On an Enhanced Hands-on Approach to Formal Logic Education

L. Joshua Crotts[0000−0002−7513−5618] and Christopher Brantley

[1] Department of Computer Science
University of North Carolina Greensboro, Greensboro NC 27401, USA
ljcrotts@uncg.edu
[2] c_brantl@uncg.edu

Abstract. In this paper we describe FLAT (the Formal Logic Aiding Tutor): an educational tool to aid students in the learning of introductory formal logic, namely propositional and first-order predicate logic. The aim is to complement the traditional textbook pedagogy with visually appealing software that works well not only for student review and practice, but also for classwork or homework submission. Lastly, we provide potential solutions to problems relating to the automatic generation of formal logic exercises such as natural language translation and random expressions.

Keywords: Education, formal logic, automatic theorem prover, logic pedagogy

1 Introduction

Many students consider formal logic a challenging subject. Both propositional and first-order predicate logic introduce computer science and mathematics related topics which may confuse introductory learners. Further, its cumbersome and esoteric notation, alongside the abundance and reliance on proofs and theoretical concepts can discourage many students. Because formal logic builds on top of itself, when a student fails to understand trivial examples as well as the underlying axioms and rules, it is likely that more complex problems and exercises will only exacerbate their confusion and frustration.

With the reliance and emphasis on critical thinking and logical thought processing in both academic and industry environments, formal logic courses are an excellent way to gain exposure to these necessary skills for debate, argumentation, research, and many more subjects [10]. Most philosophy majors are required to take a course in formal logic, which provides an introduction to propositional logic and quantifier theory.

With the reliance and emphasis on critical thinking and logical thought processing in both academic and industry environments, formal logic courses are an excellent way to gain exposure to these necessary skills for debate, argumentation, research, and many more subjects [10]. Most philosophy majors are required to take a course in formal logic, which provides an introduction to propositional logic and quantifier theory.

1.1 Definitions

Throughout this paper, we will use terms and definitions concerning propositional and first-order predicate logic. For convenience, they are labeled here. Note that there exist

differences in notation and definition depending on the source, but ours are relatively universal to typical standards in their contexts.

Definition 1 (Well-Formed Formula/Sentence). *A well-formed formula (abbreviated as wff) is a propositional or first-order predicate logic sentence that is, by some classification, properly or well defined. In our case, the meaning of properly defined comes through our language grammar for both classes of logic.*[3]

Definition 2 (Connective). *A connective is a n-ary operator that coalesces n well-formed formulas together.*

Definition 3 (Model). *A model of a PL formula F is a truth value assignment of all atoms in F.*

Definition 4 (Constant). *A constant in a FOPL formula F is a lowercase letter between a and t. Constants represent static entities in some domain D.*

Definition 5 (Variable). *A variable in a FOPL formula F is a lowercase letter between u and z. Variables represent placeholders or generalizations of some domain D in a predicate.*

2 Motivation

Automatic logic tutors and theorem provers exist in many dimensions and formats, ranging from downloadable and executable software to modern and lively web applications. From our investigations, however, these systems and software often do not provide a beginner-friendly experience, nor do they provide the functionality we want students to engage with. Others like Near et al. [19] introduce fast theorem provers written in functional programming languages, but their broad intention is not to teach students, particularly non-computer science students.

For starters, there exist plenty of online truth table generators that work well not only in the formal logic domain, but also electrical engineering, computer science, and (discrete) mathematics domains. Some even provide immediate feedback for the user as they attempt to derive the truth table by hand [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 [15]. Beyond this, Lukins et al. [17] described and built the P-Logic Tutor system for propositional logic: a Java Web Start (JNLP) system. Today, their provided link is offline, so there is no way of evaluating or testing its functionality compared to its more modern counterparts. From the details the authors provide, though, students could enter their own data into the program and receive feedback on its correctness. One significant downside to the P-Logic Tutor is that it only covers/handles propositional logic across all its units and tools, as its name suggests. Moreover, its usage required students to log in for purposes of improving and personalizing the experience, a mandate that other systems lack. Requirements like this dissuade users from the tool who are not affiliated with their university. Another software-based

[3] The grammar is listed as a `.g4` file here: `https://tinyurl.com/FLATG4`

solution (i.e., executable outside the browser) is LEGEND by Vlist [24]. 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. Cerna et al. [6] developed **AX**olotl: a clean Android formal logic tutor which includes several types of proofs and tutorials for deriving examples, though its reliance on a file protocol to load examples is a bit cumbersome for the non-savvy student or instructor. Further, it appears to focus heavier on an accelerated natural deduction curriculum, whereas FLAT attempts to target absolute beginners at the material. Almost all systems we investigated only allow for propositional logic proofs or evaluation because of first-order predicate's infinite nature when applying universal quantifier rules as well as the general difficulty curve over propositional logic.

