Short Programming Project

Course code: `WBCS15002`

Credits (EC): `5 points (140 hours, 14 hours/week)`

1 General information

Project title: `LPL Fitch` to LaTeX converter.

Student name and signature: Ghidirimschi

Nicolae Ghidirimschi, s3197395

Date: December 1, 2020

First supervisor (name + signature): [illegible]

prof. dr. Gerard Renardel de Lavalette

2 Project description

2.1 Introduction and Motivation

Due to its advantages, typesetting reports and homeworks in LaTeX is a frequent requirement for students of both Computing Science and Artificial Intelligence Programmes at the University of Groningen. Besides allowing students to develop the skill of producing documents that have a scientific and professional look, LaTeX reports are also arguably easier and more pleasant to grade. The Introduction to Logic (CS) course, albeit recommending it, has not made mandatory to present LaTeX typeset reports. Due to the many specific notations and graphical elements that are used in the homework exercises, it is a quite laborious and challenging task even for somebody proficient and used to the tool, all the more for a first year student who has only recently been introduced to both Logic and LaTeX. One type of exercise typically occurring in all homework assignments that is particularly difficult to typeset (as of its notation) are Fitch proofs. Albeit specific libraries exist that significantly simplify it, particularly `fitch.sty` [5], the task remains time-consuming and error-prone.

2.2 Proposal

We propose a tool that will generate the LaTeX code of a provided `LPL Fitch`[1] proof. We suggest that providing the students with this tool, together with a LaTeX template that would give good examples of typesetting other types of exercises that are part of the homeworks, could make typesetting Introduction to Logic assignments in LaTeX a much easier task. This can motivate more students to submit more professionally looking assignments, especially when submitting digitally. Moreover, the tool might be useful for the teaching staff, as for example happened in the year 2019 iteration of the Introduction to Logic (CS), when the tool was successfully used to generate the solutions of all Fitch proof questions that were part of the Tutorial material.

[1]a third-party software that comes free with the recommended book Language, Proof and Logic [1] used in the Introduction to Logic course as an environment for writing and verifying the validity of Fitch Proofs. Because of its friendly interface and its feature to instantly verify the validity of the developed proof, some students prefer to write their proofs using this tool.

2.3 State of the art

We were able to identify several tools that allow the construction of logical proofs in Fitch notation, validity verification and their export to LaTeX. `fitchJS` [4] is an open-source web application that uses a simple plain text environment for the construction of the proofs. It further allows the verification of the validity of the constructed proof, and it's export to LaTeX. Similarly, `ProofMood` [3] achieves the same objectives, except it uses a more graphical environment for proof construction. On the other hand, it is not open-source and is only available in Korean. Unfortunately none of the existing tools we could find allow the possibility to import `LPL Fitch` exported proofs. Since all of them force the user to use a different environment than `LPL Fitch`, they do not solve our problem. Reusing `fitchJs` is not very convenient, as instead of compiling `fitch.sty` macros that represent the proof, the challenge becomes compiling a compatible plain-text representation of the proof which we could supply as input to `fitchJs` (which is in essence the same problem). Reusing the validity checker part would constrain us to work in `JavaScript`, which given our programming experience and arguments described in Subsection 3.1 will not make it easier to achieve the proposed aims for this Short Programming Project than to write this part de novo.

2.4 Current/possible extensions

The tool is currently extended with a validity checker for only the propositional logic rules, plus the `= Intro/Elim` rules. This was implemented as to detect "shortcut" applications of rules which are regarded as valid by the `LPL Fitch` program, and were otherwise considered invalid in the context of the Introduction to Logic (CS) course[2]. Since `LPL Fitch` is not open-source, and no such option is provided by the creators of the software, it is not currently possible to directly calibrate its validity checker to disallow specific shortcuts or to enforce extra constraints. The tool might be used to detect the violation of such conventions/extra constraint.

The tool might also serve as a foundation for future extensions. One initially planned extension was the generation of the LaTeX code of the double negation translation of the input proof.

