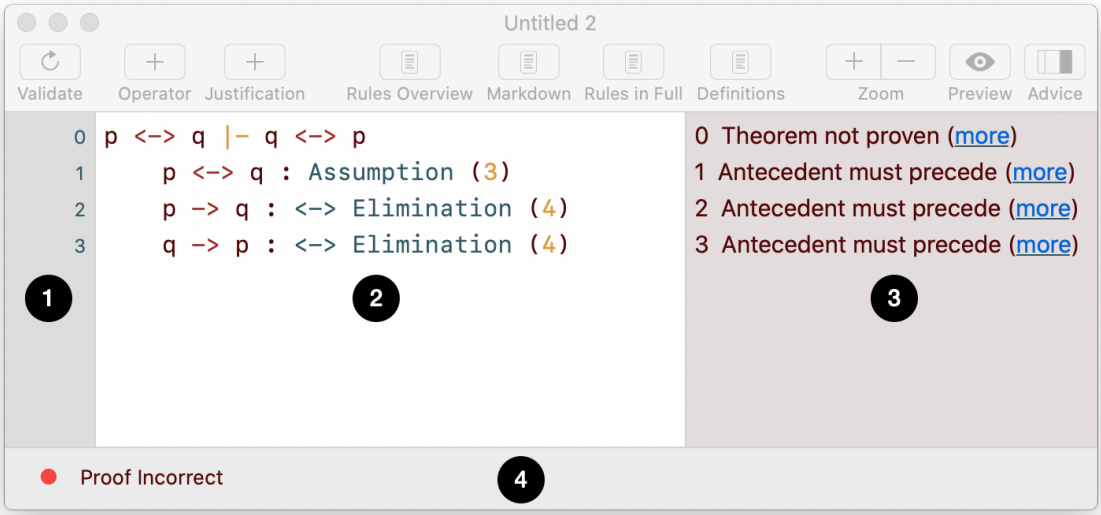


Figure 6: Proof Incorrect

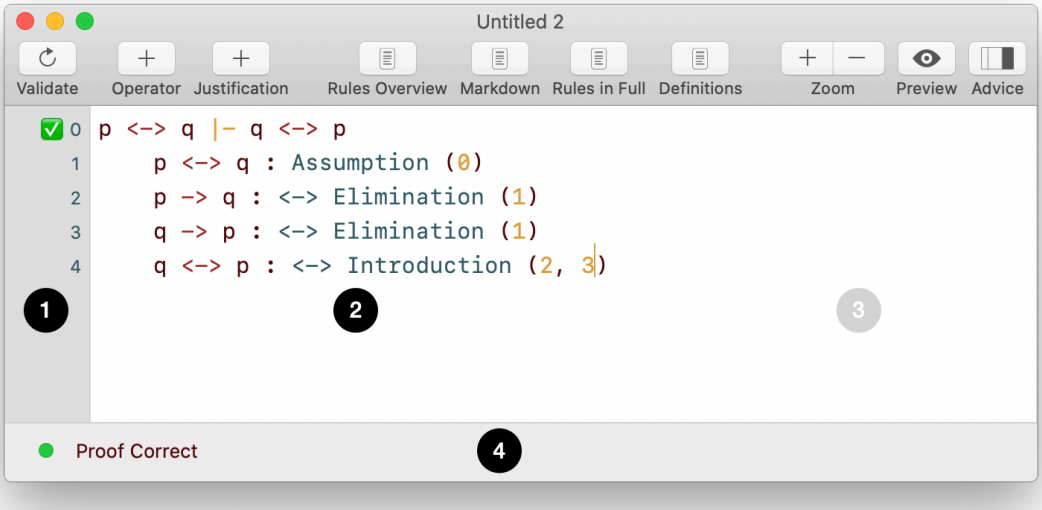
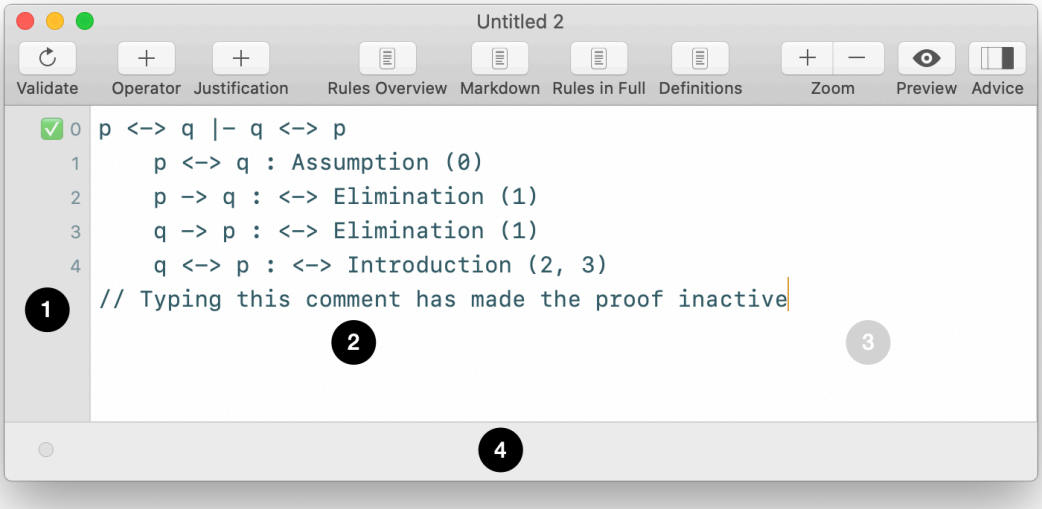


Figure 7: Proof Correct

Section 1 (the circled '1') in both figures is a panel showing line numbers. This panel also indicates when a theorem is correct via a green tick (Figure 7, Line 0). To maintain the key principle of simplicity (see Section 3.2 Design Considerations), line numbers are integers increasing by one unit per line; they do not encode scope (i.e.

level of indentation, which is commonly seen in a final proof in a form like 1.2.1).

Section 2 is the statement view, where the user types their proof in a simple markdown format (see Section 3.2.1 Functional Requirements). This is presented as a document to capture two key experiences suggested by the briefing statement: 'The user comfortably enters a proof' and 'Note that a user interface aims to prevent a user from making mistakes, whereas an exercise tool gives freedom to make mistakes' (see Appendix: 8.1 Original Brief). This document-based entry ensures that the user will never be interrupted with queries about, for instance, which line numbers an inference rule corresponds to, nor will they be constrained to enter only correct lines. The user is free to enter any text they wish. As long as a line roughly approximates a proof line,[1] it will be parsed as such. Lines can be (i) a theorem (see Figure 6, Line 0), (ii) a justified line (Lines 1-3), or (iii) a comment (see Figure 8[2]). Blank lines are permissible and encouraged. The proof is validated (i.e. tested for correctness) when the user requests it; at this point, any well-formed line will be coloured to indicate its syntax is understood. This 'validated state' is deliberately fragile; as soon as the user types anything else into the main text view, the state becomes 'inactive'—as indicated by a loss of syntax colouring, dimming of the line view (Section 1), the advice view (Section 3) and loss of colour in the proof status area (Section 4), which you can see in Figure 8. The state will be 'active' once more only after the user has requested a new validation.



1. Spacing, capitalisation and punctuation like the comma are ignored.
2. The comment will receive a line number only after a proof validation request by the user.

Figure 8: Inactive State

Section 3 is the advice view, which appears when a proof is incorrect in order to give the user feedback on their errors, and disappears when all errors are resolved. An advice line comprises (i) a line number, which helps in matching the advice to its line, (ii) a brief statement of no more than 23 characters providing a statement of the error, and (iii) a hyperlink (in the traditional blue) reading 'more'. This implements the briefing document's direction to '[check the proof] and explain mistakes' (see Appendix: 8.1 Original Brief). Maintaining the principle of simplicity, Baker Street provides the short statement, which might be enough of a hint.[3] A student must click the hyperlink to get more info. Thus, the user interface remains uncluttered and minimises the cognitive load on the student (see Section 2.3 Why Students Find This Challenging); this also touches on the Apple Human Interface guideline of progressive disclosure (cf. Section 4.4.1 Software Engineering Approach).

Clicking the hyperlink produces a transient 'popover' window with further details. Important terms receive colour highlighting. For examples of these, see Figure 9. These errors are *tactical* (i.e. contextualised) following Van Ditmarsch (1998); see Section 2.4 Computers Can Help. They are generated following Baker Street's attempts to satisfy assumptions or inference rules. An example of this for the rule 'IF Elimination' is shown in Table 11.

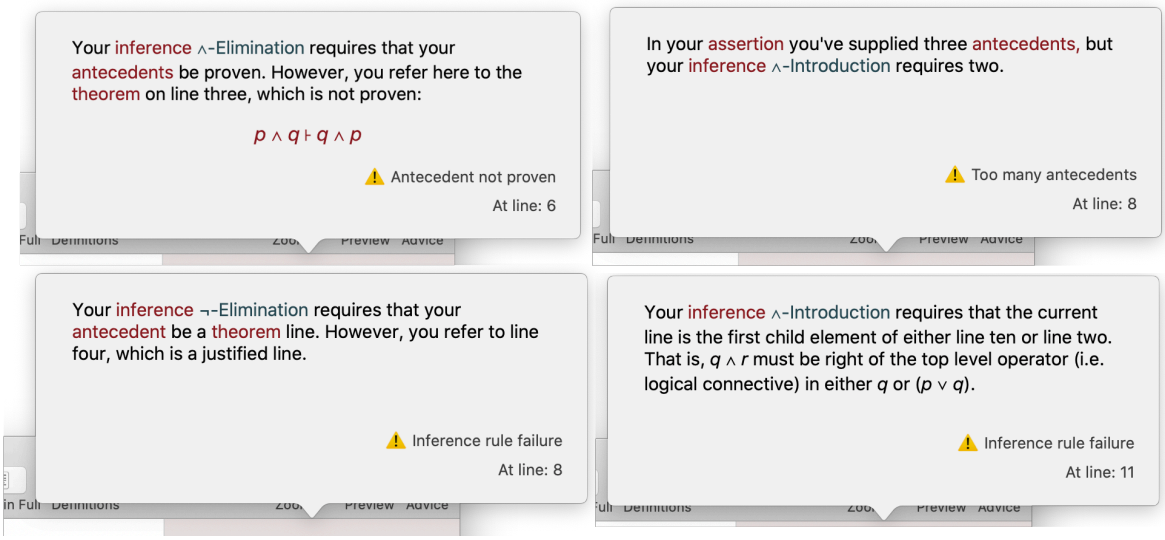


Figure 9: Examples of Advice Popovers

3. Lines may contain several errors. Baker Street will present the single most important based on a priority assigned to each error type.

Check	Type	Yes	No
Does the number of antecedents match the arity of the inference rule?	**General**	Continue	Fail; compute advice message for user
Do all antecedents come earlier in the proof than the current line?	**General**	Continue	Fail; compute advice message for user
Do multiple antecedents refer to different proof lines?	**General**	Continue	Fail; compute advice message for user
If an antecedent must be proven for this rule, is it actually proven?	**General**	Continue	Fail; compute advice message for user
Are all antecedents justified lines?	**Specific to IF-Elimination**	Continue	Fail; compute advice message for user
Does the first antecedent have an IF top level operator?	**Specific to IF-Elimination**	Continue	Fail; compute advice message for user
Is the second antecedent the first child of the first antecedent?	**Specific to IF-Elimination**	Continue	Fail; compute advice message for user
Is the current line the second child of the first antecedent?	**Specific to IF-Elimination**	Mark line as correct	Fail; compute advice message for user

Table 11: The Satisfaction Process used to Test the User is Using IF-Elimination Correctly in Their Proof

Section 4 is a footer reserved for indicating the status of the proof with a coloured dot and optional text. This status takes three forms, which you can see in Figure 2. The status is refreshed only when the user requests validation. Because validation takes place on a background thread, the UI remains fully responsive; the amber dot with animated spinner tells the user that the app is doing work at that time. Users on a modern Mac will rarely see amber.

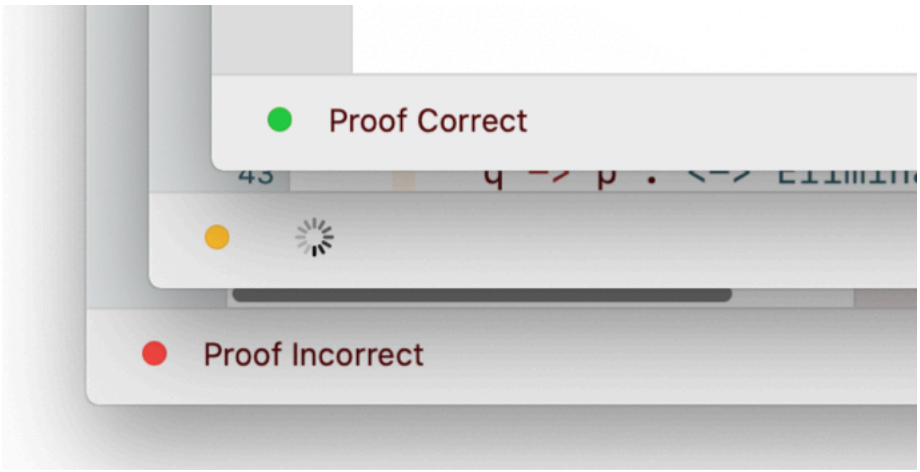


Figure 2: Proof Status Indicator

In terms of how the three window sections work together, all are zoomable via a toolbar item (which has a menu equivalent and keyboard shortcut). All vertical splits between the sections are resizable. Additionally, any resizing of the advice view will be honoured the next time it appears (this helps the user fit advice text into its panel when zoom is increased). All sections maintain the same vertical position, so that lines are always in vertical alignment. The colour scheme is shown in more detail in Appendix: 8.5 Palette. No sections permit word wrapping; as shown in Figure 10, when content is obscured in the statement view, a transient horizontal scroll bar appears, indicating to the user that they can scroll to see more. When advice content is obscured, text is truncated with the ellipsis character.[4]

4. During testing, it was more natural to simply resize the advice view using its vertical split rather than use a horizontal scroll bar.

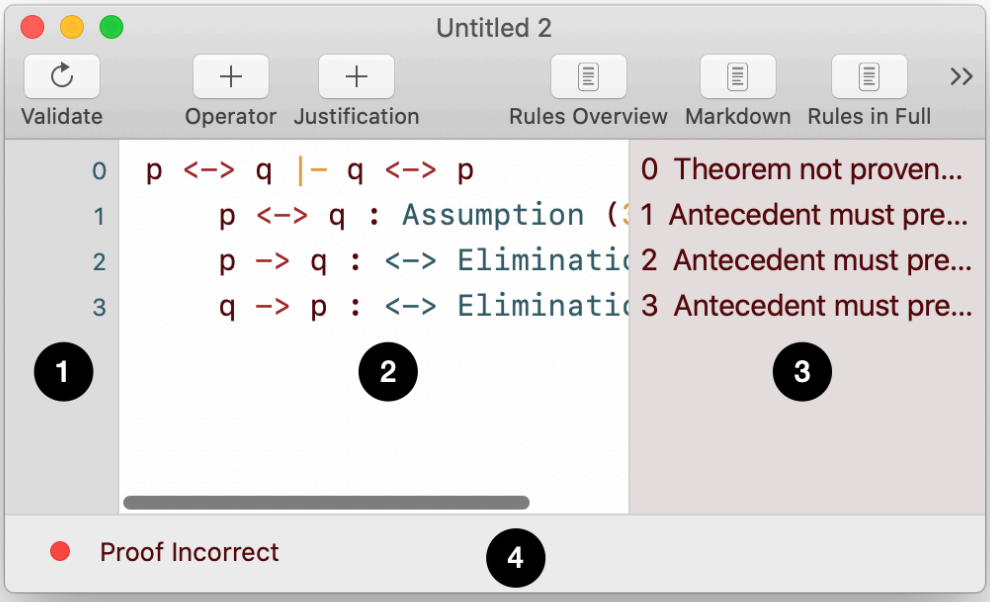


Figure 10: Content Obscured

5.3 Preferences

In keeping with the theme of simplicity (see Section 4.4 Methodology), I took the decision to eschew any capability for the user to set preferences explicitly, beyond manually zooming window contents.

