CS 3110 Lecture 1
Course Overview

Ramin Zabih
Cornell University CS
Fall 2010

www.cs.cornell.edu/courses/cs3110

Course staff

- Professor: Ramin Zabih
- Graduate TA’s: Joyce Chen, Brian Liu, Dane Wallinga
- Undergraduate consultants: Ashir Amer, Jacob Bank, Boris Burkov, Steve Gutz, Oneek Iftikhar, Gautam Kamath, Nyk Lotocky, Katie Meusling, Lucas Waye, Greg Zecchini

Course meetings

- Lectures Tuesdays and Thursdays
- Recitation sections Mondays and Wednesdays, 2:30 and 3:35, HLS314
  - A third section will be added shortly, at a time that helps out the students
  - Email to class, before Monday morning
- New material in lecture **and** section
  - You are expected to attend both!
- Class participation counts

Course web site

- www.cs.cornell.edu/courses/cs3110
- Links to lecture notes are not yet live
  - Will appear after lecture
  - One more reason to actually attend!
- Course material, homework, software, announcements

Course news group

- cornell.class.cs3110
- This should be your default way to ask questions
  - If you use email, no one else will benefit from the response
    - Your classmates almost certainly want to know!
- But don't give out solutions
- Monitor the newsgroup regularly

Coursework

- 6 problem sets due Thursday 11:59PM
  - PS1 will be out on Tuesday 8/31
- Electronic submission via CMS
- Four single-person assignments, then two two-person assignments
  - You'll have 3 weeks for the big assignments
  - There will be checkpoints
- Two prelims plus a final
- 6 small in-lecture quizzes

Grading

- Roughly speaking we will follow the usual CS3110 curve (centered around a B/B+)
- Problem sets & exams count about the same, quizzes & participation count a little
  - I’m mostly interested in what you know at the end of the class
- I don’t drop an assignment or exam, but I use your overall qualitative performance
  - There is no strict numerical formula

Late policy

- You can hand it in until we start grading
  - After that, no credit
- Be sure to save whatever you currently have done, and save frequently
  - CMS is your friend
  - Be certain you have submitted something, even if it isn't perfect and you are improving it
- If you have an emergency, talk to me or to Joyce Chen before the last second
- Qualitative grading algorithm!

Academic integrity

- Strictly enforced, and easier to check than you might think
  - Automated tools, etc.
- Exams count a lot
- To avoid pressure, start early
  - We try hard to encourage this
  - Take advantage of the large veteran staff

What this course is about

- Programming isn't hard
- Programming **well** is **very** hard
  - Huge difference among programmers (10x or more)
- We want you to write code that is:
  - Reliable, efficient, readable, testable, provable, maintainable… **beautiful**!
- Expand your problem-solving skills
  - Recognize problems and map them onto the right abstractions and algorithms

Thinking versus typing

- The sooner you start writing code, the longer it will take you to get done
  - "A year at the lab bench will save you an hour at the library"
- Fact: there are an infinite number of incorrect programs
  - Corollary: the chances that small random tweaks to your code will result in the right answer are ε
  - If you find yourself changing < to <= in the hopes that your code will start working, you're in trouble
- Lesson: think before you type!!

Rule #1

- Good programmers are lazy
  - Never write the same code twice (why?)
  - Reuse libraries (why?)
  - Keep interfaces small and simple (why?)
- Pick a language that makes it easy to write the code you need
  - Early emphasis on speed is a disaster (why?)
- Rapid prototyping!

Key goal of CS3110

- Master key linguistic abstractions:
  - procedural abstraction
  - control: iteration, recursion, pattern matching, laziness, exceptions, events
  - encapsulation: closures, ADTs
  - parameterization; higher-order procedures, modules
- Mostly in service to rule #1
- Transcend individual programming languages

Other goals

- Exposure to software eng. techniques:
  - modular design.
  - unit tests, integration tests.
  - critical code reviews.
- Exposure to abstract models:
  - models for design & communication.
  - models & techniques for proving correctness of code.
  - models for space & time.

Choice of language

- This matters less than you suspect
- You need to be able to learn new languages
  - This is relatively easy if you understand programming models and paradigms
- We will be using OCaml, a dialect of ML
- Why use yet another language?
  - Not to mention an obscure one??
- Main answer: OCaml programs are much easier to think about

Why OCaml?