The overarching problem is twofold: firstly, solvers may give answers (and sometimes detailed derivations), but they may not enhance a student's understanding if they copy an answer but retain nothing else from doing so. Secondly, consider what a confused student may do when working on assignments: go online for assistance. Suppose a student is stumped on a troubling problem, and they search "propositional logic", "first-order predicate logic", or similar terms. The first few pages link to solvers that, again, are fulfilling but likewise are far too complicated for their current audience. Additionally, they are likely to encounter propositional and first-order predicate logic lectures from other universities or curricula. External sources pose the issue of encountering connectives that they do not use or have not (yet) been taught. Applying logic rules and axioms through a computer and receiving instant results helps students understand the fundamentals since they see and interact with a practical example of these abstract and theoretical concepts. A potential way of expressing these concepts (particularly in first-order predicate logic) comes through the declarative logic programming language Prolog. The standard modus ponens inference rule, for instance, is the building block of logical consequence in Prolog [11]. The issue with this and similar approaches, though, is that it relies on the student having knowledge of not only programming, but non-trivial concepts related to logic programming such as backtracking, unification, and recursion – practices that a beginner may be unfamiliar with.

3 Implementation

We set a goal of creating a tool that provides students an alternative to their traditional textbooks and other, sometimes unorthodoxy, methods of learning in favor of a visual and highly interactive interface. Our tool is called the Formal Logic Aiding Tutor, abbreviated as FLAT[4]. This application is based off an earlier work created by a group of undergraduate seniors in their respective capstone computer science course called LLAT (the Logic Learning Assistance Tool). This software was briefly presented and demoed at the local undergraduate creativity expo. The extension is to refine and provide an improved pedagogical experience, rather than being merely a tool that solves problems as inputted by the user, as well as to include new features and algorithms. Unlike many modern applications, FLAT was developed and built as a desktop application rather than mobile or web using the Java programming language. The motivation

[4] FLAT is located on GitHub `https://tinyurl.com/FLATREPO`

and reasoning were we wanted to have the support of fully developed and established libraries with full control over the tech stack in one location. Additionally, because formal logic uses symbols that are difficult to copy and paste (or type) on a touchscreen device, we decided to completely exclude mobile devices. We implemented the frontend using JavaFX and use several remote APIs (Application Programming Interfaces) for various features as described in the following subsections.

3.1 Features

Upon starting FLAT, an empty canvas greets the user with informational panes on all sides. The left displays usable connectives for the wff input field. The right will show information about a symbol and its uses (with examples, definitions, and alternate notations) and are enabled via CTRL-clicking the corresponding symbol in the left pane. The top presents three drop-down options for choosing algorithms to evaluate wffs with. The bottom is the text input field, or wff input area. The solve button to the left is used after an input is entered. Then, the user should select an appropriate algorithm from the drop-down lists which are automatically tailored to what the user inputs (e.g., if a student enters two wffs, only algorithms that use two or more wffs are displayed, or if they enter a predicate logic formula, only predicate logic algorithms are available). From there, they press ”Apply”. This will create several diagrams and results depending on the applied algorithm. These separate screens are switchable via buttons underneath the algorithm selector dropdowns. Beside these buttons is a ”Result” keyword, where ”determiner” algorithms display true or false (described in section 3.2). Trees in the center pane may be adjusted by zooming or dragging the mouse while clicking on a node/subtree, implemented with a custom tree-drawing algorithm.

The top pane lists three drop-downs: File, Export, and Help. In File, there is a settings pane where the user can customize primary and secondary colors of their environment to aid in accessibility (shown in figures 1 & 2). There also exists a limitations section where timeouts of algorithms may be altered (these are described further in section 4.4). Users may also export their work in the form of a parse or truth tree, or truth table via LaTeX source or PDF. PDF creation calls a remote API, so an internet connection is required for this function.

