CROTTS, LARRY JOSHUA, M.S. Construction and Evaluation of a Gold Standard Syntax for Formal Logic Formulas and Systems. (2022)
Directed by Dr. Stephen R. Tate. 61 pp.

Classical logic plays a significant role in computer science where formal proofs eventually make their way into a student's curriculum via discrete mathematics, philosophy logic, or some other medium. We traditionally see propositional logic in Boolean algebra, conditional statements, program and data structure definitions and invariants, and much more. First-order logic likewise exists throughout the world of mathematics, but more prominently in everyday language. Accordingly, a solid understanding of classical logic is paramount. Natural deduction, as the name suggests, is a method of reasoning about an argument using natural intuition, and as a result, it appears quite frequently as a topic of study in introductory logic courses. Due to its relevance, natural deduction intelligent tutors and solvers are widespread on the internet and in the classroom to improve the pedagogical appeal of logic. In this thesis, we present and solve two questions. The first is an open and proposed research question where in we evaluate the efficacy of publicly-available natural deduction tutors/solvers with the prospect of determining what inherently defines understandability and difficulty in natural deduction proofs. In the second question, we investigate the problem of various logic syntaxes that unnecessarily intermingle. With this, we propose a gold standard language for zeroth and first-order logic with the goal and hope of tutor/proof systems adapting to said language to ease the currently laborious task of system evaluation.

CONSTRUCTION AND EVALUATION OF A GOLD STANDARD SYNTAX FOR FORMAL LOGIC FORMULAS AND SYSTEMS

by

Larry Joshua Crotts

A Thesis Submitted to
the Faculty of The Graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Greensboro
2022

Approved by

_____
Committee Chair

To my Viola.

APPROVAL PAGE

This thesis written by Larry Joshua Crotts has been approved by the following committee of the Faculty of The Graduate School at The University of North Carolina at Greensboro.

Committee Chair _____
Stephen R. Tate

Committee Members _____
Insa R. Lawler

_____
Chun Jiang Zhu

_____
Date of Acceptance by Committee

_____
Date of Final Oral Examination

ACKNOWLEDGMENTS

I would like to extend my gratitude and thanks to the esteemed Dr. Steve Tate for not only overseeing and advising this thesis, but also for being a fantastic mentor and professor throughout my time at UNC Greensboro. I sincerely appreciate Dr. Nancy Green for introducing me to the wonderful world of academic-level computer science research, as well as Dr. Insa Lawler in the Philosophy department for introducing me to the exciting adventure that is formal logic and its pedagogical impact. Their unwavering guidance, mentorship, and insight influenced me to pursue graduate school.

Outside of UNCG, I thank my parents for their love and support throughout my education. I also thank two of my best friends: Audree Logan and Andrew Matzureff, for their love, support, and friendship from high school to now.

It also cannot go without saying that I am forever grateful for the support from my loving wife Viola. You never let me down.

Finally, I am deeply indebted to Mr. Tony Smith: my former Advanced Placement[1] Computer Science teacher. Without him, I would not be where I am now. Thank you for seeing (and ultimately helping me reach) my potential.

[1] `https://ap.collegeboard.org/`

PREFACE

The basis for this research stems from my love of teaching. When I took my first introduction to formal logic course as an undergraduate, I was taken aback by its amazing appeal and relation to computer science. From my semesters serving as a tutor/teaching assistant in the Philosophy department at UNC Greensboro, I saw many students that struggled with this material. The problems ranged from its confusing syntax, proof techniques, and esoteric notation. At that time, I thought to myself, "Why not make a tool that helps students understand it better?" Of course, that question had already been answered and deeply investigated across multiple disciplines, but I knew that there had to be more. Once I began my exploration, I quickly realized that online solvers, theorem provers, proof assistants, and similar tools do not have a systematized input format, and testing their algorithms was far more cumbersome than I initially expected and anticipated. This evolved into the desire for a gold standard syntax for both zeroth and first-order logic systems.

Table of Contents

List of Figures

List of Tables

CHAPTER I

INTRODUCTION

1 Overview

Classical formal logic is a subset of philosophy that branches into related disciplines such as computer science, statistics, mathematics, and similar sciences. Logic, however, is taught in non-science fields like communication studies to reinforce critical thinking and improve deductive skills for argumentation. Per the Stanford Encyclopedia on Classical Logic [43], logic is a tool used for studying correct and analytical reasoning in both formal and informal languages. Its existence spawned questions ranging from its use in mathematics as an aid to disambiguate problems and proofs to considering it as an extension to natural language. As Hatcher [18] states, due to an increased viewing of rhetoric and opinion versus factual knowledge in modern social media, the need for strong logical thinking abilities is crucial for evaluating, analyzing, and debating against arguments and claims. Hatcher, likewise, mentions that standard logical deductive forms such as methods of inference and syllogisms serve as critical components for a student's ability to determine the validity of an argument and the relation (or lack thereof) of premises to conclusions. A desire for valid and sound arguments from students constitutes and contributes to wider adoption of formal logic classes in universities, or at the very least, the pedagogy of invalid arguments

on how to refute incorrect and, sometimes egregious, contentions. Formal logic's relation to computer science, in particular, is extensively prevalent when dealing with Boolean logic via sequential and iteration statements, mathematical proofs, set theory, reasoning about program correctness, and so much more. The relevance of logic in our daily lives is unprecedented and sometimes understated.

2 Contribution

We produce several key contributions in this thesis, including:

- FLAT: Formal Logic Aiding Tutor, a software for formal logic pedagogy.
- ANDTaP: Automatic Natural Deduction Tutor and Prover, a web-based natural deduction tutor.
- Comparison and evaluation of three publicly-available natural deduction proof tools.
- Creation of a gold standard syntax for zeroth and first-order logics.

3 Thesis Content

This thesis is broken up into four primary components. Chapter I introduces definitions, background, and our problem definitions. Chapter II reviews the related literature for prior work in this area. Chapter III discusses our two methods of research, being our natural deduction and formal logic tutoring system: FLAT (Formal Logic Aiding Tutor), as well as the creation of a gold standard for formal logic syntax and semantics (i.e., the creation of a standardized grammar for logic language evaluation).

Finally, chapter IV discusses and analyzes the results outlined in chapter III, as well as future work and consequences of our research.

4 Terminology

Before we continue, we will define some terms frequently used in formal logic-related work.

Definition 1 (Logic)**.** A philosophical definition of *logic* is that it is a scheme of deductive reasoning defined by the laws of validity. In computer science, its definition extends this idea to electrical circuitry (i.e., how a computer ought to perform a given task).

Definition 2 (Well-Formed Formula)**.** A *well-formed formula*, according to [40] is a string (formula) of syntactically-correct characters which conform (well-formed) to a language grammar.

Definition 3 (Proposition)**.** A *proposition* is a statement or claim that is either true or false.

Definition 4 (Proof)**.** A *proof* is a style of argument which establishes the truth of a proposition.

Definition 5 (Theorem)**.** A *theorem* is either a well-known and proven proposition, or one that can be proven.

Definition 6 (Natural Deduction)**.** *Natural deduction* [15] is a form of proof where the method of reasoning stems from human, or natural, intuition.

Definition 7 (Syntax)**.** *Syntax* defines the well-formed property of a formula in a language grammar.

Definition 8 (Grammar)**.** A *grammar* is the definition of syntax rules of a language. Largely in computer science contexts, grammars determine whether a string of characters belongs to a language.

CHAPTER II

RELATED WORK

In this chapter, we will discuss the related work and prior contributions to the discipline of natural deduction pedagogy, as well as efforts to modernize and increase its effectiveness for students with less background in, for example, mathematics.

Extending formal logic to a technological education is not a new idea—there exist many online solvers, provers, and programming languages designed to suit the needs of logic students, or those that use formal logic in some manner. In addition to these, we will also mention more powerful theorem provers that are aimed at experts/more experienced users.

Many of the systems we describe below, including our own, are a type of intelligent interactive tutor. A goal of such a system is to mimic the relationship between instructors and their student(s). Correspondingly, if a student starts to struggle with any given topic, the tutor ought to recognize the mistakes made, correct them, and then lead them down the intended path. Similarly, it should provide accurate and customized hints to a problem when requested, even if the student has not necessarily yet gone astray. Taking this a step further may call for dynamic generation of questions tailored to the individual student based on current progress. Automatic tutors and more so intelligent interactive tutors have a host of benefits: reduced stress on teachers

by not having to create unique content for all students (a feat almost impossible unless the class size is appropriately small), and research has demonstrated that students perform better on assessments in one-on-one tutoring sessions [52].

1 Formal Logic Tutors

1.1 Propositional Logic

Propositional logic, which we will interchangably refer to as zeroth-order logic (or in other disciplines as sentence logic, sentential logic, Boolean logic, combinatorial logic, or propositional calculus), according to Hein [19], is a language of propositions that conform to rules. Propositional logic is comparatively simpler than first-order predicate logic described in section II.1.2—it does not use variables, constants, or quantifiers of any kind. Rather, in this language, there are four binary (two-place/two-arity) connectives: logical conjunction, logical disjunction, logical implication, and the biconditional, as well as one unary (one-place/one-arity) operator: logical negation.

Table 1. Common Notation in Propositional Logic

Semantic Meaning	Operator/Connective
Logical Conjunction	$\wedge$, &, $\cdot$, "and"
Logical Disjunction	$\vee$, |, $\|\|$, +, "or"
Logical Implication	$\supset$, $\rightarrow$, $\Rightarrow$, $>$, $\Longrightarrow$ "if", "then"
Logical Biconditonal	$\leftrightarrow$, $\Longleftrightarrow$, $\equiv$, $\Leftrightarrow$, "iff", "if and only if"
Logical Negation	$\neg$, $-$, !, $\sim$, "not"

Because of the reducible nature of propositional logic to simple structures and representations, there exist plentiful online truth table generators that provide detailed and immediate feedback for users while solving problems and well-formed formulas. Further, such generators work well not only for formal logic, but also computer science,

mathematics, and electrical engineering, allowing students to enter a Boolean truth value (i.e., true/false) for an operand or proposition and the computer will determine if it is valid or invalid for an arbitrary cell [13]. An apparent drawback is that they require a student to have prior experience with the underlying logic or preexisting knowledge of entering values into a truth table [27], a sometimes undesired prerequisite. The problem is that many systems are not aimed at teaching, but rather serve as a solution or complementary aid to students or others who have a full understanding of the material.

Lukins et al. [29] developed the P-Logic Tutor: a propositional logic tutor with several key functions including a truth table generator, parse tree viewer, tautology/satisfiability determiners, a built-in theorem prover, all of which are supplemented with learning adaptability that generates feedback for students. It was developed as a Java Web Start (JNLP) applet. In their report, they describe an experiment where students across two discrete mathematics courses evaluated its performance, where they received generally positive reviews with some small limitations that students found cumbersome. Unfortunately, their provided link is now offline, meaning there is no way to investigate either the source code or even use the application in attempt to test it against modern alternatives. One other significant downside to the P-Logic Tutor is that while it covers propositional logic well as its name suggest, it completely lacks support for first-order logic. Additionally, the system had the requirement where students (or any user) had to log in for performance monitoring purposes. This, in turn, severely limits the testability to only those at, in this instance, Wake Forest University.

Another software-based solution (i.e., executable outside the browser) is LEGEND by Vlist [48]. LEGEND is untestable as it is closed-source and unavailable to the

public, but it allows the user to prove and generate proofs from a (simple) given propositional formula.

