CS 3110 Lecture 1
Course Overview

Ramin Zabih
Cornell University CS
Fall 2011

www.cs.cornell.edu/courses/cs3110
Course staff

- **Professor**: Ramin Zabih
- **Graduate TA’s**: Joyce Chen, Alex Fix
- **Undergraduate section TA’s**:
  - Ashir Amer, Gautam Kamath, Katie Meusling
  - Possibly more to come
- **Undergraduate course consultants**:
  - Lots! Complete list available shortly
- You have a large and veteran staff to help you - make good use of them!
Course meetings

- Lectures Tuesdays and Thursdays
- Recitation sections Mondays and Wednesdays, 11:15, 2:30 and 3:35
  - A fourth section will be added shortly, at a time that helps out the students (probably in the evening)
  - Details in email, before Monday morning
- New material in lecture and section
  - You are expected to attend both
- Class participation counts
  - Please go to the same section
Course web site

- [www.cs.cornell.edu/courses/cs3110](http://www.cs.cornell.edu/courses/cs3110)
  - Will be live later today
  - Course material, homework, announcements, etc.

- Suggested readings will include a complete set of course notes
  - Nearest equivalent to a textbook
  - But the lectures and sections are definitive

- Links to lecture notes live after lecture
  - Sketchy, but most accurate summary

- Goal is to help, not replace attendance!
Piazza will be set up shortly
- This is an experiment,
- The old approach (news group) doesn’t scale up

Assignments will be handed in via CMS
- Everything except exams and quizzes
- Grades for everything recorded on CMS
Coursework

- 6 problem sets due Thursday 11:59PM
  - Exception: PS1 (out today) is due Tuesday 9/6
- 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 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, especially as shown on the final exam

- I don’t drop an assignment or exam, but I use your overall qualitative performance
  - Look for a pattern of your overall performance
  - But the bottom third of class isn’t likely to get an A
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. Alex or Joyce 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
  - When exam scores differ from problem set scores, we typically go with exam scores

- 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 $\epsilon$
  - If you find yourself changing $<$ to $\leq$ in the hopes that your code will start working, you’re in trouble

- Lesson: think before you type!!
CS3110 challenges

- In previous programming courses smart students can get away with bad habits
  - “Just hack on the code all night until it works”
  - Can solve the entire problem by yourself
  - Write the whole program before testing any part(!)

- A bit like basketball; CS3110 ≈ NBA
  - Professionals need good work habits & right approach

- You will also need to think rigorously about programs and the models behind them
  - Think for a few minutes, rather than type for days!
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.

- **Rigorous thinking about programs!**
  - Proofs, somewhat like high school geometry
Choice of language

- This matters less than you suspect
- Must 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, they 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:

\[
\text{let } x = f(0) \text{ in } x+x \quad \text{same as} \quad f(0)+f(0)
\]
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

- **Functional**
  - Lisp
  - Scheme
  - SML
  - OCaml
  - ML family

- **Imperative**
  - Fortran
  - C
  - C++
  - Java
  - JavaScript
  - Perl
  - Matlab
  - Pascal

- **Object-Oriented**
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
Example 1: Sum Squares

\[ y = 0; \]
\[ \text{for } (x = 1; x \leq n; x++) \{ \]
\[ \quad y = y + x*x; \]
\[ \} \]
Example 1: Sum Squares

```c
int sumsq(int n) {
    y = 0;
    for (x = 1; x <= n; x++) {
        y += x*x;
    }
    return n;
}

let rec sumsq (n:int):int =
    if n=0 then 0
    else n*n + sumsq(n-1)
```
Example 1: Sum Squares Revisited

Types can be left implicit and are then inferred: \( n \) an integer, returns an integer

```ocaml
define sumsq =
  if \( n=0 \) then 0
  else \( n \times n + \text{sumsq}(n-1) \)
```

Example 1a: Sum f’s

Functions are first-class objects, used as arguments returned as values

```ml
let rec sumop f n =
  if n=0 then 0
  else f n + sumop f (n-1)
```

```ml
sumop cube 5
sumop (function x -> x*x*x) 5
```
Example 2: Reverse List

List reverse(List x) {
    List y = null;
    while (x != null) {
        List t = x.next;
        x.next = y;
        y = x;
        x = t;
    }
    return y;
}
Example 2: Reverse List

let rec reverse lst =
  match lst with
  | [] -> []
  | h :: t -> reverse t @ [h]

Pattern matching simplifies working with data structures, being sure to handle all cases
Example 3: Pythagoras

```haskell
let pythagoras x y z =
    let square n = n*n in
    square z = square x + square y
```

Every expression returns a value, when this function is applied it returns a Boolean value.
Why ML?

- ML (esp. Objective Caml) is the most robust and general functional language available
  - Used in financial industry: good for rapid prototyping.

- ML embodies important ideas much better than Java, C++
  - Many of these ideas still work in Java, C++, and you should use them…

- Learning a different language paradigms will make you more flexible down the road
  - Likely that Java and C++ will be replaced by other languages
  - Principles and concepts beat syntax
  - Ideas in ML will probably be in next gen languages
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