The right pane also has a switch to enable ”Practice Mode” in FLAT. This allows students to input problems and then test their understanding of the algorithms and the interpretation of their formulas. When this mode is enabled, the solver functionality is temporarily disabled so the students do not immediately see the answer to their input. Each algorithm has a prompt that tells the students what to do, as well as an information section describing in greater detail how they should approach the problem. Figure 1 demonstrates this where the student is tasked with proving a FOPL formula with natural deduction. Besides natural deduction, students derive truth tables, open/closed branches on truth trees, logical properties of formulas, operations related to variables/operators, and more. Section 3.2 along with table 1 provide thorough explanations.

Because some students learn foreign concepts better in their native language, and the fact that formal logic is relatively universal due to its close ties to mathematics and computer science, we implemented a language translation caching algorithm using Google Translate. These translations are then saved to a local file so the application does not

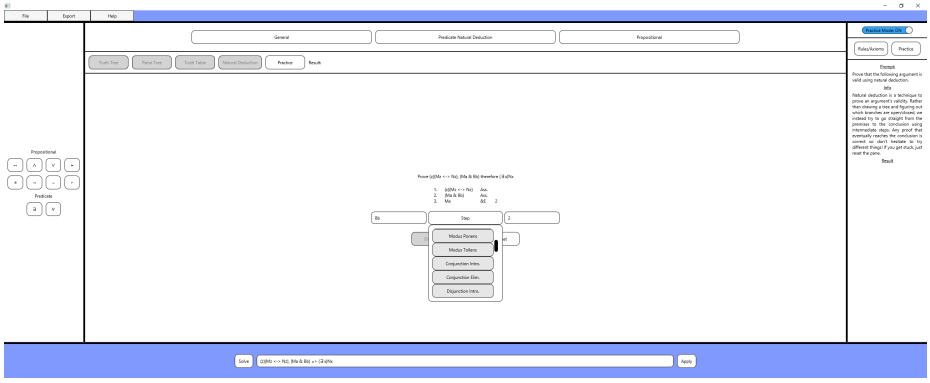

Fig. 1: Practice mode enabled with natural deduction.

recompute translations whenever languages are switched. We tested several languages, including Arabic, French, and Spanish for accuracy. We found that Latin-derived languages translate correctly more often than languages such as Japanese, Chinese, and Arabic. Our application is limited to Google's translation capabilities, so perhaps choosing a different API may yield improved results.

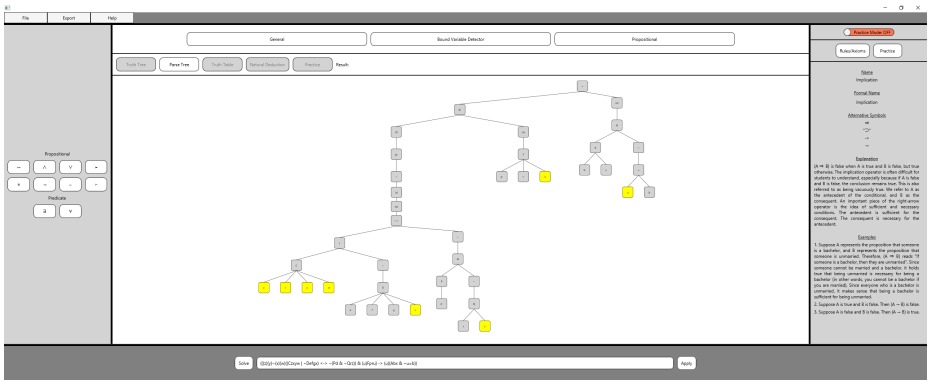

Fig. 2: FLAT home screen with bound variable detector.

3.2 Algorithms

Formal logic courses employ several types of exercises for students to demonstrate their understanding of the material. Many of the algorithms we chose to implement stem from our local introduction to formal logic course. However, these exercises and algorithms are derived from other formal logic courses and as such apply to a broad range of classes.

