# CS720

## Logical Foundations of Computer Science

### Lecture 1: course structure, Coq basics

Tiago Cogumbreiro

Do computers do

what we tell them to?

# How do we talk to computers?

How do we talk to computers?

With programs

# How do we construct a program?

How do we construct a program?

We write **code** and we give it to a compiler/interpreter

# Does the code match our intent?

# Does the code match our intent?

- Do we check inputs/outputs? Eg, for an input of x, expect an output of y
- **Do we check *all* inputs/outputs?** Eg, the result is a sorted list
- Do we check resource usage? Eg, takes under X-seconds to run
- Do we check all resource usage? Eg, takes at most X-second for any run

# Does the code match our intent?

- Do we check inputs/outputs? Eg, for an input of x, expect an output of y
- **Do we check *all* inputs/outputs?** Eg, the result is a sorted list
- Do we check resource usage? Eg, takes under X-seconds to run
- Do we check all resource usage? Eg, takes at most X-second for any run

# How do we even assess our intent?

# Does the code match our intent?

- Do we check inputs/outputs? Eg, for an input of x, expect an output of y
- **Do we check *all* inputs/outputs?** Eg, the result is a sorted list
- Do we check resource usage? Eg, takes under X-seconds to run
- Do we check all resource usage? Eg, takes at most X-second for any run

# How do we even assess our intent?

- How do we convince ourselves that our intent is correct? Tests, coverage, audit, **logic**
- How do we convince others that our intent is correct? Tests, coverage, audit, **logic**

# Does the code match our intent?

- Do we check inputs/outputs? Eg, for an input of x, expect an output of y
- **Do we check *all* inputs/outputs?** Eg, the result is a sorted list
- Do we check resource usage? Eg, takes under X-seconds to run
- Do we check all resource usage? Eg, takes at most X-second for any run

# How do we even assess our intent?

- How do we convince ourselves that our intent is correct? Tests, coverage, audit, **logic**
- How do we convince others that our intent is correct? Tests, coverage, audit, **logic**

# Does the compiler/interpreter preserve the intent?

Welcome to

# ***Programming Language Theory***

# About the course

- **Course web page:** `cogumbreiro.github.io/teaching/cs720/s22/`
  - Office hours
  - Syllabus
  - Course schedule
- **Gitlab** to share homework assignments
- **Discord** for communication (announcements, links)
  Discord is preferable to email!
- **Gradescope** for homework submission

# About the course

- A programming course (Coq)
- A theoretical course (logic)
- A forum to practice paper presentation (PhD)

# Course structure

- Course: 28 lectures
- 12 homework assignments (85%) + (1 paper presentation + 12 presentation reviews (15%))
- **No exams**; around 1 homework assignment per week; assignments are not small (but with practice, you can do them quickly)

---

***Course structure inspired by UPenn's CIS500; their grading is stricter (12 homework assignments + midterm + exam).***

# Homework (85%)

- No late homework. Late homework = 0 points.
- Homework is your personal individual work.

> It is ***acceptable*** to discuss the concept in general terms, but ***unacceptable*** to discuss specific solutions to any homework assignment.

# Grading

- Work is **partially** graded by Gradescope.
- Unreadable solutions will get 0 points.
- If Gradescope gives you 0 points, then your grade is 0 points.
- Some questions are manually graded by me.

# Presentation (15%)

- Each paper is handled by 1 group of students
- Groups will have 2 students, 1 group has 3 students
- 1 paper = 1 group
- Each student must present for 10 minutes
- Each student must review their colleagues presentations (~22 presentations)

# Textbooks

- Logical Foundations (Software Foundations - Volume 1). Benjamin C. Pierce, ***et al***. 2021. Version 6.1.
- Programming Languages Foundations (Software Foundations - Volume 2). Benjamin C. Pierce,***et al***. 2021. Version 6.1.

## Recommended

- Types and programming languages. Benjamin C. Pierce. 2002.
- Software foundations @ YouTube
- Oregon PL Summer School Archives (in particular: 2013, 2014, )

# Programming language semantics

- Describes a **computation model**
- Defines the set of possible behaviors through some primitives
- Mathematically precise properties of a computation model

# Bird's eye view

Here is what we will learn

# How do check if a program is correct?

## Does the program meet the intent?

```
let division (a b: int) : int
  requires { true }
  ensures  { exists r: int. a = b * result + r /\ 0 ≤ r < b }
=
  let q = ref 0 in
  let r = ref a in
  while !r ≥ b do
    invariant { true }
    q := !q + 1;
    r := !r - b
  done;
  !q
```

**Examples:** WhyML, Dafny.

# How does the compiler check if a program is correct?

```
let division (a b: int) : int
=
  let q = ref 0 in
  let r = ref a in
  while !r ≥ b do
    q := !q + 1;
    r := !r - b
  done;
  !q
```

