LPL Fitch to LaTeX converter

Final Report

Nicolae Ghidirimschi

January 22, 2021

1 Problem description

General: Implement a tool that accepts a `LPL Fitch` exported proof and outputs the corresponding LaTeX code (using the `fitch.sty` package macros), whether all logical formulae comprising it are well-formed and if the proof is valid or not (with the option to check extra pre-defined validity constraints).

1.1 Input

The input consists of a single or multiple `.html` file(s) generated by the `LPL Fitch` tool (using the Export function). The user can specify the location of the files, as well as the artifacts he wants to generate using the Graphical User Interface. A preview of the GUI is presented in Figure 1.

1.2 Output

The output is displayed using the Graphical User Interface and consists of the following:

1. The LaTeX code (as supported by the `fitch.sty` library) of the input proof(s) and a conversion status notification. If any supplied file is corrupted, the tool will only output a parsing error for that particular file.
2. * A message specifying whether all logical formulas of the given proof are well-formed. If this is not the case, the row number of the first logical formula will be returned in a generated error.
3. * A message specifying whether the provided proof is valid or not. If this is not the case, the row number of the first invalid inference will be returned in a generated error.
4. * An error specifying the row number of the first inference violating extra (pedantic) validity constraints. These are listed in Appendix: Extra Validity Constraints.

* - this artifact is optional (can be enabled/disabled in the Graphical User Interface of the tool) and is disabled by default. It can only be enabled if the artifact higher in the list is also enabled.

An example of a generated LaTeX proof and its source code can be found in Appendix 8.