The back-end part of the tool could also be used for automating (at least partially) the grading of Fitch proof exercises. Particularly, by extending the validity checker to verify the predicate logic rules as well, the completely correct proofs, or proofs containing specific mistakes could be easily graded automatically. The `Themis` platform could then be used to embed the program and allow the students to submit their `LPL Fitch` proofs directly. However, automating the grading of the rest of the proofs is a bigger challenge. A grading heuristic that was typically used to grade invalid proofs during the Introduction to Logic (CS, 2018-2019) was the following simple formula: $p - i$, where p was the maximum number of points for the proof and i - the minimum number of steps that were required to be corrected/added in order to make the proof valid. Given a reasonable sized proof and the precondition that it is possible to infer the goal based on the premises, we expect it should be realistic to compute i, and therefore automate the grading of all Fitch proofs. Unfortunately, this is beyond the scope of this Short Programming Project and would require more research.

3 Methodology and timeline

3.1 Programming language

We have decided to implement the proposed tool in `Java` based on several considerations. Foremost, due to the way `Java` programs are executed, this will allow us to achieve a high level of portability. Particularly, the users will not have to undergo any installation process, because the program bytecode (the `Java` executable) is cross-platform. The high portability will also ensure that any tool component, particularly the GUI, will be functional independently of the user hardware and operating system (as long as both are able to support the `Java Running Environment`). Secondly, there exist multiple `Java` libraries that will be useful in developing particular components of our tool (we will refer to them in the next subsection). Together with the automatic

[2]a particular example of this is the application of `= Elim` rule. In the context of Introduction to Logic (CS, 2017-2019) course, using the `= Elim` rule with cited steps `a = b` and `P(b)` to infer `P(a)` was considered invalid, while being reported as valid by `LPL Fitch`. Students were required to explicitly prove `b = a` and use it instead of the first cited step.

memory management inherent to `Java`, this will significantly ease the development process and will make the tool easy to maintain and extend in the future. Lastly, due to being executed in a virtual medium (`Java Virtual Machine`), the tool will hardly interfere (or tamper, in case of malfunction) with the system on which it is executed.

3.2 Tool components

In order to simplify the development process, we decided to modularize the tool into 4 components and 4 corresponding phases, as visualized in Figure 1. Before we describe them, it is worth noting that because `LPL Fitch` is not open-source, it is difficult to parse the default generated `.prf` files. However, we can bypass this limitation by using the export feature to convert the `.prf` proof into a `.html` representation.

