Motivation

Show a bunch of computer program that demonstrate that

- While programming, we typically make a bunch of careless mistakes.
- Some bugs are hard to find by running tests.
- Similar expressions may have a different meaning in different programming languages.
- It can be very hard to find correctness proofs, but in some of these cases it can be very easy to check a given proof.
- Even for “simple” programs (few boolean variables) it can be very tedious to analyze correctness.
TODO: Add everything that was shown in the lecture to the slides.
Propositional logic
We presume that all of you know propositional logic. Propositional logic is one of the basic concepts in computer science, it has applications in many areas but there exist several terminology and several notations.

Goals of this chapter are

- recall the basic ideas of propositional logic
- fix the notation and terminology that we use in this lecture
- ease the presentation of first order logic (next chapter)
- introduce the idea of a proof system
Syntax of Propositional Logic

**Definition**

Let $\mathcal{V}_{\text{PL}}$ be a set whose elements we call *propositional logical variables*

1. **false** is a PL formula
2. for each $X \in \mathcal{V}_{\text{PL}}$, $X$ is a PL formula
3. if $F$ is a PL formula, then $\neg F$ is a PL formula
4. if $F_1$ and $F_2$ are PL formulas, then $(F_1 \land F_2)$ is a PL formula

**Abbreviations**

\[
\begin{align*}
\text{true} & := \neg \text{false} \\
F_1 \lor F_2 & := \neg(\neg F_1 \land \neg F_2) \\
F_1 \rightarrow F_2 & := (\neg F_1 \lor F_2) \\
F_1 \leftrightarrow F_2 & := (F_1 \rightarrow F_2) \land (F_2 \rightarrow F_1)
\end{align*}
\]
Terminology

We call true, false atoms.
If \( X \in \mathcal{V}_{PL} \), we call \( X \) an atom.
If \( F \) is an atom, we call \( F \) and \( \neg F \) a literal.
We call the symbols \( \neg, \land, \lor, \rightarrow, \leftrightarrow \) logical connectives.

Notation

We may omit parentheses

- use the following order of precedence for logical connectives:
  \( \neg, \land, \lor, \rightarrow, \leftrightarrow \)
- use the convention that binary operators are right-associative
Semantics

We call `true` and `false` *truth values* and we call a mapping \( \rho : \mathcal{V}_{PL} \to \{\text{true, false}\} \) a *variable assignment*.

**Definition**

The *evaluation* is a mapping that takes a PL formula \( F \) and a variable assignment \( \rho \) and is defined as follows.

1. \( \llbracket \text{false} \rrbracket_\rho \) is `false`
2. for each \( X \in \mathcal{V}_{PL} \), \( \llbracket X \rrbracket_\rho \) is \( \rho(X) \)
3. \( \llbracket \neg F \rrbracket_\rho \) is \( \begin{cases} 
\text{true} & \text{if } \llbracket F \rrbracket_\rho \text{ is false} \\
\text{false} & \text{if } \llbracket F \rrbracket_\rho \text{ is true}
\end{cases} \)
4. \( \llbracket F_1 \land F_2 \rrbracket_\rho \) is \( \begin{cases} 
\text{true} & \text{if } \llbracket F_1 \rrbracket_\rho \text{ is true and } \llbracket F_2 \rrbracket_\rho \text{ is true} \\
\text{false} & \text{otherwise}
\end{cases} \)

**Definition**

1. We call a PL formula \( F \) [*satisfiable*] if there is a variable assignment \( \rho \) such that \( \llbracket F \rrbracket_\rho \) is `true`
2. We call a PL formula \( F \) [*valid*] if for all variable assignments \( \rho \) the evaluation \( \llbracket F \rrbracket_\rho \) is `true`
Which of the following formulas is satisfiable, which is valid?

- $F_1 : P \land Q$
  satisfiable, not valid
- $F_2 : \neg(P \land Q)$
  satisfiable, not valid
- $F_3 : P \lor \neg P$
  satisfiable, valid
- $F_4 : \neg(P \lor \neg P)$
  unsatisfiable, not valid
- $F_5 : (P \rightarrow Q) \land (P \lor Q) \land \neg Q$
  unsatisfiable, not valid

Is there a formula that is unsatisfiable and valid?
TODO explain truth table
Finding satisfying assignments for PL formulas can be a time-consuming task. In practice, we use tools to solve this task. Tools that are specialized in finding satisfying assignments for PL formulas are called *SAT solvers*. Later in this lecture, we will use tools that are called *SMT solvers*. Every SMT solver is also able to find satisfying assignments for PL formulas, but SMT solvers are typically not highly optimized for this task. Since performance is not an issue for us, we will not learn how to use a SAT solver and start to use SMT solvers right now.

Users communicate with an SMT solver via so called *SMT scripts*. An SMT script is a file that contains a list of commands. In order to get a satisfying assignment for a PL formula $F$, we need only the following four commands.

1. First, we write `(define-fun X Bool)` for each propositional variable $X$ in our formula $F$.
2. Then, we write `(assert F)` and have to write the formula $F$ using the infix notation that is defined at the following URL http://smtlib.cs.uiowa.edu/theories-Core.shtml. E.g., for PL formulas $F_1, F_2$ we write `(and F_1 F_2)` instead of $(F_1 \land F_2)$.
3. Next, we write `(check-sat)`.
4. Finally, if the formula is satisfiable and we want to see a satisfying assignment, we can write `(get-model)`.

There are several SMT solvers available, we propose to use Z3 because it is also available via web interface. https://rise4fun.com/z3/