CMPE 58S: Sp. Tp. Computer Aided Verification

Project: Interactive Propositional Logic Engine using Natural Deduction

Mehmet Utkan Gezer

2018700060

December 6, 2018

1 Introduction

Propositional logic, also known as propositional calculus, is the branch of logic dealing with propositions. Natural deduction is a way of handling propositions, whereby we apply a collection of rules to infer new conclusions from zero or more premises, all of which themselves are propositions.

For a more detailed introduction, we will be giving some definitions for propositions and natural deduction, along with its semantics.

1.1 Propositions

Propositions are declarative sentences with a truth value either **true** or **false**, and can be recursively defined as the composition of some other, smaller, propositions. Smallest of propositions are called *atomic* propositions, which are *indecomposable* and are given a unique symbol for the declaration they make.

Compositionals are propositions composed of other propositions, i.e. propositions that are not atomic. In propositional logic, there are 4 different ways of obtaining a *compositional*:

1. Negation ($\neg$): Negation of a proposition. E.g. "it does <u>not</u> rain" is the negation of "it rains". If the original proposition has the truth value **true**/**false**, then its negation has the truth value **false**/**true**, respectively.

2. Conjunction ($\wedge$): Conjunction of two propositions. E.g. "it does not rain <u>and</u> I am 25" is the conjunction of the propositions "it does not rain" and "I am 25". The compositional has the truth value **true** only if both of its constituents have the truth value **true**, and otherwise **false**.

3. Disjunction ($\vee$): Disjunction of two propositions. E.g. the non-exclusive sense of the statement "it is sunny <u>or</u> I am 5" is the conjunction of the propositions "it is sunny" and "I am 5". The compositional has the truth value **false** only if both of its constituents have the truth value **false**, otherwise (if either one or both of its constituents have the truth value **true**) the compositional is **true**.

4. Implication ($\rightarrow$): Implication of the second proposition (*consequent*) by the first proposition (*antecedent*). E.g. "<u>if</u> it is sunny <u>then</u> I am happy" is the implication of the proposition "I am happy" by

the proposition "it is sunny". The compositional has the truth value **false** only if the antecedent and consequent are **true** and **false**, respectively.

To formally define the *language* of propositional logic's *well-formed formulas*, we give its Backus Naur form (BNF) as follows:

$$\begin{array}{rcl} \langle\phi\rangle & ::= & \langle\text{atom}\rangle \mid \Big(\neg\langle\phi\rangle\Big) \mid \Big(\langle\phi\rangle\wedge\langle\phi\rangle\Big) \mid \Big(\langle\phi\rangle\vee\langle\phi\rangle\Big) \mid \Big(\langle\phi\rangle\rightarrow\langle\phi\rangle\Big) \\ \langle\text{atom}\rangle & ::= & p \mid q \mid r \mid \ldots \mid p_1 \mid p_2 \mid \ldots \end{array}$$

While this definition requires each use of a compositional operator to introduce a new pair of parenthesis for the newly generated proposition to be a well-formed formula, in practice we encounter propositions with many of those parenthesis omitted. In absence of parenthesis to enforce an explicit precedence of operator application, following precedence conventions are consulted:

- $\neg$ binds most tightly, followed by $\wedge$, $\vee$, and finally $\rightarrow$, in the given order.
- Implication ($\rightarrow$) is right-associative, i.e. the rightmost implication shall be evaluated the first.

Some books, including *Logic in Computer Science* by Huth and Ryan [1], regard $\wedge$ and $\vee$ operations as equal in precedence, in which case a proposition like $p \vee q \wedge r$ should either be regarded as unintelligible, or the same as $((p \vee q) \wedge r)$. In our program, we will be adhering to the above listed convention.

