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. 39 pp.

The abstract page is a required component of the thesis/dissertation. The abstract should be a brief summary of the paper, stating only the problem, procedures used, and the most significant results and conclusions. Explanations and opinions are omitted. Remember to include the necessary information regarding any multimedia components included in the document. The abstract must be approved by your advisor/committee chair.

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 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 de jure input format, and testing their algorithms was far more cumbersome than I initially expected. This evolved into the desire for a gold standard syntax for both zeroth and first-order logic systems.

Table of Contents

List of Tables

List of Figures

CHAPTER I

INTRODUCTION

1 Overview

Formal logic, otherwise known as 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 communicative studies to reinforce critical thinking and improve deductive skills for argumentation. Per Stanford's Encyclopedia on Classical Logic [26], logic is a tool used for studying correct 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 [26]. As Hatcher [9] 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 with how to refute incorrect and, sometimes egregious, contentions. Formal logic's relation to computer science, in particular, ... **talk about how we can use formal logic for mathematical proofs, Boolean logic for circuitry, set theory, reasoning about program correctness (Hoare logic) etc.**

2 Contribution

3 Thesis Content

This thesis is broken up into three primary components. Chapter 1 introduces definitions, background, and our problem definitions. Chapter 2 reviews the related literature for prior work in this area. Chapter 3 discusses the primary 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).

4 Terminology

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

Definition 1 (Well-Formed Formula)**.** A *well-formed formula*, according to [24] is a string (formula) of syntactically-correct characters which conform (well-formed) to a language grammar.

Definition 2 (Proposition)**.** A *proposition* is a statement or claim that is either true or false.

Definition 3 (Proof)**.**

Definition 4 (Theorem)**.**

Definition 5 (Natural Deduction)**. Begin with a general definition, then the history**

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 a weaker 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. 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). In particular, 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 hints to a problem if 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. Automatic tutors and intelligent interactive tutors have a host of benefits: reducing stress on teachers by not having to create unique content for all students (a feat almost impossible unless the class size is small), and research has demonstrated

that students perform better on assessments (cite the intelligent tutors book 15 or the paper it referenced) in one-on-one tutoring sessions.

Show brief examples of other tutoring systems **e.g., algebra, ...**

1 Formal Logic Tutors

1.1 Propositional Logic

Propositional logic, also known as zeroth-order logic (or in other disciplines as sentence logic, sentential logic, Boolean logic, combinatorial logic, or propositional calculus), according to Hein [11], 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
Logical Conjunction	$\wedge$, &, $\cdot$, "and"
Logical Disjunction	$\vee$, |, $\|\|$, +, "or"
Logical Implication	$\supset$, $\rightarrow$, $\Rightarrow$, "if", "then"
Logical Biconditonal	$\leftrightarrow$, $\equiv$, $\Leftrightarrow$, "iff"
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 [7]. 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 [16], 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.

Propositional logic tutors... Abraham, Lukins, Croy, ...

Lukins et al. [19] 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 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 [27]. 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. [18] 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.

1.2 First-Order Logic

Cerna et al. [4] 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.

Mauco [20], Grivokostopoulou... [8]

1.3 Problem/Solution Generators

Ahmed et al. wrote..... Hladik... Amendola... LLAT... Graham defines a semantic tableau...

2 Specialized Logic/Theorem-Proving Programming Languages

What is a theorem prover? Ultimately, it's 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.

An early adaptation of a theorem proving logic programming language is Prolog, which uses horn clauses, or a sequence of literals separated by disjunctions, to prove goals. .....

Another form of theorem proving is known as inductive logic programming, coined by [cite Muggleton]. In summary: .....

Coq..., αleanTAP,..., Hoare logic, ...

3 Boolean Satisfiability Solver Input Formats

Cook proved that the Boolean satisfiability problem SAT is $\mathcal{NP}$-Complete [5], 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 to 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 [14] 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... **use this source [13]**

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, 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 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. Determining what exactly constitutes difficult in a logic class is tricky for the same reason. That is, it is a non-trivial job to evaluate whether one problem or topic is inherently more challenging than another.....**continue here**

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 the 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.

1.2 Experiment 2: Determining the Efficacy of Natural Deduction Software

Give a brief overview of what we did. A longer description is provided later.

