Data Structures and Functional Programming

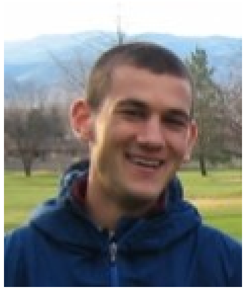
Course Overview

http://www.flickr.com/photos/rofi/2097239111/

Nate Foster and Michael George
Cornell University
Spring 2014

Course staff

- **Instructor:** Nate Foster
  - Joined Cornell in 2010 from Upenn
  - Research area: programming languages



- **Instructor:** Michael George
  - Joined Cornell in 2013 from Cornell
  - Research area: programming languages

TAs: Jonathan DiLorenzo, Ben Greenman, Ben Carriel, Muhammad Khan, Arjun Biddanda, Jianneng Li, Harris Karsch

- >10 person-years on 3110 course staff

Consultants: *many*

Course meetings

Lectures: Tuesday and Thursday 10:10-11am

Recitations:

New material in lecture *and* recitation

- You are expected to attend both

Class participation counts

- Please stick to the same section

Consulting: lots of hours (see website)

Course web site

http://www.cs.cornell.edu/Courses/cs3110
- Course material
- Homework
- Announcements

Includes a complete set of course notes
- Nearest equivalent to a textbook
- But the lectures and sections are definitive

Links to lecture notes will go live shortly after lecture

Goal is to help, not replace attendance!

Piazza and CMS

piazza Ask. Answer. Explore. Whenever.

- Online discussion forum
- Monitored by TAs/consultants
- Ask for help, but don’t post solutions

CMS

- “Course Management System”
- Assignments and grades posted here

Coursework

6 problem sets (+ 1 optional)
- Due Thursdays at 11:59pm
- Optional PS #0 (out today) due Thurday 2/30
- Electronic submission via CMS

4 x individual assignments

2 x two-person assignments
- 3 weeks for the big assignments
- There will be intermediate checkpoints

2 preliminary exams and a final

Grading

Rough breakdown:

- 45% problem sets
  - automatic grading for correctness
  - manual grading for design
- 30% prelims
- 20% final
- 5% participation (lecture, section, piazza,...)

We expect the median grade to be in the B/B+ range.

Karma

This material is fun and interesting

- You are encouraged to explore on your own
- We'll give you suggestions for things to try
- But come up with your own too!

But...Karma is completely optional and will not affect your grade

Late policy

Two free “slip days”

- Due Saturday at 11:59PM
- Penalties applied if you run out

No-compile grace

Due Saturday at 11:59PM

Small diff for a penalty

Save your code and submit early and often

- CMS is your friend
- Submit early...you can always resubmit

If you have a emergency (e.g., medical, family) talk to Nate before the last second

Academic integrity

Two requests:

1. You are here as part of an academic community. Act with integrity.

2. If you aren't sure whether some type of collaboration is allowed, ask!

...and one note:

We use automated software to detect cheating.
It works.

Special Needs and Wellness

We will provide reasonable accommodations to students who have a documented disability (e.g., physical, learning, psychiatric, vision, hearing, or systemic).

If you are experiencing undue personal or academic stress at any time during the semester (or if you notice that a fellow student is), contact me, Engineering Advising, or Gannett.

What this course is about

Programming isn’t hard

Programming **well** is **very** hard

- Programmers vary greatly
- 10X or more difference in skills

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

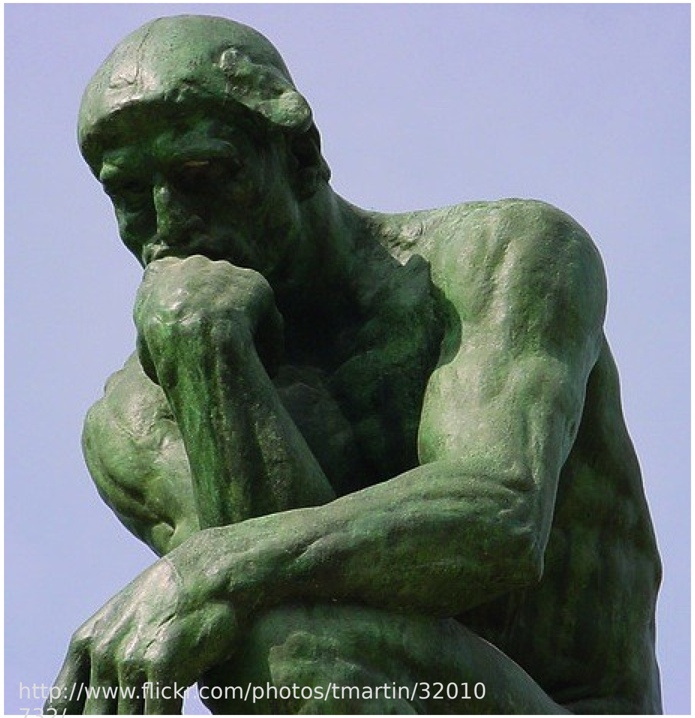
"A year at the lab bench saves an hour at the library"