1.2 Natural deduction

Natural deduction is a way of reasoning about a given set of propositions and inferring new ones from them. Using a collection of *proof rules*, natural deduction allows us to come up with *conclusions* starting off by a set of *premises*. This relation between premises and conclusions are formalized by expressions called *sequents*, such as:

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi.$$

Sequents with no premises are also valid sequents, and are called *theorems*.

Rules of natural deduction, also known as the previously mentioned proof rules, are at the heart of natural deduction, and also of this project. They allow us to establish the validity of newer propositions, using the previously validated list of propositions.

At any step, we may introduce an *assumption* to the proof, which, however, will introduce an assumption box along with it. The top line of the assumption box is drawn right above the step where the assumption is introduced. We may close an assumption box after any step. Assumption boxes restrict the access to the propositions within from the steps outside. As a result, a proof rule may only be applied to propositions with following properties:

- Their validity must have been previously established, and
- Either they must be outside of any assumption box, or the assumption box they are in must not have been closed, yet.

We refer to such proof steps as the *accessible steps*.

A proof of a sequent is complete when the validity of the sequent's conclusion is established using only the rules of natural deduction and starting off with only the premises of the sequent. It is important to note that an established proposition may not be the conclusion if it is found within an *assumption box*.

We have taken Huth and Ryan [1] as our resource for the natural deduction rules. Here is a list of all natural deduction rules we have embedded into our program, given in sequents:

Name		**Rule**
Conjunction	Introduction	$\phi, \psi \vdash \phi \land \psi$
	Elimination #1	$\phi \land \psi \vdash \phi$
	Elimination #2	$\phi \land \psi \vdash \psi$
Disjunction	Introduction #1	$\phi \vdash \phi \lor \psi$
	Introduction #2	$\psi \vdash \phi \lor \psi$
	Elimination	$\phi \lor \psi, \boxed{\phi \cdots \chi}, \boxed{\psi \cdots \chi} \vdash \chi$
Implication	Introduction	$\boxed{\phi \cdots \psi} \vdash \phi \to \psi$
	Elimination	$\phi \to \psi, \phi \vdash \psi$
Negation	Introduction	$\boxed{\phi \cdots \bot} \vdash \neg\phi$
	Elimination	$\phi, \neg\phi \vdash \bot$
false	Elimination	$\bot \vdash \phi$
Double negation	Introduction	$\phi \vdash \neg\neg\phi$
	Elimination	$\neg\neg\phi \vdash \phi$
MT (Modus Tollens)		$\phi \to \psi, \neg\psi \vdash \neg\phi$
PBC (Proof by Contradiction)		$\boxed{\neg\phi \cdots \bot} \vdash \phi$
LEM (Law of Excluded Middle)		$\vdash \phi \lor \neg\phi$
Copy		$\phi \vdash \phi$

Figure 1: Rules of Natural Deduction

The boxed premises such as $\boxed{\phi \cdots \psi}$ on implication introduction, is an assumption box in the proof that starts right before the proposition ϕ and ends right after the proposition ψ. To make a step on the proof, one of the proof rules given above must be used. To use a rule, we need accessible propositions and/or assumption boxes that fits to the rule's sequent's propositions. Only if we are able to fulfill those *requirements* of the rule, we then may establish the validity of a new proposition according to the definition of the rule, and specifically the rule's sequent's conclusion. In natural deduction, this is the only way to make a proof step and approach to the ultimate conclusion.

Accessibility of an assumption box is defined similar to the accessibility of individual proof steps. This time, not a single step but the assumption box as a whole should be accessible as a single entity.

A proof step is a declaration, and should have a *rationale* stated next to it. Following is a complete list of valid rationales for a proof step:

Rationale	**Can be used next to propositions which...**
Premise:	... are found in the premises of the sequent to be proved.
Proof rule:	... fit to the conclusion of a proof rule, and only if the corresponding premises of that proof rule is available and validated in a previous step. Those steps must be referenced in the rationale in the same order as they appear in the proof rule.
Assumption:	... appear as the first proof step of an assumption box.