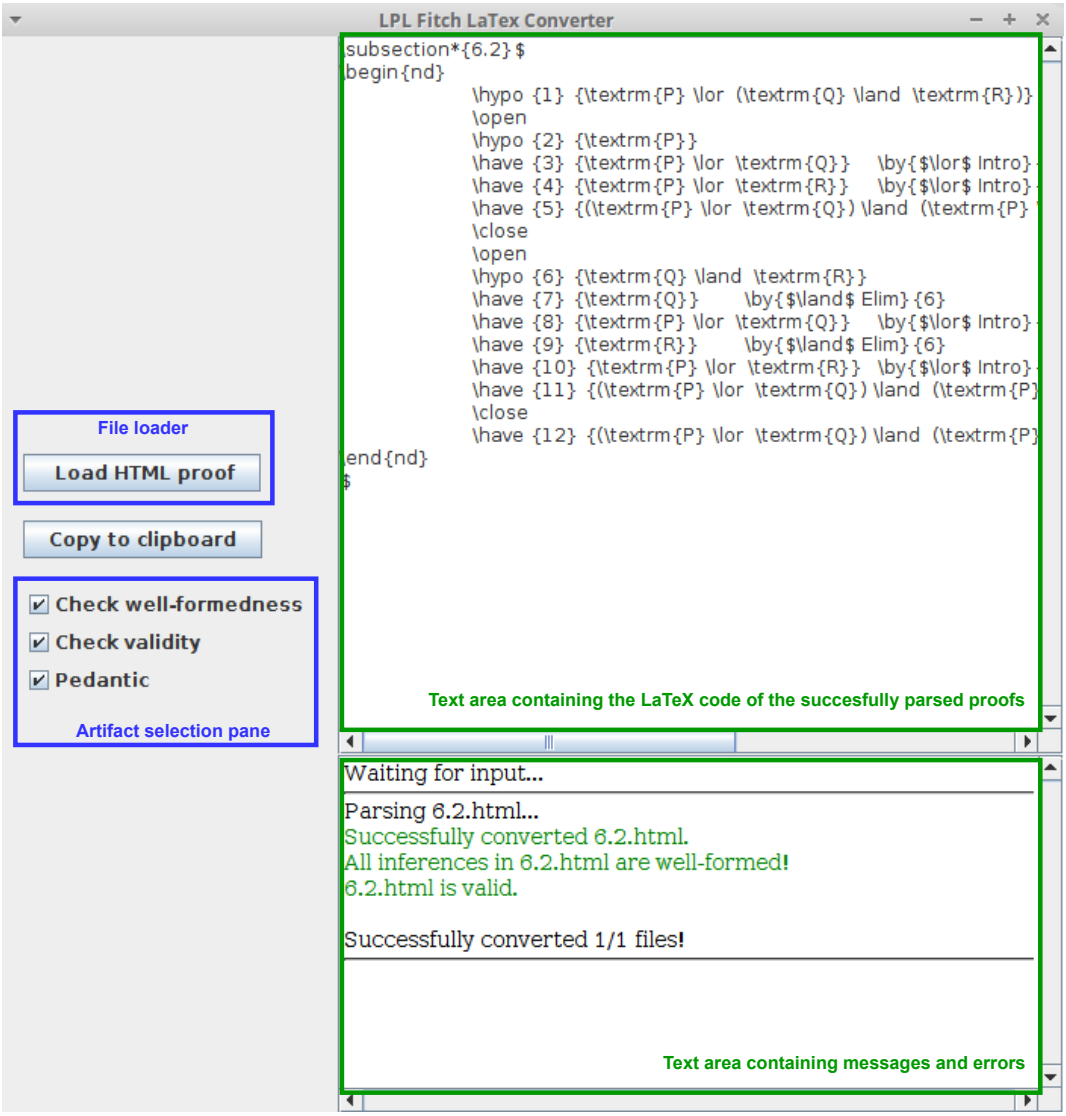

Figure 1: The Graphical User Interface and its components

2 Problem analysis

We choose a Divide-and-Conquer approach in order to tackle our problem. As mentioned in the Project Proposal, we decided to modularize the tool into 4 individual components (as visualized in Figure 2), each having an ingoing transition (phase) from one of the other components, or the initial component, which is represented by the `.html` exported `LPL Fitch` proof(s). By modularizing our tool into distinct components, we are able to separate the concerns and develop the tool in an incremental fashion.

The first component of our tool is the Concrete Syntax tree data type, which will be instantiated by traversing a `.html` input file (which can be viewed as an expression tree as well) and converting each element to one of the data types described in Figure 3. As can be seen, these data-types use a rather abstract representation for the logical formulae, particularly a simple string representation. For the purpose of compilation this is sufficient, since in the compiled LaTeX code of the proof logical formulae will also be represented as string, the only distinction being that some ASCII characters (particularly the one representing logical connectives/other symbols) will be replaced with LaTeX commands. The same will apply to the rules and the cited steps.

However, for the purposes of verifying the validity of the proof, the CST (Concrete Syntax Tree) represen-

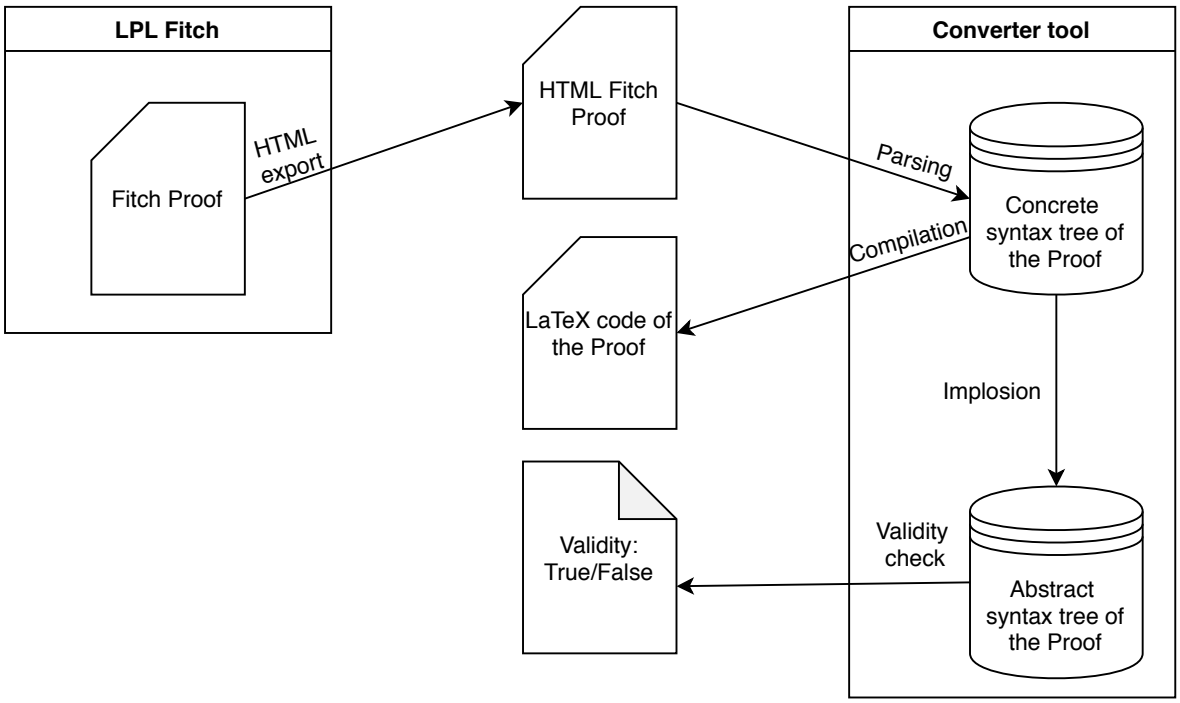

Figure 2: The main components of the Converter tool and their corresponding phases.