Lodder et al. [28] created LOGAX, which is a tool for students to learn and construct axiomatic proofs with feedback and hints. While useful in its specific domain, we focus on natural deduction proofs instead of axiomatic proofs for simplicity and approachability for non-computer science students.

Mostafavi et al. [31] have written several papers on their Deep Thought tool with several iterations of improvement ranging from on-demand step hint generation to data-driven approaches to problem generation. Their work largely aims to improve the pedagogy of deductive logic via mastery learning leveling components which dynamically change based on student performance, with higher proficiency in problem solving leading to harder problems e.g., longer proofs, different axioms, etc. Their system also modifies the difficulty of problems to fit a student's mastery level (e.g., if a student has trouble answering a particular proof, they are given an easier alternative). An eventual conclusion reached was as the "intelligence" of a tutor increased, the number of successful students likewise increased.

In [44], Siev describes the Automated Proof Search system AProS: a proof generation system which works in tandem with their proof tutor. It is used in their Open Learning Initiative[1] project at Carnegie Mellon University. Though, their system and pedagogical pipeline strays away from only propositional and first-order logic in favor of theory of computing topics such as Turing machines, computability, set theory, and others. In addition, the links provided on their website are either out of date or incorrect, because when trying to access their *Logic & Proofs* course, it redirects users to the OLI website instead. Plus, the fully-featured tool is not provided, and their

[1] `https://oli.cmu.edu/`

Java Web Start version is not easily accessible in the modern web, where Java in the browser is, to a great extent, deprecated.

In [50], Verwer et al. briefly describe their propositional logic tutor Bop: a Fitch-style proving system. Like many other systems, their provided link is also offline, so we had no way of assessing its performance.

1.2 First-Order Logic

Cerna et al. [6] developed **AX**olotl: a unique tutor due to its Android implementation. **AX**olotl includes several types of proofs and tutorials, though its reliance on a file protocol to load problem sets or questions is a bit cumbersome for the non-savvy student or instructor as they describe. Plus, its curriculum is aimed at computer scientists with what appears to be a strong background in logic, lambda calculus, and type theory.

In [30] and [12], Mauco and Felice et al. show educational software for increasing the appeal of first-order logic to introductory students. With this idea, they developed FOLST and LogicChess. The former, FOLST, is a graphical application that shows an illustrated scene with predicates that define said scene e.g., isSleeping(x), isOnTheGrass(x) when referring to a farm. Students must enter facts that describe the illustration using these given predicates. LogicChess is similar in that students are provided predicates to describe a scene, but with the difference that it the scene is modeled as a chess board with chess pieces. According to their reports, students may modify the model which updates the underlying logic formulas, which may then be checked for validity and satisfiability.

Perikos et al. in [36] developed a web-based system to help students translate sentences from natural language to first-order logic. Specifically, their approach is to

ask students to find verbs, adjectives, and nouns from the text. Then, find connectives, predicates, and quantifiers. Finally, create groups of related predicates and repeatedly connect them until a final formula is achieved. They conclude with a suggestion to the research community that announces the problem of dynamically-generated hints as manually analyzing all possible outcomes of a formula is far too laborious and impractical.

Dostálová et al. [10] describe ORGANON: a web-based tutoring system for both propositional and predicate logic with randomly-generated mutations from a database of problems. Unfortunately, from the time that their paper was written, the system had very limited functionality, only supporting conjunctive/disjunctive/prenex normal form conversion exercises, with others in the works. We could not find any current information about ORGANON.

1.3 Problem/Solution Generators

In this section, we will briefly mention work related to automated problem generation (i.e., automatic creation of problems for students to solve). Practice makes perfect with logic, and textbooks often only provide final answers; not the complete answer with shown work. As such, unmotivated students may merely copy the the answer without retaining any relevant information, an undesired outcome. A tool, or algorithm, that dynamically generates problems and solutions there of serves as excellent remediation or recitation.

In [1], Ahmed et al. created a framework and tool for automatically generated natural deduction problems and solutions. They explain the significance of problem generation on the grounds that it prevents rampant plagiarism, and copyright infringment from publishers. Plus, they reiterate the idea that not only does practice help,

but practice tailored to the student satisfies that role to a greater extent.

Other work has gone into generating algebra practice problems (see [45]), in addition to random boolean expressions for SAT and SMT solvers (see [3]).

2 Specialized Logic/Theorem-Proving Programming Languages

What is a theorem prover? Characteristically, it is a system that uses a combination of built-in rules, axioms, to prove a provided formula or theorem. It searches for an ordering of rules, axioms, and intermediate formulas which satisfy the theorem and show it holds true.

A historical theorem-proving logic programming language is Prolog, which uses Horn clauses, or a sequence of literals separated by disjunctions, to deductively prove goals. It was popularized behind the belief that logic programming ought not to be restricted to human intellect and architecture, but rather be structured around solving provided problems and assumptions that a human may easily understand when reading. Execution of a Prolog program, as hinted above, instructs the system to, satisfy a goal, given relevant background information i.e., facts and rules [46].

Another form of theorem proving which is affiliated with artificial intelligence is inductive logic programming, coined by Muggleton in [32]. In summary, with inductive logic programming, we are provided with a data set of facts and hypotheses to inductively prove an general observation from the specific. A commonly-presented equation is, we have a set of positive examples E^+, a set of negative examples E^-, background knowledge B, and we wish to find a hypothesis H such that $B \wedge H \vdash E^+$ and $B \wedge H \nvdash E^-$ [5].

Coq is a functional programming language that at its core is a proof assistant

[37]. It has been historically used for program verification in compiler optimization with LLVM (Low-Level Virtual Machine) and C, cryptographic security environment reasoning, and even fundamental breakthroughs in group theory and the famous four-color theorem [16]. As it is a functional language, it is trivial to reason about the consequences of a function or a segment of code. That is, functions should not, for example, produce side effects, and thus have the sole purpose of computing a result.

Isabelle [34] is a proof assistant with built-in functional programming that predominately focuses on higher-order logic via induction proofs, lists, and natural numbers. It also allows for inductively-defined sets, and logic proofs similar to those we describe in our report. Finally, it supports lambda calculus expressions, recursion, and proofs there of which parallel to type theory paradigms. As a consequence, it bears resemblance to functional languages e.g., ML, F#, and OCaml. While it is a powerful language and tool, we find it to be bulky, similar to that of Coq's presentation for introductory logic students. Villadsen et al. [51] created NaDeA: a natural deduction system which integrates with Isabelle to generate and verify theorems. Their premise states that students should be provided with a background to formally verify the correctness of software, and NaDeA helps build these prerequisite skills.

Floyd-Hoare logic is a system of reasoning used to prove statements about programs, and otherwise verify the correctness of software. With Hoare logic, we state pre-conditions and post-conditions of an arbitrary segment of code, then prove those conditions. In [37], Pierce et al. implement Hoare logic in Coq, but it originates back to the mid-20th century in [23].

In summary, there are several other theorem provers which all serve similar purposes [42], [26], [35], [33] for program/software verification, hardware verification, formal logic/higher-order logic proofs, type theory, mathematics with set theory, and other

topics related to automated reasoning due to the impracticality of and error-potential human proofs. Because this thesis is limited to natural deduction and introductory formal logic pedagogy, we will omit the remaining discussion of these tools.

3 Standardizing Logic Syntax

Interestingly, the International Organization of Standards has a dedicated section to logic symbol syntax in their quantities and units for mathematics standard [14]. Alas, it only addresses a very small subset of widely-used connectives as shown in Table 2. Plus, even though many other mathematical symbols are standardized and adopted in practice (e.g., set notation), logic notation never received the same level of attention from its audience.

Table 2. ISO Logic Symbols

Semantic Meaning	Operator
Logical Conjunction	$\wedge$
Logical Disjunction	$\vee$
Logical Negation	$\neg$
Logical Implication	$\Rightarrow$
Logical Equivalence	$\Leftrightarrow$
Universal Quantifier	$\forall$
Existential Quantifier	$\exists$

We could only find one attempt outside the ISO at formally standardizing propositional and first-order logic syntax, and it was for a classroom mathematics setting. In [11], Dougherty attempts to standardize logic symbols for his calculus courses. He noted that several symbols are used, sometimes erroneously, and other times in an understating context (i.e., using an implication when a biconditional is better suited). He proposes that the connectives $\rightarrow$ and $\leftrightarrow$ ought to be used when making a small claim

that returns true or false, whereas $\implies$ and $\iff$ are for tautological statements. Dougherty also dislikes implicit precedence, in favor of brackets and parentheses for grouping binary connectives so as to not unnecessarily confuse students, a shared sentiment. Importantly, the premise of his paper is that standardizing logic syntax helps students clarify their arguments and better illustrate the intended idea behind a proof. The downsides are that students may often worry about what symbol to use when rather than focusing on the concepts (we present a similar argument in chapter III with our experimentation of natural deduction software). Additionally, standardizing logic syntax usage in one class is helpful for that individual class, but without a universal formalization, the practicality and portability is thereby limited.

4 Boolean Satisfiability Solver Input Formats

Cook proved that the Boolean satisfiability problem SAT is $\mathcal{NP}$-Complete [7], providing the Cook-Levin theorem used in computational complexity reduction proofs. Boolean satisfiability answers the question, "Given a Boolean logic formula F, is there an assignment of truth values that makes F true?". Because this problem is $\mathcal{NP}$-Complete, there exists no (efficient) polynomial-time solution. Because of the usefulness of SAT with program verification, graph coloring, constraint satisfaction, artificial intelligence, electronic circuitry correctness verification, and more, the need for heuristically-fast SAT solvers was evident. A lecture by Heule and Martins [22] describes several SAT solvers including DIMACS, CaDiCaL, SAT4J, UBCSAT, and PySAT. Though, having a plethora of SAT solvers to choose from is rather meaningless to those who need them–rather, they want to know which one is the fastest. To test SAT solvers against one another, SAT Competitions came to light, as did the benchmark

submission guidelines detailing the required input format [21]. For fair and consistent evaluation, SAT solvers that participate in this competition must use the standardized DIMACS format. Formulas are entered in conjunctive normal form where numbers represent literals/atoms. Each line designates a clause in the formula. Figure 1 shows an example of the DIMACS format alongside its logic (well-formed formula) representation. As reported by the SAT solver Varisat[2], there are several extensions and variants of DIMACS, but the SAT competitions website[3] strictly states that any deviation from the required input and output formats is unacceptable.

```
p cnf 3 2
-1 2 -3 0
 -3 -1 0
```

(a) DIMACS Format

$$(\neg x \vee y \vee \neg z) \wedge (\neg z \vee \neg x)$$

(b) Formula Representation

Figure 1. DIMACS Format Example Input

[2] https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html
[3] http://www.satcompetition.org/

CHAPTER III

METHODS

In this chapter, we explain our evaluation method and metrics for assessing three publicly-available natural deduction systems against our prover. Additionally, we construct a formal definition for a standardized and uniform syntax for writing and, more importantly, testing differing logic systems and algorithms.

1 Evaluation of Natural Deduction Systems

To begin, while plenty of research papers and projects on formal logic tutors, natural deduction proving systems, and proof assistants exist (see chapter II), they are few and far between when viewed from the public, non-academic eye. That is, many only remain relevant in their academic research circle, and have either little purpose or minimal exposure outside to a "real audience". Further, current research efforts focus more on improving their current tool rather than performing direct comparisons with others. The issue with head-to-head comparisons is the mystery of a reliable metric: how do we measure "success" in a formal logic tutor without user evaluation? In other words, what metric is viable for determining the efficiency, or effectiveness, of a tutoring/proving/assistant system for formal logic?