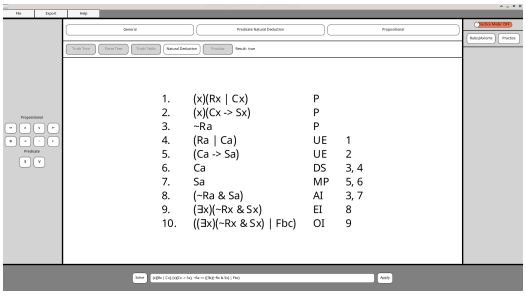
1.2.1 FLAT: Formal Logic Aiding Tutor

FLAT began as a collaborative project which extended LLAT: the Logic Learning Assistance Tool [6]. 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 IV), 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 (cite Gödel?).

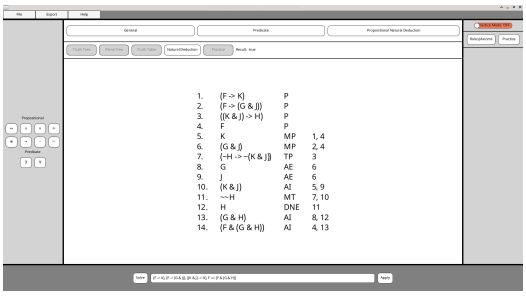
Mention FLAT's drawbacks (e.g., can't solve some "simple" proofs, weak proofs by contradiction...)

Table 1. 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 $F_{\mathcal{M}} \neq F'_{\mathcal{M}}$.



(a) Predicate Logic Proof



(b) Propositional Logic Proof

Figure 1. Two Natural Deduction Proofs Generated by FLAT

1.2.2 ANDTaP: Another 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 to a locally-installed (desktop) program.

Now that we have thoroughly discussed FLAT and ANDTaP, we will now explain, then discuss, the methods used to investigate several natural deduction systems in 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 a generated proof contains). Beginning students that receive a large and unwieldy result are likely to ignore its significance. Though, because the length of a proof

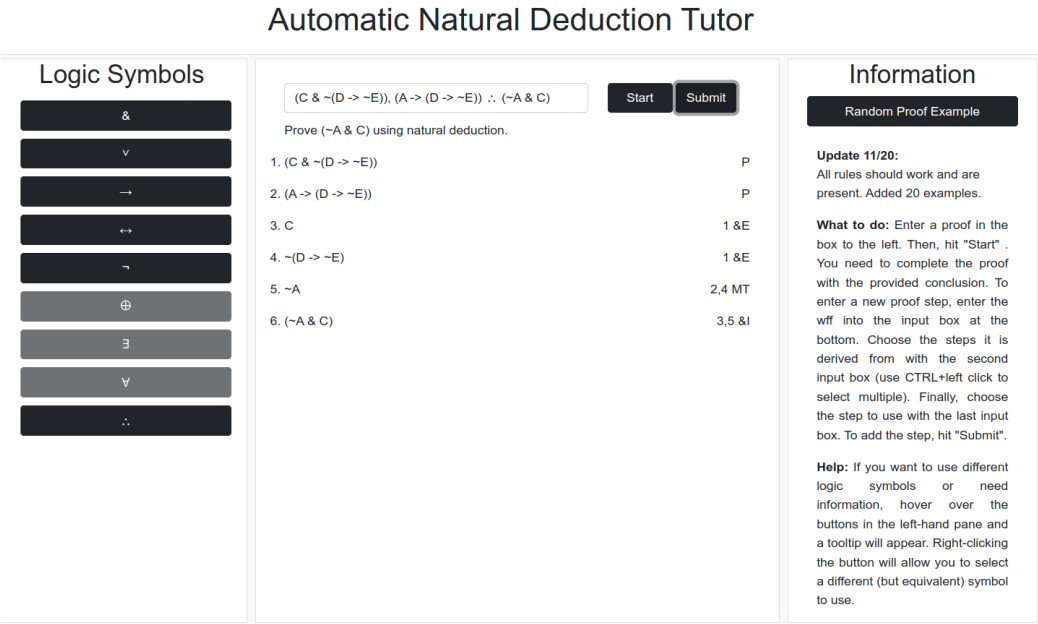


Figure 2. ANDTaP Tutoring System. ...

does not report a conclusive answer to its complexity, we recorded the number of rules/axioms a system used 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. 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). 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.