5.4 Scoping

Scoping (analogous to line depth) is an important part of understanding a natural deduction proof and, arguably, difficult to picture when indicated by fractional numbers alone (e.g. 1.2.1). Scope misunderstanding can lead to frustrating errors that are trivial to make and difficult to spot using pencil and paper (where scope can be challenging modify without starting over). As shown in Figure 3, Baker Street

indicates scope with (i) indentation and (ii) subtle coloration for immediate visual apprehension. Note that the green tick for each correct theorem also helps surface the scope structure.

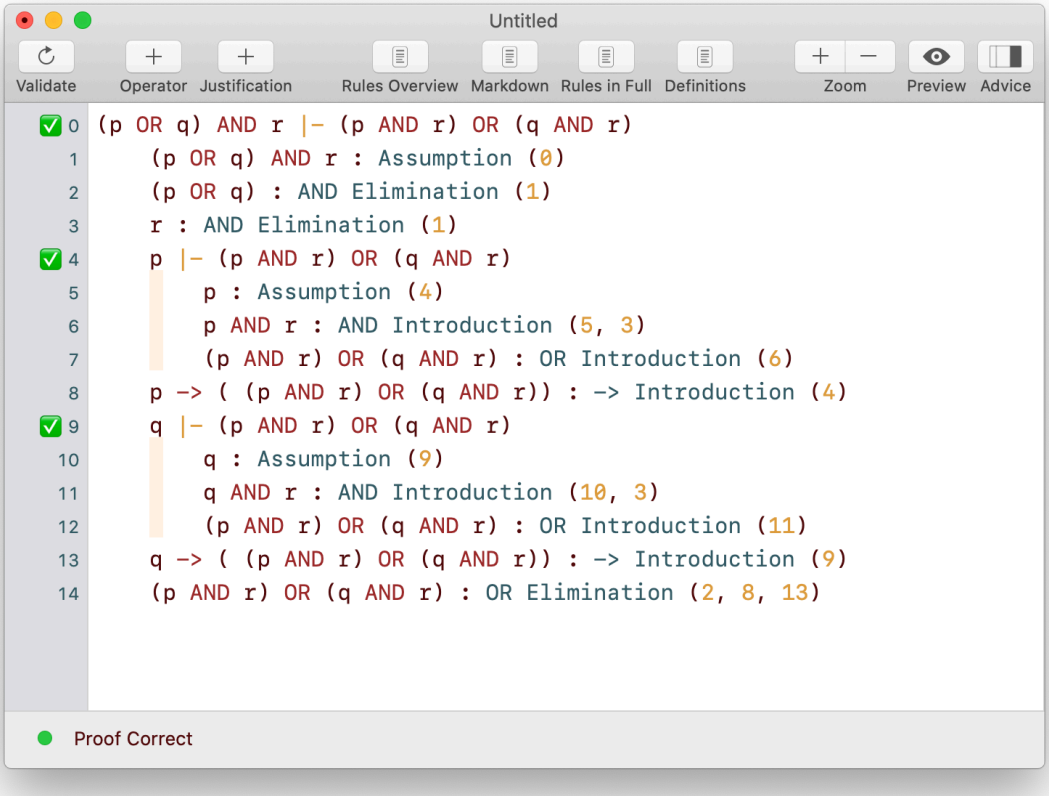


Figure 3: Indicating Scope with Pale Vertical Lines

5.5 Toolbar

A toolbar is a standard macOS interface feature. Baker Street comes with a customisable set (Figure 11). In customisation mode, toolbar item labels are more expressive. Users can show or hide the toolbar, choose features to add, show or hide text beneath the icons, and change their size. Icons for the 'reference' items and 'Toggle Advice Panel' were created for Baker Street (see Appendix: 8.4 Final Custom Icon Design).

Figure 11: Customising the Toolbar

5.6 Autocompletion

All important functions within the app are accessible from the menu and toolbar. They all come with keyboard shortcuts. The toolbar/menu items *Operator* and *Justification* trigger autocompletion at the text caret in the statement view for logical operators/connectives and justification types (i.e. inferences rules or assumption) respectively. While adds an important element of discoverability for the markdown needed to enter these, the most common case should remain manually typing an operator or justification. In Figure 12, we see the user using this feature to select the operator.

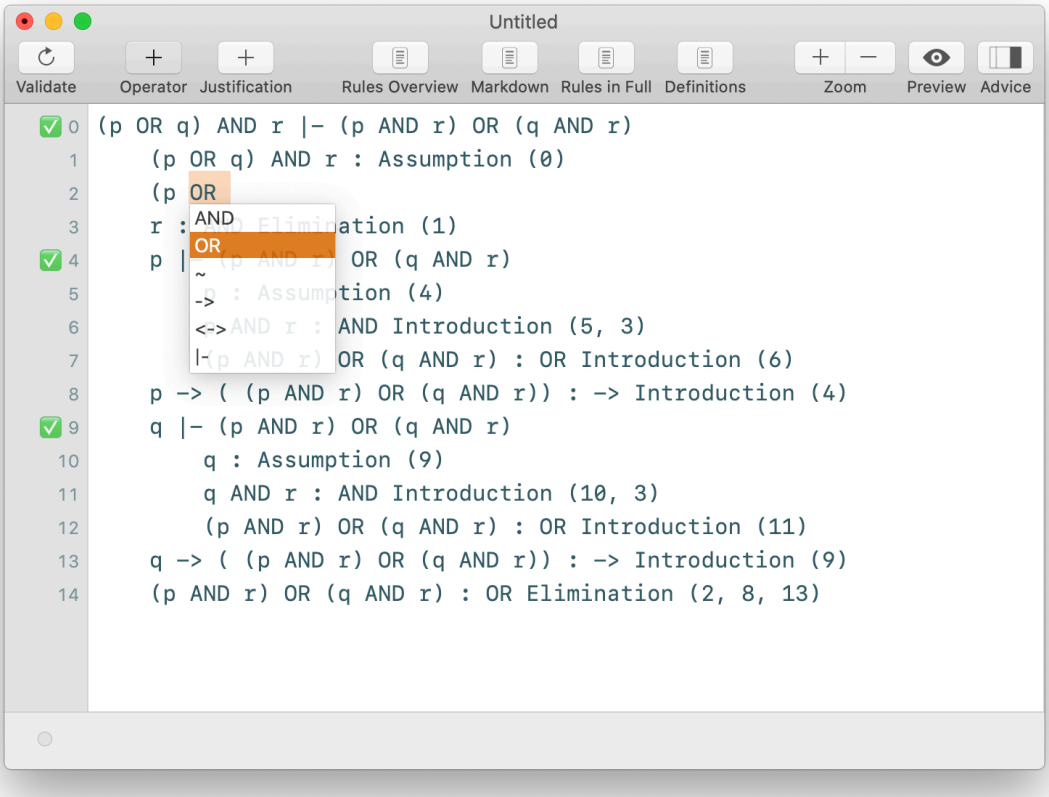


Figure 12: Selecting an Operator

5.7 Help

The *Help* menu provides access to the tutorial but also example proofs showing every supported inference rule in action. These launch new document windows that allow a user to see a successful application of the rule. Figure 13 shows the menu being used.

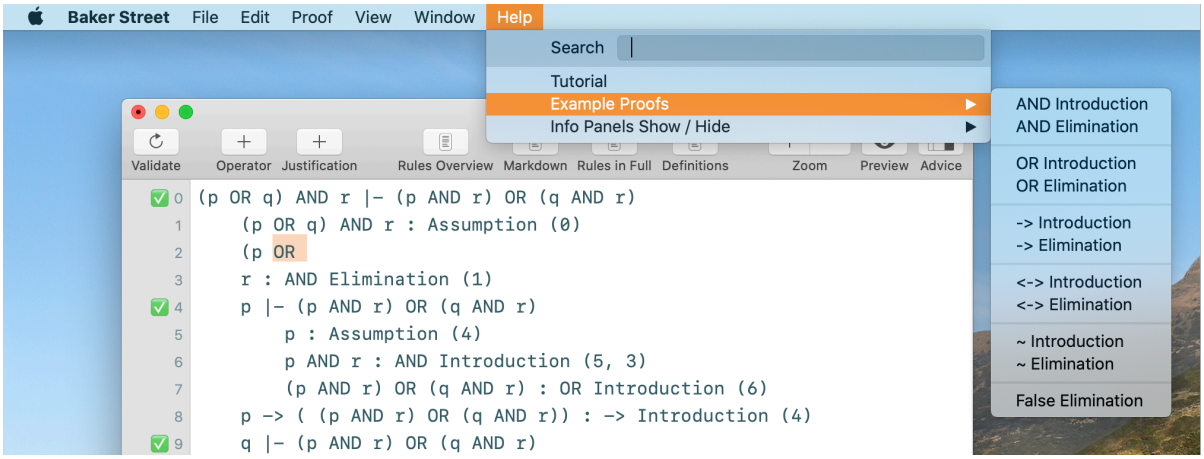


Figure 13: Help Menu

5.8 Floating Document Windows

The toolbar in Figure 11 shows *Definitions Reference*, *Rules Reference*, *Markdown Reference*, *Full Rules Reference*, and *Overview Rules Reference*. When clicked, these produce the floating windows, pictured in Figure 14. Their independence of the main window is designed to serve the briefing request, 'The user experience should be similar to writing a proof on paper' (see Appendix: 8.1 Original Brief) in that these act like information sources off to one side, to be consulted as the user requires. With the exception of the *Overview Rules Reference*, each floating window is scrollable, resizable, zoomable, and has a text search field. Some floating windows contain example proofs that follow the colouring of Baker Street's exported proofs (see Section Floating Preview Panel).

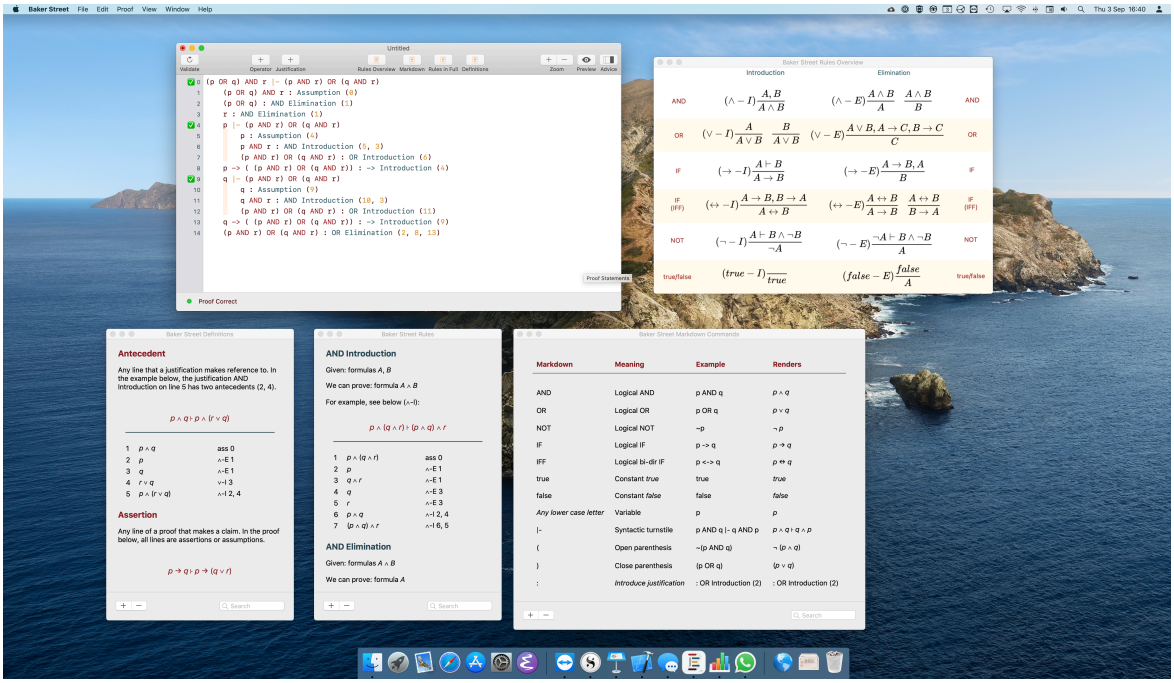


Figure 14: Floating Windows. Note: These do not obscure the main window

5.9 Floating Preview Window

Baker Street provides a preview feature accessible from the menu and toolbar. This produces a floating preview window that updates whenever the proof is manually validated by the user. Here, the proof is presented in a form that is closer to the one a user might see in handouts. Figure 15 shows two examples, one simple and one complex.