- RDZ's favorite feature: OCaml makes certain common errors simply impossible
  - More precisely, fail at compile time
  - Early failure is very important (why?)
- OCaml is a functional language
  - More on this in a second
- It is statically typed and type-safe
  - Lots of bugs are caught at compile time

Imperative Programming

- Program uses **commands** (a.k.a **statements**) that *do* things to the **state** of the system:
  - x = x + 1;
  - p.next = p.next.next;

- Functions/methods can have **side effects**
  - int wheels(Vehicle v) { v.size++; return v.numw; }

Functional Style

- **Idea:** program without side effects
  - Effect of a function is *only* to return a result value

- Program is an **expression** that **evaluates** to produce a **value** (e.g., 4)
  - E.g., 2+2
  - Works like mathematical expressions

- Enables **equational reasoning** about programs:
  - if x = y, replacing y with x has no effect:

`let x = f(0) in x+x` same as `f(0)+f(0)`

Functional Style

- Binding variables to values, not changing values of existing variables

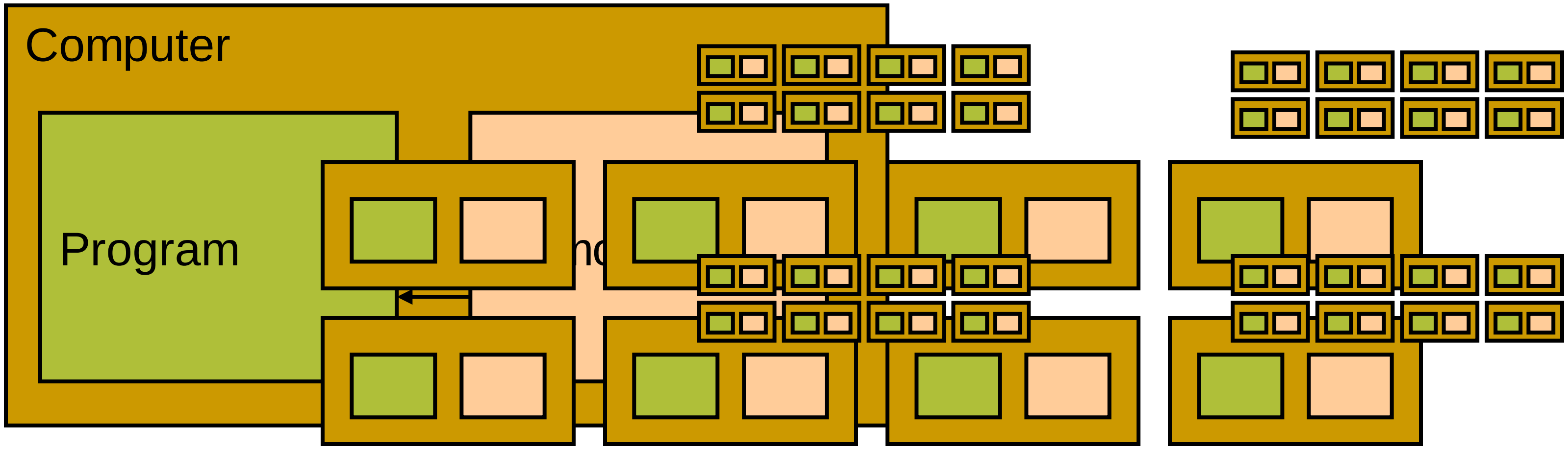
- No concept of **`x=x+1`** or **`x++`**

- These do nothing remotely like **`x++`**

```
let x = x+1 in x
let rec x = x+1 in x
```

- Former assumes an existing binding for **`x`** and creates a new one (no modification of **`x`**), latter is invalid expression