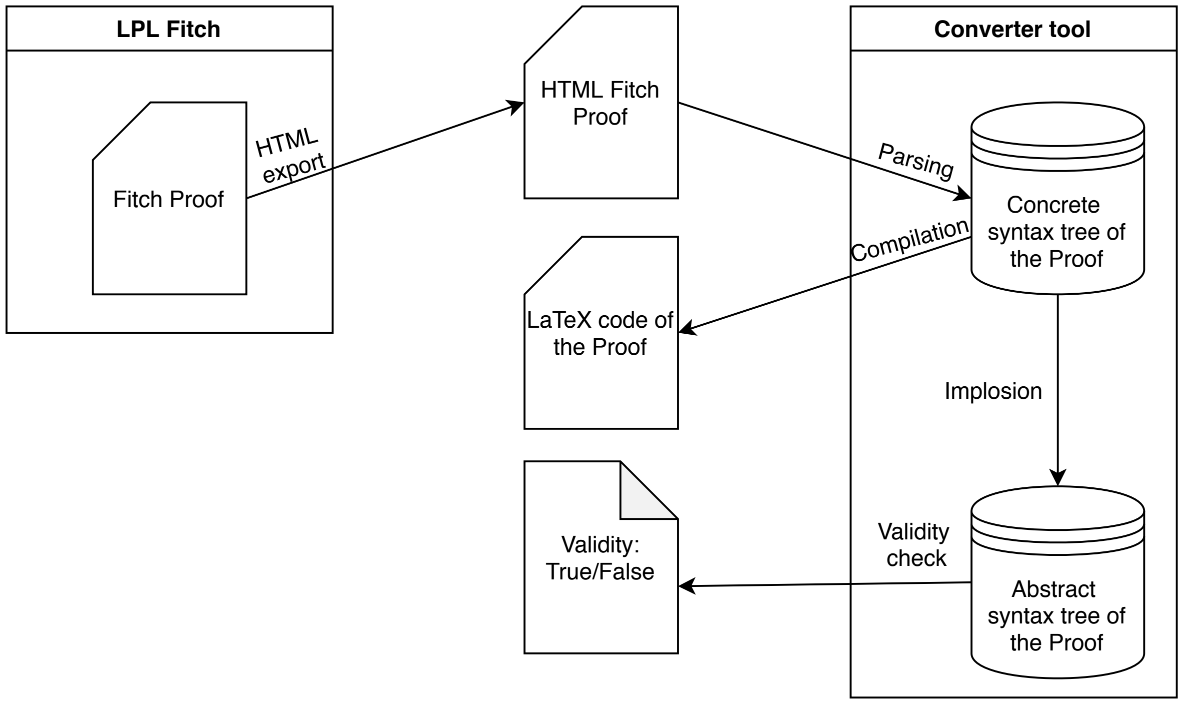


Figure 1: The main components of the Converter tool and their corresponding phases.

The first component of our tool is the Concrete syntax tree data type, which will be instantiated by parsing a `LPL Fitch` Proof exported as `HTML`. At this phase, we will make use of the `jsoup` [2] library to tokenize and traverse the input HTML code. As a result, we will either instantiate a `Proof` object according to the specification of Figure 2, or will return a parsing error, in case the traversal of the `HTML` code encountered a problem.

The next component is the LaTeX code of the `Fitch` proof and the corresponding compilation phase. The code is generated during the traversal of the Concrete syntax tree and the transformation of each visited node to one of the corresponding LaTeX macros provided by the `fitch.sty` library.

Similarly to the previous phase, during the Implosion we traverse the Concrete syntax tree, and instantiate a corresponding Abstrast syntax tree object. The main difference between the Concrete and the Abstract syntax tree is that we will define abstract data types for the string data fields of `Premise` and `Inference`. Particularly, the `wff` field will become an interface, which will be extended by a class corresponding to every logical operator and quantifier (conjunction, disjunction, etc.) and by the class of atomic propositions. The rule and the cited steps will form an enumeration class. The Implosition step will only generate an Abstract syntax tree representation if all the logical formulas that are part of the proof (either as part of an inference or a premise) are well-formed, and will otherwise generate an error. For brevity, we will omit the visual representation of the Abstract syntax tree from this document