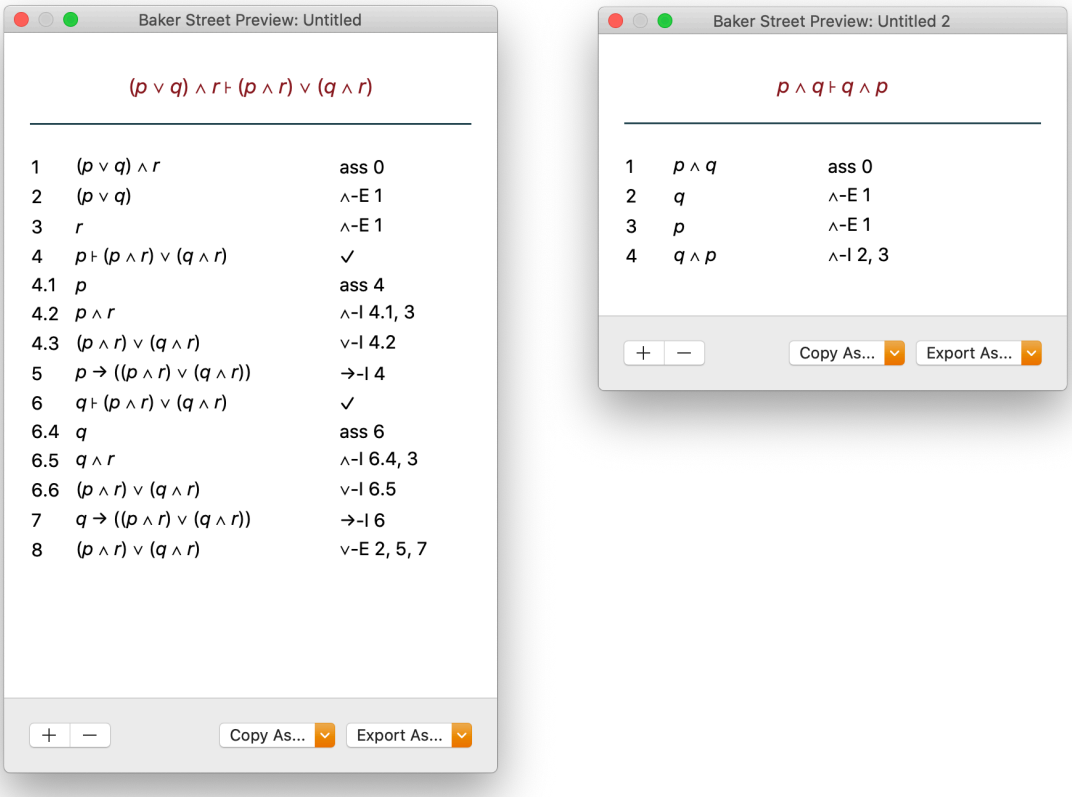


Figure 15: Example Previews

The preview window is zoomable, uses a common colouring cue for the top level theorem, and permits several copying or export actions. A copy action pastes formatted text to the clipboard, while an export action saves it to a file. The formats are LaTeX (LaTeX – A document preparation system, 2020), HTML (HTML, n.d.), and Markdown (Gruber, 2004). Figure 16 shows how each of these renders in appropriate applications:

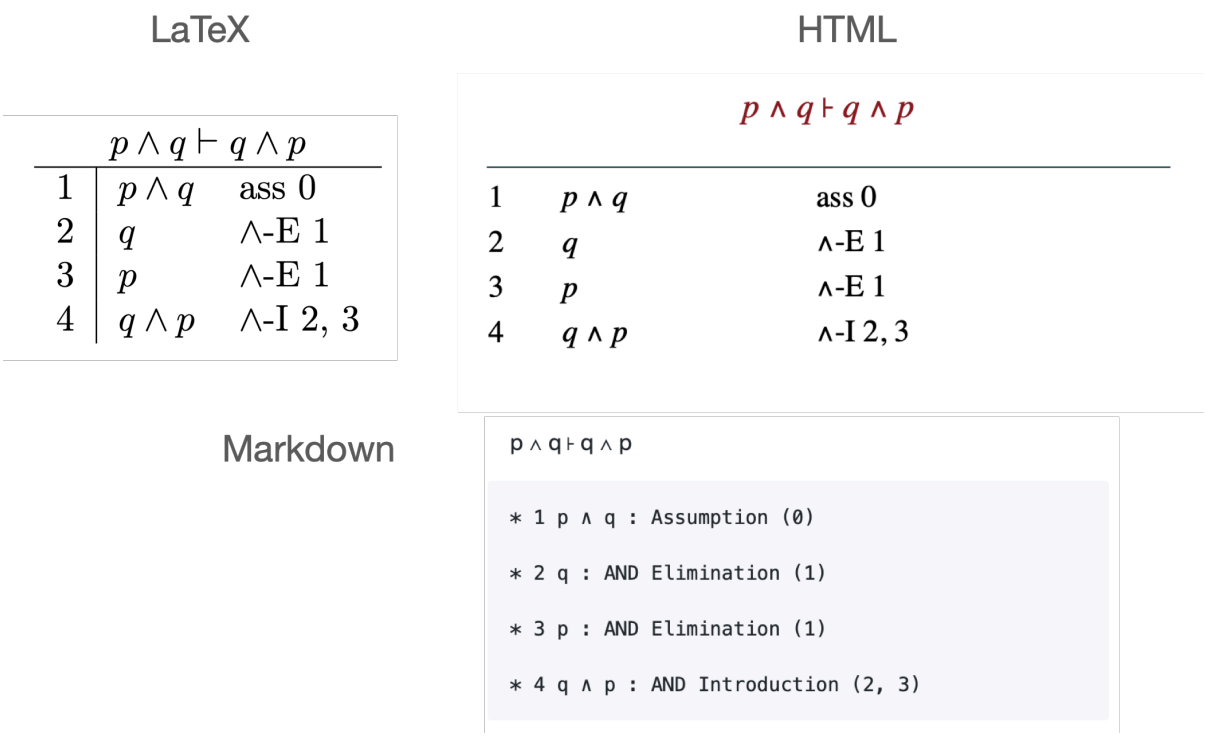


Figure 16: Rendered Examples of Export Formats. Note: The HTML uses the Baker Street colour palette.

5.10 Dark Mode

If macOS switches to dark appearance, Baker Street will honour this immediately. Figure 17 shows what happens to Figure 14 (with the preview floating window added on the right).

Figure 17

6 Evaluation

6.1 Overview

In this chapter, I will take a more self-reflective view on the state of Baker Street as well as the development process. Much of this necessarily overlaps with with Chapter 4 Implementation, given that the development issues remain the same.

6.2 What Works and What Doesn't Work

6.2.1 In General

In general, I'm satisfied with Baker Street. Much of it works effectively. The user interface is straightforward and its features discoverable via the toolbar and the menu. The *Help* menu provides a bare-bones tutorial and my sense is that the app represents a typical macOS experience. Mac apps tend to be designed with the idea that they do one thing, and do it well. I'm not sure if Baker Street allows users to enter proofs perfectly well, because the markdown format is novel, but this is the solution I elevated over others (e.g. a floating palette of special symbols that the user selects with a mouse, or a large set of keyboard shortcuts, both of which can be frustrating). The app launches quickly, validates proofs almost instantly, and provides a basic suite of tools and information for a student to create their proof, check it, and export it.

The tailored feedback works well but could be made more helpful if Baker Street produced a proof itself—somehow fitting the proof that the student is aiming for. Hints could then include the number of subproofs, the next inference rule to use, and so on. Ultimately, I ran out of time to do this. That said, one hint-like feature that did make it into the final version was the check that a (sub)theorem is provable (i.e. the user is warned that they cannot prove a theorem, so there is no point in including it in the proof) (see Section 6.3.12 Theorem Provability).

6.2.2 App Store Sandboxing

I wanted to implement a feature where Baker Street would export the proof as a PDF. The simplest way would have been to hand off the LaTeX to a command line tool like *latexmk* and invoke it as an argument to *sh* (i.e. the Bash or other default shell). Unfortunately, sandbox security restrictions for the Mac App Store mean that an app cannot easily launch arbitrary third-party executables. This was one example where sandboxing limited the app. On balance, I find this acceptable given the benefits of App Store distribution. Additionally, exporting a single proof as a PDF is perhaps not ideal. A student is more likely to put the proof into a larger document for later rendering. In this case, pasting exported LaTeX or Markdown is probably more flexible.

6.2.3 Known bugs

To be clear, the bugs below exist in the version of Baker Street submitted in the corpus. During the writing of my dissertation, I looked into these and fixed them. GitHub releases v1.0 and v1.0.1 *do* contain these bugs. They are fixed in v1.0.2. The Mac App Store version (v1.0.2) *does not* contain these bugs.

Bug: Scroll Synchronisation

For the interface to feel integrated, it is important that the line view, statement view and advice view keep in vertical synchronisation so that content in any view corresponds precisely with content in its adjacent views. However, this fails when the statement view has text content larger than the current window. This behaviour is shown in Figure 18, and is referenced in GitHub Issue #5 (Baker Street GitHub Repository, 2020).

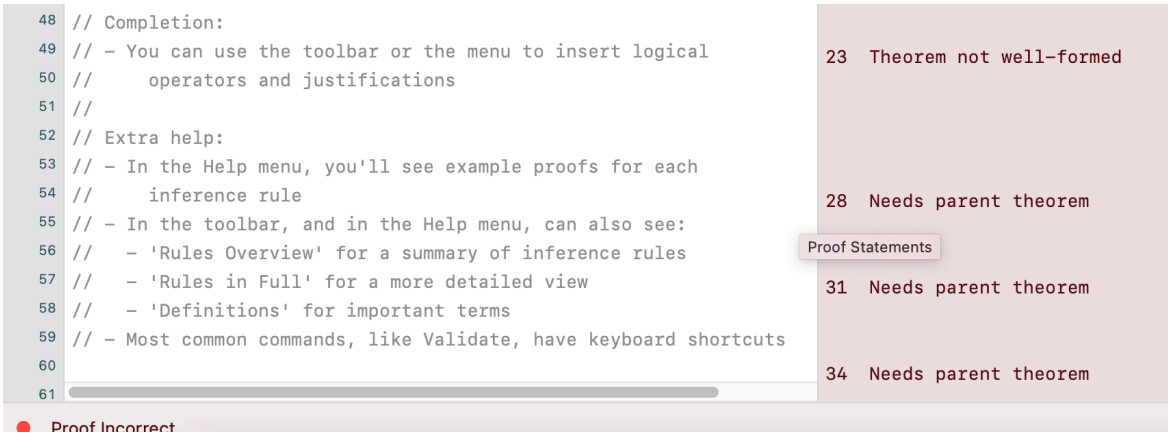


Figure 18: Scroll Sync Failure

Bug: HTML Exported in Dark Mode Renders with White Text on White Background

This bug is due to the manual colour override in my HTML exporter not paying attention to the OS interface colour. This is GitHub Issue #2 (Baker Street GitHub Repository, 2020).

Bug: Example Proofs Should be Saveable

When a student uses the *Help* menu to view an example proof in a new Baker Street document window, this window behaves like a document in all respects apart from one: it is not saveable. Intermittent crashes are also possible. This is because my manual creation of new windows failed to add them to the array of windows held by the *NSWindowController()* that manages window resources. This is GitHub Issue #1 (Baker Street GitHub Repository, 2020).

6.2.4 Accessibility

By being a Macintosh app, Baker Street inherits a rich accessibility framework. This includes enhancements for visual impairments, hearing, mobility, and learning (Apple Accessibility, n.d.). I've built on this by including tooltips for the main windows, having descriptive titles for menus, and so on. Additionally, each window is zoomable.

However, I'm aware that there is a reliance on communicating via colour. For instance, syntax highlighting cues the student that a formula or proof line is well-

formed. An accessibility preference that allows the user to toggle colour mode (e.g. replacing colour with typographical changes like bold and italic) would be useful. In other areas, such as the footer where I have a 'Proof status' button, I make sure to accompany colour changes with text.

6.3 Challenges

6.3.1 Learning Swift

At the beginning of development I had almost no experience writing Swift, although I had written a toy 'merge sort' algorithm in December as a taster. I have little familiarity with C; most of my experience is Python and Java (during this MSc). Fortunately, Swift was relatively straightforward to pick up, but being on a steep learning curve meant that my earlier Swift code required more refactoring to become well-formed, idiomatic Swift (greatly helped by the Wenderlich Style Guide, n.d.; Mathias & Gallagher, 2017; and numerous web resources).

Several non-trivial bugs were traced to the difficulty of maintaining state within *for* loops, particularly when nested, and I began to appreciate the power of using a more functional approach. In Figure 19, for instance, a single function concatenates the string output of *makeDefinition()* into the string *body* using a closure. The code is shorter and easier to debug.

```
859             let body = definitions.reduce("") {
860                 x, y in x + makeDefinition(for: y) }
861
```

Figure 19: Functional Programming Example from Baker Street/Examples.swift

6.3.2 Planning vs. Doing

Early on, I had created a risk analysis, a plan for work, as well as lists of functional and nonfunctional requirements (see Section 4.4 Methodology, and Appendix: Log,

3 June). These provided a useful overarching framework, but I soon found that I was not referring to them much during the coding process. Indeed, my supervisor pointed out that my early deliverable gave the impression I was using the Waterfall (McConnell, 2004) method for my approach, when my understanding of the problem domain was not sufficient to create such a detailed plan. As I worked, my main goal was to prototype the application as rapidly as possible. For this, I had a set of features that I considered essential and others that had a lower priority. Overall, this worked out to be a good solution; I spent little time on planning once I'd begun in earnest, and benefitted from the clarity that the initial planning had given me (i.e. what the app should do and how). Because I was not working as part of a team, leaving the plan aside and ploughing on did not put the project at a disadvantage. This was a good strategy because the coding in itself was quite time consuming.

I had intended to make good use of comment-based documentation both outside (in terms of state and class diagrams) and inside (making use of Swift's Markdown-based documentation system). I started well enough with the lexer/parser structs and classes, but soon reached a situation with other parts of the codebase where the refactoring was too constant, and the final deadline too pressing, to properly document things. Some of the documentation for later classes is good (for instance, the detailed notes on how the *scrollViewSync()* function works, see Section 6.2.3 Known bugs), but in other places it is much briefer. Where it is briefer, I tried to make the code 'self documenting', while recognising that this can be an excuse to avoid writing good comments.

I mitigated this further with *protocols*. Swift is considered to take a Protocol-Oriented Programming (Mathias & Gallagher, 2017) approach rather Object-Oriented (McConnell, 2004). Protocols are blueprints telling the compiler that a type, such as a struct, agrees to conform to a protocol. Having made extensive use of protocols in the source code, these should also serve as a form of documentation.

Ultimately, I consider the code base to have an adequate amount of documentation, though it could be clearer and more extensive.

6.3.3 Versioning

I started to use full version control only towards the end of the development

process, largely because much of my work in the early stages focused on code common to various parts of the app and it made sense to work on a single branch. Plus, Xcode has a way of dealing with version control that I found difficult to grasp initially; I relied on my normal hourly Time Machine backup strategy to access old versions as necessary. Once the app was mature, I learned more about Xcode's version control integration and started using branches to work on different features (see Section 4.4.5 Source Control). Thus, I made extensive use of branches for later features/bugs, but did not miss this functionality early on.

6.3.4 Testing

6.3.4.1 Test Driven Development

This is considered by some to be A 'gold standard' for development (McConnell, 2004). Using this method, tests are written first, fail, and then the code is adapted to make the tests succeed. Prospective tests then serve as a roadmap for development.

I had intended to follow this approach, but there was frankly too much overhead. I tried to plan out in detail what architecture I would need—classes, structs and functions—but this was too abstract for me to properly anticipate what the roles and responsibilities of each would be. It felt effective for me to write the code, then refactor when opportunities for DRY and KISS (see Section 4.4.1 Software Engineering Approach) presented themselves. I wrote tests afterwards, focusing on extra tests for bugs or edge cases.

The inference rules could flag errors to the user in many different cases (see Figure 9). Unfortunately, a given error might be 'protected' by other errors that would be flagged first; so it was often difficult to craft a proof that would cause a certain error to be produced. This means that several (perhaps the majority) of inference-specific errors have not been tested.

6.3.4.2 User Testing

Ideally, I would have had several beta testers look at the app during development. This would have been possible—i.e. recruits might have come from several places, including the current MSc cohort as well as interested online communities (e.g. https://www.reddit.com/r/logic)—but would have introduced more overhead that, during development, I did not feel would have helped the app enough. Throughout, I had a strong idea of how the app should work.

My user testing largely involved myself (i.e. a student who would want to use such software) and my wife, who is interested in logic but has not been formally taught natural deduction. This, added to the straightforward nature of the app in terms of UI interactions, and my unit testing strategy, felt like a reasonable compromise given my resource constraints.

6.3.5 Deal Breakers

There were two features that I considered 'deal breakers' in the sense that, without them, I did not think the app would be considered releasable as anything other than a toy implementation. That is, there were two essential user interface behaviours that I considered essential for a basic app like Baker Street: *undo* and *scroll synchronisation*.