Trends against imperative style

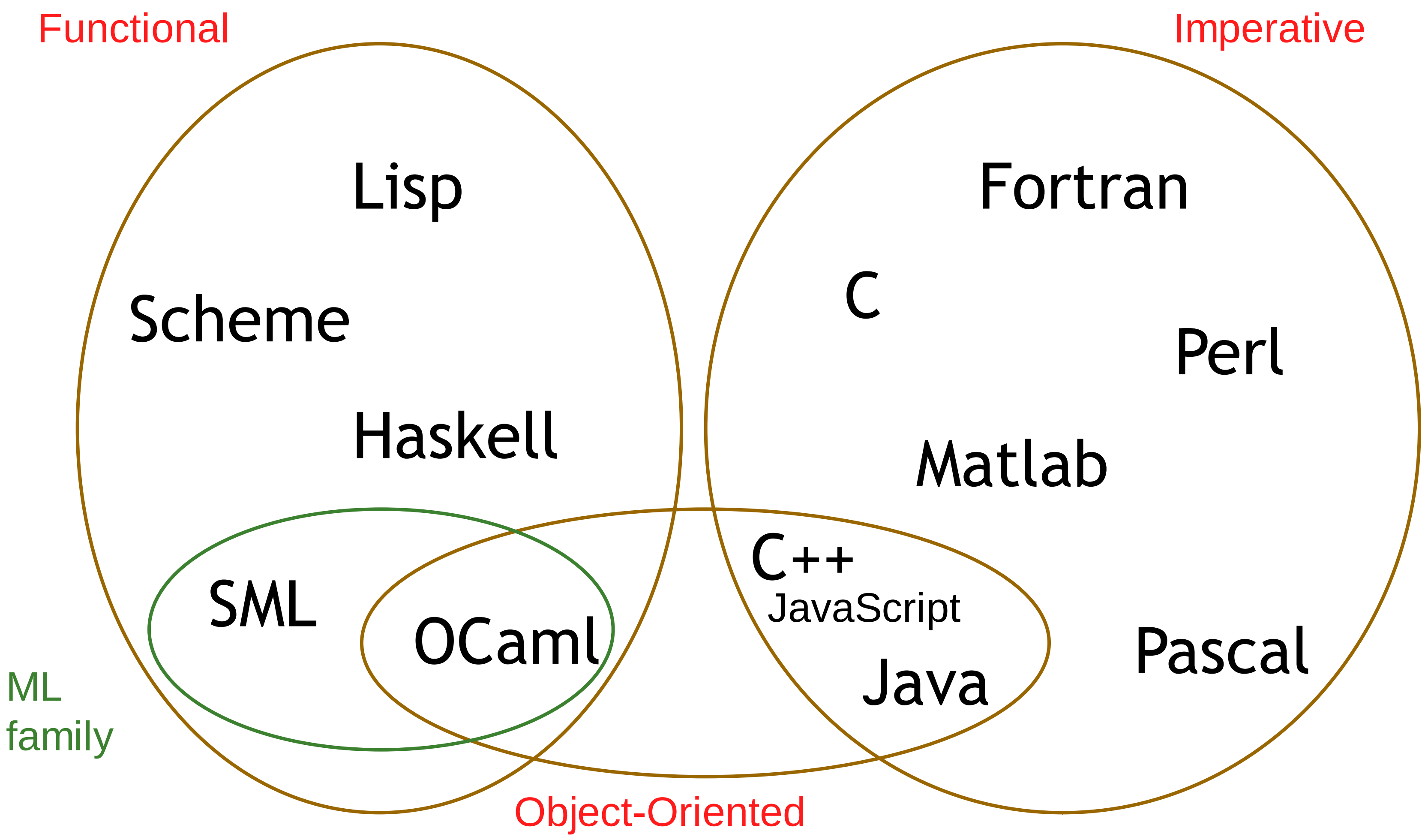


- **Fantasy:** program interacts with a single system state
  - Interactions are reads from and writes to variables or fields.
  - Reads and writes are very fast
  - Side effects are instantly seen by all parts of a program
- **Reality today:** there is no single state
  - Multicores have own caches with inconsistent copies of state
  - Programs are spread across different cores and computers (PS5 & PS6)
  - Side effects in one thread may not be immediately visible in another
  - **Imperative languages are a bad match to modern hardware**

Imperative vs. functional

- ML: a *functional* programming language
  - Encourages building code out of functions
  - Like mathematical functions; f(x) always gives the same result
  - No side effects: easier to reason about what happens
  - Equational reasoning is easier
  - A better fit to hardware, distributed and concurrent programming
- Functional style usable in Java, C, …
  - Becoming more important with fancy interactive UI's and with multiple cores
  - A form of encapsulation – hide the state and side effects inside a functional abstraction

Programming Languages Map



Imperative “vs.” functional

- Functional languages:
  - Higher level of abstraction
  - Closer to specification
  - Easier to develop robust software

- Imperative languages:
  - Lower level of abstraction
  - Often more efficient
  - More difficult to maintain, debug
  - More error-prone

Rough schedule

- Introduction to functional programming (6)
- Modular programming and functional data structures (4)
- Reasoning about correctness (4)
- ***Prelim 1***
- Imperative programming and concurrency (4)
- Data structures and analysis of algorithms (5)
- ***Prelim 2***
- Topics: memoization, streams, managed memory (5)
- ***Final exam***