Here is an example of a proof for the sequent $p \vee q, \neg q \vee r \vdash p \vee r$:

1.	$p \lor q$	premise
2.	$\neg q \lor r$	premise
3.	$q \lor \neg q$	*LEM*
4.	q	assumption
5.	$\neg q$	assumption
6.	$\bot$	$\neg_e$ 4, 5
7.	r	$\bot_e$ 6
8.	r	assumption
9.	r	$\lor_e$ 2, 5–7, 8–8
10.	$p \lor r$	$\lor_{i_2}$ 9
11.	$\neg q$	assumption
12.	p	assumption
13.	q	assumption
14.	$\bot$	$\neg_e$ 13, 11
15.	p	$\bot_e$ 14
16.	p	$\lor_e$ 1, 12–12, 13–15
17.	$p \lor r$	$\lor_{i_1}$ 16
18.	$p \lor r$	$\lor_e$ 3, 4–10, 11–17

Figure 2: Proof for the sequent $p \lor q, \neg q \lor r \vdash p \lor r$.

Some remarks on this proof:

1. Note that on step #3, we use the rule *LEM* without any argument. It really does not depend on any one of the previous steps, however, it actually has a hidden argument, q, making it establish the validity of $q \lor \neg q$ and not, for example, $\neg r \lor \neg\neg r$.

2. Note the order of arguments on step #14. With respect to the rule of negation elimination ($\neg_e$), latter of its two arguments must be the negation-applied of the former, which is why we first refer to q and then $\neg q$, and not the other way around. Also note that negation-applied is not the same as

negation-discarded, although they semantically are the same.

3. Note how assumption boxes are referred to with ranges, i.e. a starting index and an ending index separated with a dash.

Hereby we conclude with our definitions on propositional logic and natural deduction, and also our introduction.

2 Interactive Propositional Logic Engine using Natural Deduction

A *propositional logic engine* is an abstract machine that works with propositional logic as its substance. One that works with the methods of natural deduction, is a propositional logic engine *using natural deduction*. With this project, we propose an *interactive* propositional logic engine using natural deduction.

Our product is a software that allows its users to build up valid proofs for any given propositional logic sequent. By stating the rule they want to use and specifying the arguments that they want to use it with, users can establish the truth of newer propositions and get them added to the proof as a step. The proof rules are embedded into the software, ensuring the validity of the proof throughout the execution.

The software comes with a command-line interface, which is a REPL[1] that reads user input for proof rules, parses and evaluates it, and re-draws the working proof to the console window, in a loop. The loop ends when the conclusion of the initially input sequent is reached.

In the following sections, we will be giving out more details on software's workflow, its the input specifications and some of its inner mechanisms. We will also mention some of its software design aspects, some of which have been made possible (with ease) of the programming language Julia [2].

2.1 Workflow

At all times, the program greets the user with the title same as the name of the project, "Interactive Propositional Logic Engine using Natural Deduction". Initially, the program asks user to provide a sequent that is to be proven.

[1]Read-Evaluate-Print-Loop

After being provided a sequent, the program enters the proof mode, where it will start displaying a working proof structure with step numbers and box visualizations with monospace box-drawing characters[2] for the assumption boxes. In this mode, the user is repeatedly asked to provide a natural deduction rule to apply. With the provided natural deduction rule, along with its parameters, the software will do either one of the following:

- If the rule is indeed applicable to the given arguments, it will add the proposition, validity of which has been established using the provided proof rule with the given arguments, and then re-draw the proof with the new step on the proof.
- If the rule is malformed:
  - The program may ask the user to re-consider their input, or
  - The program may fail and exit.

The program is designed to apply a properly provided natural deduction rule, and only apply a properly provided natural deduction rule. However, as stated above, it may fail to recover promptly and ask for the user to retry if an input is malformed.

Apart from the natural deduction rules, some other statements can be provided to the program to perform the opening and closure of an assumption box, and, for user's convenience, to undo the last action.