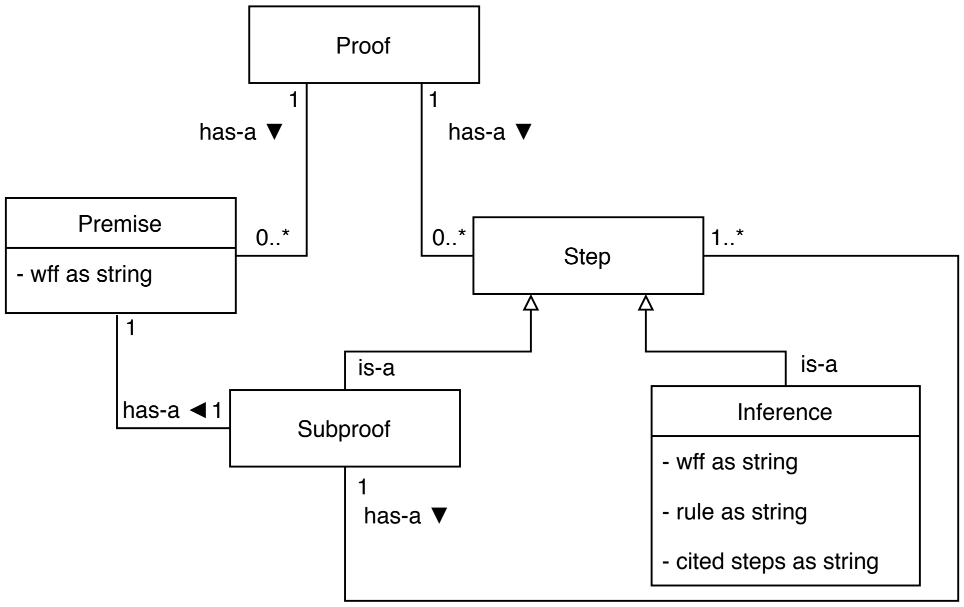
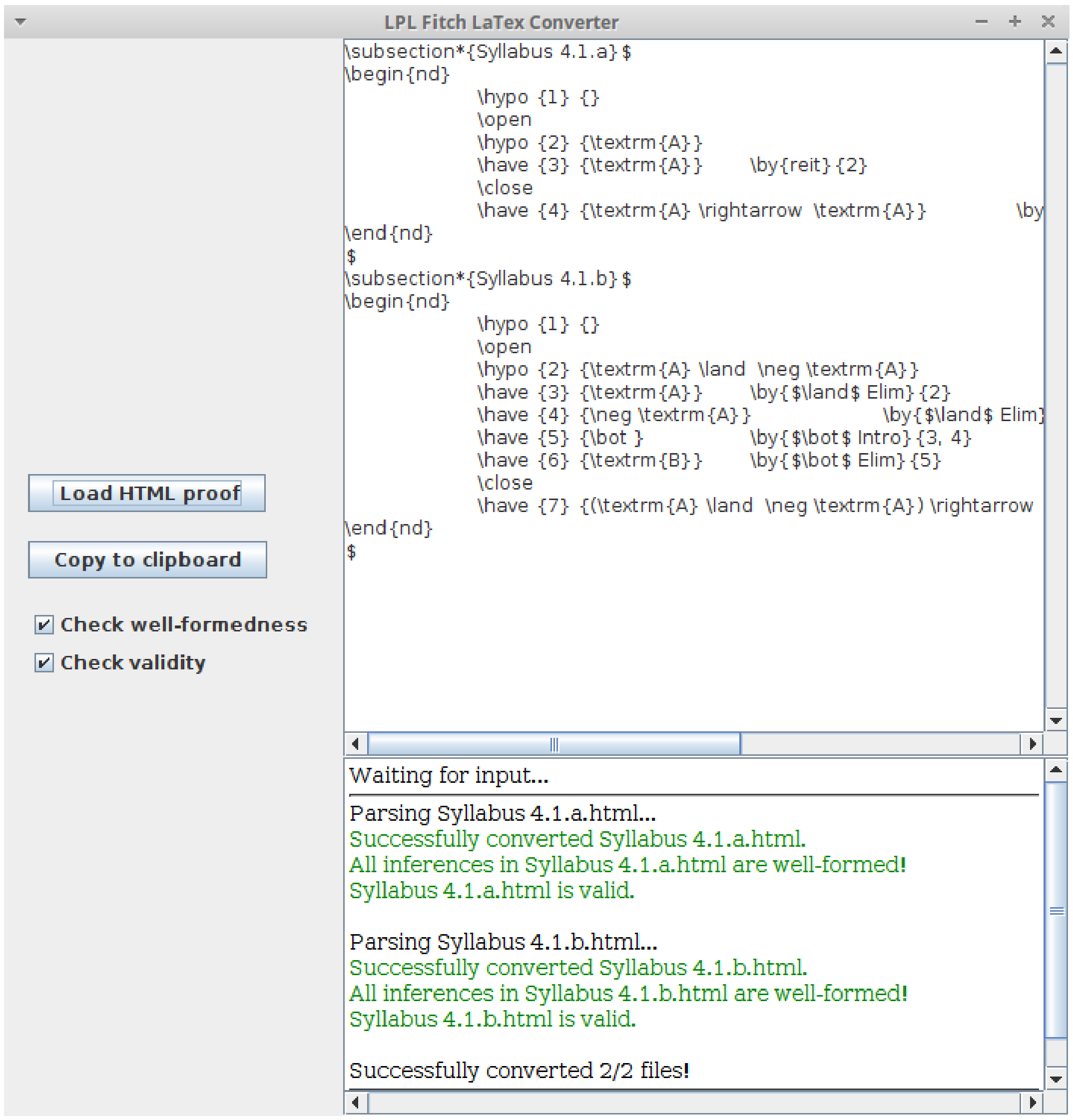


Figure 2: The representation of the Proof Concrete Syntax tree Data Type.

