Logic Software Requirements
It's only worth doing if we produce something dramatically better than anything else available and that will be rapidly adopted and enhanced by many universities, and will be useful for decades. The idea is not to create software for one particular logic course but a platform that will work with different logic courses. Instructors from different universities can draw on a database of contributed exercises and solutions; and students can interact with their instructors and other users.

Note. While writing these requirements I realised that they describe multiple projects. Some or all of these should probably be separate. Several are not specific to logic at all.

> * A system for (i) creating exercises of various kinds (see C, 'Types of Exercise' below), (ii) assembling exercises into series, (iii) presenting exercises and series of exercises to students, and (iv) enabling students to answer exercises.
>
> > This will require components that allow users to create and manipulate (i) sentences of a formal language, (ii) proofs, (iii) truth tables and (iv) possible situations.
>
> * A system for automatically marking submitted answers to exercises.
>
> > This will require components that (i) parse sentences of a formal language, (ii) check proofs, (iii) verify truth tables, (iv) convert a sentence in a formal language into a canonical form, and (v) evaluate the truth of a sentence against a possible situation.
>
> * Something that allows (i) secretaries and instructors to create courses (large units) and classes (smaller units), (ii) secretaries and instructors to control admission to classes and courses, (iii) instructors to be assigned to classes and courses, (iv) students to join classes and courses.
>
> * Something that enables students to (i) find and select an individual exercise or a series of exercises; (ii) see the selected exercise or series of exercises to be tackled, (iii) save their work on exercises, (iv) submit exercises, (v) review and modify the answers they gave to exercises, (vi) see their marks on submitted exercises, (vii) request help from instructors or each other, (viii) see feedback on exercises and responses to requests for help, (ix) respond to requests for help
>
> * Something that enables instructors to (iii) assign series of exercises to courses or classes, (iv) respond to requests for help, (v) see students' submitted exercises, (vi) give marks to students' submitted exercises, (vii) give feedback on students' submitted exercises, (viii) see a tabular overview of students' submissions and marks by course or class and by date range.

So part of the project as currently described in these requirements involves tools for electronic submission & marking of work that are not specific to logic. Could we save work by finding existing software that provides some or all of the components that aren't specific to logic?

A. Requirements: outline

1. It's free.

2. It works in browsers (not just in one browser, but not necessarily in all browsers).

3. The user interface (a) is easy even for philosophers to understand; (b) doesn't break the users' flow when they're working on problems; and (c) is pleasant.

4. Students' work is never lost: even if the network connection fails or their computer crashes or the server(s) goes down, at most 60 seconds should be lost.

5. It can be used for an entire logic course (no other software is needed).

6. Textbook compatibility (and ideally agnosticism):

> 6(a) It can use exactly the syntax of FOL and the system of proof (Fitch) described in *Language, Proof and Logic* (I can provide you with a copy);
>
> 6(b) [optional] it can also be configured to use other syntax and systems of proof for compatibility with whatever textbook a university uses.

7. Instructors (and maybe all users) can create new exercises and new series of exercises and assign these to their students.

8. Feedback:

> 8(a) Students can request help with particular exercises from (i) the system (automated feedback) or (ii) their own instructors or (iii) anyone;
>
> 8(b) Instructors can can mark students' exercises and give feedback after the exercises have been submitted;
>
> 8(c) Where possible feedback is automatically generated or recycled from an earlier case where a student made the same mistake and an instructor or student commented.

9. Instructors can view students' progress over a specified range of dates or range of exercises for both (a) an individual student; and (b) the whole class or whole course.

10. The code can be taken over and extended by a completely new team.

11. Others not formally connected with the project can provide extensions (e.g. an

alternative syntax, or a new type of exercise).

12. There are automated tests which at least provide coverage of the core logic parts (e.g. of the proof checker).

B. Goals but not strictly requirements

* It's easy to set up on a simple server and requires only simple maintenance.

* It scales to thousands of simultaneous users without generating significant costs (lots of network traffic is fine).

C. Types of exercise

1. Generic exercises that can be automatically evaluated. Student is given a picture or some text or both and required to provide an answer, either a number or a selection.

2. Generic exercises that cannot be automatically evaluated. Student is given a picture or some text or both and required to provide an answer which an instructor (or in some cases other users) can mark.

3. Truth tables: Student is given a sentence of FOL (e.g. (P & !Q)) and required to produce a truth table for it. The instructor can specify whether the reference columns appear and may optionally provide part of the answer.

4. Truth table mistakes: student is given a truth table and required to identify errors in it.

5. Evaluating sentences

> 5(a) The instructor specifies a possible situation and one or more sentences. The student has to determine, for each sentence, whether it is true or false in the possible situation.
>
> 5(b) The instructor specifies one or more sentences. The student has to determine whether the sentence is necessarily false, necessarily true or contingent.
>
> 5(c) The instructor specifies one or more sentences in FOL (e.g. [all x][Cat(x) -> Red(x)]). The instructor may also partly specify a possible situation. The student has to modify the possible situation (which may be blank) so that the sentences are all true (or all false). The instructor may specify restrictions on what the student cannot do: (i) cannot create new objects; (ii) cannot destroy

objects; (iii) cannot move existing objects; (iv) cannot rename existing objects; (v) cannot alter existing objects' properties.

6. Proofs and counterexamples.

(a) Instructor specifies premises and a conclusion, and optionally includes further parts of a proof. The student has to complete the proof.

