Data Structures and Functional Programming
Course Overview

Nate Foster
Cornell University
Spring 2012
Course staff

Instructor: Nate Foster
- Joined Cornell last year from UPenn
- Research area: programming languages
- Functional programmer since 1998

TAs: Shrutarshi Basu (coordinator), Ashir Amer, Stuart Davis, Gautam Kamath, Katie Meusling, Greg Zecchinni

Consultants: many

You have a large and veteran staff. Make use of them!

Office hours in Upson 360 Sunday-Thursday from 7-9pm
Additional office hours Thursday from 5-7pm
Course meetings

**Lectures:** Tuesday and Thursday 10:10-11am

**Recitations:** Monday and Wednesday, 2:30 and 3:30
- A third section will be added, at a time that helps out students with conflicts (probably in the evening)
- We’ll pick the time at the end of class today

New material in lecture *and* recitation
- You are expected to attend both

Class participation counts
- Please stick to the same section
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

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

CMS

- “Course Management System”
- Built by Andrew Myers (with help from lots of students)
- Assignments and grades posted here
Coursework

6 problem sets
  • Due Thursdays at 11:59pm
  • PS #1 (out today) is due Thursday 2/2
  • Electronic submission via CMS

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

6 (small) quizzes in lecture

2 preliminary exams and a final
Breakdown:
- 45% - Problem sets
- 5% - Quizzes (lowest dropped)
- 30% - Preliminary exams (lower exam weighted less)
- 20% - Final exam

Will follow the usual CS3110 curve
- Centered around a B/B+
Late policy

You can hand it in until we start grading
  • 15% penalty / day
  • After we start grading, no credit

Save your code and submit early and often
  • CMS is your friend
  • Be certain you have submitted something, even if it isn’t perfect and you are improving it

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

Strictly enforced

Easier to check than you might think
  • We compare submissions using automated tools

Unpleasant and painful for everyone involved

To avoid pressure, start early
  • We try hard to encourage this
  • Take advantage of the large veteran staff
  • Let Nate know if you run into difficulty
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!
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!
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!
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
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 easy to reason about
Awesome OCaml feature: many common errors simply impossible
- More precisely, they are caught at compile time
- Early failure is very important (why?)

Functional language
- Programs have a clear semantics
- Heavy use of recursion
- Lots of higher-order functions
- Few side effects

Statically typed and type safe
- Many bugs caught by compiler
Imperative Programming

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**

- \( \text{int wheels(Vehicle v) \{ v.size++; return v.numw; \}} \)
Functional Style

Idea: program without side effects

- Effect of a function is *only* to return a result value

Program is an **expression** that can be **evaluated** to produce a **value**

- For example, evaluating 2+2 yields 4
- Just like mathematical expressions

Enables **equational reasoning** about programs:

- if x equals y, replacing y with x has no effect:
- let x=f(0) in x+x equivalent to f(0)+f(0)
Functional Style

Bind variables to values, don’t mutate existing variables

No concept of \( x = x + 1 \) or \( x++ \)

These do nothing remotely like \( x++ \)

\[
\text{let } x = x + 1 \text{ in } x \\
\text{let rec } x = x + 1 \text{ in } x
\]

The former assumes an existing binding for \( x \) and creates a new one (no modification of \( x \))

The latter is an 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:** 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**
**Functional** programming languages

- Encourages building code out of functions
- \( f(x) \) always gives same result
- No side effects: easier to reason about what happens
- Better fit to modern hardware, distributed systems

Functional style usable in Java, C, Python…

- Becoming more important with interactive UI’s and multiple cores
- Provides a form of encapsulation – hide the state and side effects inside a functional abstraction
Programming Languages Map

- Functional: Lisp, Scheme, Haskell, OCaml, SML
- Imperative: Fortran, C, Perl, Matlab, Pascal, C++, Java
- Object-Oriented: JavaScript
- ML family: ML
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

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

```ml
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 then inferred.

For example, in following, typechecker determines that \( n \) is an integer, and \texttt{sumsq} returns an integer

```plaintext
let rec sumsq n =
  if n=0 then 0
  else n*n + sumsq(n-1)
```
Example 1a: Sum f’s

Functions are first-class objects

Can be used as arguments and returned as values

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

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

```java
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

```
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 OCaml?

Objective Caml is one of the most robust and general functional languages available
  • Used in financial industry
  • Lightweight and good for rapid prototyping

Embodies important ideas better than Java, C++
  • Many of these ideas work in Java, C++, and you should use them…

Learning a different language paradigms will make you a more flexible programmer down the road
  • Likely that Java and C++ will be replaced
  • Principles and concepts beat syntax
  • Ideas in ML will likely be in next-generation languages
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