Having an Abstract syntax tree representation, it is now possible to verify the validity of the Proof by traversing it and verifying that each rule application is valid (according to the natural deduction axioms). The result of the validity check is a simple Boolean value - true if the Proof is valid, false otherwise.

3.3 User interface

We aim for a minimalist and user-friendly graphical interface, where inputs and outputs will be entered and displayed in the same application window. The current design is demonstrated in Figure 3. The user-interface is connected with the back-end using Model–View–Controller (MVC) pattern.



```
\subsection*{Syllabus 4.1.a}$
\begin{nd}
            \hypo {1} {}
            \open
            \hypo {2} {\textrm{A}}
            \have {3} {\textrm{A}}      \by{reit}{2}
            \close
            \have {4} {\textrm{A} \rightarrow  \textrm{A}}          \by
\end{nd}
$
\subsection*{Syllabus 4.1.b}$
\begin{nd}
            \hypo {1} {}
            \open
            \hypo {2} {\textrm{A} \land  \neg \textrm{A}}
            \have {3} {\textrm{A}}      \by{$\land$ Elim}{2}
            \have {4} {\neg \textrm{A}}              \by{$\land$ Elim}
            \have {5} {\bot }           \by{$\bot$ Intro}{3, 4}
            \have {6} {\textrm{B}}      \by{$\bot$ Elim}{5}
            \close
            \have {7} {(\textrm{A} \land  \neg \textrm{A}) \rightarrow
\end{nd}
$
```

```
Waiting for input...

Parsing Syllabus 4.1.a.html...
Successfully converted Syllabus 4.1.a.html.
All inferences in Syllabus 4.1.a.html are well-formed!
Syllabus 4.1.a.html is valid.

Parsing Syllabus 4.1.b.html...
Successfully converted Syllabus 4.1.b.html.
All inferences in Syllabus 4.1.b.html are well-formed!
Syllabus 4.1.b.html is valid.

Successfully converted 2/2 files!
```

Figure 3: Demonstration of the current GUI of the Converter tool.

3.4 Timeline

An approximate time-line is provided in Figure 4. As indicated, the majority of the coding part of the project was already finished approximately one year before the moment of writing of this document. Based on the commit history and project statistics, we are certain that the coding part alone took more than 140 hours and the tasks described in the Gantt chart happened as visualized in Figure 4.

The milestones of the project correspond to each component described in Subsection 3.2, complemented with a milestone for the final product in 2019, and a Final milestone for 2020. The 2019 Milestones were achieved in a Waterfall fashion, as each component of the project was based on the previous ones. The only exception was the implementation of the GUI, which was done in an iterative manner throughout the weeks, as to accommodate for each newly added component. The 2020 work packages aim to both improve the already existing tool by incorporating new updates to it, provide a good source code documentation and provide a project description and final report discussing the plan, the realisation and the general structure of the software developed. This milestone will only be reached after the completion of the Project description and Deployment work packages, which besides the specified tasks/deliverables, will also include a demonstration of the final product to the supervisor. It might also be preceded by preliminary feedback.

References

[1] David Barker-Plummer, Jon Barwise, and John Etchemendy. *Language, Proof, and Logic: Second Edition.* Center for the Study of Language and Information/SRI, 2nd edition, 2011.

[2] Jonathan Hedley. `jsoup Java HTML Parser`, 2020.

[3] Joohee Jeong. `Proofmood`, 2020.

[4] Michael Rieppel. `fitchJS`, 2020.

[5] Peter Selinger. LaTeX macros for Fitch style natural deduction, 2020.