(b) The instructor provides a mistaken proof. The student has to correct it.

(c) The instructor specifies then premises and conclusion of an argument. The student has to decide whether the argument is valid or not. If it's valid, she should prove it. If it's not valid, she should produce a counterexample to the argument.

6. Translation to FOL.

6(a) The instructor provides and English sentence and specifies a list of names and predicates which the student can use. The student has to provide a translation of the English sentence into FOL.

6(b) Like 6(a) but from FOL to English

D. User interface: basics

1. Users must be able to enter and manipulate **sentences** of FOL (e.g. [all x](Cat(x))); these sentences must be displayed in a way that looks nice; and it must be easy to enter the sentences using a keyboard only.

Note. It would be confusing if students have to use two sets of symbols, one for the textbook and another for the software. One way around this might be to use words in the software, with the option of using symbols e.g.:

[exists x]((not Cat(x)) -> Dog(x))

and

[exists x]((~Cat(x)) arrow Dog(x))

should both be parsed as the same sentence.

2. Users must be able to see whether a sentence they have entered is syntactically correct. (Ideally it would be possible for different instructors to specify different rules of syntax. Minimally these must include the syntax used in *Language, Proof & Logic*).

3. Users must be able to create and manipulate **proofs**, including proofs which contain

subproofs. The proofs must be displayed as they would be drawn (i.e. subproofs are indented).

4. Users must be able to see whether a proof they have entered is correct; and if it is incorrect, where the mistakes are.

5. Users must be able to create and manipulate **truth tables**.

6. Users must be able to see whether the truth table they have entered is correct, and if not where the mistakes are.

6. Students must be able to create and manipulate **possible situations**. This includes specifying (i) how many things exist; (ii) which properties they have (from a fixed range of properties, e.g. Square(x), LeftOf(x,y), ...); (iii) which, if any, names an object has (from a fixed range of names, a, b, c, ...).

7. Students must be able to see whether a sentence of FOL is true in a possible situation.

8. Students must be able to see whether a possible situation is a counterexample to an argument.

9. Students must be able to see which rows (if any) of a truth table are counterexamples to an argument.

E. Users & user groups

1. There are three types of user: secretary, instructor and student.

2. Each student belongs at most one course (large groups) and and at most one class (small groups).

3. Each class belongs to exactly one course.

4. Zero or more secretaries can be the secretaries responsible for zero or more courses.

5. Zero or more instructors can be the instructors responsible for zero or more courses.

6. Zero or more instructors can be the instructors responsible for zero or more classes.

7. Students can join courses and classes.

Definition. A student's instructors (if any) are the instructors responsible for her class and course.

8. Instructors and secretaries can make their courses restricted to certain students.

> *Note.* This is vague because I'm not sure what an effective and simple implementation would be. The idea is that, as an instructor, I don't want to end up unknowingly marking work from lots students who aren't actually taking my course.

9. Instructors and secretaries can create both courses (large groups) and classes (small groups).

F. Organisation of exercises

1. Each exercise has a unique name associated with a permanent url

2. Exercise descriptors:

> 2(a) Exercises can be tagged (e.g. 'quantifier proof', 'hard')
>
> 2(b) Exercises can have a brief description.
>
> 2(c) Exercises can be assigned a difficulty level (easy / normal / advanced)

4. Users can create series of exercises

5. Exercises can be added to series in a specific order

6. Exercises can be added to a series as an option set (e.g. EITHER this OR that OR the other)

7. Users can search for exercises by tag, description or popularity.

8. Instructors can assign series of exercises to courses and classes.

9. Students can select a series of exercises to complete (those assigned by an instructor are the default selection but students can select any series)

G. Feedback

1. Instructors can give boolean or numeric marks to exercises submitted by any student who is a member of a course or class that they are an instructor of.

2. Where possible an exercise is given a boolean or numeric mark automatically.

3. Instructors can give narrative feedback (e.g. `Is negation elimination helpful here?') on exercises submitted by a student who is a member of a course or class that they are an instructor of. (This is possible even where the mark is generated automatically.)

4. Instructors can review the progress of a student who is a member of a course or class that they are an instructor of.

5. Instructors can see an tabular overview of the progress of all students on a course or class for exercises submitted within a date range.

6. Students working on an exercise can request automated feedback (where available) before or after submitting the exercise. (E.g. automated feedback might say: line 4 of your proof is incorrect.)

7. Students working on an exercise can request recycled feedback (where available) before or after submitting an exercise. I.e. feedback that a human gave when another student submitted this answer.

8. Students working on an exercise can request narrative feedback from one of their instructors before or after submitting an exercise.

9. Students working on an exercise can request narrative feedback from anyone (including other students) before or after submitting an exercise. (*Note*. PhilSoc members might volunteer to help.)

H. Submission of exercises

1. A student's work on an exercise is saved automatically even if they don't submit it, so they can return to that exercise later and continue where they left off.

2. No one can view a student's exercise unless they have either requested help or submitted it.

?3. Students may submit a single exercise multiple times (repeat attempts).

?4. Marks and feedback apply to a particular submission. (e.g. student submits an answer to an exercise; it is marked; student submits a second answer which now needs another mark).

?5. Where a student has submitted several answers, the mark for the most recently submitted exercise is the mark that student receives for the exercise.

Note. I'm concerned that 3, 4 & 5 add complexity to the requirements --- in effect they mean we require a system of versioning for student's answers. Perhaps we should drop this requirement.