6.3.5.1 Undo

For any document-based app, the user will expect and require a history of their typing actions. That is, triggering an undo should take them meaningful one step back through their interactions with, for instance, a text field. In Baker Street, two actors have the ability to write to the central statement view: the user and the app itself (following, for instance, validation, where the text is tidied and coloured). I had an ongoing problem that app-initiated writes to the text field were not being included in the OS-level mechanisms responsible for maintaining undo behaviour. This took a long time to get working, partly because the text view, being an element designed for user interaction, does not easily expose the appropriate methods. This was a case of 'fighting the API', where my use of an element ran

somewhat counter to its intended function. An exacerbating factor was the limited documentation on my intended use-case. In the end, this was solved calling a special method in the text view to provide it with 'before' and 'after' text that could be passed on the OS-level undo manager. Without this, the app would have had unpredictable and confusing undo behaviour.

6.3.5.2 Scroll Synchronisation

It was important to match the vertical position of all views (line, central text and advice view). I spent several days on this over the course of the project, failing each time, until succeeding when much of the remainder of the app was mature. The difficulty lay in (i) identifying which method would tell me that the contents of a scroll view had moved, (ii) learning a design pattern based on notifications[1], and (iii) noting the 'before' and 'after' scroll coordinates and the position of the user's caret in the central text view. After I had a rudimentary form of this working, the app experienced serious runtime crashes due to a recursion—if, for instance, the user scrolled the central view, the app would scroll the others, but these others would trigger scroll in the central view, which would restart the loop. The final implementation involved several functions, some of which needed special treatment to be made visible to the Objective-C runtime (the notification design pattern is not yet fully available to Swift). The code is working reasonably well but had at least one serious bug (see Section 6.2.3 Known bugs) at the time of corpus submission. Without this syncing behaviour, however, line numbers and their statements would become detached; a new user interface would have been required (perhaps using a single text window).

6.3.6 Implementing the Proof

One of the major implementation decisions was how to represent a proof. This went through several iterations and finished in a state that is workable but not entirely satisfactory in terms of Model-View-Controller separation (Matthias & Gallagher, 2017). The architectural paradigm used by Swift is often called Protocol-Oriented

1. In this pattern, objects register events with the Notification Center. Other objects can ask to told via notification when certain events occur.

Programming (Swift, n.d.; cf. Object-Oriented Programming, Mathias & Gallagher, 2017). Protocols, which are similar to interfaces in Java, and heritable, simply describe some functions and properties. Classes or structs (or enumerated types) may then adopt them. In order to understand how a proof is represented, some reference to the protocols used in Baker Street is necessary, and these are provided in Figure 20, along with their inheritance (children point to their parents; thus BKInspectable inherits from BKLine).

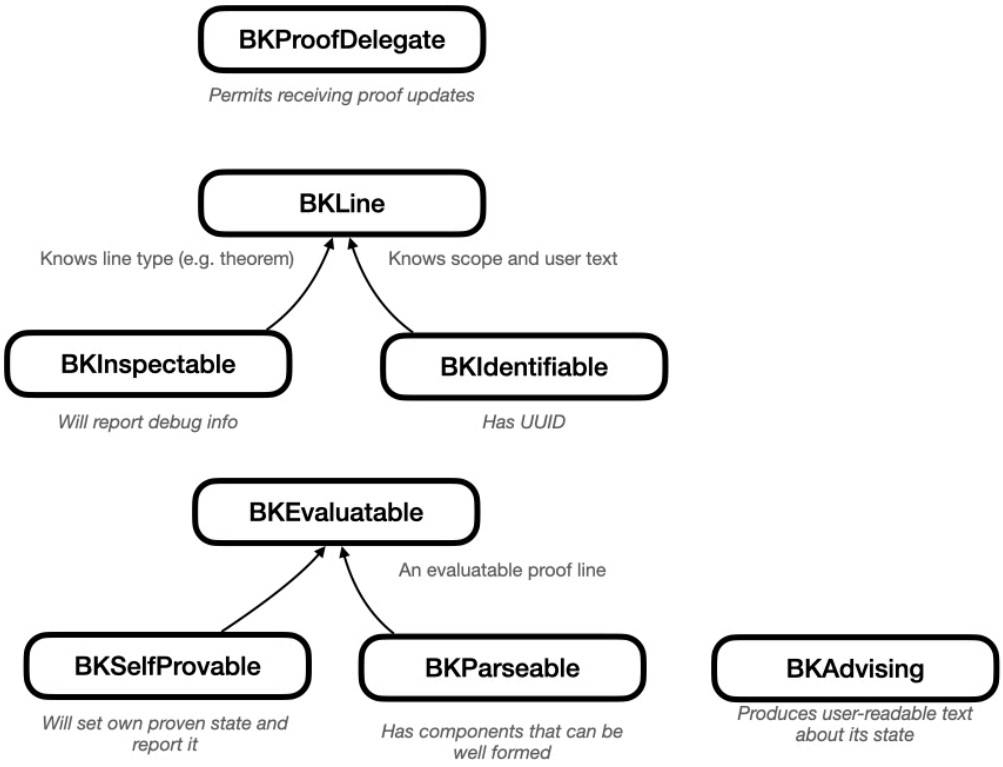


Figure 20: Protocols Used in Baker Street

These protocols were created iteratively in combination with the complex of classes and structs[2] presented in Figure 21. This figure shows the entities as well as the relationships themselves in a manner similar to a UML diagram, which I have simplified for clarity (McConnell, 2004). An entity's box provides its title and responsibilities, while a second box, appended to it, indicates its conformance to

2. It is considered good Swift style to use referenced objects derived from classes when dealing with an entity that has some permanence (e.g. a window, or the proof itself in this case). Structs are value types and better suited to entities that do not have an independent existence (e.g. a helper role).

the protocols in Figure 20. Many of these classes/structs were created within a separate Swift Playground for rapid iteration before being placed into the larger project.

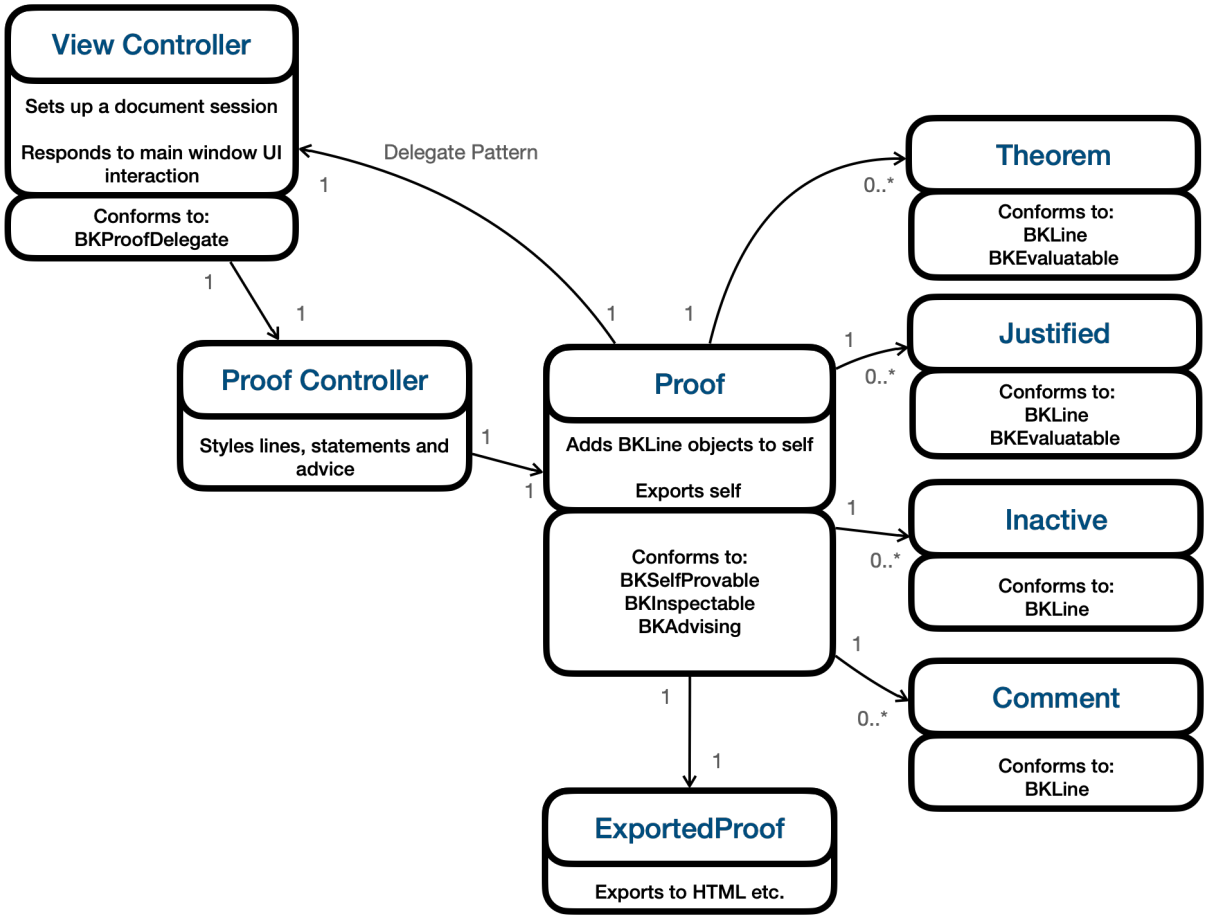


Figure 21: Proof Structure

The general workflow for the proof complex in Figure 21 is as follows: The user interacts with the View Controller (which controls the UI for a given main window) to request proof validation. In a background thread, the View Controller creates a Proof Controller object (passing it the user's proof text), which then instantiates a Proof object using the user text. The Proof object goes through each line, classifies it (as Theorem, Justified, Inactive, or Comment) and creates sub objects as necessary. Theorems and Justified objects have the property of self-evaluation. When all evaluatable objects report that they have completed self evaluation, the Proof creates an ExportedProof; the View Controller is notified of this as a *delegate*[3]

3. To be a delegate means conforming to the BKProofDelegate, which requires the receiver to contain a proofDidCompleteExport(withExportedProof: ExportedProof) method.

and stores the Exported Proof, which is used by the PreviewView Controller to create the floating preview window. The Proof Controller then takes the completed Proof object and creates a view of it (i.e. counts line numbers, collects the syntax-highlighted lines, as well as any advice). When the Proof Controller has finished its instantiation, the view is then made available for the View Controller to update the content of the user interface.

This workflow is reasonably efficient but has the problem that the View Controller receives an Export Proof from the Proof (i.e. one representation of the proof) via the *delegate* pattern and a second, styled version of the proof from the Proof Controller (i.e. another representation of the proof). It would be better to combine these so that the Proof passes its Exported Proof object back to the Proof Controller, which then calls the View Controller with a single object representing the proof. This would better confirm to the Model-View-Controller approach.

So the current architecture does work, but it does not comply with MVC one-hundred per cent. If it did, refactoring would be improved, along with portability and clarity—all important indicators of code quality (McConnell, 2004).

6.3.7 Optimisation

In Baker Street, a user can look up reference materials such as definitions. These definitions contain proofs, all of which are rendered on the fly (principally so that they can be formatted using the same mechanism as a user's proof, providing consistency). In an earlier increment, these were computed at the point of demand, i.e. when a user requested the floating panel by clicking a toolbar button. This led to a delay of approximately 0.1 seconds, or longer—certainly noticeable. The Apple Human Interface Guidelines make clear that the main execution thread should only be used to update the UI, so I refactored the reference windows (e.g. those showing the definitions, etc.) to render on a background thread as soon as a main document window was created, and placed in a cache. When the user requests a reference window, it then appears with rendered contents without delay. (On the topic of optimisation, the proof itself is also being calculated in a background queue; Baker Street shows a status spinner to keep the user informed, see Section 5.2 The Three Panels.)

6.3.8 Advice View Panel Animation

The advice view panel on the right hand side of the window will only appear if the proof is incorrect (i.e. there is advice to show). I was not happy with the original implementation of this, in which the panel immediately disappears when the proof is complete. If the user is looking elsewhere, it could lead to a sense that the app has changed but the user is not sure what. I wanted to animate the closing. That is, the panel would 'ease' closed and open over a second or so, clearly indicating to the user that it was changing state. However, I was not able to implement this because of issues with the rendering: the animation itself worked, but the text on the sliding panel was being stretched. Ultimately, I had to abandon the animation. I still think the disappearance of the panel is too abrupt.

6.3.9 Parsing with ANTLR

My formula parser tokenises the formula, converts from infix notation to postfix notation using the Shunting Yard algorithm (Dijkstra, 1961), and evaluates using a standard postfix evaluation algorithm. My original version of this code, which I wrote in the early stages of my Swift journey, was quite monolithic and difficult to test. I came across a third-party framework called ANTLR in a student dissertation (ANTLR, n.d.; see Appendix: Log, 2 June). ANTLR is a mature open source product that will take a simply-expressed finite grammar and return code (in your chosen languages) parsing data according to the grammar. I experimented with it, but, in the end, decided to stick with my own parser. Its code would have been faster but I might not have fully understood it. There was also a problem integrating the framework into Xcode. Finally, it would have created a dependency, which I wanted to avoid. Ultimately, creating my own lexer/parser worked well enough.

6.3.10 File Format

Originally, I had intended to use the JSON specification for Baker Street data files (Introducing JSON, n.d.). This would have had the advantage of simplicity from a programming point of view, since Swift provides built-in support for exporting variables to JSON, and importing them back (see Appendix: Log, May 28).

However, while the output file contents were human readable to an extent, they were not as readable as the Baker Street markdown in the main statement view of the app. I decided to use the latter for the file format, despite this leading to problems stemming from the fact that I could not write extra information to the file, such as explicit scope level for a statement. Instead, the file would need to be read and evaluated, and variables like scope re-computed. This was a good decision in retrospect. Baker Street data files remain plain text, easily understood, small, and portable. They will be useful long after the app is discontinued.

6.3.11 MacOS Versions

The version of Baker Street submitted for this dissertation works only with macOS Catalina (10.15) and onwards. I did look into expanding support back to Sierra (i.e. 10.12, released in 2016), but when I set the build target to this version of macOS, there were too many API errors for me to fix by the deadline. (Support for Sierra and subsequent versions was added shortly after the deadline.)

6.3.12 Theorem Provability

Late in development, in August, I began the implementation of a feature that would warn the user if a theorem could not be computed. In other words, if a theorem was not provable, it would not be possible for its overall proof to be provable (unless it is considered a redundant part of the proof). This was quite challenging to implement. First, my understanding of what makes a theorem provable needed several exchanges with my rather patient supervisor to iron out. Second, it involved rearchitecting the formula evaluation code to produce truth tables. Third, my original formula evaluator only produced a 'well formed' Boolean result; it needed to be changed, non-trivially, to evaluate to a semantic value, i.e. *true* or *false*. Fourth, I wrote an extensive new class that attempted to compare multiple truth tables by joining them in the manner of an SQL query; this was, really, a blind alley because it was a buggy implementation of an IF connective. I scrapped this code and replaced it with a single, flawlessly executing instruction to create a 'super' formula from several others using an IF connective, passed to the Formula struct I'd already written and tested.

