1 Software

Our modern societies heavily depend on software, and this dependence is likely to grow. Software is important, and it is also beautifully complex. It is complex first because of its sheer size: estimates place Google’s software at about 2 billion lines of code, and Microsoft’s Windows operating system at about 50 million lines of code\textsuperscript{1}; a pacemaker has about 100 thousand lines of code, the Boeing 787 airplane has more than 10 million, and a modern high-end car has about 100 million\textsuperscript{2}; some estimates place the size of new software produced every year to the hundreds of billions of lines of code\textsuperscript{3}.

But even very small programs can be extremely complex. The famous Collatz conjecture states that the following program terminates for all possible inputs:

```plaintext
n := input a natural number;
while (n > 1)
    if (n is even)
        n := n/2;
    else
        n := 3*n + 1;
```

The Collatz conjecture is an open problem in mathematics.\textsuperscript{4} It is a conjecture (i.e., something we believe is true), but not a theorem (i.e., not something we have proven). The fact that this 6-line program defies the understanding of our best mathematicians tells us that there is something inherently complex and challenging about software. Software is the most complex artifact that humans have ever constructed. Understanding software is an important intellectual challenge for humanity.

2 Software Science

Science is knowledge that helps us make predictions. The key word is predictions. The stronger the science, the stronger the predictions it can make. Software science helps us make predictions about the programs that we write. Will my program terminate? Is my program correct? What does correct even mean? Will my program produce a correct output? When exactly is an output correct? Should the input satisfy any conditions in order for the output to be correct? Etc.

\textsuperscript{1}According to \url{https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/}.
\textsuperscript{2}According to \url{https://www.visualcapitalist.com/millions-lines-of-code/}.
\textsuperscript{3}According to \url{https://cybersecurityventures.com/application-security-report-2017/}.
\textsuperscript{4}If you solve it, you will become famous. See also: \url{https://www.quantamagazine.org/can-computers-solve-the-collatz-conjecture-20200826/} (thanks to Samuel Lowe for suggesting this article).
3 This Course

Testing and proving: This course is an introduction to the science of software. You have already written programs. You have taken and will take more courses that teach you how to program. In most programming courses you will focus on checking program correctness by testing. Testing is very important, but as Dijkstra famously said: “Program testing can be used to show the presence of bugs, but never to show their absence!”

In this course we focus on proving program correctness. Proving is a stronger guarantee than testing. Testing checks only some inputs, whereas a proof is usually about all possible inputs. So proofs offer stronger predictions about our programs.

Logic: But in order to prove that a program is correct, we must first define what exactly do we mean by correct. For that, we will use logic. Logic is first of all a language. Contrary to natural languages (English, Greek, etc.), logic is precise and unambiguous. We can debate endlessly about the meaning of love and politics, but the meaning of a logical formula is not a matter of opinion. It is mathematically defined. This is very important because it helps avoid errors of communication. Miscommunication can be catastrophic in love and politics, but also in engineering projects.

Specification and verification: In this course we will use logic for several things. We will use logic to express properties of programs. Collectively these properties define what it means for a program to be correct: they specify the program. This is called program specification. We will also use the rules of logic to prove those properties. Proving that a program satisfies its specification is called program verification.

In this course we will learn:

- to read functional programs with types
- to write functional programs with types
- to read formal specifications written in logic
- to write formal specifications in logic
- to read proofs
- to write proofs.

LEAN: This semester, we will use the LEAN theorem prover: https://leanprover.github.io/. We will write programs in LEAN’s programming language, we will write specifications in LEAN’s logic, and we will write proofs using LEAN’s proof system.

Install LEAN on your personal computer as soon as you read this:

IMPORTANT: YOU SHOULD INSTALL LEAN 3, NOT LEAN 4!!

We found most helpful the instructions provided here: https://leanprover-community.github.io/.

Other theorem provers: The goal of this course is not to teach you LEAN. The goal is to introduce you to the science of software, formal logic, formal specification, and formal verification. We are using LEAN as a tool and as a means to an end, rather than the end itself. LEAN is just one of many tools that could be used for this purpose. Examples of other such tools are:

- ACL2s: http://acl2s.ccs.neu.edu/

5Logic goes far beyond what we will see in this course. Logic is the foundation of mathematics. It is also the foundation of language, reason, and philosophy.
The above list is by no means exhaustive. This is an active area of research, and new tools are being developed or new capabilities are added to existing tools all the time. Each tool has its own pros and cons, just like different programming languages and systems have their own pros and cons. Nevertheless, some basic concepts and principles are common to all these tools. It is these concepts and principles that we strive to teach you in this course, and it is these concepts and principles that you should strive to learn.

Having fun with proofs: Proving theorems with a tool like LEAN is a lot of fun. It’s like playing a game. The goal of the game is to prove the theorem. This is like solving a puzzle, or finding our way out of a maze. We will learn which moves to make to help us find the exit of the maze. WARNING: this game can become addictive!