Whenever a proof step with a proposition exactly the same as in the conclusion of the initially provided sequent appears and is outside of all the assumption boxes, the program then declares success and exits. This concludes the workflow of the program, and also this section.

2.2 Input specifications

Our program accepts 3 different category of inputs: Propositions, sequents, and natural deduction statements with their arguments. We will provide their specifications separately for a more structured view.

[2]`https://en.wikipedia.org/wiki/Box-drawing_characters`

2.2.1 Propositions

Refer to the Section 1.1 for the BNF specification for the well-formed formulas on propositions. As stated before in that section, we will not expect the user-provided formulas to be well-formed, and tolerantly accept propositions according to precedence rules described in again the same section.

In general, and with respect to the BNF specification, a proposition consists of atoms, operator symbols for negation, conjunction, disjunction, and implication operators, and finally parenthesis. We extend this list of constituents with constants for tautology and contradiction, which we have seen to take place in propositions, both in the proof rules (Figure 1) and the example proof (Figure 2) we gave.

Since we cannot expect users to input Unicode characters like $\wedge$ for the **and** operation, we instead accept sensible ASCII alternatives to represent these operations. We do not accept the Unicode originals, even if the user somehow manages to type them down. Here is the full list of those alternatives, next to their Unicode originals:

Unicode	**Alternative**	
	#1	**#2**
$\top$	`T`	`TRUE`
$\bot$	`F`	`FALSE`
$\neg$	`!`	
$\wedge$	`&`	`*`
$\vee$	`|`	`+`
$\rightarrow$	`->`	
()	`()`	`[]`
$\mathtt{q}_1$	`q1`	

Atomic propositions which are symbolized with a single letter should simply be input using that letter. Atoms with a subscript number should have their number appended right next to them, and not as a subscript. More specifically, all the atoms should be in the form of the following regular expression:

```
[a-z][0-9]*
```

There can be as many spaces around the tokens as the user may input, as well as no space at all. A couple of example propositions our program would accept are as follows:

```
(((!p) & q) -> (p & (q | (!r))))  | (p1 -> (p2 -> (p3 -> p4)))
!p & q -> p & (q | !r)            | p1 -> p2 -> p3 -> p4
!p&q   ->   p&((q)|!r)            | (((p1->p2->p3->p4)))
```

All 6 of the inputs are valid, and the ones on the same column are three different ways of writing the same proposition. The top ones are explicit and well-formed formulas, while the remaining 4 are dependent on conventions of precedence. The bottom ones are untidy examples, where there are a superfluous spaces and then sometimes none. Last examples also exhibit extra parenthesis, which are adequately handled by the parenthesis elimination routine in our program.

2.2.2 Sequents

Refer to the Section 1.2 for the definitions of a sequent. Recall that a sequent with no premises is also a valid sequent, also known as a theorem.

List of premises in a sequent are to be separated with commas (`,`) in our program, just like we do it on paper. The symbol $\vdash$ should be substituted by `=>`. If the user desires to provide a theorem, the sequent symbol `=>` can also be omitted altogether.

Conclusion of the sequent may not be omitted, as it is essential for a sequent. However, if the user desires to use the program for natural deduction without any pre-destined conclusion, he or she may simply provide an unattainable conclusion, such as an atomic proposition `q1234` that does not take place in any one of the premises. This should allow them to run the program indefinitely, as it should be impossible for them to establish the validity of an atomic proposition that does not exist in any one of the premises.

As with the propositions, there can be as many spaces around the tokens as the user feels like, as well as no space at all. A couple of example sequents our program would accept are as follows:

```
!q -> !p => p -> !!q
p -> (q -> r), p, !r => !q
p -> q -> r  , p  , !r=>!q
=> (q -> r) -> ((!q -> !p) -> (p -> r))
(q -> r)   ->  ((!q -> !p) -> (p -> r))
```

Here the sequents in rows 2 and 3, and then the ones in rows 4 and 5 are equivalent, i.e. will be interpreted as the same by our program.