This code worked in the end, but the lessons to take from it are clear: understand what the feature is meant to be doing (i.e. define it), and think about what other aspects of the code can be re-employed.

7 Conclusion and Future Work

7.1 Overview

Following the brief and the issues outlined in the literature review, Baker Street helps proof production by reducing the cognitive load on the user via a simple interface with clear indicators of correctness such as syntax highlighting. It provides simple, to-the-side documentation, and contextualised, constructive help designed to guide the student towards a correct proof. It never interrupts the student with modal dialogues. It should be useable without training; compared to Java or web-based apps, it is fast, intuitive, responsive, builds on native device functionality, has offline storage, adequate security, export facilities, and data longevity.

7.2 Future Work

Solve the Proof and Provide Hints

At present, Baker Street makes no attempt to produce a proof solution. This would be useful because the app could then provide a hint about the next step. For instance—and on the proviso this could be for one solution of several—the app could indicate the expected number of subproofs needed, assumptions, and inferences. It could then reveal these one at a time, upon request. The app should be careful not to reveal the final proof too easily, however, since this may detract from the learning experience.

In this vein, it is sometimes the case that a student will attempt to justify a formula unnecessarily; that is, the formula can be derived from information already present elsewhere in the proof (Olaf Chitl, personal communication). A warning to this effect would be useful.

Reduce Reliance on Colour

It has already been noted that Baker Street communicates important information

using colour. An accessibility preference that allows the user to switch to non-colour mode (i.e. replacing colour with typographical changes like bold and italic) would be useful. See Section 6.2.4 Accessibility.

Printing

This is a standard feature of most macOS apps and should be included.

Increased Grain of Contextualised Help

On June 10th (see Log) I started work on a feature that would take a syntactically incorrect formula and tell the user which part of it might need correcting. This was abandoned in the drive to complete other, core features—bearing in mind that an incorrect formula is still indicated as such—but this would be a useful teaching aid.

Animate the Advice View Panel

This panel disappears when the proof is incorrect, and the disappearance is too fast. The app would benefit from an animation to make clear to the user what is happening (see Section 6.3.8 Advice View Panel Animation).

Reduce the Effort of Writing Justifications

Instead of requiring the user to write:

```
p -> q : Assumption (0)
```

The user could write this, or something similar:

```
p -> q ass 0
```

Custom Rulesets

This was suggested by a user on Reddit (link to post). At the moment, the logical connectives and their truth tables (as well as semantic values) are hardcoded into the app. It should be possible to describe arbitrary rulesets.

Extensibility

It is common in Xcode applications to use bespoke frameworks (i.e. frameworks beyond those that tend to be used as standard, such as the foundation framework that is imported at the beginning of most Swift files in Baker Street). To facilitate an iOS version of the app—and to simplify the Mac version—it would be useful to spin off the proof-validation code into a framework, which can then be imported. Such a framework would then become a dependency. A framework manager like CocoaPods would then be need to manage the dependency (CocoaPods, n.d.).

8 References

American Psychological Association (2020). *APA Style.* Retrieved September 3, 2020, from https://apastyle.apa.org.

ANTLR (n.d.). *ANother Tool for Language Recognition*. Retrieved September 4, 2020, from https://www.antlr.org.

Apple Accessibility (n.d.). Retrieved September 4 from https://www.apple.com/uk/accessibility/mac/.

Apple Human Interface Guidelines. (n.d.). Retrieved June 26, 2020, from https://developer.apple.com/design/human-interface-guidelines/

Baker Street GitHub Repository (2020). Retrieved September 1, 2020, from https://github.com/OolonColoophid.

C (programming language) (n.d.). In *Wikipedia*. Retrieved June 26, 2020, from https://en.wikipedia.org/wiki/C_(programming_language).

CLOC GitHub Repository (2020). Retrieved September 2, 2020, from https://github.com/AlDanial/cloc.

CocoaPods (n.d.). A dependency manager for Swift and Objective-C Cocoa projects. Retrieved September 4, 2020, from https://cocoapods.org.

Dijkstra, E. W. (1961). *An ALGOL 60 Translator for the X1*. ALGOL Bulletin Supplement, 10.

Gruber, J. (2004). Markdown. Retrieved September 1, 2020, from https://daringfireball.net/projects/markdown/.

Isted, T. (2010). *Beginning Mac Programming*. USA: The Pragmatic Programmers.

Introducing JSON (n.d.). Retrieved September 4, 2020, from https://www.json.org/json-en.html.

Hahn, J., & Bussell, H. (2012). *Curricular use of the iPad 2 by a first-year undergraduate learning community*. *Library Technology Reports*, *48*(8), 42-47.

HTML (n.d.). In *Wikipedia*. Retrieved September 7, 2020, from https://en.wikipedia.org/wiki/HTML.

Kaur, D., Koval, A., & Chaney H. (2017). *Potential of using iPads as a supplement to teach math to students with learning disabilities. International Journal of Research in Education and Science (IJRES)*, *3*(1), 114-121.

Knuth, D. (1997). *The Art of Computer Programming* (3rd Edition). Reading, MA, USA: Addison-Wesley.

LaTeX – A document preparation system (2020). Retrieved September 7, 2020, from https://www.latex-project.org.

Lemonde-Labrecque, G. (2020). *Sentential Logic Truth Tree Solver*. Retrieved September 6, 2020, from http://gablem.com/truth-tree-solver/sentential-logic.

Lévy, M. (2019, December 2). *Natural Deduction*. Retrieved September 6, 2020, from http://teachinglogic.liglab.fr/DN/index.php.

Logic. (n.d.). In *Wikipedia*. Retrieved June 26, 2020, from http://en.wikipedia.org/wiki/Logic.

Mathias, M. & Gallagher, J. (2017). *Swift Programming: The Big Nerd Ranch Guide (2nd Edition)*. Atlanta, Georgia: Big Nerd Ranch.

McConnell, S. (2004). *Code Complete (2nd Edition)*. Redmond, USA: Microsoft Press.

Nayuki (2018, July 26). *Propositional Sequent Calculus Solver.* Retrieved September 6, 2020, from https://www.nayuki.io/page/propositional-sequent-calculus-prover.

Natural deduction proof editor and checker (n.d.). Retrieved September 6, 2020, from https://proofs.openlogicproject.org.

Nissanke, N. (1999). *Introductory Logic and Sets for Computer Scientists*. Harlow, Essex: Addison-Wesley.

Oracle Java. (n.d.). Retrieved June 26, 2020, from https://www.oracle.com/uk/java/

Ray Wenderlich Style Guide (n.d.). Retrieved September 1, 2020, from https://github.com/raywenderlich/swift-style-guide.

Schwarz, W. (2020, July 4). *Tree Proof Generator.* Retrieved September 6, 2020, from https://www.umsu.de/trees/.

Swift. (n.d.). Retrieved June 26, 2020, from https://swift.org

Van Ditmarsch, H. (1998). User interfaces in natural deduction programs. In R. C. Backhouse (Ed.), *Informal Proceedings of the Workshop on User Interfaces for Theorem Provers* (Eindhoven, The Netherlands, July 13-15, 1998) (pp. 173-180). (Computing Science Reports; Vol. 98/08). Eindhoven: Technische Universiteit Eindhoven.

8.1 Original Brief

An Education Tool for writing proofs in CO884

Individual

MSc programmes: all

Difficulty: Flexible, at least medium

Requirements: CO884, good programming skills

In CO884 several different proof methods for propositional logic are taught:

- algebraic equivalence
- natural deduction
- resolution

All exercises are currently being done with pencil and paper. The aim of this project is to choose one of these methods and build an educational tool that supports writing proofs using that method.

The tool provides questions of various degrees of difficulty. The user comfortably enters a proof; the tool checks it and explains mistakes. Note that there exist many correct proofs. So the tool cannot simply compare the students answer with a given solution, but must parse the formulae given by the student into abstract syntax tree representations and apply proof methods itself. A good user interface is essential for this project. The user experience should be similar to writing a proof on paper. Note that a user interface aims to prevent a user from making mistakes, whereas an exercise tool gives freedom to make mistakes. You have to decide how to resolve this conflict. If the project time allows it, the generation of hints on how to complete a partial answer could be considered.

❖

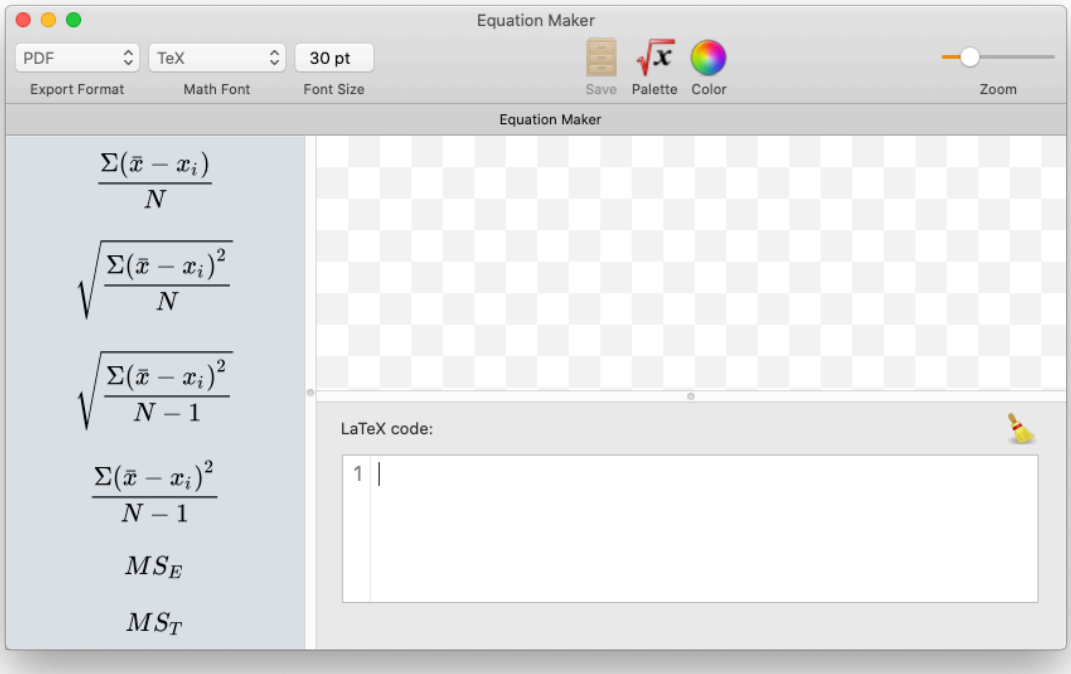
The tool should run on different platforms and be easy to distribute. A Java application will meet that requirement. An alternative is a web page with embedded Javascript, which would make distribution and platform-independent use even easier.

8.2 Initial Design Notes

Initial design

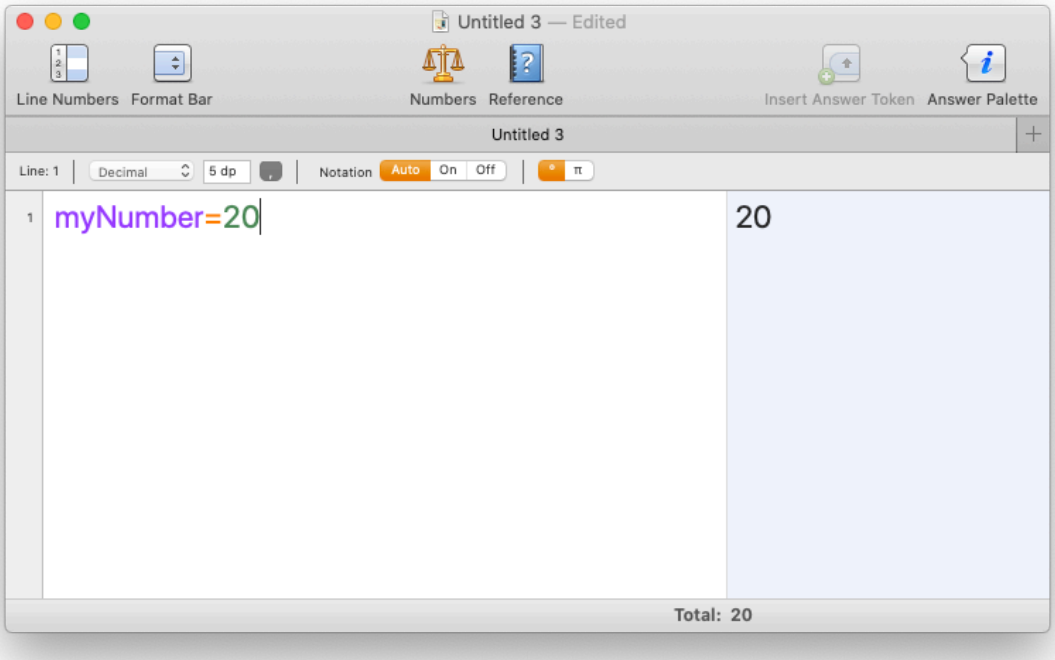
Proof Navigator View

On the left, a list of deletable prior proofs (just the main statement visible) would be nice. A little tick if the proof is complete. This would represent a workspace of all proofs (no actual saving of individual files). (cf. Equation Maker):

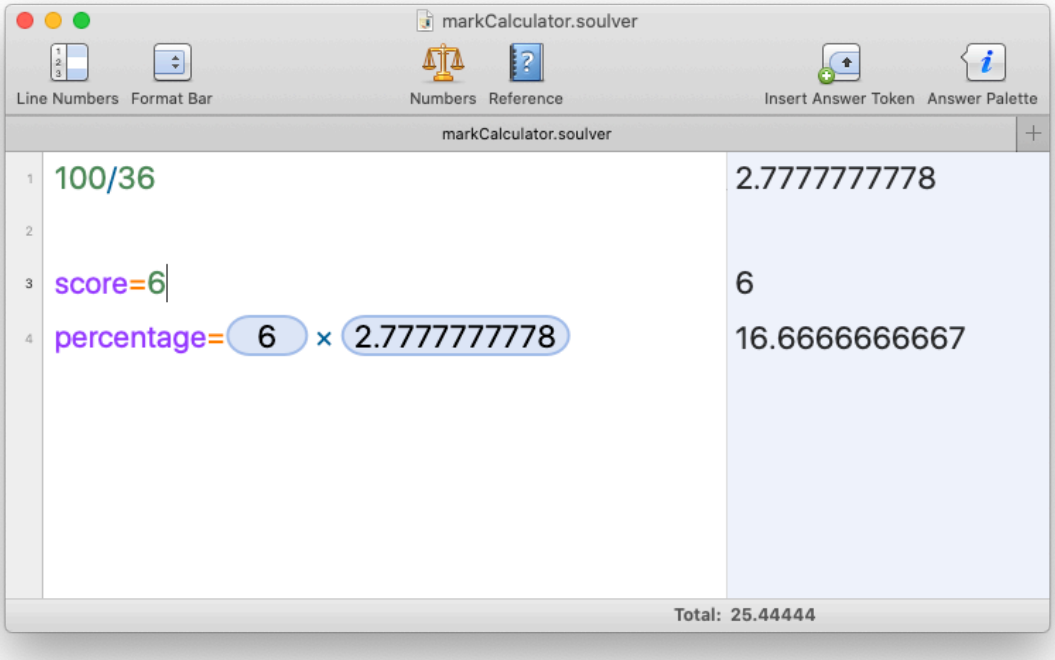


Proof Main View

In the proof window, parse it line by line. Colorisation something like Soulver:



Line numbering like Soulver would also be a good idea:



In the main window, we should have indentation for a subproof, much like Xcode or any other program that indents code:

```
21
22  // Can report whether itself is proven
23  protocol BKSelfProvable {
24      var proven: Bool { get set }
25      var scope: [BKLine] { get set }
26
27      mutating func setProven()
28      func getProven() -> Bool
29
30  }
```

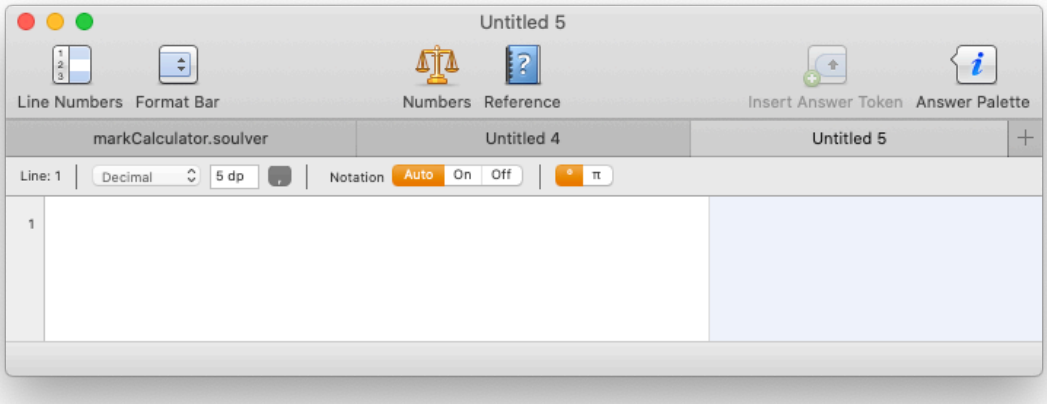
Showing the user feedback is important. Perhaps a spinning wheel while linting.