How to succeed in this course: You learn by experimenting, asking questions, and making mistakes. Making mistakes is great (as long as you are not catastrophic mistakes, like drinking and driving and car crashing). Fortunately, computer science provides you with a very safe environment for making mistakes: the worst that can happen is that your program doesn’t compile, or that it doesn’t return the right result. Big deal. In this course, what can go wrong? Maybe LEAN does not accept your function definition and you don’t see why. Or maybe your function doesn’t work as expected. Or you cannot complete a proof. Etc. Try to experiment to see what goes wrong. For LEAN specific things, consult the LEAN documentation. Ask questions when you are deadlocked. There are no stupid questions.

A good way to know whether you are learning what you are supposed to be learning is whether you are able to do all the homework problems by yourself. If you are, you will do well in the course. If you are not, you should be worried. Come to our office hours if you are worried.

4 These Lecture Notes

These lecture notes will be sparse. This is intentional. Their aim is not to be a comprehensive textbook, but rather to guide you in the course (like a map). The philosophy of the course is learning by doing. Is there any other way to learn really?

In particular, these lecture notes are not about learning LEAN. There are many many good resources on LEAN freely available online: examples, tutorials, online textbooks, and many more. References to those will be provided as the course progresses.

These lecture notes will be permanently under construction. They will be updated regularly as we advance in the course. The latest version will serve as the reference point. Please look at the date of these notes, compare it to the date in your own copy, and use the latest version.

5 Other Reading

Documentation on LEAN: There is a lot of documentation available on LEAN from the following websites:

- https://leanprover.github.io/
- https://leanprover-community.github.io/
Unfortunately, there is no single document that matches exactly what we present in this course, so you will have to collect information from multiple sources.\footnote{This is also what you will have to do in your “real life” outside the university.} Also, much of the LEAN documentation is under construction and/or incomplete. We recommend starting with this (although there is a lot from the link below that we will \textit{not cover/emphasize} in this course, like type theory, dependent types, etc., for instance):

- \url{https://leanprover.github.io/theorem_proving_in_lean}

You can also consult the reference manual (unfortunately the programming part is missing):

- \url{https://leanprover.github.io/reference/}

You can also look directly at the LEAN code, libraries, etc.:

- \url{https://github.com/leanprover/lean/tree/master/library/init}

For those interested in using LEAN for formal mathematics, here’s a couple of links:

- \url{https://leanprover-community.github.io/mathematics_in_lean/index.html}

**Type Theory:** LEAN is based on so-called \textit{type theory} which studies \textit{type systems}. LEAN has a type system, and many (typed) programming languages also have type systems. Type systems are fundamental in programming (languages), but also in logic and the foundation of mathematics. However, we will not study type systems nor type theory in this class, as our main focus is to learn how to do proofs by doing. Those interested in type theory can consult the references below:

- \textit{Types and Programming Languages} by Benjamin C. Pierce.
- \textit{Advanced Topics in Types and Programming Languages} by Benjamin C. Pierce, editor.
- A short introduction to LEAN’s type system can be found here: \url{https://leanprover.github.io/theorem_proving_in_lean/dependent_type_theory.html}.

as well as relevant courses in programming languages.

**Software Foundations:** \url{https://softwarefoundations.cis.upenn.edu/}. \textit{Software Foundations} is a book series available online. It goes much further than we do in this course, but its first part (Volume 1) serves as good reading material for this course. \textit{Software Foundations} uses a different theorem prover, called Coq. LEAN is quite similar to Coq, and you should be able to follow and re-do most of the things described in \textit{Software Foundations} in LEAN. We often borrow exercises from \textit{Software Foundations} and adapt them to our course. We thank the authors of \textit{Software Foundations} for making the series freely available.

**Other Courses:** In addition to the \textit{Software Foundations} online series, there is a number of courses available online which are related to our course. Here’s a partial list for those interested:

- \textit{Logic and Proof} at CMU: \url{https://leanprover.github.io/logic_and_proof/}. This course is also based on LEAN.
- \textit{Logical Verification} at Vrije Universiteit Amsterdam: \url{https://lean-forward.github.io/logical-verification/2020/}. This course is also based on LEAN.
- \textit{Semantics of Programming Languages} at TU Munich: \url{http://www21.in.tum.de/teaching/semantik/WS1920/}. This course is based on another theorem prover called Isabelle.
**Textbooks:**  *THERE IS NO REQUIRED TEXTBOOK FOR THIS COURSE.* For those interested in learning more about logic and its use in computer science in general and specification/verification in particular, here are some textbooks:

- *Logic in Computer Science: Modelling and reasoning about systems*, by Huth and Ryan [9].
- *Handbook of Practical Logic and Automated Reasoning*, by Harrison [6].
- *The Calculus of Computation - Decision Procedures with Applications to Verification*, by Bradley and Manna [2].

For those interested in learning more about verification and formal methods:

- *Model Checking*, by Clarke, Grumberg and Peled [3].
- *Principles of Model Checking*, by Baier and Katoen [1].
- Several books on the *SPIN Model Checker* by Holzmann [7, 8].
- *Handbook of Model Checking*, by Clarke, Henzinger, Veith, Bloem [4].

**The history of logic, in comics:**  The following is a wonderful book on the history of logic and foundations of mathematics, written by famous computer scientist Christos Papadimitriou:


## 6 Course Outline

To be populated as we go along.

## 7 Summary of Proof Tactics

To be populated as we go along.

## 8 Allowed LEAN Library Axioms/Theorems

To be populated as we go along.
References