2.2.3 Natural deduction statements

Natural deduction rules have no syntax, but they are rather rules to be applied to true propositions in order to establish the truth of newer ones. When a new proposition is introduced using a natural deduction rule, however, we have to specify the rule as a rationale next to that proposition in a very specific way.

In our program, we adopt this rationale syntax for the *natural deduction statements.* In general, a natural deduction statement should start with the identifier of the corresponding natural deduction rule, followed by the list of parameters it takes. Here, parameters are either one of the following three:

- Line number, with the placeholder `#`.
- An interval of line numbers, with the placeholder `#-#`.
- A proposition, with the placeholder `φ`.

The following table specifies all of the natural deduction statements that our program accepts: First two columns have the names of the statements, and corresponds to the names of the natural deduction rules if the statement is for a rule. The next column is for the statement's identifier, and the following columns contain placeholders for one of the three different types of parameters as listed above, to indicate which type of a parameter that specific natural deduction statement expects.

The last 3 statements on the table are the only ones that do not have a corresponding natural deduction rule. First two are there to open and close assumption boxes, and the last one is for the convenience of the user, allowing them to undo the effects of the last (non-`undo`) statement they have provided.

Name		**Identifier**	**Arguments**		
			#1	#2	#3
Conjunction	Introduction	`andi`	`#`		
	Elimination #1	`ande1`	`#`		
	Elimination #2	`ande2`	`#`		
Disjunction	Introduction #1	`ori1`	`#`	φ	
	Introduction #2	`ori1`	`#`	φ	
	Elimination	`ori1`	`#`	`#-#`	`#-#`
Implication	Introduction	`impi`	`#-#`		
	Elimination	`impe`	`#`	`#`	
Negation	Introduction	`negi`	`#-#`		
	Elimination	`nege`	`#`	`#`	
false	Elimination	`bote`	`#`	φ	
Double negation	Introduction	`negnegi`	`#`		
	Elimination	`negnege`	`#`		
MT (Modus Tollens)		`MT`	`#`	`#`	
PBC (Proof by Contradiction)		`PBC`	`#-#`		
LEM (Law of Excluded Middle)		`LEM`	φ		
Copy		`copy`	`#`		
Assume		`assume`	φ		
Conclude		`conclude`			
Undo		`undo`			

The `#` should be replaced by a single line number out of one of the line numbers that are present in the working proof so far. `#-#` should be replaced by two line numbers separated by a dash (`-`), where the first line number should be of the line right after the start of an assumption box opening line, and the second line number should be of the line right before the end of an assumption box closing line. The φ should be replaced by any proposition, as previously described at the propositions' input specification.

The user may separate the identifier of the statement from the first argument either with one or more space characters (` `), or a comma (`,`) with any amount of space characters around it. The arguments should also be separated from each other in a similar fashion.

All the arguments are necessary, and the program will not proceed when they are missing. The program tolerates an invalid natural deduction statement input when the user forgets to provide the necessary φ argument, by letting the user try again. However, forgetting to provide other types of arguments, or providing different types of arguments than necessary, may possibly not be tolerated and result in an error and therefore cause an abrupt termination of execution.

Following lines are examples to natural deduction statements accepted by our program:

```
LEM q
nege 4, 5
nege 4 5
bote 6 p->r
assume (p->n) & (n->p)
conclude
ore 2, 5-7, 8-8
undo
```

Note how we may keep or omit commas as argument/identifier separators. The parser is able to distinguish arguments simply from their forms, and by utilizing the knowledge that no natural deduction statement expects multiple propositions on its arguments.

2.3 Internal representations

A handful of abstract type and structure declarations are to be found on the first lines of our program's source code, with a couple more in its rest. Explicit typing in Julia allows the programmer to leverage the multiple-dispatch feature of the language.