tation is clearly insufficient. To circumvent the limitations imposed by the string representation, we define an Abstract Syntax Tree data type, which will be instantiated by imploding the CST. During the implosion phase, we will visit every node of the input CST and parse every field represented by a string to an object of a new abstract data type. These new abstract data types are represented in Figure 4. For the sake of brevity, the Figure omits many similar subclasses of the `Logical Formula` and `Rule` . Particularly, similarly to conjunction, there exists a subclass corresponding to each logical connective extending both the aforementioned interfaces. Using the lower abstraction level of the AST data type it is straightforward to apply the validity constraints and verify the validity of the proof.

We use the following **extended Backus–Naur form** grammar, with augmented Regex expressions (represented by `r'')` in order to parse the logical formulae:

⟨*identifier symbol*⟩ ::= r'[a-z][A-Za-z0-9]*'

⟨*predicate symbol*⟩ ::= r'[A-Z][A-Za-z0-9]*'

⟨*variable*⟩ ::= r'[a-z]'

⟨*term*⟩ ::= ⟨*identifier symbol*⟩ ['(' {⟨*term*⟩ ','} ⟨*term*⟩')']

⟨*atomic formula*⟩ ::= '⊥'
| ⟨*term*⟩ '=' ⟨*term*⟩
| ⟨*predicate symbol*⟩ ['(' {⟨*term*⟩ ','} ⟨*term*⟩')']

⟨*formula*⟩ ::= ⟨*atomic formula*⟩
| '¬' ⟨*formula*⟩
| ⟨*formula*⟩ '∧' ⟨*formula*⟩
| ⟨*formula*⟩ '∨' ⟨*formula*⟩
| ⟨*formula*⟩ '→' ⟨*formula*⟩
| ⟨*formula*⟩ '↔' ⟨*formula*⟩
| '(' ⟨*formula*⟩ ')'
| '∀' ⟨*variable*⟩ ⟨*formula*⟩
| '∃' ⟨*variable*⟩ ⟨*formula*⟩

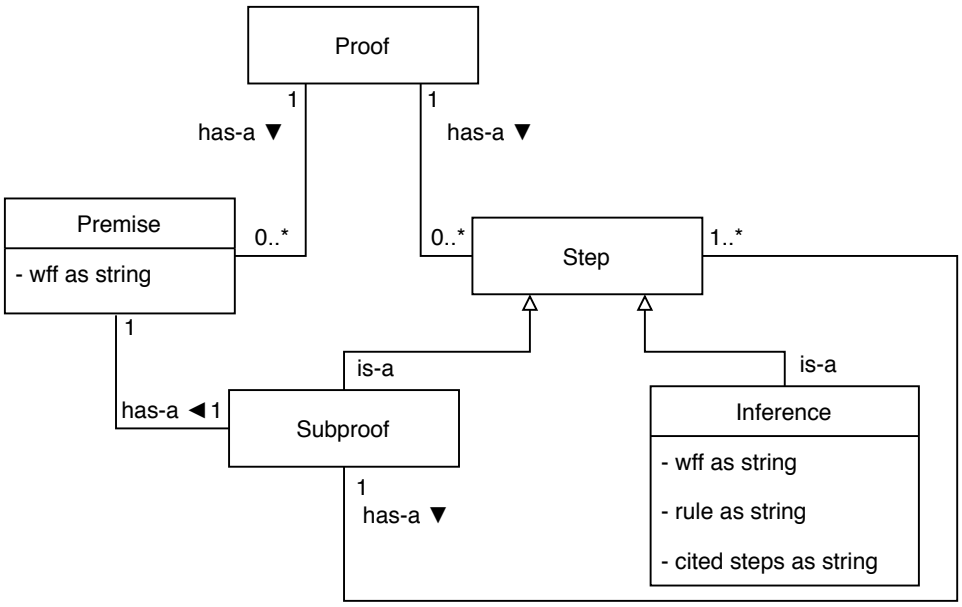

Figure 3: The representation of the Proof Concrete Syntax tree Data Type.