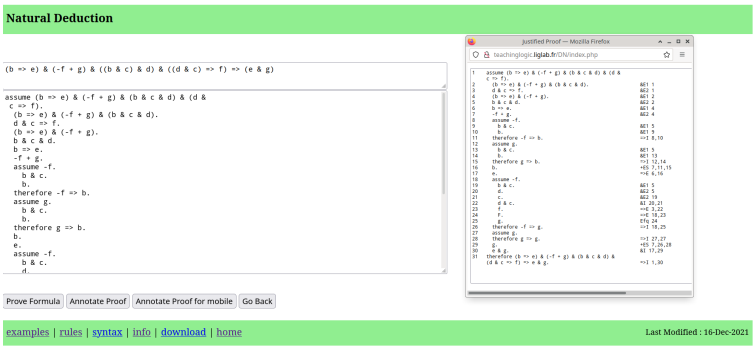


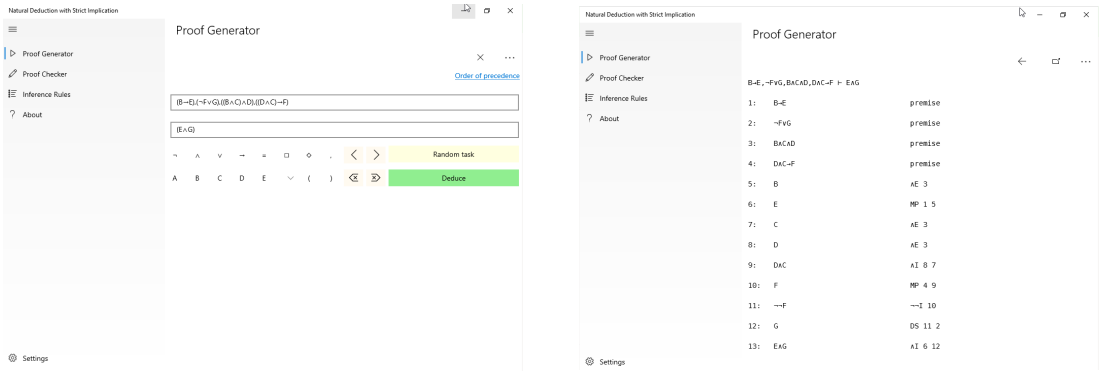
Figure 3. LIGLAB's Natural Deduction. Example proof showing the user interface and proof annotations.

Natural Deduction is a Windows 10 application designed by Jukka Häkkinen, and is the second natural deduction software we investigated. 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: it's 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. 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.

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



(a) A subfigure (b) A subfigure

Figure 4. A figure with two subfigures

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.

Table 2 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

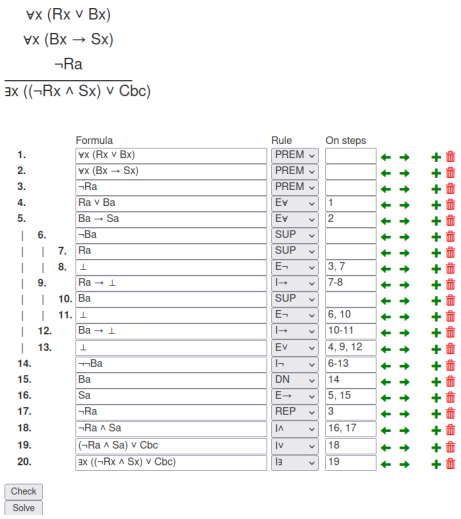


Figure 5. TautLogic's Predicate Natural Deduction. Write a caption...

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.

Table 2. 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

Discuss the general findings, then the challenges with inputting data... which lead to the desire for a gold standard!

Table 3. 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 4. 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 5. 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 e.g., 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..... **continue here**

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.

2.1 Zeroth-Order Logic Well-Formed Formula Representation

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

The bijective function f maps two sets $f\colon X \to Y$, 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 $|X| = |Y|$. N is unary logical negation, C is binary logical conjunction, D is binary logical inclusive disjunction, E is binary logical exclusive disjunction, I is binary logical implication, B is binary logical biconditional (if and only if), T is the truth function, and F is the false function. Note that the arity of any $\phi \in X$ must match the arity of its corresponding output connective in Y.