1.1 Experiment 1: A Student-Driven Approach for Difficulty Metrics

All students are not unique, and a formal logic class may contain students across different disciplines. As a result, what is difficult to one student who has yet to see logic may be easier to another student who programs or otherwise works with Boolean operators on a daily basis. Determining what exactly constitutes difficult in a logic class is tricky for the same reason. In other words, it is a non-trivial job to evaluate whether one problem or topic is inherently more challenging than another.

Given the varied opinions of students, we wanted to perform an observational study on our software to determine what kinds of proofs students find difficult, or where they potentially go astray from an expert's solution to a problem. In particular, our goal was to analyze a student's abilities to complete propositional logic natural deduction proofs, picking between two sets of premises and conclusions to say which looks harder to prove, as well as choosing a proof out of two that appears more difficult, with the hopeful outcome of finding common patterns between proofs/formula choices, then analyzing said patterns to understand why those were said to be more difficult.

The study was implemented as two Qualtrics surveys. The first consisted of ten[1] carefully-selected pairs of premises that either prove the same conclusion, or a slightly-altered version of the conclusion. The second survey contained ten[2] pairs of proofs where each proof in the pair derives the same conclusion, but take different approaches to doing so. Finally, we had a third component where students would use our ANDTaP system (see section 1.2.2) to prove ten natural deduction problems[3]. A cheat-sheet of all rules was provided. We would measure how long the student took

[1] `https://tinyurl.com/QualtricsSurvey-1`
[2] `https://tinyurl.com/QualtricsSurvey-2`
[3] `https://tinyurl.com/ANDTaP-Questions`

to solve the proof (provided they were able to), what rules were utilized, and what mistakes were made. Participants who fully completed the study were entered into a drawing to win a $50 Amazon Gift Card.

Unfortunately, due to a lack of participants in the Qualtrics surveys ($n = 2$) and ANDTaP ($n = 0$), we were unable to perform any significant analysis of the results. It is unclear why the survey was unsuccessful, but the lack of an incentive or requirement via a class grade is almost certainly a suspect. Perhaps integrating one of our tools into a course, similar to other previous experiments, could yield improved participation results. We will, however, analyze and discuss the data received from the two Qualtrics survey participants in chapter IV.

1.2 Experiment 2: Determining the Efficacy of Natural Deduction Software

Experimenting between systems that perform related tasks appears easy at first glance, but quickly diverges into chaos if a consensus on differentiation is not reached. In this section, we describe the two systems we created, aimed at improving the pedagogy of introductory formal logic. With this, we also thought about testing the algorithms and subsystems against other available options to students and teachers. Of course, as we have mentioned, without students to assess, such a task is quite challenging because an expert's opinion on what "is better" may reflect their own implicit bias and not that of "common users" of the software. Due to this dilemma, our metric evolved into the complexity of a generated natural deduction proof. But, then again, what is complexity to an expert versus a beginning, or even intermediate, student? Is it fair to say that a proof which uses more axioms from its base set is less complex than another (system)? Should we consider a proof in comparison to what other systems generate; namely, if one system generates a proof that wildly diverges

from an "expert solution", ought we see this as a negative? In our analysis, we will answer several of these questions.

1.2.1 FLAT: Formal Logic Aiding Tutor

FLAT began as a collaborative project which extended LLAT: the Logic Learning Assistance Tool [8]. This extension brought along a core component to formal logic proofs: the ability to prove or disprove a proposition via natural deduction. Not only does the system have an algorithm for automatically proving a clause of formulas (see appendix 1), it also includes a tutoring system allowing students to, step-by-step, write a proof. FLAT supports both zeroth and first-order logics, with heuristics to prevent infinite proofs that comes with the semidecidability of first-order logic as proven by Turing in [47].

(a) Predicate Logic Proof (b) Propositional Logic Proof

Figure 2. Two Natural Deduction Proofs Generated by FLAT

We will briefly explain some algorithms/subsystems, as well as other unique features of FLAT. For starters, students can export truth tables, truth trees (semantic tableaux) [39], and natural deduction proofs as .pdf and .tex source files. This allows for easy submission and modification of, for instance, practice problems.

FLAT is translatable into a number of languages other than English via a Google Translate API to ease students into the world of the new language of formal logic

without having to strictly learn English (note: this feature is highly experimentable and does not always produce optimal translations).

Table 3 provides a list of a subset of the FLAT algorithms/subsystems that students can use to practice. Table 4 shows the axiom base set for natural deduction proofs in FLAT.

Like many existing systems, FLAT has its share of drawbacks. One of those is its tutoring system is rather primitive compared to others—while it has a "tutoring mode" for almost all supported algorithms and subsystems, it does not generate exercises nor learn from an individual student's mistakes, which deems it non-intelligent according to the standard definition of an intelligent tutor [52]. Also, there is no rhyme or reason why some first-order logic well-formed formulas infinitely loop whereas others do not (we suspect this is due in part to how it selects steps to complete a proof). Furthermore, the natural deduction system in FLAT is missing several key axioms e.g., commutativity and associativity as explained in section 1.2.2, which largely limits the set of solvable proofs. Moreover, when it cannot solve a proof, the system tries to perform a proof by contradiction, which sometimes leads it astray. We plan to drastically improve all features in future editions of FLAT.

1.2.2 ANDTaP: Automatic Natural Deduction Tutor and Prover

ANDTaP is a smaller, web-based version of FLAT's natural deduction implementation. It supports a subset of its proving capabilities, but a superset of the tutoring functionality. Some natural deduction rules, e.g., associativity and commutativity, that are not present in FLAT work as intended in ANDTaP. The choice to use a web-based client for ANDTaP rather than a desktop application was highly influenced by the desire to allow students to use it wherever they want instead of being restricted

Table 3. Subset of Implemented Algorithms in FLAT

Algorithm	Definition
Truth Tree	A truth tree is a description of the truth interpretations of a logic formula F.
Truth Table	A truth table is a sequence of true and false values evaluated for all models of a PL formula F.
Free Variable Detector	Finds all free variables in a FOPL formula F. An occurrence of a variable $v \in F$ is free iff there is no quantifier Q that binds v in its scope.
Bound Variable Detector	Finds all bound variables in a FOPL formula F. An occurrence of a variable $v \in F$ is bound if there is a quantifier Q that binds v in its scope.
Open Sentence Determiner	A FOPL formula F is open if $\exists v \in F$ such that v is free.
Closed Sentence Determiner	A FOPL formula F is closed if $\forall v \in F$, v is bound.
Ground Sentence Determiner	A FOPL formula F is ground if F does not contain any variables.
Main Operator Detector	A unary or binary connective c is the main operator of a logic formula F if it is the first-parsed operator when recursively evaluating F. If F contains no connectives, then there is no main operator.
Vacuous Quantifier Detector	A quantifier q in a FOPL formula F is vacuous if it does not bind any variable v in its scope.
Logical Tautology Determiner	A logic formula F is a logical tautology if it is true in every interpretation/model.
Logical Falsehood Determiner	A logic formula F is a logical falsehood if it is false in every interpretation/model.
Logical Contingency Determiner	A logic formula F is a logical contingency if it is neither a logical tautology or logical falsehood.
Logically Consistent Determiner	Two logic formulas F, F' are logically consistent if there a model $\mathcal{M}$ such that F and F' are true and $F_{\mathcal{M}} = F'_{\mathcal{M}}$.
Logically Contradictory Determiner	Two logic formulas F, F' are logically contradictory if there is no model $\mathcal{M}$ such that $F_{\mathcal{M}} = F'_{\mathcal{M}}$.
Logically Contrary Determiner	Two logic formulas F, F' are logically contrary if there is at least one model $\mathcal{M}$ that is false and $F_{\mathcal{M}} = F'_{\mathcal{M}}$, and there does not exist a model $\mathcal{M}'$ that is true and $F_{\mathcal{M}'} = F'_{\mathcal{M}'}$.
Logically Implied Determiner	Two logic formulas F, F' are logically implied if there does not exist a model $\mathcal{M}$ such that $F_{\mathcal{M}}$ is true and $F'_{\mathcal{M}}$ is false.
Logically Equivalent Determiner	Two logic formulas F, F' are logically equivalent if there does not exist a model $\mathcal{M}$ such that $F_{\mathcal{M}} \neq F'_{\mathcal{M}}$.

Table 4. FLAT Natural Deduction Axioms

Axiom Name	Definition
Modus Ponens (MP)	$(\phi \rightarrow \psi) \wedge \phi \therefore \psi$
Modus Tollens (MT)	$(\phi \rightarrow \psi) \wedge \neg\psi \therefore \neg\phi$
Hypothetical Syllogism (HS)	$(\phi \rightarrow \psi) \wedge (\psi \rightarrow \gamma) \therefore (\phi \rightarrow \gamma)$
Disjunctive Syllogism (DS)	$(\phi \vee \psi) \wedge \neg\phi \therefore \psi$
Disjunction Introduction ($\vee$I)	$\phi \therefore (\phi \vee \psi)$
Conjunction Elimination ($\wedge$E)	$(\phi \wedge \psi) \therefore \phi$
Conjunction Introduction ($\wedge$I)	$\phi \wedge \psi \therefore (\phi \wedge \psi)$
Material Implication (MI)	$(\phi \rightarrow \psi) \leftrightarrow (\neg\phi \vee \psi)$
Biconditional Breakdown (BCB)	$(\phi \leftrightarrow \psi) \therefore (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$
Biconditional Introduction (BCI)	$(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \therefore (\phi \leftrightarrow \psi)$
Double Negation Introduction (DNI)	$\phi \therefore \neg\neg\phi$
Double Negation Elimination (DNE)	$\neg\neg\phi \therefore \phi$
Transposition (TP)	$(\phi \rightarrow \psi) \leftrightarrow (\neg\psi \rightarrow \neg\phi)$
Constructive Dilemma (CD)	$(\phi \vee \psi) \wedge (\phi \rightarrow \gamma) \wedge (\psi \rightarrow \omega) \therefore (\gamma \vee \omega)$
Destructive Dilemma (DD)	$(\neg\gamma \vee \neg\omega) \wedge (\phi \rightarrow \gamma) \wedge (\psi \rightarrow \omega) \therefore (\neg\phi \vee \neg\psi)$
DeMorgan's Law (DeM)	$\neg(\phi \wedge \psi) \leftrightarrow (\neg\phi \vee \neg\psi)$
	$\neg(\phi \vee \psi) \leftrightarrow (\neg\phi \wedge \neg\psi)$
	$\neg(\phi \leftrightarrow \psi) \leftrightarrow \neg((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$
	$\neg\exists\phi \leftrightarrow \forall\neg\phi$
	$\neg\forall\phi \leftrightarrow \exists\neg\phi$
Existential Elimination ($\exists$E)	$\exists x Px \therefore P\alpha$ where α is not previously-used
Existential Introduction ($\exists$I)	$P\alpha \therefore \exists x Px$
Universal Elimination ($\forall$E)	$\forall x Px \therefore P\alpha$ where α is previously-used

to a locally-installed (desktop) program.

Now that we have thoroughly discussed FLAT and ANDTaP, we will now explain the methods used to investigate several natural deduction systems via head-to-head comparisons against one another.