Toolbar

We should be able to give a local hint somehow (i.e. say what a 'legal' next step would be). What if the inference rules were in a bar across the top, and those that don't fit would be absent? Some system for entering the terms that would fit the particular inference rules also needs to happen.

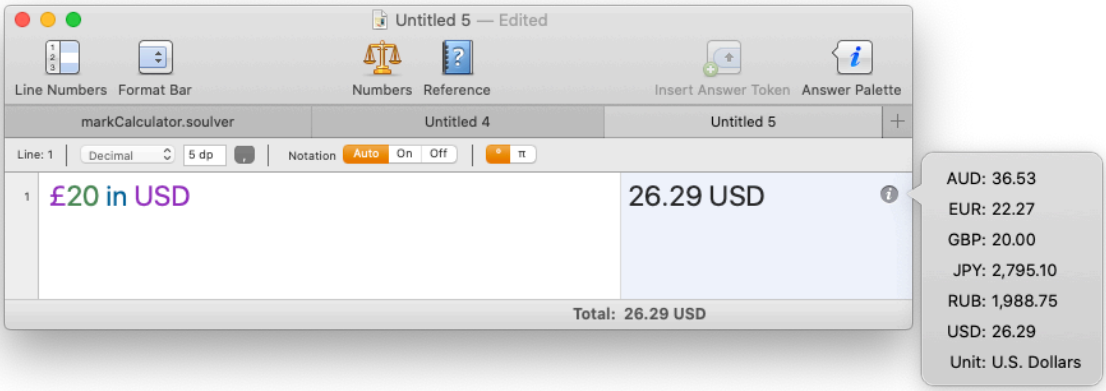
Tool to export proof to PDF via Latex.

Tabs would be nice, like Soulver, with a different proof in each.



Justification View

In the right-hand info window (adjustable size), show justifications somewhat in the style of Soulver:



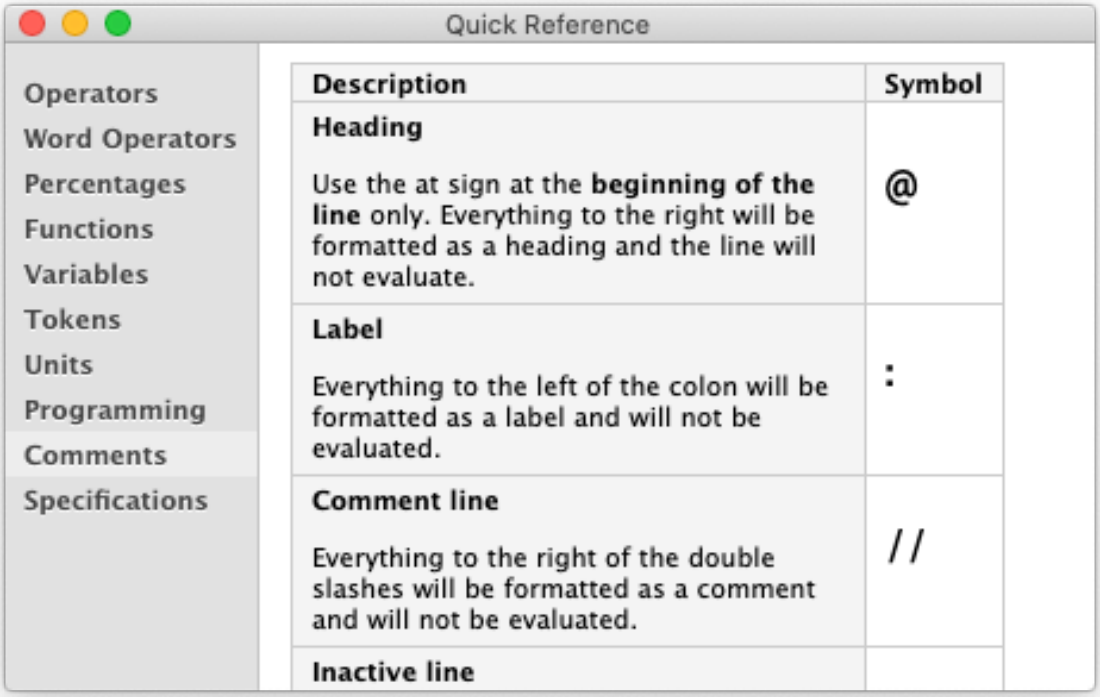
This could have a tick for those lines that are 'good'.

Use a palette for people to select variables, connectives and perhaps inference rules. Like Equation Maker's palette:

Errors could be little warning triangles in the column to the left hand of the particular view (e.g. left of statements)

Reference Materials

The 'Reference' toolbar is a good way to present materials to the user:



Interaction Notes

Some thoughts on interaction:

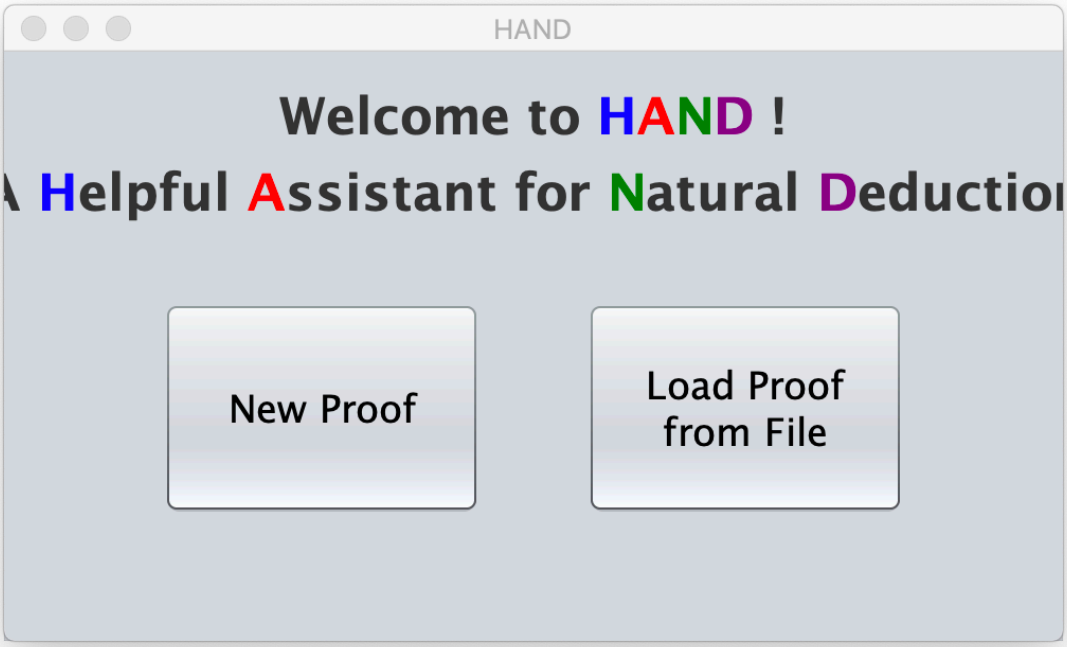
1. User types in text box (freely in main view; right panel has dropdowns of possible next steps, justifications and lines)
2. The TextController checks:
   - Is there a well formed proof statement?
     - If not, stop processing, indicate error to user
     - If so, and there is an error flag, clear the flag
   - Are all formulae, subproofs and justifications well formed?
     - If not, stop processing, indicate flag error lines

Poorly formed formulae should lose their syntax highlighting.

❖

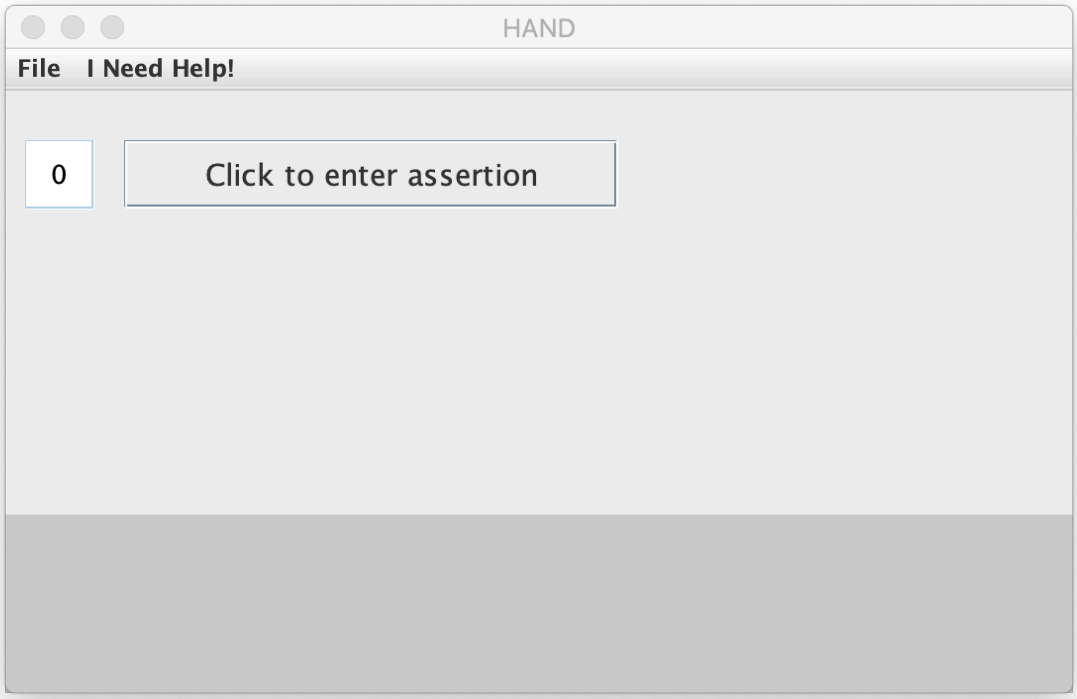
8.3 HAND - in depth

When HAND first starts, the user is presented with a window that asks whether they wish to begin a new proof or load from a file:



This is useful but it would be better to start with a new proof; the user can then load a proof from a file if needs be. (Note that the second line of text is clipped on my Mac, which may well be a function of using Java.) At this point in the app, there is no application-specific menu system available, which can be problematic from an accessibility as well as design point of view. There is only a menu called 'Main'. Within this, if we click 'About', we see are told that the application is 'java Version 1.0 (1.0)'—this is likely to be placeholder text created by the author's Independent Development Environment (IDE). It's important for a user to have information about a running program before interacting with it (ref).

Having clicked on 'New Proof', the user is presented with this window:



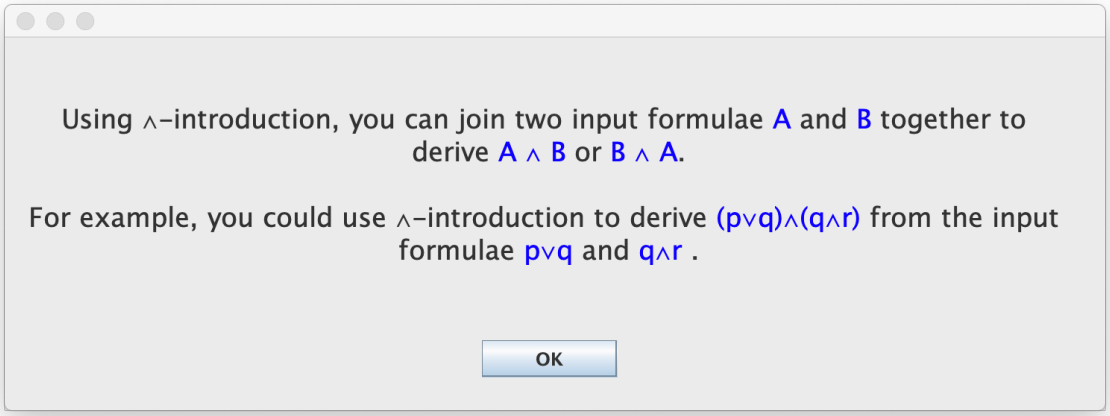
This is clean, minimalistic interface that allows the user to concentrate on the proof itself. Throughout, however, there is no menu system at the top of the screen other than 'Main'; the menus on this application are in a toolbar area at the top of a window (which is not consistent with Apple's Human Interface Guidelines, ref). The menu system itself contains:

- File
    - New Proof
    - Open Proof
    - Save
    - Save As
    - Print
    - Exit
- I need help!
    - Help

These are simple and clear. However, the keyboard shortcuts do not work on my iMac when the menu is opened; furthermore, because they are based on the 'control' key as a modifier, they will conflict with standard command line keyboard shortcuts. On the Mac, the key space is divided, typically, into Unix and Mac; Mac