We label this section ”Algorithms”, but realistically, these are the features and ”subprograms” that the user may perform on their input. Users can input a formula to first determine if it is well-formed or not. As stated in section 1.1, the definition of well-formed depends on our grammar for the logic classifications. Section 3.3 describes this implementation in further detail.

Some schools, textbooks, subjects, etc. use wildly different notation. So, to reduce as much confusion among students as possible, we enforce parenthetical precedence around well-formed formulas, but allow students to mix and match symbols provided that the resulting formula remains well-defined. For example, the first-order logic sentence ”For all x, if x satisfies P then x satisfies Q and a does not satisfy P” is symbolically represented as (x)(Px $\rightarrow$ (Qx & $\sim$Pa)) but is also semantically equivalent to ($\forall$x)(Px $\supset$ (Qx $\wedge$ $\neg$Pa)). Different sources explicitly state a precedence for unary (quantifiers, negation) and binary connectives (conjunction, disjunction, biconditional, implication, exclusive-or). FLAT, though, prefers the use of parentheses as it requires students to enforce their own ordering of operands to lower the likelihood of common mistakes in their input. Allowing for implicit operator precedence is beneficial for those who understand the material in greater detail than a novice who may unintentionally make simple typos in their work.

Table 1 provides a subset of algorithms we have implemented. Some algorithms are only for propositional logic or first-order predicate logic (but not both), yet many work with both. Likewise, some algorithms require more than one well-formed formula to produce a meaningful result. Each algorithm is supplemented with a short description. Most algorithms fall into one or two categories: determiners or detectors. Detectors return a list of results. This list can be empty, a singleton, or more. For instance, if the user runs the Free Variable Detector on a first-order predicate logic sentence, it will return a list of all free variables in that sentence. On the contrary, determiners return a Boolean result for the supplemented algorithm. For example, if the user runs the Open Sentence Determiner on a first-order predicate logic sentence, it will return true if the sentence is open, and false otherwise. Some algorithms we have implemented are not listed as they require further investigation. The most complex of which is the structure that builds the underlying representation of the user’s input: the parse tree, although a more accurate description would be the abstract syntax tree.

3.3 Parsing Input

According to Aho et al. [1], a parse tree is a pictorial representation of the top-down evaluation of a string according to some grammar, whereas an abstract syntax tree (abbreviated as AST) is a structure where parents of a node are operators, and the children are operands. Each node in the AST represents a well-formed formula that may be structured recursively of sub-well-formed formulas. The initial parse tree is used to ensure the formula entered is valid on a syntactic level. ANTLR (ANother Tool for Language Recognition), a lexing and parsing library [20] has been used. If the user enters an invalid formula, a relevant error message is displayed. Since a significant component of formal logic relies on the structure of a wff, quality and meaningful error messages are paramount as they help the user understand where they went wrong (see Figure 4). After

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

the formula is validated, it is sent forward to the AST representation. As mentioned earlier, well-formed formulas are recursive in nature, so using the AST as a backbone for algorithm results was obvious. If a well-formed formula W is constructed with several sub-well-formed formula children and we want to compute the result of some algorithm A, we impose a paradigm that ”detector” algorithms always return a list of well-formed formula nodes. Consequently, if the algorithm has reference to the root of the AST, it can pinpoint which nodes are answers produced by A, and display those in the graphical interface. For example, suppose the user wants to find all bound variables to the formula F = ((x)(Px & Qyx) $\equiv$ $\sim$(y)(Pya & $\sim$Qbx)). Recall the definition of a bound variable from Table 1. From this, we see that the first two occurrences of x are bound to the universal quantifier (x). Similarly, the second occurrence of y is bound by the negated universal quantifier $\sim$(y). All other variable occurrences are free. Representing this as an abstract syntax tree follows naturally and therefore, an algorithm can perform a linear search to compute the query while knowing exactly what type of node resides at any arbitrary step. Computationally, this implies no backtracking or any unnecessary invariants that do not contribute to the algorithm–-everything is embedded in the structure of the formula, so algorithms need to do very little if the underlying representation is, for whatever reason, altered.