Systematically evaluating natural deduction software is complicated. For starters, as we previously mentioned, we still need a metric of analysis: how to directly compare one system to another. The best and systematic way to do this, excluding the possibility of a human study, is via the overall size of a proof (i.e., the number of lines

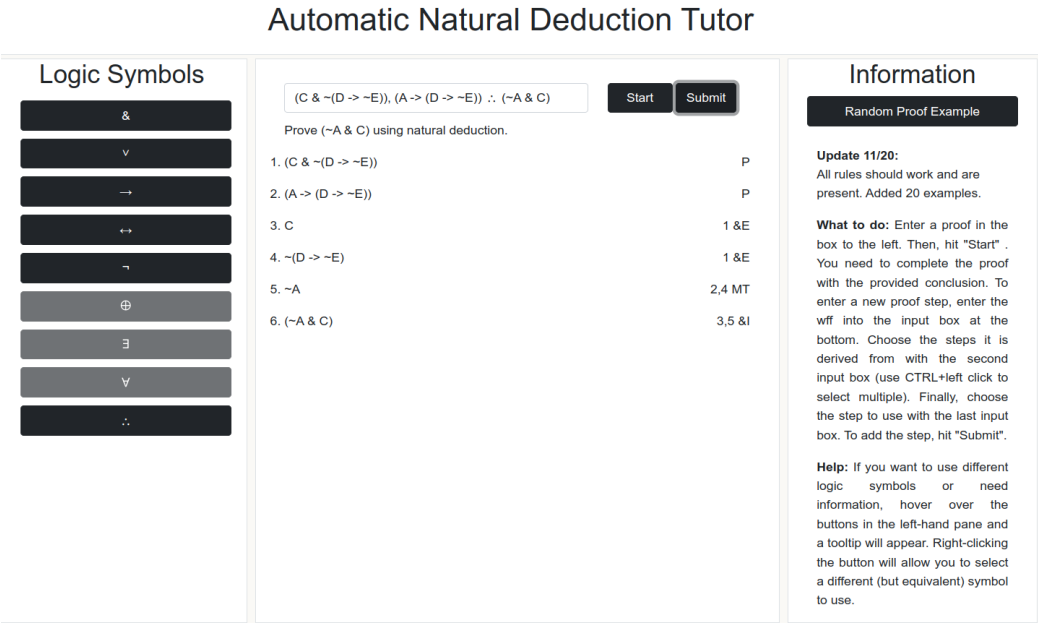


Figure 3. ANDTaP Tutoring System

a generated proof contains). Beginning students that receive a large and unwieldy result are likely to ignore its significance out of frustration. Though, because the length of a proof does not strictly correlate with its complexity, we recorded the number of rules/axioms a system used to solve a proof compared to its base set size. The idea behind this decision is to provide insight into how often a system takes advantage of its axiom set, instead of trying to strictly use direct proofs (e.g., in Hilbert systems) or proofs by contradiction.

We collected and created 288 well-formed formula propositional and first-order logic proofs from various logic and computer science textbooks [19] [25] [49]. We manually converted each proof to the required syntax for all four system, then recorded the number of lines in the annotated proof as well as how many axioms/rules were used. Before we reveal and explain the results, we will briefly review each investigated system.

The first tool we analyzed was a web application for proving propositional logic natural deduction formulas by Laboratoire d'Informatique de Grenoble (LIGLAB) [17]. While their tool includes a few other argument verification tools (e.g., semantic tabuleaux solver for modal logic S4, a resolution prover for first-order logic), we focused only on its propositional logic proving ability. Users enter premises as a series of conjunctions followed by an implication to the conclusion. This syntax follows the standard proof idea which says if all premises are true, then the conclusion must be true (in other words, the premises logically imply the conclusion). Beginning students or those using an ever-so-slightly different notation may be frustrated to discover that they have to convert their entire input to this rigid standard to parse it correctly. Such restrictions mean that users must focus on formatting their input to what the system requires rather than what it outputs as a result. We did find that their prover solved every propositional logic proof in our suite, but we found that because the system has a small baseset of theorems/axioms, almost every proof is a proof by contradiction, resulting in several nested proofs which can be hard to decipher.

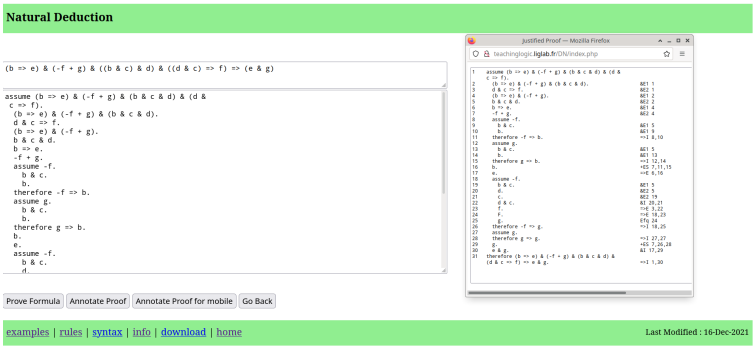


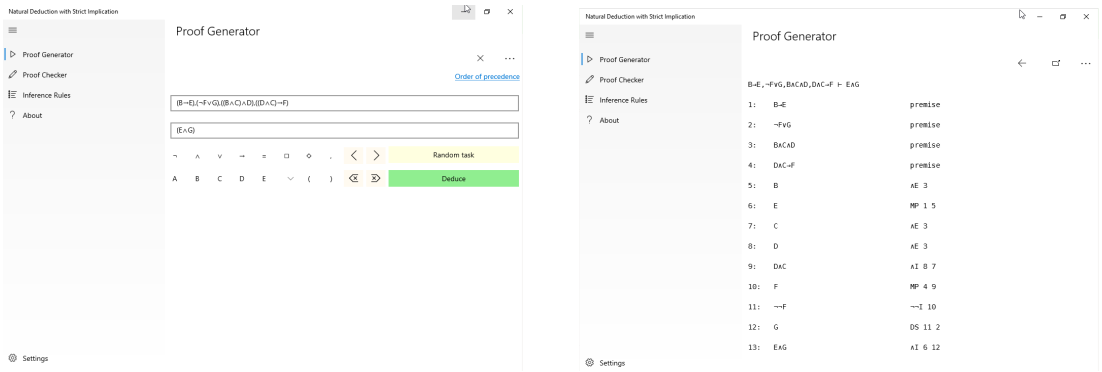
Figure 4. LIGLAB's Natural Deduction
Example proof showing the user interface and proof annotations.

Natural Deduction is a Windows 10 application designed by Jukka Häkkinen, and is the second natural deduction software we investigated [24]. This system includes

both a proof generator and a proof checker. While it primarily focuses on modal logic (specifically, the modal logic system S5), it has a propositional logic prover because modal logic natural deduction semantics are a superset of those of propositional logic. We noted that its interface is clean and very elegant to use. Likewise, its ability to prove both theorems and premise-conclusion style proofs is helpful. We also found its performance on par, if not faster than other similar software. However, Natural Deduction has a severe drawback: its proof generation capabilities, or somewhat of a lack thereof. While it generates short and simple proofs for a subset of our test suite, for others, the proofs were unmanageably long and so cumbersome that a student would, realistically, never look through them. In addition to this significant issue, we discovered that the system places an arbitrary limit on the length of a premise set and its corresponding conclusion. Along a similar vein, the system refuses any proof that contains more than seven propositions/atoms, even if the proof contains no connectives - only atoms (e.g., A, B, C, D, E, F, G, H, $\therefore H$). Lastly, the system automatically converts connectives to a recognizable format e.g., $>$ to $\rightarrow$, and uppercases any entered propositions. It gets confused, though, if the user enters a symbol it does not recognize (e.g., & instead of $\wedge$), and erroneously replaces symbols that otherwise together represent one into two separate symbols e.g., $=>$ into $\equiv\supset$. These restrictions do not entirely detract students from the application; however, they exemplify the types of downfalls that other systems do not have.

TAUT-Logic is a web application designed by Ariel Roffé, and assisted by the Buenos Aires Logic Group [41]. Unlike the first two, TAUT-Logic supports first-order predicate logic, and is the only easy-to-use system of its kind that we found which is not out of commission nor abandoned. In addition to its natural deduction toolset, it supports basic set theory, truth table generation, and model truth.

Figure 5. NaturalDeduction Windows Application Solving a Proof



(a) Proof Input to System (b) Proof Output Solution with Annotations

One awkward characteristic of this system is that its propositional logic application only supports lowercase letters for atoms. Additionally, similar to Natural Deduction, TAUT-Logic supports a total of nine different atoms, ranging from o to w. Supporting only nine atoms may be enough for some problems/proofs, but this restriction seems rather arbitrary for natural deduction, whose problem complexity should not grow strictly in terms of the number of atoms. Another odd design choice is the preference of English phrases for connectives instead of symbols such as, for example, "implies" versus $\rightarrow$ or $\supset$. Because all other systems we tested only utilize symbols (with the exception of FLAT, as it supports both), the process of converting each formula and symbol to the appropriate format was tiresome. Lastly, for some reason, the biconditional connective is not supported, requiring students to either convert them to a conjunction of implications, or omit the proof altogether.

Regarding TAUT-Logic's performance, we discovered that it is roughly on par in terms of speed, but fails on moderately complex propositional and first-order logic proofs. We also saw a general increase in the produced proof size compared to the other systems.

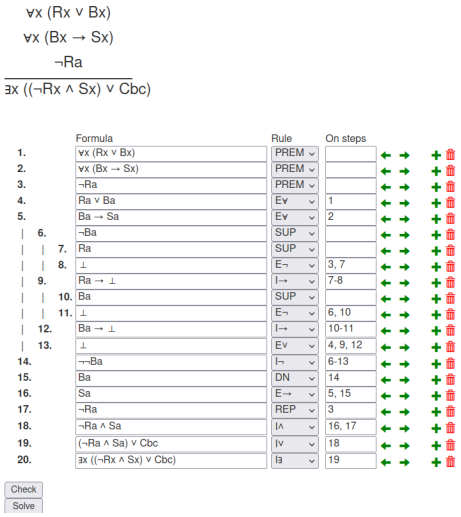


Figure 6. TautLogic's Predicate Natural Deduction
Write a caption...

1.2.3 System Comparison Results

Table 5 shows the tabulated results, including the average number of applied distinct rules compared to the base set size, the average length of all proofs, and the average success rate. Note: systems that did not support first-order logic contributed to a lower overall success rate. Consequently, we further subdivided the data into two groups: one with only propositional logic formulas ($n = 203$), and another with only predicate logic formulas ($n = 85$). Also note that, when a system cannot solve a proof, this, in turn, lowers the average number of steps and applied distinct rules as those count as zero towards the average. Thus, we created another table to show non-zero total averages.

In chapter IV we will discuss the findings to our experiment, as well as the challenges we encountered when formatting and inputting the formulas from our dataset.