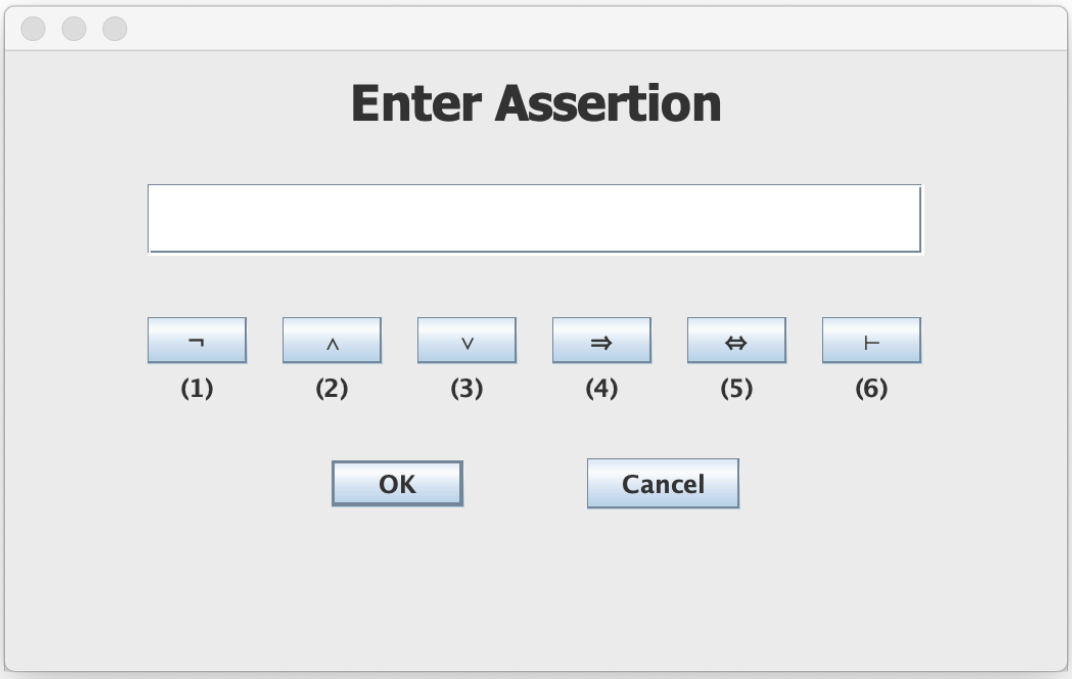
shortcuts will begin with the 'command' key modifier. The help system in HAND is well constructed in its simplicity, though some things could be improved. For instance, the contrast ratio between the centre test and its drop-down box is quite small, making this difficult to read and therefore a challenge for accessibility:

'Next option' is selected, though this is an option itself. When the drop-down is activated, the user can select any inference rule. These present the user with separate windows that provide information (in a font considerably larger than the system font) about the rule:



The Value of Free Form Input

On the main window, when one clicks 'Enter Assertion', a new window appears:



This is a well laid-out, clear interface that works well. However, the user is constrained to enter one line at a time. This can be contrasted with other language interpreters. For instance, here is XCode:

```
8
9  import Foundation
10
11 /// A logical operator associativity type, e.g. left associative.
12 public enum OperatorAssociativity {
13     case leftAssociative
14     case rightAssociative
15 }
```

In the above example, the text itself isn't important, but the paradigm used by XCode (and many other language interpreters) is that the user is free to code and make mistakes. These are then highlighted and the user given the opportunity to correct then. For example, we can introduce a mistake into the code above:

```
9  import Foundation
10
11 /// A logical operator associativity type, e.g. left associative.
12 public num OperatorAssociativity {          4  Closure expression is unused
13     case leftAssociative                      Enum 'case' is not allowed outside of an enum
14     case rightAssociative                     Enum 'case' is not allowed outside of an enum
15 }
16
```

Only then can we see our errors. This permits the user to view the text area as more of a 'playground'. If the input is constrained, as is the case with HAND, entering a proof can be a laborious process, even after several attempts at proofs.

In the XCode examples above we can see *syntax highlighting*. This is way of representing meaningful units in a stream of text using colour. It can help the user to learn a grammar (not so important with the limited grammar of logical formulae) and indicate quickly that a 'sentence' - in this case, a formula - is well-formed.

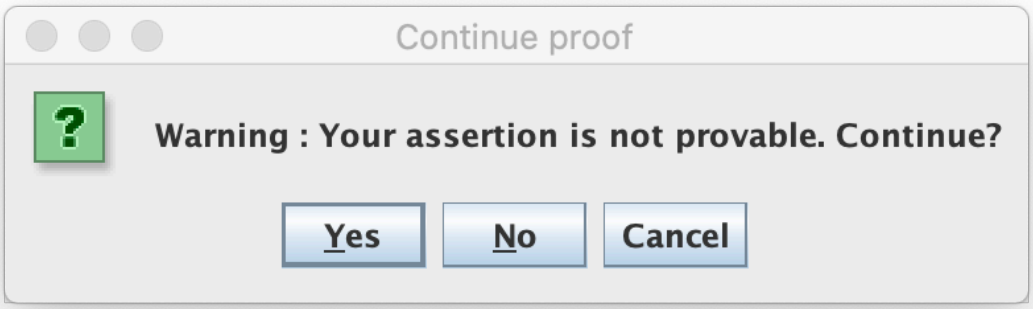
Relatedly, one issue with entering logical symbols is that they don't appear on a standard keyboard. One solution is to use numbers that cause the appropriate symbol to be inserted, which is the approach taken by HAND. Another is to use a form of shorthand, or markdown, that can represent the symbols. For instance, AND for the 'and' operator, ~ for the 'not' operator, and and so on.

The HAND structured input requires the user to enter line numbers for each assertion. Line numbering is not, in itself, something that requires user input, and

adds to the overhead that user (who is trying to solve a proof). For each line, the user must select, using the mouse, the type of line (e.g. a subproof); this is another comparatively labour-intensive process involving at least two mouse clicks. An improvement would be to have the user achieve this without leaving the keyboard. Another restriction at this point is requiring the user to select the line type before they enter their assertion, despite the assertion being to the left of the line type; this is counter to the left-to-right ordering common in most interfaces. Avoiding this constraint would be useful.

Feedback

HAND provides useful feedback as the user progress through the proof. For instance, if an assertion is entered that is not provable, the user sees this:

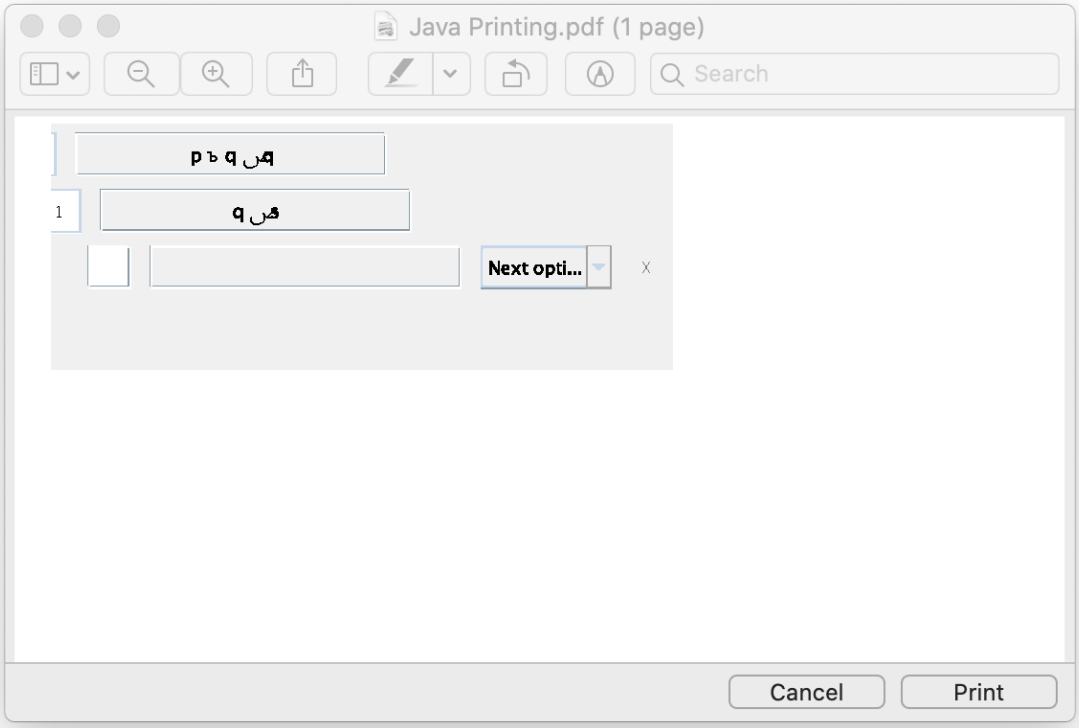


This is indeed useful; however, it is modal (i.e. prevents editing the main window while this popup is open). It would be better to quietly inform the user in the same way a teacher would point out that the student is off track, barring the way. Students would likely prefer a scaffolded (ref) approach, which could indicate:

- what inference rules could make the current assertion possible
- whether the top-level theorem is provable
- how many subproofs might be required

Interoperability

HAND uses a proprietary file format. Without HAND, these cannot be interpreted. If HAND stop working, these files will be inaccessible. Unless absolutely necessarily, it would be desirable to have the file formatted as a simple text file. HAND does have a print option, but this is currently not working. Below is an example of a two-line proof:



A printout would be useful as a properly formatted proof as it might appear in a Latex document. Export to Markdown or HTML might also be useful.

8.4 Final Custom Icon Design

App icon

First iteration

Second iteration

For this icon, I adopted the rounded rectangle design recommended for the next major release of macOS (11, codenamed Big Sur).

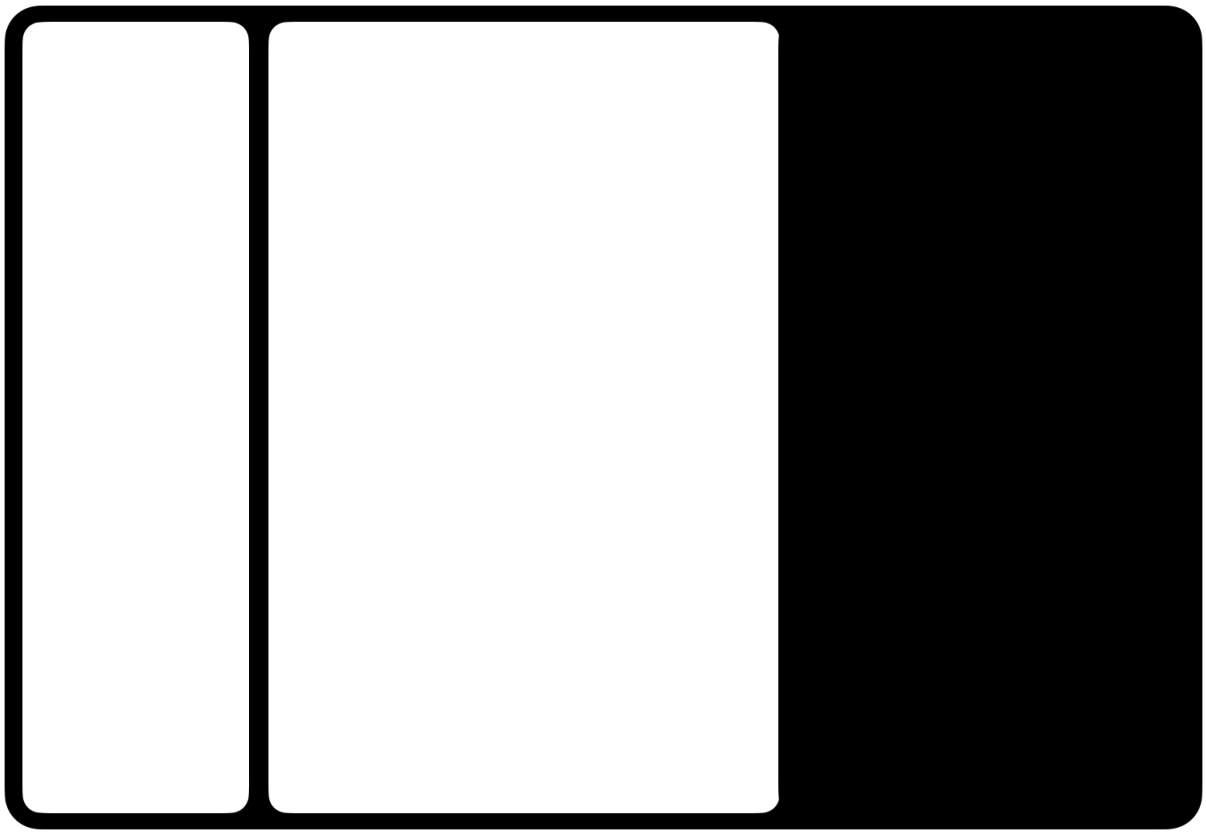
Third iteration

This version reduces contrast and flattens the coloured line elements.

Custom Toolbar Icons

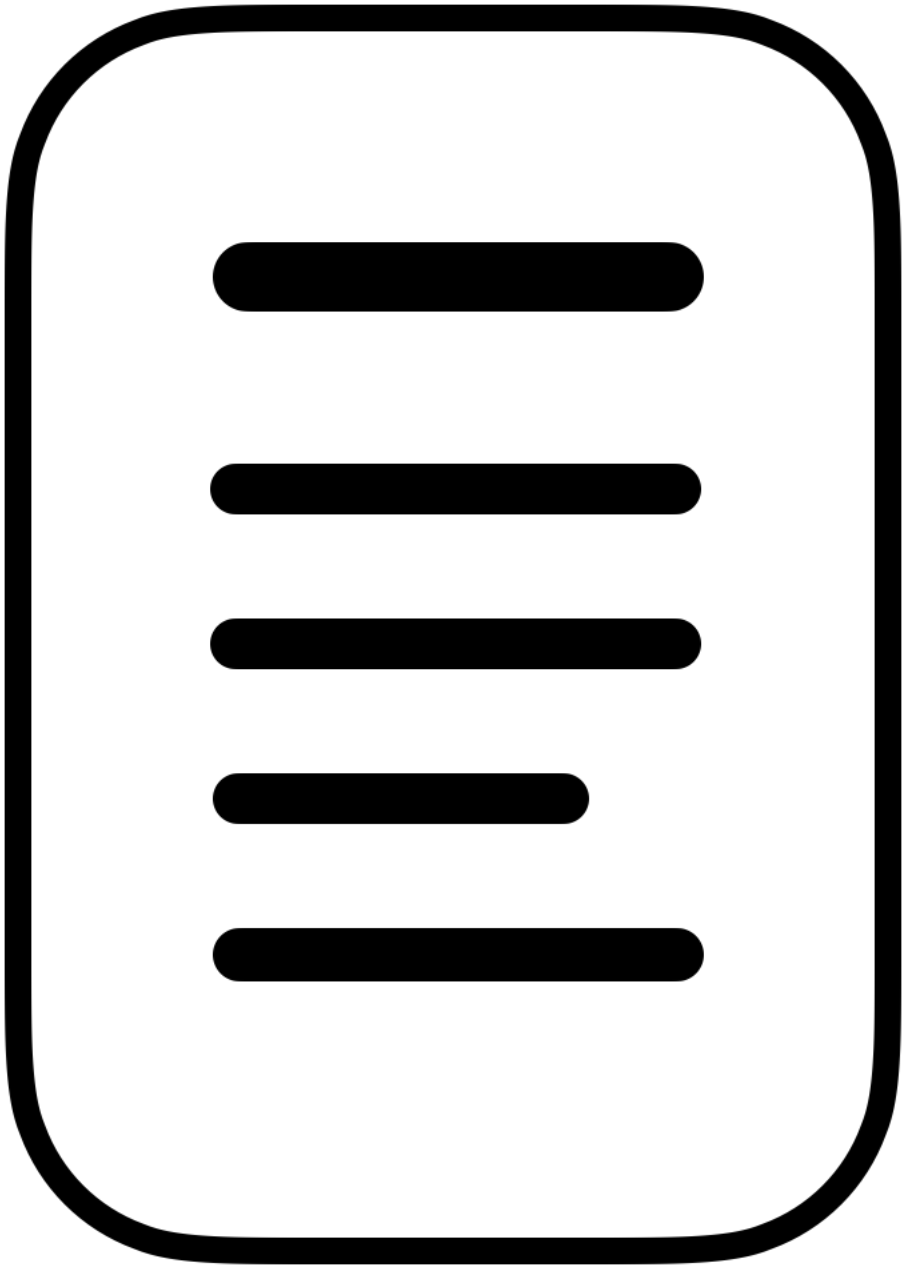
Advice Panel

This icon is designed to imitate the layout of the main window. In Xcode, it is imported as a ‘template’, which means the opaque colour (black, in this case) can take on the highlight colour of the operating system (blue by default).