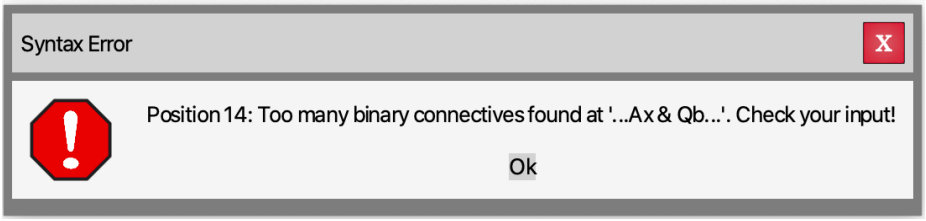

Fig. 3: Syntax error when evaluating (x)(Px $\rightarrow$ Ax & Qbc).

3.4 Random Formula Generation

Another prominent algorithm missing from Table 1 is the generation of random propositional and first-order predicate logic formulas. Hladik [12] and Amendola et al. [2] have investigated this problem at a theoretical level, whereas our approach is much simpler. We constructed a probabilistic recursive algorithm to generate formulas that aims to create unique and well-defined wffs while avoiding complicated methods that try to perfect the generation. Since FLAT does not automatically try to solve generated formulas, it was of less importance to implement a highly efficient (and provably correct) algorithm. Our method uses static probabilities for generating unary and binary connectives which decrease as each are used/generated. Random formulas are recursively constructed such that as they grow in complexity, the more likely it is for generation to terminate (to prevent incoherently long formulas). Predicates, constants, and atoms are chosen at random, but in order to maintain a level of consistency, these are saved to a list. Thus, when generating more predicates, constants, and atoms, we can either poll from this list or generate unused ones. Additionally, because predicates that share the same letter but

different arities are not well-formed, storing previously-generated predicates with their arities helps tremendously. The following is a list of four formulas generated from FLAT that students may use for practice. 1. ∼(V → (M & ∼V)); 2. ∼(Lvv ↔ ∼(∃w)∼(z)(Laa → Qua)); 3. ∼∼(Buujx → ∼(∃u)(∃v)Bvjwq); 4. ∼(∼(v)Nvvo & Gemme);

3.5 Argument Validity

Determining whether an argument is deductively valid or not is crucial in formal logic as it determines whether we can deduce the soundness of the argument. FLAT provides two methods of proving if an argument is valid: semantic tableaux and natural deduction.

Graham [21] defines a semantic tableau as a structure with connected nodes and branches to tips or leaves, with a root node. This is effectively the definition of a n-ary tree. A semantic tableau, or truth tree, can be used to prove an argument is valid or not by listing the premises and negated conclusion from top to bottom, then deriving contradictions. A truth tree branch is closed if there is a branch from a node with wff P to a node with wff ∼P. In other words, there exists a node with its negation as an ancestor. If a truth tree contains only closed branches, it is valid. Truth trees offer a different perspective on the validity of an argument, because students only have to derive contradictions. Rather than being required to decide which rule to apply at all steps, building a tree is almost uniform and mechanical. As such, creating propositional logic truth trees is unambiguous since they are always decidable. Conversely, first-order predicate logic is only semi-decidable due to the universal quantifier and identity operators. Since these operators can be used infinitely many times, the student must be able to determine where to apply the appropriate rule when searching for a contradiction. For software, however, this is trickier because, even though an algorithm may eventually derive a truth tree, it may be so large and unwieldy that it is almost meaningless for anyone investigating it. FLAT uses heuristics to determine when a truth tree ought to apply a universal or identity decomposition. For example, only reapplying constants that have previously been used on a branch, and an iteration timeout to prevent infinite generation.

Natural deduction is the other popular method of determining an argument's validity. FLAT uses a recursive goal-searching algorithm to determine if a list of premises and conclusion are deductively valid. We will describe our approach and its comparison to other software in the next section.