Table 5. Data Analysis of Propositional and First-Order Logic Proofs

System	Avg. Success Rate	Avg. No. Steps	Avg. No. Distinct
FLAT	**84.72%**	**5.86**	**2.79**
TeachingLogic	70.49%	12.41	3.77
NaturalDeduction	56.60%	13.50	3.38
TAUT-Logic	73.96%	13.34	4.46

Table 6. Data Analysis of Propositional Logic Proofs

System	Avg. Success Rate	Avg. No. Steps	Avg. No. Distinct Rules
FLAT	**85.22%**	**5.94**	**2.67**
TeachingLogic	99.01%	17.48	5.30
NaturalDeduction	79.31%	19.03	4.73
TAUT-Logic	76.35%	15.59	4.71

Table 7. Data Analysis of Predicate Logic Proofs

System	Avg. Success Rate	Avg. No. Steps	Avg. No. Distinct Rules
FLAT	**83.53%**	**5.62**	**3.05**
TAUT-Logic	68.24%	7.46	3.76

Table 8. Non-Zero Data Analysis of All Proofs

System	Avg. No. Steps	Avg. No. Distinct Rules
FLAT	**6.92**	**3.30**
TeachingLogic	17.61	5.35
NaturalDeduction	23.85	5.96
TAUT-Logic	18.04	6.03

2 Gold Standard for Formal Logic System Syntax

There are several reasons why a standardized grammar does not necessarily already exist for formal logic. Firstly, symbol usage varies widely from one subject to the next. Case in point, notation used in computer science may contain subtle yet

important differences from philosophy-esque logics. Secondly, preexisting sources such as textbooks, websites, professors, and others all use preferential notation (i.e., they use what they think is correct, what they were taught, or what is otherwise preferred in their respective discipline), providing an amalgamation of symbols for students to use and reference which, therefore, leads students and automatic systems astray when expecting one syntax yet receive something completely different. Thirdly, propositional logic learning platforms may or may not include certain operators. For example, because it is trivial to represent the biconditional (if and only if) binary operator as a conjunction of implications, it is certainly possible, albeit rather rare, to omit its symbolic representation from a language. Such omissions cause problems when evaluating formulas either automatically or by hand due to the extended requirement of deriving an equivalent format in a language. In terms of computability, this does not pose a significant issue because any connective in zeroth-order logic can be equivalently represented by, for instance, either $\{\wedge, \neg\}$ (NAND) $\{\vee, \neg\}$ (NOR) due to their functional completeness property as proved by Post in [38]. While not an algorithmic problem, it is inconvenient to rewrite formulas by hand to fit a restrictive system when, for instance, testing different automatic logic systems (see Table 9, TAUT-Logic).

We propose a formal definition that aims to solve most of these problems. One component of this definition allows users to create their own logic language definition as they see fit for their situation. This language is then translatable into a gold standard format, which we will define syntactically and semantically.

The reason we formalize the language definition is to allow different logic systems with varying syntax—some use lower-case atomic formulas, while others may restrict the alphabet to a subset. This definition allows different connective alphabets to map

to the same symbol in the gold standard which provides a seamless translation to and from various host logic languages (i.e., the language of the implementing systems, assuming it does not, by default, use the gold standard internally). Another benefit of using prefix connective notation is its disambiguation of precedence, which we will supplement with further discussion at the end of the next section.

2.1 Zeroth-Order Logic Well-Formed Formula Representation

We will start with a small example and then generalize the approach to achieve a well-defined representation. Suppose we have a set of premises $P = \{(A \leftrightarrow B), \neg(C \wedge D), C, \neg B\}$ with the conclusion $c = (\neg A \wedge D)$. Converting this proof into the three systems we tested is laborious at best and is increasingly tiresome the more systems we wish to evaluate. Table 9 displays the required syntax to parse an equivalent representation of w. The desire for a uniform standard to rapidly test multiple systems without manual intervention is readily apparent.

Table 9. Required Syntax to Parse Proof (P, c)

Natural Deduction System	Syntax
TeachingLogic	$(p$ <=> $q)$ & -$(r$ & $s)$ & r & -q => $(-p$ & -$s)$
NaturalDeduction	$(A \equiv B), \neg(C \wedge D), C, \neg B \therefore (\neg A \wedge \neg D)$
TAUT-Logic	$(A$ implies $B)$ and $(B$ implies $A)$, not$(C$ and $D)$, C, not $B \therefore$ (not A and not $D)$

Let $M(\mathcal{L}, w)$ be the operation that "applies" the zeroth-order logic language $\mathcal{L}$ to the well-formed formula w. Let $\mathcal{L}$ be a pair (δ, ζ) where δ is an connective mapping function, and ζ is an atomic literal mapping function.

The bijective function δ maps two sets δ: $X \rightarrow Y$, where X is the set of input connectives defined by $\mathcal{L}$, and $Y \subseteq \{N, C, D, E, I, B, T, F\}$ is the set of output

connectives defined by our grammar, where $|X| = |Y|$. Table 10 shows the meaning of each connective in Y. Note that the arity of any connective $\phi \in X$ must match the arity of its corresponding output connective $\psi \in Y$.

Table 10. Connective Symbols for Zeroth-Order Logic Gold Standard

Semantic Meaning	Connective Symbol
Logical Negation	N
Logical Conjunction	C
Logical (Inclusive) Disjunction	D
Logical Exclusive Disjunction	E
Logical Implication	I
Logical Biconditional	B
Logical Tautology	T
Logical Falsehood	F

The bijective function ζ maps two sets ζ: $A \to B$, where A is the set of atomic literals $\phi \in A$ where ϕ is an atomic literal used in w, and B is the set of output atomic formulas a_j where $j \in [1, |A|]$.

We can now define the Polish (Łukasiewicz), or prefix notation grammar G used to create a standardized notation for zeroth-order logic. This notation takes inspiration from the syntax of the programming language Scheme with its parenthesization of connectives and operands. For this, we must extend the definition of typical Extended Backus-Naur Form to account for multiple-arity connectives. Thus, we introduce the notation <x—R> to indicate that x is a variable used in the EBNF rule R, and $\{\Lambda\}^x$ to denote exactly x applications of Λ. In the grammar, α is the arity of a connective.

$\langle atomic \rangle$ ::= ‘`a`’ (‘`1`’ | ‘`2`’ | ...)

$\langle connective \rangle$ ::= ‘`N`’ | ‘`C`’ | ‘`D`’ | ‘`E`’ | ‘`I`’ | ‘`B`’ | ‘`T`’ | ‘`F`’

$\langle \alpha\text{—}wff \rangle$ ::= $\langle atomic \rangle$ | ($\langle connective \rangle$ [‘ ’] $\{\langle wff \rangle\}^{\alpha}$)

We shall reiterate that our goal is to create a language pipeline that allows for easy conversion between one language $\mathcal{L}$, the gold standard $M(\mathcal{L}, w)$, and another arbitrary language of the same class $\mathcal{L}'$. Symbolically, $\mathcal{L} \Leftrightarrow M(\mathcal{L}, w) \Rightarrow \mathcal{L}'$. We will now make note of converting a gold-standard formula w' to a language $\mathcal{L}'$ such that Y', the connective set of w' is disjoint from the connective set of $\mathcal{L}'$: Y (that is, there exists an operator in the gold standard not supported by the target language $\mathcal{L}'$). In this scenario, $\mathcal{L}'$ must augment its definition with a pattern-matching replacement function $\mathcal{R}$. $\mathcal{R}$ should take a connective as input, and output an equivalent representation that satisfies the grammar of language $\mathcal{L}'$. As an example, suppose $w' = (B\ a_1\ a_2)$. When converting w' to a target language not identical to the original source language (i.e., $L' \neq L$) that does not support the biconditional connective, we could define $\mathcal{R}(B) = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ where $\rightarrow, \wedge \in Y$, and ϕ, ψ are arbitrary well-formed formulas in $\mathcal{L}'$. So, when we convert from the gold standard into a target language, any instance of the biconditional B is replaced by $\mathcal{R}$ into a recognizable format. At a minimum, any set of connectives Y for any zeroth-order logic language must be functionally complete to achieve this goal [38]. One note regarding the expansion well-formed formulas via $\mathcal{R}$ is its growth rate consequence. Namely, if the transformed formula is reflective (i.e., $(A \rightarrow B) \wedge (B \rightarrow A)$), it expands exponentially.

2.1.1 Zeroth-Order Logic Example 1

Let us take a "standard" propositional logic language $\mathcal{L}$ and a formula w. $\mathcal{L}$ consists of two functions δ and ζ where

$$\delta\colon \{\supset, \wedge, \vee, \leftrightarrow, \neg\} \mapsto \{I, C, D, B, N\}$$
$$\zeta\colon \{A, B, C\} \mapsto \{a_1, a_2, a_3\}$$

We will let $w = A \supset (B \leftrightarrow \neg C)$. Thus,

$$M(\mathcal{L}, w) = (I\ a_1\ (B\ a_2\ (N\ a_3)))$$

This representation reads naturally from left-to-right as follows: "An implication of a_1 and a biconditional of a_2 and negated a_3". We consider all connectives as first-class functions in our definition.

While prefix notation is not as readable as the infix w, it creates a uniform standard for testing zeroth-order logic systems. What's more is that this application process is reversible; given $M^{-1}(\mathcal{L}', w')$ where $\mathcal{L}' = (\delta^{-1}, \zeta^{-1})$ and $w' = M(\mathcal{L}, w)$, we can reproduce w using a simple stack-and-pop parsing evaluation approach (deterministic push-down automaton).

2.1.2 Zeroth-Order Logic Example 2

Quine's syntax in [49] for propositional logic is slightly different from modern variants. Specifically, his use of dots and colons removes superfluous parentheses when defining operator precedence. Largely, we will ignore this notation in favor of his parenthesized form. In addition, Quine uses an empty string ε to represent conjunction

(e.g., S_1S_2 represents a conjunction between two well-formed formulas S_1 and S_2). Finally, negations on a single atom p are condensed with a vertical overbar $\overline{p}$. To compensate for the digital representation, we will keep the negation in front of the atom (e.g., $-S_1$ where S_1 is a well-formed formula). Now, we define $\mathcal{L}$ with functions δ and ζ where

$$\delta\colon \{\supset, \varepsilon, \vee, \equiv, -\} \mapsto \{I, C, D, B, N\}$$
$$\zeta\colon \{p, q, r, s\} \mapsto \{a_1, a_2, a_3, a_4\}$$

We will let $w = -((p \vee q)(-r \vee s)) \supset (-(p \vee q)s)$. Thus,

$$M(\mathcal{L}, w) = (I\ (C\ (N\ (D\ a_1\ a_2))\ (D\ (N\ a_3)\ a_4))$$
$$(C\ (N\ a_1\ a_2)\ a_4))$$

This representation reads as "An implication where the left-hand side is a conjunction between a negated disjunction of a_1 and a_2, and a disjunction of negated a_3 and a_4. The right-hand side of the implication is a conjunction between a negated disjunction of a_1 and a_2, and a_4."

We assume the incoming formula is unambiguous according to its language grammar specification out of simplicity, but for completeness, we will define a precedence mapping function for the incoming formula Γ. By enforcing prefix notation in the gold standard, we no longer have to deal with the inherent complexities of operator precedence present in the commonly-used infix notation.

Let Γ be an injective function that maps the set of connectives δ to $\mathbb{N}$, namely $\delta \mapsto \mathbb{N}$. Γ is designed to give connectives in δ a priority level, where the closer its

mapped natural number is to zero, the higher its priority. We define priority as the precedence of a connective. When a system defines Γ, it implies that any ambiguous well-formed formula in its corresponding language is parsable without parenthesization. Γ, as an algorithm, automatically adds parentheses to disambiguate the formula.