`Atom` and `Neg` types are for atomic propositions and negated propositions, respectively, and both are categorized under `Unit` abstract type, to carry out the information that they need not be parenthesized when incorporated into other proposition types. `And`, `Or`, and `Imp`, on the other hand, are for conjuncts, disjuncts, and implications, respectively. They are categorized under `Compound` abstract type, which lets us tell that they have to be parenthesized when found as an element of another proposition.

Types `True` and `False` are of abstract type `Literal`, which is a subtype of `Unit` again. Both `Unit` and

`Compound` are subtypes of `Formula`, which is an alias for *proposition* in our program.

A `Sequent` consists of an *array of formulas* for premises, and a *formula* the conclusion.

Our final structure is for the proof steps, and they contain the proposition and the rationale of the step. One of the biggest challenges in developing our program was to find the adequate and efficient way to represent the assumption boxes in proof steps. After several trials and attempts, we have decided to include that bit of information also in our `ProofStep` structure with the Boolean fields `indent` and `outdent`, where `indent` means that the current step has an assumption box starting right above it, and `outdent` means that an assumption box ends right after the current proof step.

An example failed attempt on encoding the information of assumption boxes in proof steps is to have a depth counter to keep the nesting index for assumption boxes. We hoped to be able to tell where to draw the opening and closure lines with this information, however, we have realized that it is bound to fail on some of the cases, and realized this early enough, luckily. An example case where this idea would fail is when there are two adjacent assumption boxes, where an assumption box ends at the n-th step, and another one opens at the $(n+1)$-th step. Those two steps would then have the same indentation level with this representation, making it look as if they are within the same assumption box.

2.4 Output specification

Aside from some title and subtitle texts and prompt messages, the program provides some error/success messages, and most importantly, the depiction of the working proof just like the one we have on this report with Proof 2 and the ones that can be found in the respective sections of the textbook from Huth and Ryan [1].

2.4.1 Error/success messages

There are 5 different messages that the program can provide, and 4 of them are error messages. Some of the errors ask the user whether it should continue regardless, or if it should just abort. You may refer to the following table for further description on what each error message means, on top of what they already say:

Message	Meaning	Prompt
`Following will be regarded as an atomic proposition: >φ<`	Indicates that there has been an error while parsing the provided proposition. Unless the program has a defect, the provided proposition must either be malformed, or it has an atomic proposition that does not fit the input specification. Abort if the former is true, continue at your own risk otherwise.	Yes
`Multiple => symbols detected`	Indicates that the sequent input had more than one sequent symbols. Continuing will make the program regard the whole input as a theorem. Termination is recommended.	Yes
`Unknown rule >id<, please retry`	Indicates that the natural deduction statement starts with an unrecognized identifier. The program will allow user to retry. Please refer to Section 2.2.3 for the list of available natural deduction statements.	No
`Ill-formed natural deduction rule, try again.`	Indicates that the provided natural deduction statement had a proper identifier, but the arguments were inappropriate. Reasons for inappropriateness could be reference to *inaccessible* proof steps or assumption boxes, `#-#` input that does not refer to an assumption box from its start to its end, or lack of an argument, and most probably a φ argument. The program will allow user to retry. Refer to Section 2.2.3 for the complete details.	No
`Target reached!`	This is a success message, indicating that the proof is complete. The program should exit with success after this message.	No

2.4.2 Depiction of the working proof

After the sequent is input, the program starts presenting a depiction of the working proof. After each successful application of a natural deduction statement, the display is cleaned and the updated depiction is drawn.

Constrained to the command-line interface, the depiction is prepared using monospace characters. Assumption boxes are prepared using box-drawing characters and cascaded boxes are depicted as if they were stack of scopes, with assumption boxes laid on top of each other with an offset.

Line numbers are printed to the left of the proof steps. Propositions for proof steps are aligned from their left, and rationale for the steps are printed as aligned from their right, with the necessary amount of spacing calculated and printed in between.