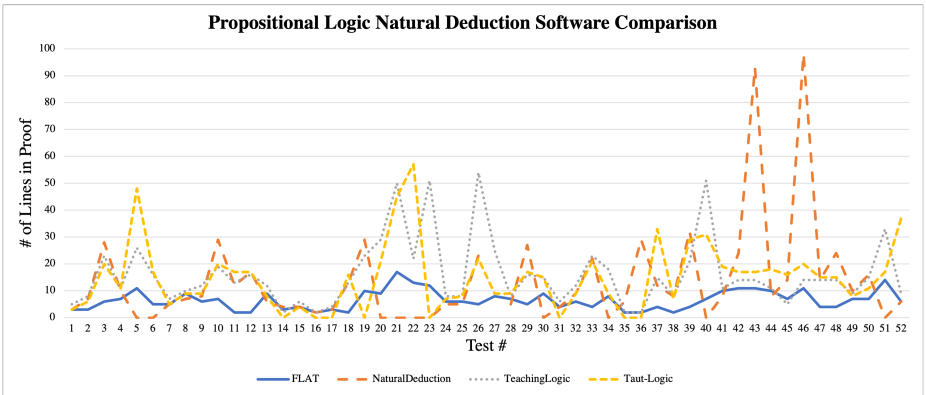

Fig. 4: Number of lines in each propositional logic proof software surveyed.

4 Results and Discussion

4.1 Related Work

Because natural deduction has close ties with discrete math, computer science, and philosophy, its appearance in online solvers is to be expected. To our surprise, however, there were not many propositional logic natural deduction generators available, and even less so for first-order predicate logic. To determine the effectiveness of FLAT's natural deduction algorithm, we compared its efficiency (measured in number of lines in the generated proof) to three different systems freely available online: TAUT from the Buenos Aires Logic Group [3], NaturalDeduction from Jukka Häkkinen [14], and Natural Deduction from the Grenoble Computer Science Laboratory [8]. Our test suite consisted of 52 propositional logic formulas varying in complexity. All 52 were deductively valid arguments with some requiring only one line to deduce the necessary conclusion[5]. We have discovered that many solvers use either an indirect or conditional proof approach to solving natural deduction problems. FLAT does not currently handle conditional proofs (with nested subproofs), so any problem that requires a conditional proof is unsolvable. None of the examples in our test suite, however, required the use of a subproof. FLAT uses several syllogisms and axioms to search for sub-goals that other systems manually derive which slows computation time and over-complicate the proof. We also found that some systems did not allow the use of certain symbols or input, such as the biconditional operator, uppercase propositions, or arbitrary letter propositions (requiring us to alter our test cases for these systems). Figure 2 shows a comparison of the four systems using the metric described above. Some test cases generate a line count of 0. This indicates that the test was unsolvable in that system due to a symbolic restriction or resource limitation.

Section 4.3 describes related work to translating natural language sentences to formal logic alongside our proposed solution.

[5] The 52 tests consist of `.in` and `.out` files; they are posted here `https://tinyurl.com/FLATTESTS`

4.2 Future Work and Consequences

Several considerations and apparent consequences arose throughout the development of the research project. The first prominent concern comes through students abusing this system on exams and homework assignments. A student can easily input a given formula to the system that they are supposed to compute by hand, then claim they performed the computation without outside assistance. Our posed counterargument is that these solutions and theorem solvers/provers exist elsewhere, and because this is a publicly available system, abuse is possible. Our hope is that its emphasis on aid and tutorial, rather than being a blatant ”plug and chug” solution will dissuade those looking for an easy workaround. To reiterate, the goal is to provide a functional and easy-to-use environment that teaches.

As described in section 3.1, FLAT has two convenient export features via PDF and LaTeX source code (.tex). A problem with the latter is that because it is easily editable, any means to protect (it) against plagiarism contradict why we implemented it from the start. Additionally, students can generate a PDF and plagiarize it as their own. To counter this, we propose placing a watermark over the graph/diagram or in the document's margins. Of course, like other solutions, this also has its flaws, which is why we stress FLAT's intended purpose and why it should be used to aid students but not serve as a personal homework solver.

As we briefly mentioned, textbooks provide examples and practice problems for students. Online learning management purveyors may, in a similar vein, sell software to universities that professors use for homework and other assessments. Systems like Pearson's MyMathLab [23] for mathematics courses generate random problems for each student so they cannot easily collaborate or search for solutions online. In formal logic, the issue is that problems sometimes do not make sense when procedurally generated. In the case of first-order predicate logic, it is possible to generate problems that are unsolvable by automatic systems. Furthermore, if an instructor would rather use and create non-symbolic problems, a generator is often too much work due to the complex nature of natural language generation. Textbooks may provide solution sets to questions, but these are sometimes locked behind expensive paywalls, are only given to instructors to circumvent those who use textbook questions as assessments, or only give brief answers instead of lengthy descriptions of the derivation to said solution. FLAT's deterministic generation algorithm creates provably well-formed formulas for both kinds of logic, but they may be (automatically) unsolvable. The upside to these is that the formulas can serve as supplemental practice for a student to solve by hand.