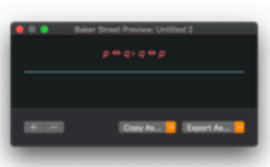


Reference Documents

❖



Baker Street Colour Palette: Foreground

Palette	Used In	Light	Dark
Any Appearance Light Appearance Dark Appearance	General proof statement; table header		
Any Appearance Light Appearance Dark Appearance	Inactive proof line; justification text; table line; table header		
	Statement turnstile; justification number		
Any Appearance Light Appearance Dark Appearance	Advice text; formula operand, operator, bracket, punctuation		

Baker Street Colour Palette: Background

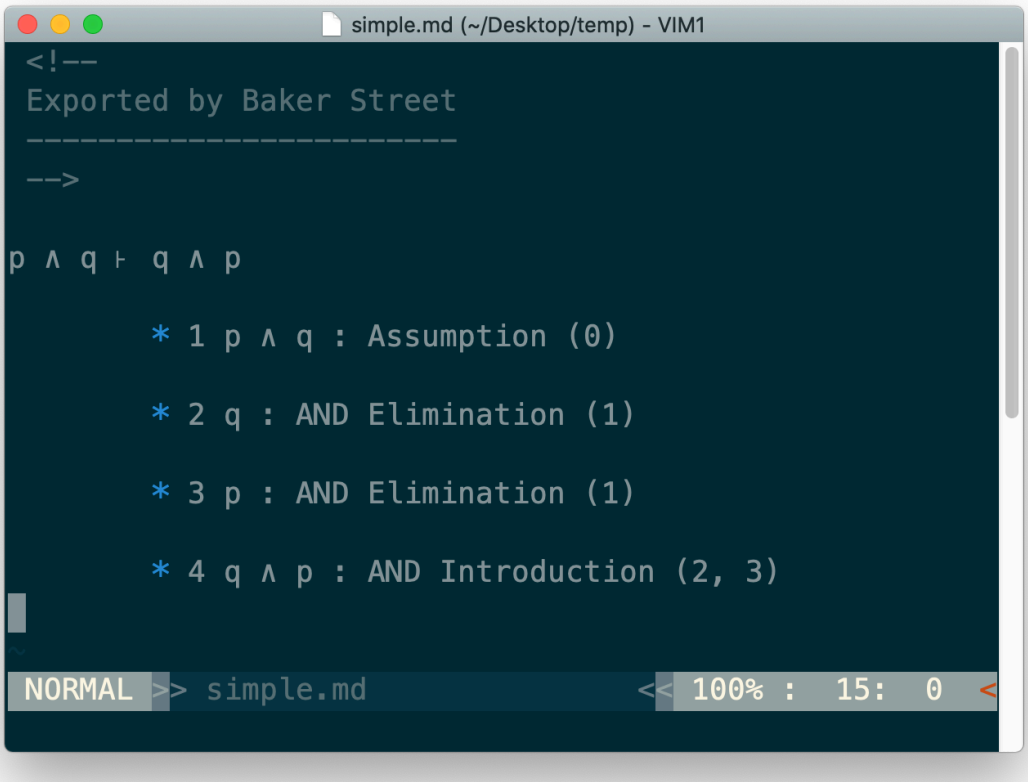
Palette	Used In	Light	Dark
	Advice text		

8.6 Example Proof Exports

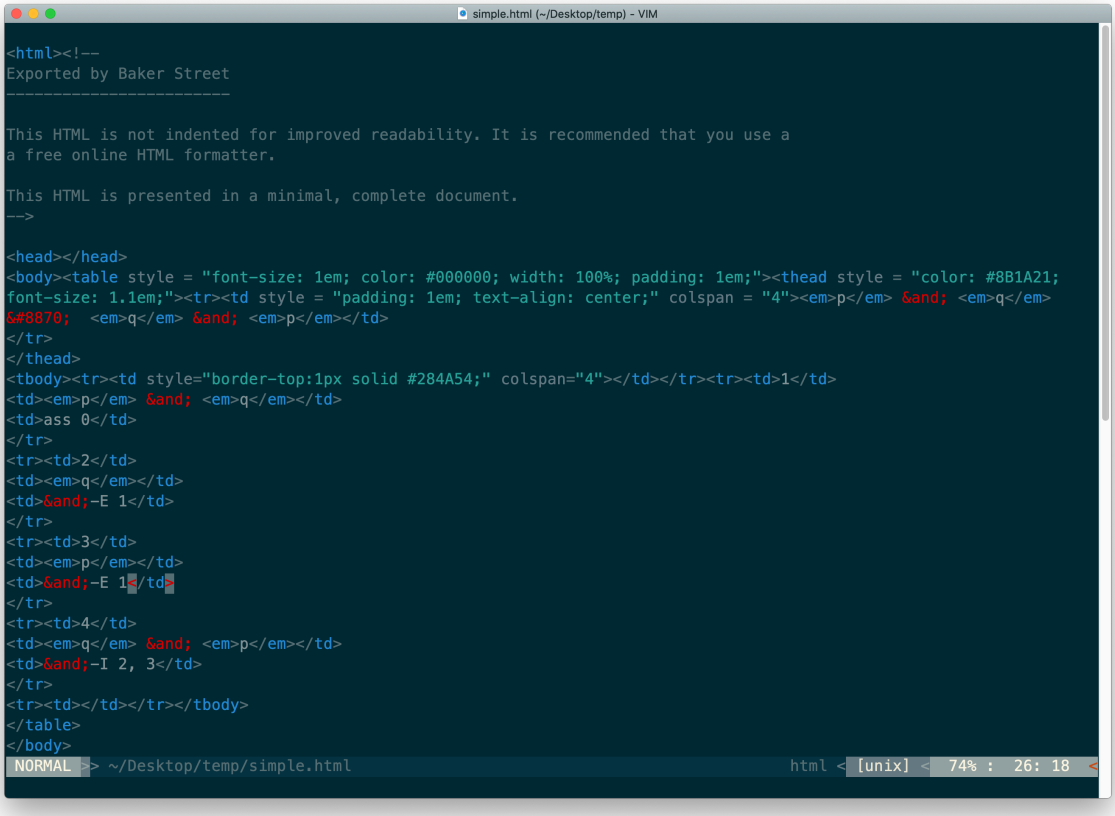
These show the raw export text for Markdown, HTML and LaTeX. Rendering of each

version in native apps is shown in 5.9 Floating Preview Window.

Markdown

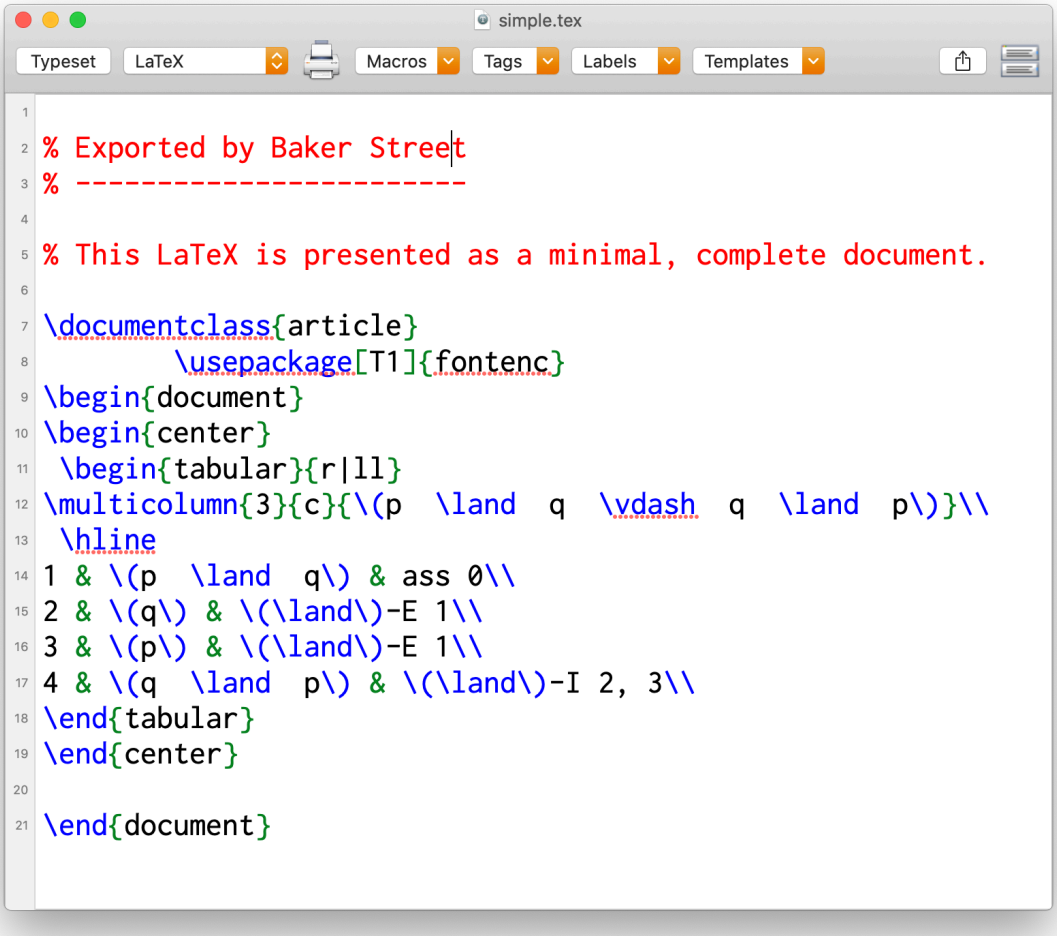


HTML


simple.html (~/Desktop/temp) - VIM
<html><!--
Exported by Baker Street

This HTML is not indented for improved readability. It is recommended that you use a
a free online HTML formatter.
This HTML is presented in a minimal, complete document.
-->
<head></head>
<body><table style = "font-size: 1em; color: #000000; width: 100%; padding: 1em;"><thead style = "color: #8B1A21;
font-size: 1.1em;"><tr><td style = "padding: 1em; text-align: center;" colspan = "4">p ∧ q
⊦ q ∧ p</td>
</tr>
</thead>
<tbody><tr><td style="border-top:1px solid #284A54;" colspan="4"></td></tr><tr><td>1</td>
<td>p ∧ q</td>
<td>ass 0</td>
</tr>
<tr><td>2</td>
<td>q</td>
<td>∧-E 1</td>
</tr>
<tr><td>3</td>
<td>p</td>
<td>∧-E 1</td>
</tr>
<tr><td>4</td>
<td>q ∧ p</td>
<td>∧-I 2, 3</td>
</tr>
<tr><td></td></tr></tbody>
</table>
</body>
NORMAL ~/Desktop/temp/simple.html
html [unix] 74% : 26: 18

LaTeX


simple.tex
Typeset
LaTeX
Macros
Tags
Labels
Templates
% Exported by Baker Street
% ------------------------
% This LaTeX is presented as a minimal, complete document.
\documentclass{article}
\usepackage[T1]{fontenc}
\begin{document}
\begin{center}
\begin{tabular}{r|ll}
\multicolumn{3}{c}{\(p \land q \vdash q \land p\)}\\
\hline
1 & \(p \land q\) & ass 0\\
2 & \(q\) & \(\land\)-E 1\\
3 & \(p\) & \(\land\)-E 1\\
4 & \(q \land p\) & \(\land\)-I 2, 3\\
\end{tabular}
\end{center}
\end{document}

8.7 Baker Street Markdown Reference

Markdown	Meaning	Example	Renders
AND	Logical AND	p AND q	$p \wedge q$
OR	Logical OR	p OR q	$p \vee q$
NOT	Logical NOT	~p	$\neg p$
IF	Logical IF	p -> q	$p \rightarrow q$
IFF	Logical bi-dir IF	p <-> q	$p \leftrightarrow q$
true	Constant *true*	true	*true*
false	Constant *false*	false	*false*
Any lower case letter	Variable	p	p
|-	Syntactic turnstile	p AND q |- q AND p	$p \wedge q \vdash q \wedge p$
(	Open parenthesis	~(p AND q)	$\neg (p \wedge q)$
)	Close parenthesis	(p OR q)	$(p \vee q)$
:	*Introduce justification*	: OR Introduction (2)	: OR Introduction (2)

8.8 Architecture Elements

Note: structs are value types, while classes are reference types.

Proof Controller (Class)

Creates a Proof struct and manages its style.

Exported Proof (Class)

Creates and holds a proof as a collection of RudimentaryProofLines.

Inference Controller (Struct)

Checks whether the inference rules (e.g. AND Introduction) in a proof are successful.

DocumentView Controller (Class)

Displays and controls floating document windows (i.e. for rules, definitions and markdown).

PreviewView Controller (Class)

Displays and controls the floating Preview window.

ImageView Controller (Class)

Displays and controls the floating Rules Overview window.

Preferences (multiple)

Holds global enumerated types, functions and constants related to appearance.

Syntax Styler (Struct)

Returns highlighted formulae and theorems.

Advice Styler (Struct)

Returns styled text for the advice popover views.

Formula (Struct)

Takes a string like “p AND q” and returns a Formula type.

Semantic Permuter (Struct)

Returns semantic permutations (i.e. a truth table) for a formula.

AdviceView Controller (Class)

Displays and controls the popup advice window.

Examples (multiple)

Stores example proofs, text for all floating windows, extends primitive String with HTML convenience functions.

8.9 Tested Elements

Note: structs are value types, while classes are reference types.

Precedence Operator (function)

This function remaps <= and < to refer to operator precedence. For example, NOT should take precedence over IF.

Tree (class)

Stores a formula in a recursive structure where tokenised elements have parent/child relationships.

Lexer (struct)

Takes a string, e.g. “p AND p” and return a flat list of lexicalised tokens.

RpnMake (struct)

Takes tokens and evaluates them.

Formula (struct)

Takes a string, e.g. “p AND p” and returns Formula object (containing a Tree, a syntax highlighted form of itself, a Boolean representing well-formedness, and so on).

Proof (class)

Take a string, e.g.

```
“””
  p OR (q AND r), p -> s, (q AND r) -> s |- s OR p

        p OR (q AND r)              : Assumption (3)
        p -> s                      : Assumption (3)
        (q AND r) -> s              : Assumption (3)
```

```
        s                       : OR Elimination (5, 6, 7)
        s OR p                  : OR Introduction (8)
    """
```

and returns Proof object (containing all Formula objects, an Export Proof, a Boolean representing correctness, and so on).