The proofs follow the following **EBNF** grammar:

⟨*proof*⟩ ::= {⟨*premise*⟩} {⟨*step*⟩}-

⟨*premise*⟩ ::= ⟨*formula*⟩

⟨*step*⟩ ::= ⟨*inference*⟩
| ⟨*subproof*⟩

⟨*inference*⟩ ::= ⟨*formula*⟩ ⟨*rule*⟩

⟨*rule*⟩ ::= ⟨*rule name*⟩ {⟨*cited step*⟩}

⟨*rule name*⟩ ::= '∧ Intro' | '∧ Elim' ...

⟨*cited step*⟩ ::= Natural number

⟨*subproof*⟩ ::= ⟨*premise*⟩ {⟨*step*⟩}-

3 Program design

We chose to implement our tool in `Java`. The Graphical User Interface is written using the `Swing` toolkit, and is connected to the Model through a Controller component, as defined by the Model–View–Controller (MVC) pattern.

The data types described in the previous subsection are translated to code in a straightforward manner: the superclasses (the ones which have ingoing is-a relationships) are implemented as interfaces, and are correspondingly extended by the subclasses, which are defined as simple classes in code. The has-a relationship is implemented as composition.

During the Parsing phase, the `.html` file is converted to an expression tree and traversed using the `jsoup`

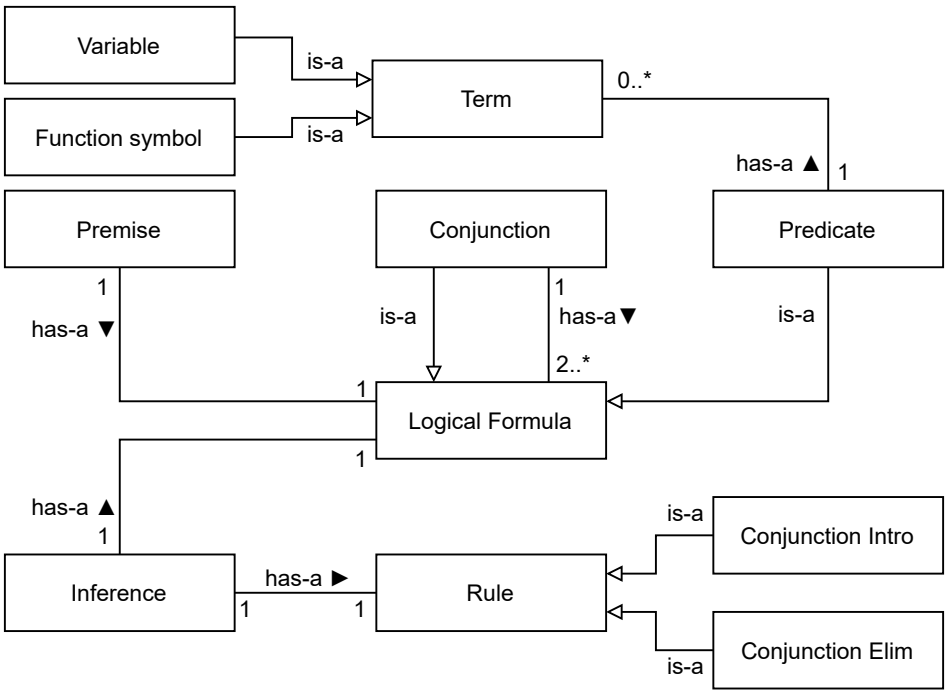

Figure 4: A reduced representation of the Proof Abstract Syntax tree Data Type.

library. During the traversal, a `Proof` object is instantiated by parsing the `html` elements according to the Proof **EBNF** grammar specified above.

The implosion is implemented as a member method of the `Proof` class and all its composed classes and is therefore invoked in a cascading fashion by all nodes of the CST object. It parses all the encountered string fields, particularly the wff ones according to the formula **EBNF** grammar listed above. The result is a new AST (which we define as `AbstractProof`) object, where all string fields are replaced with references to objects of type `AbstractInference` or `AbstractPremise`, which are implemented according to the model defined in Figure 4.

We use the `apache commons` library for some utility functions, particularly for string and arraylist manipulations.

4 Evaluation of the program

We have extensively tested our program on various input files. More specifically, we have created and exported `LPL Fitch` proofs corresponding to every formal proof exercise which was considered part of the Tutorial material of the 2019-2020 Iteration of the Introduction to Logic course (around 50 exercises). All the proofs were successfully converted to LaTeX and correctly identified as both well-formed and valid. This allowed us to conclude that our tool does not produce *false negatives*. In order to verify the possibility of *false positives*, we have deliberately introduced various well-formedness and validity mistakes in the many of the same previously used inputs. Albeit, during the initial testing phase, the tool was unable to detect some of these, we have continuously updated it until we could have not reproduced *false positives* anymore. The overall extensive evaluation process makes us confident that our tool does not contain any significant bug.