4.3 Natural Language to Formal Logic

Translation of prose to formal logic syntax provides students a general introduction into how words and phrases are symbolized. Singh et al. [22] use a machine learning and neural network approach to tackle this problem whereas Bansal [4] uses constraints and rules to match keywords with first-order predicate logic symbols. Our potential computerized solution is to use an ad-hoc generator that pattern matches to symbols and notations which create a problem that makes sense in some domain. For instance, suppose an instructor wants their students to practice translating (English) sentences to first-order

predicate logic, but wishes to not spend a lot of time manually writing custom problems (with a similar desire to not use published book or online problems). A phrasal template could be used for an algorithm to insert cue words, creating an arbitrary number of premises and conclusion. This, of course, could generate invalid arguments which may or may not be desired, which can be prevented by only allowing the generation of valid argument forms e.g., modus ponens, modus tollens, or any valid logical syllogism. Sound sentences, or those that ”produce” semantically correct information rely on relations between words. A set of propositions such as ”All cows are televisions.”, ”No television is a war.”, ”Therefore no cow is a war.” convey a deductively valid argument but make no sense semantically.

Bansal [4] uses a corpus of sensible, manually annotated natural language examples which were then converted to first order logic. However, our idea is to procedurally generate arbitrary examples that make syntactic and semantic sense. A lexicon of determiners and connectives could serve as the phrasal template, but nouns, adjectives, verbs, and others should not be predetermined. There has been some previous experimentation with this idea in Stanford's CoreNLP framework [18], the Natural Language Toolkit (NLTK) [16], and other systems that use semantic parsing such as Boxer [5]. We plan to experiment further with CoreNLP since it incorporates POS (part-of-speech tagging), named-entity recognition, coreferencing, and relation extraction. These components, if used together, could build a rich corpus of arguments, thus providing students the necessary tool to translate many examples of English sentences to formal logic syntax. Perikos et al. [9] present a conversion process similar to our idea, but we plan to extend it to not require a corpus of terms or propositions to poll from, meaning the instructor does not have to provide a dictionary themselves.

4.4 Limitations

We acknowledge the lack of certain features and capabilities in FLAT. As discussed in section 3.5, the system enforces a timeout on the maximum number of iterations a proof or generator may continue without finding a solution (this also holds true for the number of permitted atoms in a truth table). Raising these numeric limits in the advanced settings menu increases the likelihood of FLAT discovering a solution, but this, in turn, increases program execution time. Further, because of the algorithms used for natural deduction and first-order predicate logic, it is possible that a proof or truth tree is not deducible due to its complexity. Modifications to the algorithm to improve heuristics and shortcuts are planned for future development to either remove these iteration-based limitations altogether or increase them.

When comparing our software to other preexisting options in section 4.1, we used the number of lines in the natural deduction proofs generated, where lower is better. Of course, any proof that appropriately and validly arrives at the conclusion is correct. Nonetheless, a simpler solution is generally better as students may struggle with any natural deduction proof. So, a generator that constructs simpler solutions is favored over one that has superfluous and cumbersome derivations. As we continue to improve the algorithms, we plan to use different metrics in measuring FLAT's performance.

We also recognize that our natural language translation method is naive. To our knowledge, however, this is the first experimentation with random premises and propo-

sitions to generate an argument, and we hope to improve this development in the near future.

Regarding FLAT's pedagogical value, we plan to add a feature that allows instructors to load a file with wffs for the student to practice. Also, improving the interface with different error messages, hints, and more user options is a desired goal.

Finally, an evaluation of the effectiveness, intuitiveness, and pedagogical impact of FLAT via an in-person observational study with students in the near future is in the works.

5 Conclusion

We have presented a new educational tool for propositional and first-order predicate logic. With several algorithms and customization features, our aim has been to provide beginner-to-intermediate students and instructors a comfortable and easy-to-use environment for learning and practicing introductory formal logic. We have also discussed groundwork for natural language to formal logic generation and translation as well as methods of generating random logic formulas.