Examples: OCaml, F#, ReasonML

# Specifying a functional language

## Language grammar

$$t ::= x \mid v \mid t\ t \qquad v ::= \lambda x\colon T.t \qquad T ::= T \to T \mid \mathtt{unit}$$

## Evaluation rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \texttt{(E-app1)} \qquad \frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}\ \texttt{(E-app2)}$$

$$(\lambda x\colon T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}\ \texttt{(E-abs)}$$

# Specifying a functional language

## Type checking rules

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\texttt{T-var}) \qquad \frac{\Gamma[x \mapsto T_1] \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}\, T_1.t_2 : T_1 \to T_2}\ (\texttt{T-abs})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash \lambda x{:}\, T_1.t_2 : T_1 \to T_2}\ (\texttt{T-app})$$

# What about all programs of a given language?

## Progress: valid programs execute one step

***Any valid program is either a value or can evaluate.***
If $\Gamma \vdash t : T$, then either $t$ is a value, or there exists some $t'$ such that $t \longrightarrow t'$.

## Subject reduction: valid programs remain valid

***The validity of a program is preserved while evaluating it.***
If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

> Can you give an example of a property?

# What we will learn in this course

## Course summary

**Specification:** logical reasoning, describing program behavior
**Abstraction:** capturing the fundamentals, thinking from first principles
**Testing:** unit and property testing

# `Basics.v`: Part 1

## A primer on the programming language Coq

We will learn the core principles behind Coq

# Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?

# Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

## Examples?

- boolean
- 4 suits of cards
- byte
- int32
- int64

# Declare an enumerated type

```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

- `Inductive` defines an (enumerated) type by cases.
- The type is named `day` and declared as a `: Type` (Line 1).
- Enumerated types are delimited by the assignment operator (`:=`) and a dot (`.`).
- Type `day` consists of 7 cases, each of which is is tagged with the type (`day`).

# Printing to the standard output

`Compute` prints the result of an expression (terminated with dot):

```
Compute monday.
```

prints

```
     = tuesday
     : day
```

# Interacting with the outside world

UMASS BOSTON

- Programming in Coq is different most popular programming paradigms
- Programming is an **interactive** development process
- The IDE is very helpful: workflow similar to using a debugger
- It's a REPL on steroids!
- `Compute` evaluates an expression, similar to `printf`

# Inspecting an enumerated type

```
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end
```

# Inspecting an enumerated type

```
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end
```

- `match` performs **pattern matching** on variable `d`.
- Each pattern-match is called a ***branch***; the branches are delimited by keywords `with` and `end`.
- Each ***branch*** is prefixed by a mid-bar (`|`) (`⇒`), a pattern (eg, `monday`), an arrow (`⇒`), and a return value

# Pattern matching example

```
Compute match monday with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.
```

# Create a function

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

# Create a function

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

- `Definition` is used to declare a function.
- In this case `next_weekday` has one parameter `d` of type `day` and returns (`:`) a value of type `day`.
- Between the assignment operator (`:=`) and the dot (`.`), we have the body of the function.

# Example 2

```
Compute (next_weekday friday).
```

yields (Message pane)

```
     = monday
     : day
```

`next_weekday friday` is the same as `monday` (after evaluation)

# Your first proof

```
Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
  simpl.        (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.
```

# Your first proof

```
Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
  simpl.       (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.
```

- `Example` prefixes the name of the proposition we want to prove.
- The return type (`:`) is a (logical) **proposition** stating that two values are equal (after evaluation).
- The body of function `test_next_weekday` uses the `ltac` proof language.
- The dot (`.`) after the type puts us in proof mode. (Read as "defined below".)
- This is essentially a unit test.

# Ltac: Coq's proof language

`ltac` is **imperative**! You can step through the state with CoqIDE

`Proof` begins an `ltac`-scope, yielding

```
1 subgoal
______________________________________(1/1)
next_weekday (next_weekday saturday) = tuesday
```

Tactic `simpl` evaluates expressions in a goal (normalizes them)

# Ltac: Coq's proof language

```
1 subgoal
______________________________________(1/1)
tuesday = tuesday
```

- `reflexivity` solves a goal with a pattern `?X = ?X`

```
No more subgoals.
```

- `Qed` ends an `ltac`-scope and ensures nothing is left to prove

# Function types

Use `Check` to print the type of an expression:

```
Check next_weekday.
```

which outputs

```
next_weekday
     : day → day
```

Function type `day → day` takes one value of type `day` and returns a value of type `day`.

# Basic.v

- New syntax: `Definition` declares a non-recursive function
- New syntax: `Compute` evaluates an expression and outputs the result + type
- New syntax: `Check` prints the type of an expression
- New syntax: `Inductive` defines inductive data structures
- New syntax: `Fixpoint` declares a (possibly) recursive function
- New syntax: `match` performs pattern matching on a value
- New tactic: `simpl` evaluates functions if possible
- New tactic: `reflexivity` concludes a goal `?X = ?X`

# Ltac vocabulary

- `simpl`
- `reflexivity`