The bijective function g maps two sets $g\colon A \to B$, where A is the set of atomic literals $\psi \in A$ where ψ 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 Scheme-syntax 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 $\{\gamma\}^x$ to denote x applications of γ. 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`’

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

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 f and g where

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

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 $\mathcal{L}' = (f^{-1}, g^{-1})$ and $w' = M(\mathcal{L}, w)$, we can reproduce w using a simple stack-and-pop parsing evaluation approach (deterministic push-down automaton).

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. 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 f to $\mathbb{N}$, namely $f \mapsto \mathbb{N}$. Γ is designed to give connectives in f 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 means 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... **continue here**

2.1.2 Precedence Mapping Example

blah blah blah

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.

2.1.3 Zeroth-Order Logic Example 2

Quine's [28] syntax 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., $S_1 S_2$ 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 f and g where

$$f\colon \{\supset, \varepsilon, \vee, \equiv, -\} \mapsto \{I, C, D, B, N\}$$

$$g\colon \{p, q, r, s\} \mapsto \{a_1, a_2, a_3, a_4\}$$

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."

2.1.4 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).

2.1.5 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 the function g in $\mathcal{L}$ as follows:

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

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.

Let $\mathcal{L}$ be a quadruple $(p,\ q,\ r,\ s)$ where p is a connective mapping function, q is

a predicate mapping function, r is a constant mapping function, and s is a variable mapping function.

The bijective function p is identical to f in the sense that it maps two sets p: $X \rightarrow Y$, 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. Z is the universal quantifier, X is the existential quantifier, and V is the identity operator. 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.

The bijective function q maps two sets q: $A \rightarrow B$, 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 r maps two sets r: $C \rightarrow D$, 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 s maps two sets s: $E \rightarrow F$, 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 H 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*⟩})
| (⟨*connective*⟩ [‘ ’] ⟨*wff*⟩$^{\alpha}$)
| (⟨*quantifier*⟩ ⟨*variable*⟩ {⟨*variable*⟩} ⟨*wff*⟩)
| (⟨*identity*⟩ ⟨*literal*⟩ [‘ ’] ⟨*literal*⟩)

The above grammar states ...**continue here**

2.2.1 First-Order Logic Example

We will, once again, use a “standard” first-order logic language $\mathcal{L}$ and a formula w. $\mathcal{L}$ consists of the four functions p, q, r, and s where

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

$$q\text{: } \{P, Q, R\} \mapsto \{L_1, L_2, L_3\}$$

$$r\text{: } \{c, d\} \mapsto \{c_1, c_2\}$$

$$s\text{: } \{x, y, z\} \mapsto \{v_1, v_2, v_3\}$$

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

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

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

References

[1] 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.

[2] 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.

[3] Cerna, D. M., Kiesel, R., and Dzhiganskaya, A. A mobile application for self-guided study of formal reasoning. In *ThEdu@CADE* (2019).

[4] 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.

[5] 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.

[6] Crotts, J., Altamimi, A., Brantley, H. B. C., and Salou-Doudou, N. A visual improvement to the pedagogy of introductory logic, 2021.

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

[8] Grivokostopoulou, F., Hatzilygeroudis, I., and Perikos, I. An intelligent system for learning first order logic to clause form conversion, 2014.

[9] Hatcher, D. L. Why formal logic is essential for critical thinking. *Informal Logic 19*, 1 (1999).

[10] Hatcher, D. L. Why formal logic is essential for critical thinking. *Informal Logic 19* (1999).

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

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

[13] Heule, M., Iser, M., Jarvisalo, M., Suda, M., and Balyo, T. Sat competition 2011: Benchmark submission guidelines, 2011.

[14] Heule, M. J., and Martins, R. Sat and smt solvers in practice, September 2020.

[15] Häkkinen, J. Naturaldeduction, 01 2017.

[16] 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).

[17] Laboratory, G. C. S. Natural deduction, 2021.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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`.

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

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

[25] Roffé, A. Propositional logic - natural deduction.