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. "Generating Hard Random Boolean Formulas and Disjunctive Logic Programs". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. Melbourne, Australia: AAAI Press, 2017, pp. 532–538. ISBN: 9780999241103.

[3] Ariel Roffé. *Propositional logic - Natural deduction*. URL: `https://www.taut-logic.com/`.

[4] Naman Bansal. "Translating Natural Language Propositions to First Order Logic". MA thesis. Kanpur, India: Indian Institute of Technology Kanpur, 2005.

[5] Johan Bos. "Open-Domain Semantic Parsing with Boxer". In: *NODALIDA*. 2015.

[6] David M. Cerna, Rafael Kiesel, and Alexandra Dzhiganskaya. "A Mobile Application for Self-Guided Study of Formal Reasoning". In: *ThEdu@CADE*. 2019.

[7] Bastion Fennell, Eysa Lee, and Thomas Kim. *Truth Table Creator*. Accessed: 2021-07-04. 2020. URL: `https://www.cs.utexas.edu/~learnlogic/truthtables/`.

[8] Grenoble Computer Science Laboratory. *Natural Deduction*. URL: `http://teachinglogic.liglab.fr/DN/`.

[9] Foteini Grivokostopoulou, Isidoros Perikos, and Ioannis Hatzilygeroudis. "Assistant Tools for Teaching FOL to CF Conversion". In: *Artificial Intelligence Applications and Innovations*. Ed. by Lazaros Iliadis, Ilias Maglogiannis, and Harris Papadopoulos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 306–315. ISBN: 978-3-642-33409-2.

[10] Donald L. Hatcher. "Why Formal Logic is Essential for Critical Thinking". In: *Informal Logic* 19 (1999).

[11] James L. Hein. *Prolog Experiments in Discrete Mathematics, Logic, and Computability*. 2009.

[12] J. Hladik. "A Generator for Description Logic Formulas". In: Edinburgh, Scotland, UK., 2005.

[13] Rodger L. Jackson and Melanie L. McLeod. *The Logic of our Language: An Introduction to Symbolic Logic*. Broadview, 2015.

[14] Jukka Häkkinen. *NaturalDeduction*. Jan. 2017. URL: `http://naturaldeduction.org/`.

[15] Kenneth R. Koedinger et al. ""Opening the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration."". In: 3220 (2004). DOI: `https://doi.org/10.1007/978-3-540-30139-4_16`.

[16] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*. 2002.

[17] Stacy Lukins, Alan Levicki, and Jennifer Burg. "A Tutorial Program for Propositional Logic with Human/Computer Interactive Learning". In: SIGCSE '02 34.1 (2002), pp. 381–385. ISSN: 0097-8418. DOI: `10.1145/563517.563490`. URL: `https://doi.org/10.1145/563517.563490`.

[18] Christopher Manning et al. "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 55–60. DOI: `10.3115/v1/P14-5010`. URL: `https://aclanthology.org/P14-5010`.

[19] Joseph P. Near, William E. Byrd, and Daniel P. Friedman. "αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic". In: *Logic Programming*. Ed. by Maria Garcia de la Banda and Enrico Pontelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–252. ISBN: 978-3-540-89982-2.

[20] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999.

[21] Graham Priest. *An Introduction to Non-Classical Logic: From If to Is*. 2nd ed. Cambridge Introductions to Philosophy. Cambridge University Press, 2008. DOI: `10.1017/CBO9780511801174`.

[22] Hrituraj Singh, Milan Aggarwal, and Balaji Krishnamurthy. "Exploring Neural Models for Parsing Natural Language into First-Order Logic". In: *ArXiv* abs/2002.06544 (2020).

[23] Kirk Trigsted, Kevin Bodden, and Randall Gallaher. *MyMathLab Developmental Mathematics: Basic Mathematics, Beginning Algebra, Intermediate Algebra – Access Card – PLUS EText Reference*. 1st ed. Pearson, 2014. ISBN: 032195405X.

[24] Christiaan van der Vlist. *A Solver and Tutoring Tool for Logical Proofs in Natural Deduction*. Bachelor's Thesis. The Netherlands, 2019.