Apart from the visualizations, the way how we format the propositions and rationales for printing might be of interest to the user/reader.

First, the rationales for the proof steps are prepared with simple string interpolation on template strings that we have embedded to our code. Every natural deduction rule based natural deduction statement prepares its own corresponding rationale for the proof step it generates. While we do not oblige users to give natural deduction rules with commas separating their identifiers and arguments, we print them with commas for a clean and uniform look. Similarly, we print them with symbols and not with the identifiers, which we had introduced for our users to conveniently type them. For example, a proper usage of statement `impi 4-6` will prepare a rationale like $\rightarrow_{\mathtt{i}}, \mathtt{4} - \mathtt{6}$.

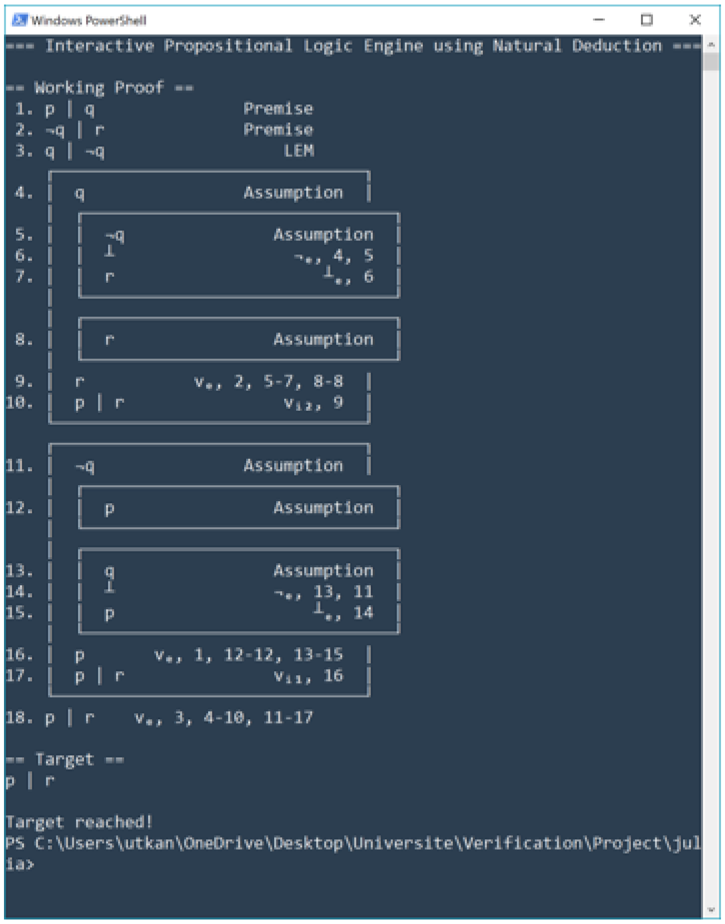
While preparing the propositions for printing, we leverage the internal representation of them. Using the distinction between types `Unit` and `Compound`, we parenthesize propositions only if their main connective is a binary operator, and only if the proposition is not at the top level. As a result, the propositions we print relies on the convention of unary operators having precedence over the binary operators. Since we keep the parenthesis for prioritizing all three binary operators, we hope that our propositions do not lose much from its clarity, while its cleaner and less-cluttered look augments its readability.

Refer to the Section 2.3 for more details on internal representations.

2.5 Example execution

To show reader a glimpse of what can be achieved using this program, here we present a screenshot of the program after it exits with success. Next to it, we have listed the first 17 of the inputs we have provided to our program, line-by-line.