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

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

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

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. **Now explain how it finds a path to a solution with that cool arrow diagram.**

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
```

CHAPTER B

RESULT GRAPHS

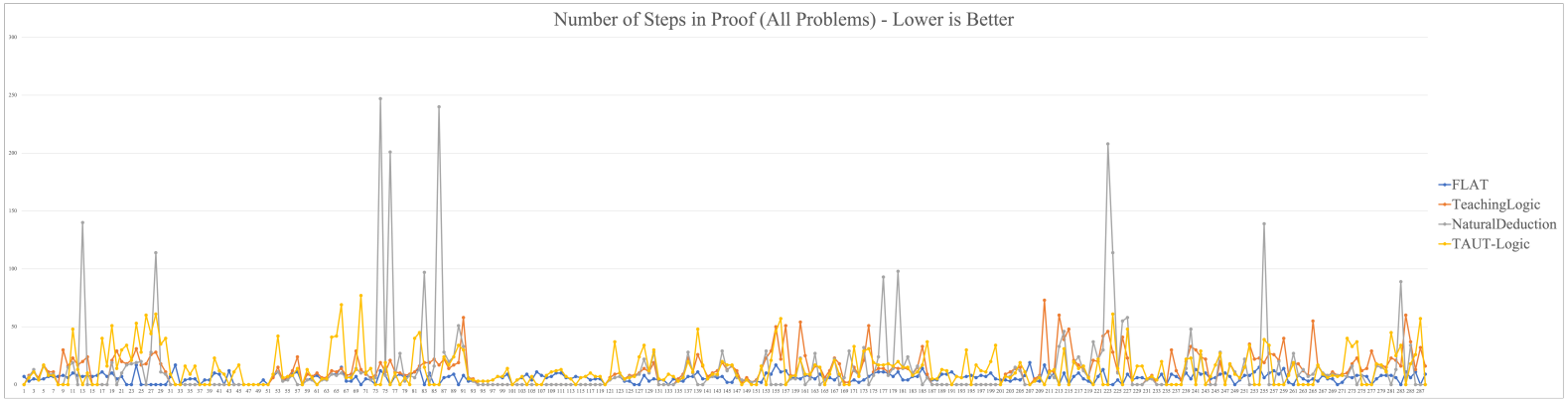


Figure 1. TautLogic's Predicate Natural Deduction. Write a caption...

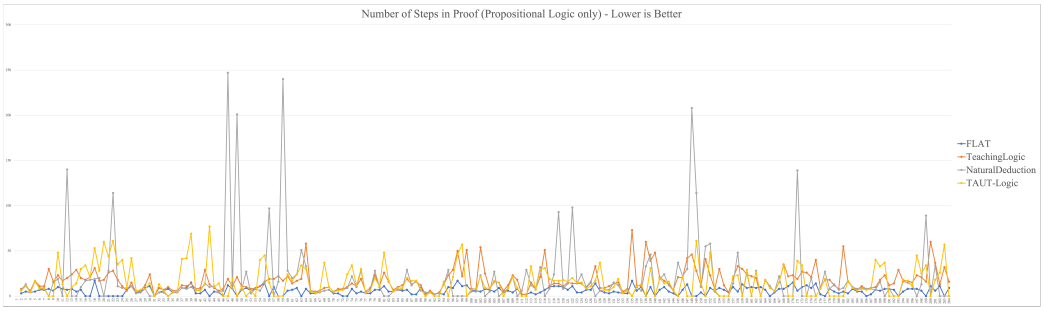


Figure 2. Propositional Logic Natural Deduction. Write a caption...

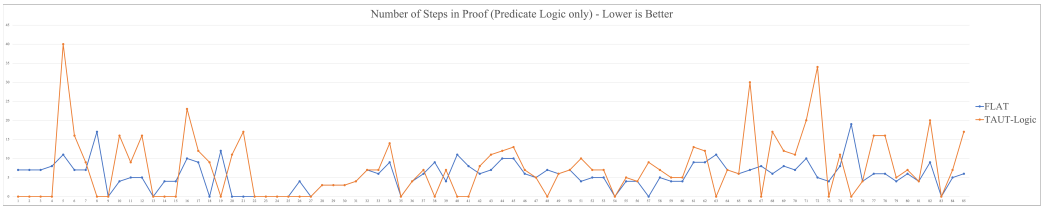


Figure 3. Predicate Logic Results Graph. Write a caption...