2.1.3 Precedence Mapping Example

We will use the same functions δ and ζ from the first propositional logic example. We will also define Γ as

$$\delta\colon \{\supset, \wedge, \vee, \leftrightarrow, \neg\} \mapsto \{I, C, D, B, N\}$$

$$\zeta\colon \{A, B, C\} \mapsto \{a_1, a_2, a_3\}$$

$$\Gamma\colon \{\supset, \wedge, \vee, \leftrightarrow, \neg\} \mapsto \{3, 1, 2, 4, 0\}$$

Now, suppose $w = A \rightarrow \neg B \rightarrow C \wedge \neg A$. The precedence function parenthesizes/disambiguates w to $((A \rightarrow \neg B) \rightarrow (C \wedge \neg A))$, which is then converted into the gold standard as

$$M(\mathcal{L}, w) = (I\ (I\ a_1\ (N\ a_2))\ (C\ a_3\ (N\ a_1)))$$

Since we consider Γ to be optional (opting for a default precedence of logical negation, logical conjunction, logical disjunction, logical implication, then logical biconditional), a system without a defined Γ should, optimally, output a warning when it parses an ambiguous well-formed formula. Defining Γ allows for any ambiguous formula to be converted into one that is unambiguous, as we previously stated, and also allows for "custom precedence" levels (i.e., if we want to bind logical disjunction higher than

logical conjunction, it is trivial to do so). Finally, some may question the associativity of connectives. We assume that all operators are left-associative, similar to traditional addition, subtraction, multiplication, and division. For those who wish to not always strictly assume left-associativity for connectives, we will now define the associativity function γ.

Let γ be a surjective function that maps the set of binary connectives $X_B \subseteq X$ to the set $\{L, R\}$, where L and R designate left and right-associativity respectively.

2.1.4 Associativity Mapping Example

We will use the same functions δ and ζ from the previous precedence mapping function example. In addition, we will define γ as

$$\delta\text{: } \{\supset, \wedge, \vee, \leftrightarrow, \neg\} \mapsto \{I, C, D, B, N\}$$
$$\zeta\text{: } \{A, B, C\} \mapsto \{a_1, a_2, a_3\}$$
$$\gamma\text{: } \{\supset, \wedge, \vee, \leftrightarrow, \neg\} \mapsto \{L, R\}$$

Where $\forall C \in X_B$, $\gamma(C) = L$ (i.e., every binary connective is left-associative). Now, suppose $w = (A \to B \to C) \wedge (\neg A \wedge \neg B)$. The associativity function disambiguates w to $((A \to B) \to C) \wedge (\neg A \wedge \neg B)$. This is converted into the gold standard as

$$M(\mathcal{L}, W) = \ (C\ (I\ (I\ a_1\ a_2)\ a_3)\ (C\ (N\ a_1)\ (N\ a_2)))$$

2.1.5 Natural Deduction Extension

It is simple to extend G to support premises and conclusions using the same syntax. We can define a new function $N(\mathcal{L}, P, c)$, where $\mathcal{L}$ is the same definition as before,

P is a set of well-formed formula acting as the premises of the proof, and c is the well-formed formula acting as the conclusion of the proof. Our new grammar G' is as follows:

$\langle atomic \rangle$ ::= ‘`a`’ (‘`1`’ | ‘`2`’ | ...)

$\langle connective \rangle$ ::= ‘`N`’ | ‘`C`’ | ‘`D`’ | ‘`E`’ | ‘`I`’ | ‘`B`’ | ‘`T`’ | ‘`F`’

$\langle \alpha\text{—}wff \rangle$ ::= $\langle atomic \rangle$ | ($\langle connective \rangle$ [‘ ’] {$\langle wff \rangle$}$^{\alpha}$)

$\langle premise \rangle$ ::= (‘`P`’ $\langle wff \rangle$)

$\langle conclusion \rangle$ ::= (‘`H`’ $\langle wff \rangle$)

$\langle proof \rangle$ ::= ($\langle conclusion \rangle$ {$\langle premise \rangle$})

The preceding grammar states that a premise is preceded by the letter P standing for *premise*, conclusions are preceded by H for *hence*, and a proof is a conclusion followed by zero or more premises (a proof with zero premises is a theorem).

2.1.6 Natural Deduction Example

Let's create a proof where $P = \{\neg(C \vee D),\ D \leftrightarrow (E \vee F), \neg A \supset (C \vee F)\}$, and $c = A$. We must redefine ζ in $\mathcal{L}$ as follows:

$$\zeta\colon \{A,\ C,\ D,\ E,\ F\} \mapsto \{a_1,\ a_2,\ a_3,\ a_4,\ a_5\}$$

Therefore,

$$\begin{aligned} N(\mathcal{L},\, P,\, c) = ((H\ a_1) \\ (P\ (N\ (D\ a_2\ a_3))) \\ (P\ (B\ a_3\ (D\ a_4\ a_5))) \\ (P\ (I\ (N\ a_1)\ (D\ a_2\ a_5)))) \end{aligned}$$

We read this as "Hence a_1 if P_1 is true and P_2 is true and P_3 is true", where P_1, P_2, and P_3 are the individual premises that comprise the argument. This style largely resembles the way we write Prolog (conditional) rules.

2.2 First-Order Logic Well-Formed Formula Representation

First-order logic is a superset of zeroth-order logic, meaning we can reuse most of our definitions from the previous section. We will, however, need to slightly redefine $\mathcal{L}$ to allow for mapping predicate definitions, constants, and variables. Further, so as to not confuse the function definitions from zeroth-order logic, we will instead use new letters to represent mapping functions unique to first-order logic semantics.

Let $\mathcal{M}(\mathcal{L},\, w)$ be an operation that applies the first-order gold standard to a logic language $\mathcal{L}$ and a well-formed formula w. Let $\mathcal{L}$ be a quadruple $(\delta,\, \varsigma,\, \chi,\, \eta)$ where δ is a connective mapping function, ς is a predicate mapping function, χ is a constant mapping function, and η is a variable mapping function.

For first-order logic, we slightly modify δ from its zeroth-order definition in the sense that it now maps two sets δ: $X \rightarrow Y$, where X is the set of input connectives defined by $\mathcal{L}$, and $Y \subseteq \{N,\, C,\, D,\, E,\, I,\, B,\, T,\, F,\, Z,\, X,\, V\}$ is the set of output connectives defined by our grammar, where $|X| = |Y|$. N, C, D, E, I, B, T, and F are identical

in both syntactic and semantic meaning to zeroth-order logic as referenced in Table 10. Table 11 shows the new operators added by first-order logic. Note that Z and X have arities dependent on the formula used, so we cannot restrict it syntactically. Identity V, on the other hand, is a binary operator.

Table 11. Connective Symbols for First-Order Logic Gold Standard

Semantic Meaning	Connective Symbol
Universal Quantifier	Z
Existential Quantifier	X
Identity	V

The bijective function ς maps two sets ς: $A \to B$, where A is the set of predicate letters $\phi \in A$ where ϕ is a predicate letter used in the wff w, and B is the set of output predicate letters L_i where $i \in [1, |A|]$.

The bijective function χ maps two sets χ: $C \to D$, where C is the set of constant letters $\psi \in C$ where ψ is a constant identifier used in w, and D is the set of output constant identifiers c_i where $i \in [1, |C|]$.

Lastly, the bijective function η maps two sets η: $E \to F$, where E is the set of variable letters $\rho \in E$ where ρ is a variable identifier used in w and F is the set of output variable identifiers v_i where $i \in [1, |E|]$.

Now, similar to zeroth-order logic, we will construct the gold standard Polish notation grammar $\mathcal{G}$ for first-order logic. Likewise, we will utilize the previously-defined notation $<x$—R$>$ to eliminate ambiguity with operator arity. Two points to note are that, because identity is a special connective in first-order logic, we restrict its syntactic definition to only constants and variables. Additionally, quantifiers have a restriction in that they must have at least one variable following their declaration, as well as a bound well-formed formula.

⟨*constant*⟩ ::= ‘c’ (‘1’ | ‘2’ | ...)

⟨*variable*⟩ ::= ‘v’ (‘1’ | ‘2’ | ...)

⟨*literal*⟩ ::= ⟨*constant*⟩ | ⟨*variable*⟩

⟨*predicate*⟩ ::= ‘L’ (‘1’ | ‘2’ | ...)

⟨*connective*⟩ ::= ‘N’ | ‘C’ | ‘D’ | ‘E’ | ‘I’ | ‘B’ | ‘T’ | ‘F’

⟨*identity*⟩ ::= ‘V’

⟨*quantifier*⟩ ::= ‘Z’ | ‘X’

⟨α—*wff*⟩ ::= (⟨*predicate*⟩ ⟨*literal*⟩ {⟨*literal*⟩})
| (⟨*connective*⟩ [‘ ’] ⟨*wff*⟩$^{\alpha}$)
| (⟨*quantifier*⟩ ⟨*variable*⟩ ⟨*wff*⟩)
| (⟨*identity*⟩ ⟨*literal*⟩ [‘ ’] ⟨*literal*⟩)

The above grammar states the following rules: constants use the prefix c with a uniquely identifying successive integer. Variables follow the same rules with the exception that variables are prefixed by v. Literals are either constants or variables. Predicates, likewise, use the same identifying principle except that they use L as a prefix. As aforesaid, the identity and quantifier operators are special in the wff definition, in that a quantifier binds a variable to a well-formed formula, and an identifier wraps two literals together. In addition to these special cases, predicates bind at least one literal, and a connective contains as many well-formed formula operands as its arity requires.

2.2.1 First-Order Logic Example

We will, once again, use a "standard" first-order logic language $\mathcal{L}$ and a formula w. $\mathcal{L}$ is a quadruple of the four functions δ, ς, χ, and η where

$$\delta\colon \{\supset, \wedge, \vee, \leftrightarrow, \neg, \forall, \exists, =\} \mapsto \{I, C, D, B, N, Z, X, V\}$$

$$\varsigma\colon \{P, Q, R\} \mapsto \{L_1, L_2, L_3\}$$

$$\chi\colon \{c, d\} \mapsto \{c_1, c_2\}$$

$$\eta\colon \{x, y, z\} \mapsto \{v_1, v_2, v_3\}$$

Suppose $w = \forall x \forall y \neg Pxyc \wedge (Qcd \vee \exists z Rz)$. Thus,

$$\mathcal{M}(\mathcal{L}, w) = (C\ (Z\ v_1\ (Z\ v_2\ (N\ (L_1\ v_1\ v_2\ c_1))))$$
$$(D\ (L_2\ c_1\ c_2)\ (X\ v_3\ (L_3\ v_3))))$$

We read this as "A conjunction of a universal quantifier that binds v_1, a universal quantifier binding v_2, bound to the negation of $L_1\, v_1\, v_2\, c_1$ and a disjunction of the following: $L_2\, c_1\, c_2$ and an existential quantifier which binds v_3, bound to $L_3\, v_3$."

CHAPTER IV

DISCUSSION AND FUTURE DIRECTION

In chapter III, we detailed our methods of experimentation, including a survey-focused and student-driven methodology, a head-to-head comparison of publicly-available natural deduction proving software, and a gold standard syntax definition for both zeroth and first-order logics. In this chapter, we will discuss our general findings, and analyze future research potential.

1 Experiment 1

Despite only receiving two submissions to our surveys for the proof interpretation study, we will still analyze their data. Both participants are computer science majors who have a background in formal logic (propositional and first-order), proofs, and otherwise discrete mathematics.

Regarding survey 1 (premises and conclusion interpretation), we consistently saw the theme that larger conclusions (i.e., conclusions with more clauses) were said to appear more difficult than those with smaller conclusions. Additionally, premises and conclusions which had more negations and disjunctions compared to implications and biconditionals were viewed similarly (e.g., $P = \{A \to B,\ B \to C\}$), $c = A \to C$ versus $P = \{\not{A} \vee B,\ \neg B \vee C\}$, $c = \neg A \vee C\}$). We suspect this is due to easily-applicable

axioms that involve implications such as modus ponens, modus tollens, hypothetical syllogism, and more. There was a disparity between the participants when choosing premises which were distributive opposites via DeMorgan's law, but we can only narrow this down to individual preference with such a low participant count.