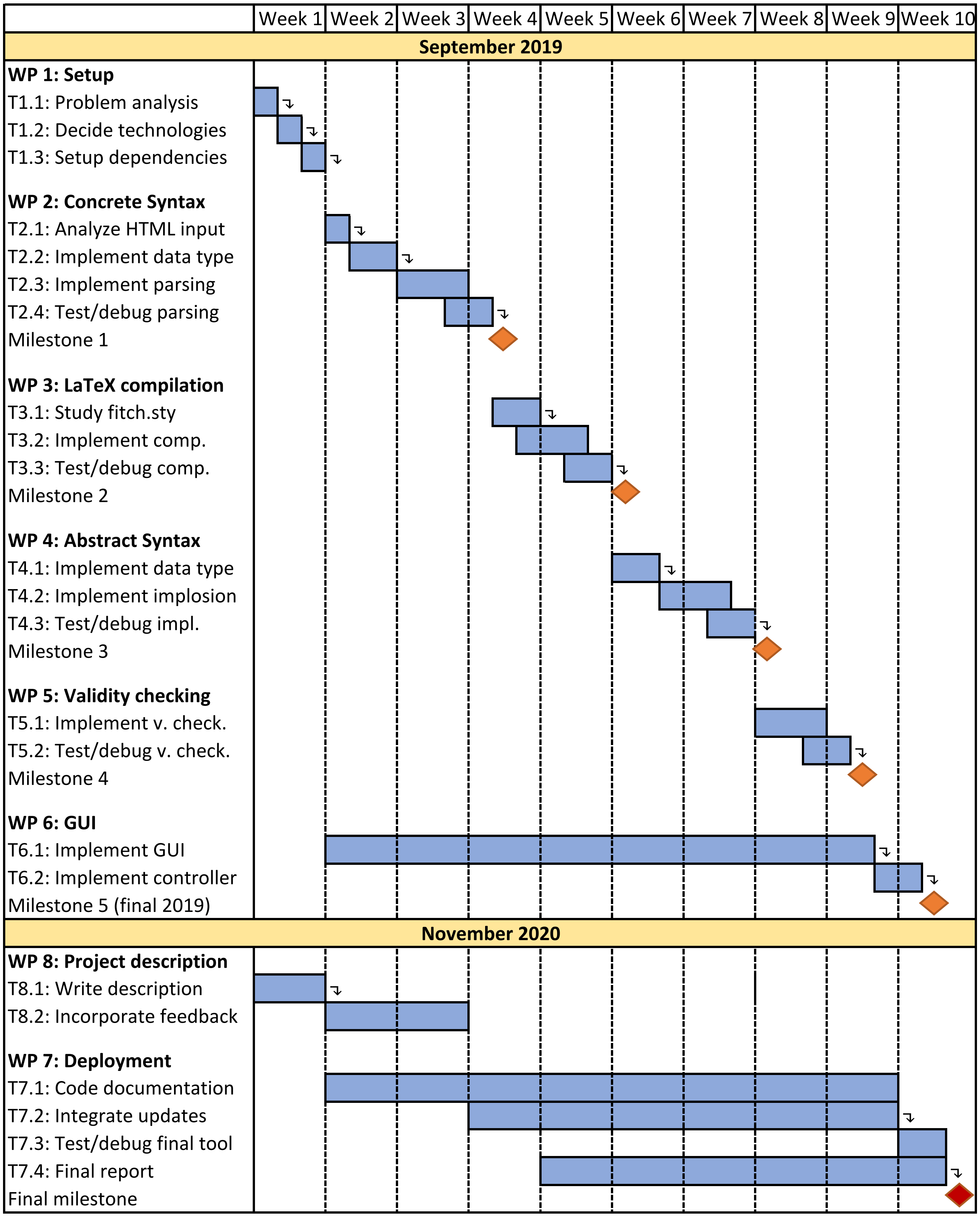


Figure 4: The Gantt chart with Work Packages (first column) and Milestones (orange/red rhombuses).