Fact: there are an infinite number of incorrect programs

Corollary: making random tweaks to your code is unlikely to help

- If you find yourself changing "<" to "<=" in the hopes that your code will work, you're in trouble

Lesson: think before you type!



http://www.flickr.com/photos/tmartin/32010732/

CS 3110 Challenges

In early courses smart students can get away with bad habits

- “Just hack until it works”
- Solve everything by yourself
- Write first, test later

CS 3110 ≈ Tour de France

- Professionals need good work habits and the right approach

Will need to think *rigorously* about programs and their models

- Think for a few minutes, instead of typing for days!

http://www.flickr.com/photos/franklintello/4349205547/

Rule #1

Good programmers are **lazy**

- Never write the same code twice
- Reuse libraries
- Keep interfaces small and simple

Main 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

Transcends individual programming languages

Other goals

Exposure to software engineering techniques:

- Modular design
- Integrated testing
- Code reviews

Exposure to abstract models:

- Models for design & communication
- Models & techniques for proving correctness
- Models for analyzing space & time

Rigorous thinking about programs!

- Proofs, like in high school geometry

Tools

We will be using OCaml
- A popular and growing functional language
- (Lots) more on OCaml soon

We will use other common programming tools
- Linux
- Git (later in the course)

For help getting going:
- PS 0
- Demo sessions Thursday and Friday
    - please try to download before coming!
- Weekend consulting

Why OCaml?

OCaml programs are easy to reason about
- variables don't change
- function output depends only on input
- well defined semantics

OCaml makes abstraction easy
- polymorphism
- higher-order functions
- modules

OCaml is safe
- many errors caught early
- “once it compiles,
  it's probably right”

Imperative style

Program uses **commands** (a.k.a **statements**) that **do** things to the **state** of the system:

- x = x + 1;
- a[i] = 42;
- p.next = p.next.next;

Functions and methods can have **side effects**

- int wheels(Vehicle v) {
  - v.size++;
  - return v.numw;
- }

Trends against imperative style

The fantasy:

- there is a single state
- the computer does one thing at a time
- in the order that I ask it to

The reality:

- there is no single state
  - programs have many threads
  - spread across many cores
  - spread across many processors
  - spread across many computers
  - each with its own view of memory
- there is no single program
  - most applications integrate multiple services
- the program you write isn't the one that runs
  - aggressive compiler optimizations

Imperative style is not well suited to modern computing

Functional Style

A program is an **expression** describing **what** to compute.

Variables never change(!)

- they are more like definitions
- function output depends only on input

Example:

```
let x = 0 in
let f y = x + y in
let x = 3 in
f 5
```

f is a function that takes in y and returns x + y

What is f 5? (vote: 8 or 5?)

Advantages of functional style

(Functional) abstraction:

- Functions can be called promiscuously
- Can pass functions as arguments to other functions
- ...and return them from functions

- Remember rule #1?

Testing and specification:

- Only one behavior to describe

Equational reasoning:

- if x equals y, then replacing y with x has no effect:
  let x = f 0 in x + x **is the same as** (f 0) + (f 0)
- (mostly)
- Useful to programmer AND compiler