Regarding survey 2 (proof interpretation), we observed an almost identical pattern as survey 1 showed—participants commonly noted that proofs with negated propositions and fewer implications appeared harder to understand. Also, as we predicted, longer proofs were considered more difficult than shorter proofs. We still are not sure if these decisions are out of interpreting the proofs or personal preference, but we hope to expand upon this study and the idea of computing a viable metric of difficulty for formal logic in future research.

2 Experiment 2

As Tables 5, 6, 7, 8, and the graphs in Figures 9, 10, 11 show, our system, FLAT, has a higher percentage of proofs that are what we consider digestable for a beginning logic student. It is desirable for a natural deduction proof to be understandable, comprehensible, and condensed enough where details are not omitted, but rather are fleshed out to provide a clear path to the solution.

While TeachingLogic's system has a greater percentage of solved proofs than FLAT, NaturalDeduction, and TAUTLogic, we found its user interface to be unintuitive particularly when entering a formula with unrecognizable syntax according to their language grammar. This odd requirement strongly deters those who wish to use a natural deduction proving software, but perhaps focus more time modifying their input to fit the system specifications than that spent understanding the generated

proof(s). Moreover, its limited axiom base set upper-bounds the kinds of proofs it can generate—resulting in several nested subproofs and proofs by contradiction. Importantly, the generated proofs are syntactically and semantically correct, but to someone with little prior experience in formal logic or mathematical proofs, it may come across as overwhelming.

A severe problem with NaturalDeduction, as we previously mentioned, is its unnecessary inconsistency—the generated proofs range from perfect (in that they match the intended expert solution) to absolutely out of control nose dives into confusion. As an illustration, suppose we want to prove $P = \{(A \leftrightarrow B), (C \leftrightarrow B)\}$, $c = A \leftrightarrow C$. FLAT's solution, which we consider the optimal solution under our axiom set, has twelve lines including the premises and conclusion (see Figure 7). NaturalDeduction, on the contrary, has 247 lines. The absurd length we discovered is not exclusive to this proof; we found several other sets of premises and conclusions which caused the system to generate unimaginably long and incomprehensible proofs. Furthermore, its input requirements are unreasonably restrictive: only allowing eight unique atoms and fourteen identical atoms combined via binary connectives e.g., $A \wedge A \wedge ... \wedge A$. We are unsure if these limitations are due to the proof generation system that NaturalDeduction uses, but because the instructions do not provide this information, we can, at best, only speculate. One additional problem we discovered is that NaturalDeduction omits premises that do not impact the final proof (i.e., premises that it deems are unnecessary), even if they were listed as premises in the original premise set P. We feel that this omission is problematic due to the relative importance of initial premises to an argument. Plus, this can lead to confusion and thoughts of "What happened to premise X?" from students.

Moreover, while not a huge downside, neither NaturalDeduction or TeachingLogic

Figure 7. Proof of $P = \{(A \leftrightarrow B), (C \leftrightarrow B)\}$, $c = A \leftrightarrow C$

1.	$(A \leftrightarrow B)$	P
2.	$(B \leftrightarrow C)$	P
3.	$(A \rightarrow B) \wedge (B \rightarrow A)$	1 BCB
4.	$A \rightarrow B$	3 $\wedge$E
5.	$B \rightarrow A$	3 $\wedge$E
6.	$(B \rightarrow C) \wedge (C \rightarrow B)$	2 BCB
7.	$B \rightarrow C$	6 $\wedge$E
8.	$C \rightarrow B$	6 $\wedge$E
9.	$A \rightarrow C$	4, 7 HS
10.	$C \rightarrow A$	3, 8 HS
11.	$(A \rightarrow C) \wedge (C \rightarrow A)$	9, 10 $\wedge$I
12.	$(A \leftrightarrow C)$	11 BCI

support first-order predicate logic. Oddly enough, NaturalDeduction, instead, supports S5 Modal Logic.

Lastly, TAUT-Logic was, as mentioned before, the only system that we tested which supports both zeroth and first-order logics. It too, though, comes with its share of cons. We provided an example of a logic system in section 2.1 which does not support the biconditional operator, requiring instead to write it as a conjunction of implications (see Table 9). Its performance is similar in that to FLAT, but we found it times out on both zeroth and first-order logic proofs for sometimes no apparent reason with an error stating that it simply could not solve the proof.

While each system that we evaluated has its share of advantages and disadvantages, the most profound problem was the inconsistency in entering proofs and well-formed formulas. Each system uses a slight variation of logic syntax which in turn means we had to convert each formula from the original format into four separate standards (FLAT, TeachingLogic, NaturalDeduction, TAUT-Logic). Our gold standard as

detailed in chapter III aims to rectify these issues with long-term goals of systems either adopting the gold standard, or creating the language pipeline for formulas to be converted from the gold standard into their syntax, so as to be an automatic process rather than manual. While it is relatively unrealistic to expect current state-of-the-art systems to rewrite their input pipeline to only support the gold standard, this is not necessarily our goal. Instead, we aspire to have the gold standard be an intermediate representation for formulas which are fed, or piped, to and from testable systems. Such an increase in standardization will undoubtedly raise the automatic testability and evaluation of logic systems.

3 Future Work

3.1 Gold Standard Extendability via Non-Classical Logic Systems

This thesis is only concerned with classical logic: zeroth and first-order logics. Though, we propose that creating a gold standard for other logics which are, perhaps, supersets of classical logic (e.g., modal) is relatively straightforward with our provided groundwork. Modal logic, however, is different from classical logics in that its necessity and possibility unary connective extensions (i.e., $\Box$ and $\Diamond$ respectively) are the dominant standard.

3.2 Intelligent Tutors

Circling back to formal logic tutors, there is still a significant amount of work to be had on the intelligence of logic tutoring systems (see chapter II). We, in particular, would like to completely move FLAT (see section 1.2.1) into the browser so students do not have to download an application on their computer. This transition is in-

progress via ANDTaP (see section 1.2.2), but has much room for improvement. A responsive and clean UI is all but mandatory in today's world of web and mobile applications, and while ANDTaP is clean, it does not scale well to the mobile platform (e.g., sizing issues, button clicks). Further, we wish to improve the tutoring system by supplementing the user with hints similar to other tutoring systems across different disciplines. Unlike many other systems which specialize in one area of logic e.g., natural deduction, however, FLAT has several tools (see Table 3) where in each could use a pedagogical upgrade via hints, automatic (satisfiable) problem generation, and classroom integration. We plan to continuously update and upgrade both systems, as well as research into improved methods of teaching logic with special emphasis on non-computer science or mathematics students.

4 Conclusion

In this thesis, we discussed FLAT: the Formal Logic Aiding Tutor, and ANDTaP (Automatic Natural Deduction Tutor and Prover). With this, we performed a head-to-head experiment comparing the efficacy of other publicly-available natural deduction systems against FLAT. We described our proposed experiment to uncover a metric of difficulty for natural deduction proofs via student intervention. Most importantly, we transitioned this discussion into the desire for a gold standard for classical logic syntax to ease logic system automated testing and evaluation. We explained a systematic method of defining the gold standard for differing logic languages as well as its associated customizability and flexibility. Both logic and, more broadly, mathematical notation serve the purpose of easy expressing ideas. Pushing efforts towards a gold standard syntax to reduce the time spent wondering about what symbols ought to be

used when enhances said expressibility.

References

[1] Ahmed, U. Z., Gulwani, S., and Karkare, A. Automatically Generating Problems and Solutions for Natural Deduction. In *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence* (August 2013), pp. 1968–1975.

[2] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[3] Amendola, G., Ricca, F., and Truszczynski, M. Generating hard random boolean formulas and disjunctive logic programs. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (2017), IJCAI'17, AAAI Press, pp. 532–538.

[4] Autexier, S., Dietrich, D., and Schiller, M. R. G. Towards an Intelligent Tutor for Mathematical Proofs. In *Proceedings First Workshop on CTP Components for Educational Software, THedu'11, Wroclaw, Poland, 31th July 2011* (2011), P. Quaresma and R. Back, Eds., vol. 79 of *EPTCS*, pp. 1–28.

[5] Bratko, I. *Prolog Programming for Artificial Intelligence*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[6] Cerna, D. M., Kiesel, R. P., and Dzhiganskaya, A. A Mobile Application for Self-Guided Study of Formal Reasoning. *Electronic Proceedings in Theoretical Computer Science 313* (Feb 2020), 35–53.

[7] Cook, S. A. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1971), STOC '71, Association for Computing Machinery, pp. 151–158.

[8] Crotts, J., Altamimi, A., Brantley, H. B. C., and Salou-Doudou, N. A Visual Improvement to the Pedagogy of Introductory Logic, 2021.

[9] Croy, M., Barnes, T., and Stamper, J. Towards an Intelligent Tutoring System for Propositional Proof Construction. In *Proceedings of the 2008 Conference on Current Issues in Computing and Philosophy* (NLD, 2008), IOS Press, pp. 145–155.

[10] Dostálová, L., and Lang, J. ORGANON — The Web Tutor for Basic Logic Courses. *Logic Journal of the IGPL 15*, 4 (2007), 305–311.

[11] Dougherty, M. M. A Standardized Logic Notation for Everyday Classroom Use.

[12] Felice, L., and Leonardi, M. C. Motivating Students through Educational Software Development, 2017.

[13] Fennell, B., Lee, E., and Kim, T. Truth table creator, 2020. Accessed: 2021-07-04.

[14] for Standardization, I. O. Quantities and units - Part 2: Mathematical signs and symbols to be used in the natural sciences and technology, January 2010.

[15] Gentzen, G. Investigations into logical deduction. *American Philosophical Quarterly 1*, 4 (1964), 288–306.

[16] Gonthier, G. Formal Proof - The Four-Color Theorem. *Notices of the American Mathematical Society 55*, 11 (December 2008).

[17] Grenoble Computer Science Laboratory. Natural Deduction, 2021.

[18] Hatcher, D. L. Why Formal Logic is Essential for Critical Thinking. *Informal Logic 19* (1999).

[19] Hein, J. L. *Discrete Structures, Logic, and Computability*, 2nd ed. Jones and Bartlett Publishers, Inc., USA, 2002.

[20] Hein, J. L. *Prolog Experiments in Discrete Mathematics, Logic, and Computability.* Portland State University, 2009.

[21] Heule, M., Iser, M., Jarvisalo, M., Suda, M., and Balyo, T. SAT Competition 2011: Benchmark Submission Guidelines, 2011.

[22] Heule, M. J., and Martins, R. SAT and SMT Solvers in Practice, September 2020.

[23] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM 12*, 10 (oct 1969), 576–580.

[24] Häkkinen, J. NaturalDeduction, 01 2017.

[25] Jackson, R. L., and McLeod, M. L. *The Logic of our Language: An Introduction to Symbolic Logic.* Broadview Press, November 2014.

[26] Kaufmann, M., Moore, J. S., and Manolios, P. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, USA, 2000.

[27] Koedinger, K. R., Aleven, V., Heffernan, N., McLaren, B., and Hockenberry, M. Opening the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration. *Lester J.C., Vicari R.M., Paraguaçu F. (eds) Intelligent Tutoring Systems 3220* (2004).

[28] Lodder, J., Heeren, B., and Jeuring, J. Generating Hints and Feedback for Hilbert-Style Axiomatic Proofs. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2017), SIGCSE '17, Association for Computing Machinery, pp. 387–392.

[29] Lukins, S., Levicki, A., and Burg, J. A Tutorial Program for Propositional Logic with Human/Computer Interactive Learning. *SIGCSE Bull. 34*, 1 (February 2002), 381–385.