In this particular execution, we were aiming to prove the validity of the sequent $p \vee q, \neg q \vee r \vdash p \vee r$. Observe how closely the depiction resembles the Proof 2.



```
=== Interactive Propositional Logic Engine using Natural Deduction ===

== Working Proof ==
 1. p | q                 Premise
 2. ¬q | r                Premise
 3. q | ¬q                    LEM
    ┌─────────────────────────────┐
 4. │  q              Assumption  │
    │   ┌─────────────────────────────┐
 5. │   │  ¬q             Assumption  │
 6. │   │  ⊥                ¬ₑ, 4, 5  │
 7. │   │  r                 ⊥ₑ, 6    │
    │   └─────────────────────────────┘
    │   ┌─────────────────────────────┐
 8. │   │  r              Assumption  │
    │   └─────────────────────────────┘
 9. │  r          ∨ₑ, 2, 5-7, 8-8  │
10. │  p | r               ∨ᵢ₂, 9  │
    └─────────────────────────────┘
    ┌─────────────────────────────┐
11. │  ¬q             Assumption  │
    │   ┌─────────────────────────────┐
12. │   │  p              Assumption  │
    │   └─────────────────────────────┘
    │   ┌─────────────────────────────┐
13. │   │  q              Assumption  │
14. │   │  ⊥              ¬ₑ, 13, 11  │
15. │   │  p                ⊥ₑ, 14    │
    │   └─────────────────────────────┘
16. │  p      ∨ₑ, 1, 12-12, 13-15  │
17. │  p | r              ∨ᵢ₁, 16  │
    └─────────────────────────────┘
18. p | r    ∨ₑ, 3, 4-10, 11-17

== Target ==
p | r

Target reached!
PS C:\Users\utkan\OneDrive\Desktop\Universite\Verification\Project\jul
ia>
```

Figure 3: Screenshot for the proof.

```
p | q, !q | r => p | r

LEM q

assume q

assume !q

nege 4 5

bote 6 r

conclude

assume r

conclude

ore 2 5-7 8-8

ori2 9 p

conclude

assume !q

assume p

conclude

assume q

nege 13 11
```

[remaining steps are omitted]

Figure 4: First 17 inputs.

3 Conclusion and Discussion

We are very content with the final version of our program. It is able to produce proofs in a very appealing way, despite being a command-line application. With the interactions of the user, it can prepare any propositional logic proof using natural deduction, having a strict enforcement on its validity at each and every step.

The software has an engine embedded within, and uses that engine at its core to consume input and provide output through a monospace terminal. Overall, the software is intended to be used as an interactive, stand-alone command-line application. However, with only some minor modifications, it can also be turned into a tool and a back-end engine for any other application.

A user may use the software non-interactively, by feeding inputs to it from a file. This way, the proofs will become easily reproducible, since when the input file is fed back again to the program, it will again provide the same depiction of the proof as a result.

With only slight modifications, we can introduce the program some switches to tell whether an input file is a successful proof or not, without printing anything else, but maybe also the sequent itself. With this, the tool can then be used as tool for instructors on giving homeworks on sequent proving using natural deduction rules.

To make the scenario more visual, you may think of the case where the instructor gives some various sequents to their pupils to prove, along with this tool. The pupils may then use the tool to interactively come up with a proof for sequents. After they are done, they can write their inputs down in an input file, or, again with a slight modification to the program, the program may already have a command-line switch to write the inputs provided to it to a file, for the convenience of students. Students then can provide this input file as their answer to the instructor, and the instructor can check their answers simply by feeding them to the program again, with the switch for just telling the result and the sequent enabled. The instructor may check the sequent, if the sequent on the input is actually the one he/she had asked, to prevent that kind of cheating. Apart from that, the instructor can be sure of the correctness of the proof, if the program says that the proof is successful, in a matter of seconds.

We believe that we have came up with a tool that is useful, which, at the same time, is aesthetically pleasing. We hope that it reaches to its use cases and that the users find it convenient to work with, and are pleased to use it. Thank you.

References

[1] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge university press, 2004.

[2] NumFocus. The Julia language. `https://julialang.org/`, 2018. [Online; accessed 2018-12-06].