Imperative “vs.” functional

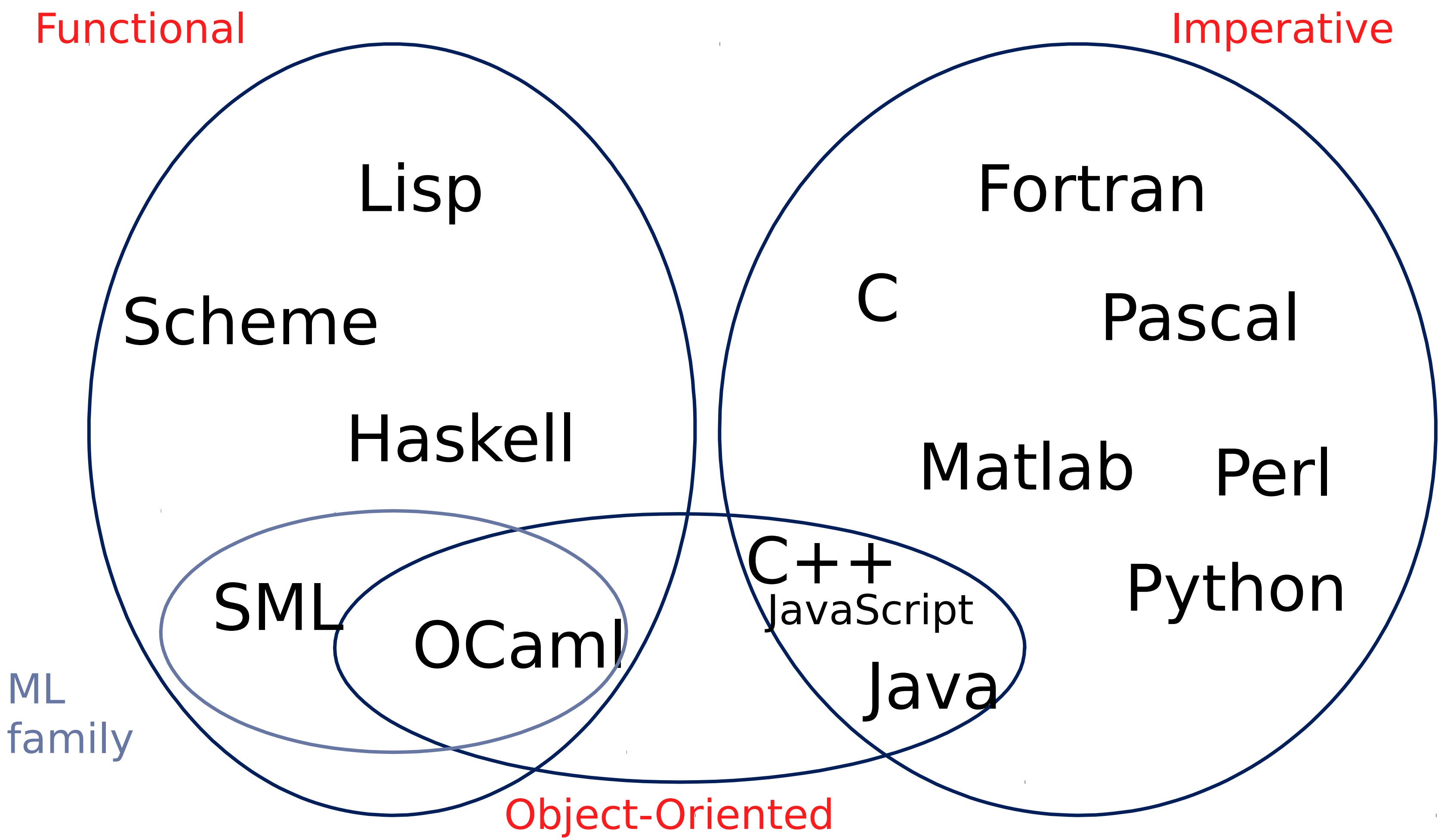
Functional languages:

- 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

Programming Languages Map



Example 1: Sum Squares

How do I program without changing variables?

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

Example 1: Sum Squares

```
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)

let rec sumsq n =
   if n = 0 then 0
   else n*n + sumsq (n-1)
```

Example 2: Sumcubes

Remember rule #1?

Create a common abstraction by passing functions as arguments:

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

```
let sumsquares x = sumof square x
let sumcubes   x = sumof cube   x
```

```
let sumcubes = sumof cube
```

```
let sumcubes = sumof (fun x → x*x*x)
```

Example 3: 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 3: 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 4: Quicksort

Describe quicksort in English.

Describe quicksort in Java:

(No).

Quicksort in OCaml:

```
let qsort l = match l with
   | [] → []
   | mid::rest →
     let left, right = partition ((<) mid) rest
     in (qsort left) @ [mid] @ (qsort right)
```

Why OCaml?

OCaml is a great language to know

- Lightweight and good for rapid prototyping
- Powerful
- Growing in popularity

OCaml is a great vehicle for ideas

- Functional programming
- Formal reasoning
- Software design
- These skills apply to all languages

Learning new languages and paradigms is useful

- Principles and concepts beat syntax
- You will think differently

Rough schedule

Introduction to functional programming (6)

Functional data structures (5)

Verification and Testing (5)

Preliminary Exam #1

Concurrency (1)

Data structures and analysis of algorithms (5)

Preliminary Exam #2

Topics: streams, λ-calculus, garbage collection

Final exam

Keep an eye on Piazza

- Demo session locations and times
- Weekend consulting times
- VM download
- PS0 release