[30] Mauco, V., Ferrante, E., and Felice, L. Educational Software for First Order Semantics in Introductory Logic Courses. In *Information Systems Education Journal* (2014), vol. 12, pp. 15–23.

[31] Mostafavi, B., and Barnes, T. Evolution of an Intelligent Deductive Logic Tutor Using Data-Driven Elements. *International Journal of Artificial Intelligence in Education 27* (04 2016).

[32] Muggleton, S. Inductive Logic Programming, February 1991.

[33] Near, J. P., Byrd, W. E., and Friedman, D. P. αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic. In *Logic Programming* (Berlin,

Heidelberg, 2008), M. Garcia de la Banda and E. Pontelli, Eds., Springer Berlin Heidelberg, pp. 238–252.

[34] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283. Springer Science & Business Media, 2002.

[35] Owre, S., Rushby, J. M., and Shankar, N. PVS: A Prototype Verification System, 1992.

[36] Perikos, I., Grivokostopoulou, F., and Hatzilygeroudis, I. Teaching Assistant Tools for NL to FOL Conversion, 01 2011.

[37] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Tolmach, A., and Yorgey, B. *Programming Language Foundations*, vol. 2 of *Software Foundations*. Electronic textbook, 2021. Version 6.1, `http://softwarefoundations.cis.upenn.edu`.

[38] Post, E. L. *The Two-Valued Iterative Systems of Mathematical Logic*. London: Oxford University PRess, 1941.

[39] Priest, G. *An Introduction to Non-Classical Logic: From If to Is*, 2 ed. Cambridge Introductions to Philosophy. Cambridge University Press, 2008.

[40] Ralston, A., Reilly, E. D., and Hemmendinger, D. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., GBR, 2003.

[41] Roffé, A. Propositional Logic - Natural Deduction.

[42] Schürmann, C. The Twelf Proof Assistant. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), TPHOLs '09, Springer-Verlag, pp. 79–83.

[43] Shapiro, Stewart, and Kissel, T. K. Classical Logic. *The Stanford Encyclopedia of Philosophy (Spring 2021)* (2021).

[44] Sieg, W. The AProS Project: Strategic Thinking and Computational Logic. *Logic Journal of the IGPL 15*, 4 (2007), 359–368.

[45] Singh, R., Gulwani, S., and Rajamani, S. Automatically generating algebra problems. In *AAAI* (April 2012), Microsoft Research.

[46] Sterling, L., and Shapiro, E. *The Art of Prolog (2nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[47] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*, 42 (1936), 230–265.

[48] van der Vlist, C. A solver and tutoring tool for logical proofs in natural deduction, 2019. Bachelor's Thesis.

[49] van Orman Quine, W. *Methods of Logic*. Harvard University Press, 1950.

[50] Verwer, S., Weerdt, M., and Zutt, J. A Tutoring System to Practice Theorem Proving in Fitch, 01 2005.

[51] Villadsen, J., From, A. H., and Schlichtkrull, A. Natural Deduction and the Isabelle Proof Assistant. In *ThEdu@CADE* (2017).

[52] Woolf, B. P. *Building Intelligent Interactive Tutors: Student-Centered Strategies for Revolutionizing e-Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

CHAPTER A

PROPOSITIONAL LOGIC NATURAL DEDUCTION ALGORITHM

This appendix describes the algorithm we used to find a natural deduction proof for propositional logic. The idea is to use a "satisfiability" algorithm (not to be confused with the Boolean satisfiability problem SAT... **is this needed? Probably not**). A wff w is satisfied when it is used in the reduction or expansion of another well-formed formula w'. In essence, if w is used to construct w', then w is satisfied. At a high level, we recursively compute goals for each premise, and if we can satisfy each goal, then the premise is satisfied. The procedure SATISFY uses rules for each axiom to determine if a premise can be either simplified or if, as a goal, it can be constructed using other premises. For example, suppose we have two premises A and B and we want to satisfy $A \wedge B$. SATISFY will recursively search through the well-formed formula (i.e., attempt to satisfy its operands from left-to-right) to determine if either have been previously satisfied. Given that A and B are already premises, and premises are, by default, satisfied, we can conlude that $A \wedge B$ is satisfiable. As another example, let us take the argument $P = \{A \supset (B \wedge \neg C),\ \neg(B \wedge \neg C)\}$ and $c = \neg A$. It is trivial to see that we can conclude c from the premises in P via a modus tollens rule. SATISFY works slightly different when this type of situation occurs. Because there is no way to individually solve intermediate formulas e.g., B, $\neg C$, the algorithm

searches for transformations and elimination rules e.g., modus ponens, modus tollens, disjunctive syllogism, transposition, material implication, etc., that may be applied to premises. In our provided example, we can apply modus tollens to the two premises and consequently satisfy $\neg A$. $\neg A$ is, therefore, added to P. The terminating condition is when c is satisfied, or equivalently, $c \in P$. Because there are numerous transformations that may be applied to premises, we will omit their direct inclusion in favor of a broad description of behaviors. To prevent unnecessary premises, once a premise is satisfied, it can never be "unsatisfied". Moreover, if a premise was constructed, it cannot be redundantly destructed or vice versa. For instance, suppose we have premises A and B. We can use a conjunction introduction $\wedge I$ rule to satisfy $A \wedge B$. The algorithm could, in theory, use a conjunction elimination $\wedge E$ rule to break $A \wedge B$ back down into its original components. Since such repeated applications blows up the size of P (leading to a potentially infinitely long proof), we heuristically prevent its occurrence.

Algorithm 1 Propositional Natural Deduction Satisfaction Algorithm

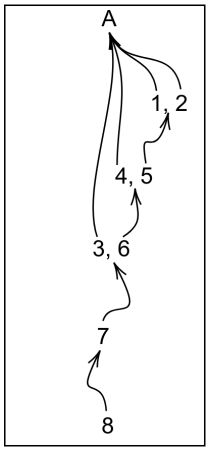
```
1: procedure PROVE(P, c)                    ▷ P is a list of premises, c is conclusion
2:     while c is not satisfied do
3:         for i ← 1 to P.length do
4:             if SATISFY(P[i]) then
5:                 P[i].satisfied ← true
6:         if SATISFY(c) then
7:             c.satisfied ← true
```

We have described how the algorithm generates a proof, but there lies the issue that unnecessary premises may have been generated as part of the path to said proof. In other words, there may be steps that contributed to a later step, but ultimately do not contribute to the proof path which derived the conclusion, deeming them as superfluous steps. Each step in the proof has parent steps which generate the child. For example, if A and B are steps to generate $A \wedge B$, we denote A and B as its parent

steps. These are used in a backwards path walk from the conclusion to find a solution to the proof. Note that there may be more than one solution to a proof; our algorithm simply picks the path which explores all parent steps until the original assumptions are reached. We denote an original assumption as the premises which were provided as part of the proof.

1.	$A \rightarrow B$	P
2.	$B \rightarrow \neg C$	P
3.	$D \vee C$	P
4.	A	P
5.	$A \rightarrow \neg C$	$1, 2$ HS
6.	$\neg C$	$4, 5$ MP
7.	D	$3, 6$ DS
8.	$D \vee P$	$7 \vee$I

(a) Propositional Logic Proof Example



(b) Path From Conclusion to Premises to Find Proof of (a)

Figure 8. Example of Finding a Valid Proof using Satisfaction Algorithm

CHAPTER B

RESULT GRAPHS

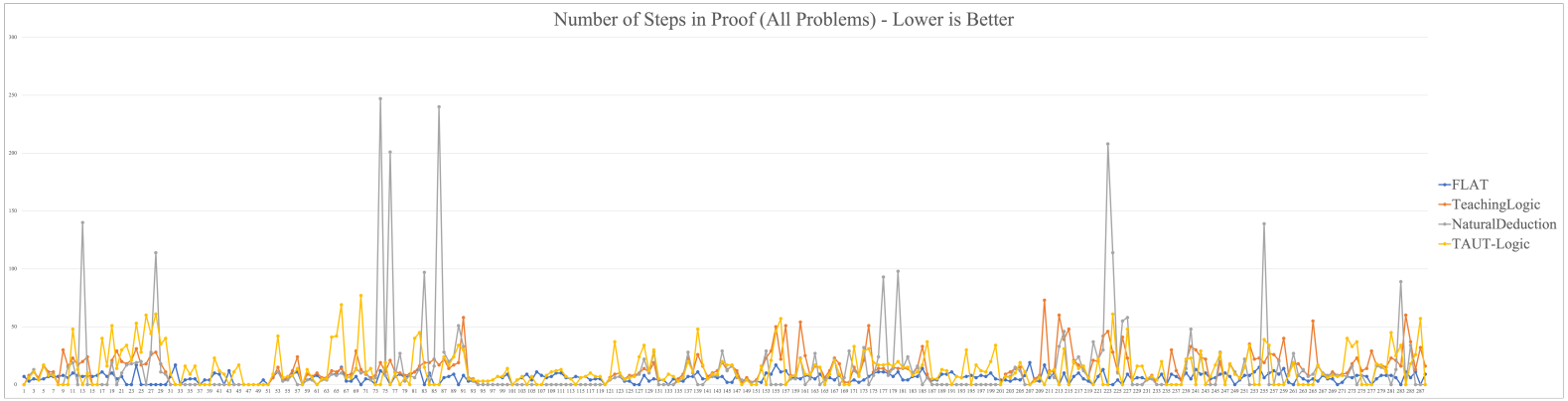


Figure 9. All Systems Natural Deduction Proof Line Count

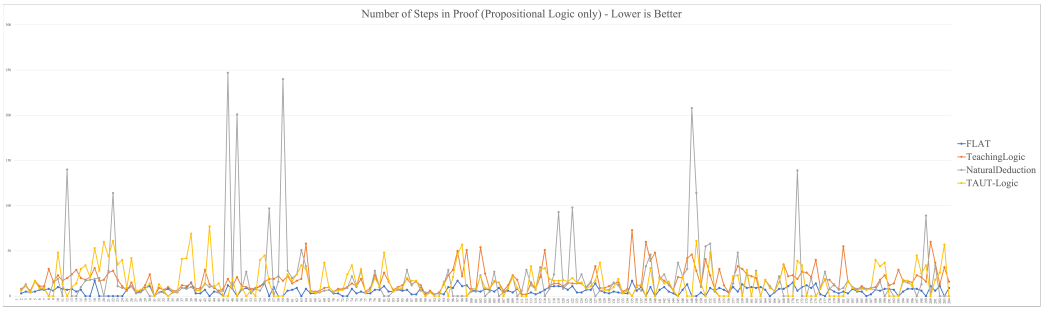


Figure 10. Propositional Logic Natural Deduction Line Count

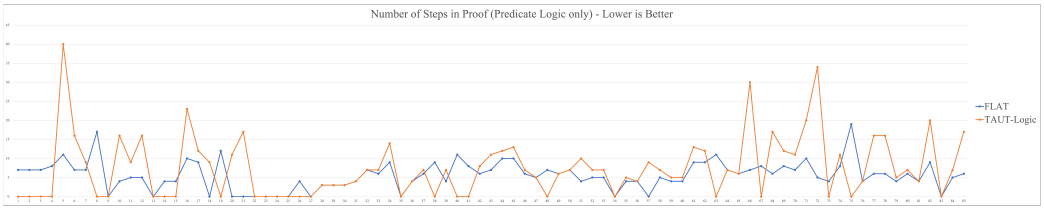


Figure 11. Predicate Logic Line Count