All the produced artifacts, particularly the LaTeX code of the input proofs, were used to generate the complete `Tutorial Exercises Answers` for the Introduction to Logic Course.

5 Extensions of the program & Process description

The current and possible suggested extensions of the program are described in Subsection 2.4 of the Proposal Document. Similarly the process description is thoroughly described in the Subsection 3.4 of the same document. For brevity, we will omit repeating them here.

6 Conclusions

Albeit developing this tool has took more time than originally planned and underwent many organizational changes (originally started as a simple initiative, then became a Honours Research Project, and finally a Short Programming Project), we find the overall experience very enriching and useful. We have learnt (sometimes the hard way) how crucial it is to have and follow a rigid plan, and how important well-calculated design decisions are. After trying to implement the tool in `C` instead of `Java` we realized how much the choice of a higher abstraction level programming language can simplify the development process. After considering to add a multitude of various extensions to the tool (such as negative translations or an automatic proof solver) we learnt how important it is to have realistic and well defined aims for a project. After implementing a parser by hand, we understood how using a parser library could have saved us a lot of development time.

Besides the things learnt, we are also pleased with our final tool. We are happy that with its help we were able to compile the solutions for the `Tutorial Exercises` for the Introduction to Logic Course, and based on our testing, we are confident that one may use it as well to compile his/her own `LPL Fitch` proofs to LaTeX and/or verify its validity (with extra constraints). We also believe that the tool can be reused for other interesting, and perhaps useful extensions or projects. We have provided several such examples in the Proposal Document.

7 Appendix: Extra Validity Constraints

1. `= Elim a, b` only allows the left hand side operand of the equality on row `b` to be replaced by the right hand side one. The replacement must occur in the formula located on row `a`.
2. $\perp$ `Intro a, b` is only valid if the formula on row `b` is the negation of the formula on row `a`. The vice-versa is not considered valid.
3. $\rightarrow$ `Elim a, b` is only valid if the formula citing the rule is the consequent of the implication on row `a` and formula on row `b` is logically equal to its antecedent.

8 Appendix: Example generated LaTeX Proof

Exercise 6.20 (LPL textbook)

1	$A \lor B$	
2	$A \lor C$	
3	$\quad A$	
4	$\quad A \lor (B \land C)$	$\lor$ Intro: 3
5	$\quad B$	
6	$\quad\quad A$	
7	$\quad\quad A \lor (B \land C)$	$\lor$ Intro: 6
8	$\quad\quad C$	
9	$\quad\quad B \land C$	$\land$ Intro: 8, 5
10	$\quad\quad A \lor (B \land C)$	$\lor$ Intro: 9
11	$\quad A \lor (B \land C)$	$\lor$ Elim: 8–10, 6–7, 2
12	$A \lor (B \land C)$	$\lor$ Elim: 5–11, 3–4, 1

Which corresponds to the following LaTeX code:

```
\subsection*{Exercise 6.20 (LPL textbook)}$
\begin{nd}
  \hypo {1} {\textrm{A} \lor \textrm{B}}
  \hypo {2} {\textrm{A} \lor \textrm{C}}
  \open
  \hypo {3} {\textrm{A}}
  \have {4} {\textrm{A} \lor (\textrm{B} \land \textrm{C})}   \by{$\lor$ Intro}{3}
  \close
  \open
  \hypo {5} {\textrm{B}}
  \open
  \hypo {6} {\textrm{A}}
  \have {7} {\textrm{A} \lor (\textrm{B} \land \textrm{C})}   \by{$\lor$ Intro}{6}
  \close
  \open
  \hypo {8} {\textrm{C}}
  \have {9} {\textrm{B} \land \textrm{C}}  \by{$\land$ Intro}{8, 5}
  \have {10} {\textrm{A} \lor (\textrm{B} \land \textrm{C})}  \by{$\lor$ Intro}{9}
  \close
  \have {11} {\textrm{A} \lor (\textrm{B} \land \textrm{C})}  \by{$\lor$ Elim}{8-10, 6-7, 2}
  \close
  \have {12} {\textrm{A} \lor (\textrm{B} \land \textrm{C})}  \by{$\lor$ Elim}{5-11, 3-4, 1}
\end{nd}
$
```