CS 312 Lecture 1
Course overview

Dan Huttenlocher
Cornell University Computer Science
Spring 2009
Course staff

- Prof. Dan Huttenlocher

- Grad TAs:
  - Jean-Baptiste Jeannin
  - Ed McTigue

- UG Consultants:
  - Andrew Owens
  - Tanya Gupta
  - Rick Ducott
  - Dane Wallinga
  - David Kupiec
  - Matt Pokryzwa
  - Jerzy Hausknecht
  - Jacob Bank
  - Nyk Lotocky

- Office, consulting hours posted on web
- Consulting Sun, Tue, Wed, Thu evenings
- Use TA, instructor office hours!
What this course is about

Helping you become expert designers and developers of valuable software systems.

1) Programming paradigms
   New programming language concepts and constructs

2) Reasoning about programs
   • Correctness
   • Performance
   • Designing for change and reuse

3) Tools
   Data structures and algorithms
Course meetings

- Lectures Tues, Thurs: Phillips 219
- Recitations Monday, Wednesday
  - Upson 109, at 2:30pm
  - Upson 109, at 3:35pm
  - Possible third section – early evening?

- New material is presented in lecture and recitation
- Attendance is expected at lecture and recitation
- Participation counts
Course web site

http://www.cs.cornell.edu/courses/cs3110

- Announcements
- Lecture notes
- Assignments
- Course software
- OCaml documentation
- Other resources
Course newsgroup

cornell.class.cs312

- A great place to ask questions!
- A great place to see if your question has already been asked
- A place to discuss course ideas
  - But don’t solve assignments for other people
Readings

- Course material in lecture notes on website
  - But you are also responsible for in-class material...

- Some other useful texts:
  - *The Objective Caml System*, Leroy et al. (online)
  - *Introduction to Objective Caml*, Hickey. (online)
Assignments

- 6 problem sets – generally due Thursdays
  - PS1 released Thursday, due Jan. 29: “OCaml Warmup”

- Mix of programming, written problems

- Submitted electronically via CMS
  - 24 hours late 10% penalty, 48 hours late 20% penalty

- Four single-person assignments (1–4)
  - Two weeks each

- Two two-person assignments (5–6)
  - Three weeks each
Exams

- Exams test material from lectures, problem sets, assume you have done assignments

- Prelim 1: March 5, 7:30 PM
- Prelim 2: April 14, 7:30 PM

- Final exam May 8, 2:00 PM

- Any makeup exams must be scheduled within the first two weeks of class
  - Check your schedule and let me know!
Academic integrity

- Strictly and carefully enforced
  - Please don’t make us waste time on this
  - Automated tools readily reveal code similarity even if you try to hide it by changing names, spacing, etc.
  - Written problems also often surprisingly easy to see

- Start assignments early and get help from course staff!
  - Use the generous late policy if you can’t make the deadline
What this course is about

Helping you become expert designers and developers of valuable software systems.

<table>
<thead>
<tr>
<th>1) Programming paradigms</th>
<th>2) Reasoning about programs</th>
<th>3) Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>New programming language concepts and constructs</td>
<td>• Correctness</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Designing for reasoning and reuse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data structures and algorithms</td>
<td></td>
</tr>
</tbody>
</table>
Why do you need this class?

- Skills and ideas that will help you become better software designers and implementers
  - 10x difference in productivity, fun, …

- A course about creating good software – just because you write it doesn’t make it good
  - Correct
  - Fast
  - Maintainable
  - Reusable

- Needed in many upper level courses
- Needed for serious programming tasks
- Needed for managing programming projects
Programming any big system
1) Programming Paradigms

- Functional programming
- Polymorphism
- Pattern matching
- Modular programming – beyond OO/classes
- Concurrent programming – multi-core and UI
- Types and type inference
- Managed memory (garbage collection)

We’ll use ML to convey these concepts

- The important part are the concepts, not the ML syntax!
2) Programming Techniques

- *Design and reasoning: critical to robust, trustworthy software systems.*

- **Design and planning:**
  - Modular programming
  - Data abstraction
  - Specifications, interfaces, data structures (the curse of bad ones)
    - Interfaces as treaties

- **Reasoning about programs**
  - Program execution models
  - Reasoning about program correctness
  - Reasoning about performance via asymptotic complexity
  - Using induction to reason about program behavior

- **Testing**
3) Data Structures & Algorithms

- Standard structures: lists, trees, stacks, graphs, etc.
  - Functional versions of these structures

- More advanced structures:
  - Balanced trees: AVL, Red-Black, B-trees, splay trees
  - Disjoint sets
  - Hash tables
  - Binary heaps

- Algorithms on these data structures
  - Analysis of correctness and performance
Imperative style

- 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 \( x++ \)

- Neither of these does anything remotely like \( x++ \)
  
  \[
  \begin{align*}
  \text{let } x &= x+1 \text{ in } x \\
  \text{let rec } x &= x+1 \text{ in } x
  \end{align*}
  \]

- 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 and it’s only getting worse.
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
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

\[ y = 0; \]
\[ \text{for (x = 1; x \leq n; x++) { } } \]
\[ \quad y = y + x^2; \]
\[ \]
Example 1: Sum Squares

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

```ocaml
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

\[
\text{let rec sumsq } n = \\
\quad \text{if } n=0 \text{ then } 0  \\
\quad \text{else } n*n + \text{sumsq}(n-1)
\]
Example 1a: Sum f’s

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

\[
\text{let rec sumop f n =}
\begin{align*}
\text{if } n=0 & \text{ then 0} \\
\text{else } f\ n + \text{sumop } f\ (n-1)
\end{align*}
\]

\[
\text{sumop (function } x \rightarrow x^3)\ 5
\]
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

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)
- Spring break
- Data structures and analysis of algorithms (5)
- Prelim 2
- Topics: memoization, streams, managed memory (5)
- Final exam
Announcements

- Problem set 1 released Thursday
  - Due January 29 at 11:59pm
  - Look on the course web site and CMS

- Consulting starts on Sunday

- **Help sessions:** getting started with OCaml+Emacs
  - This Thursday 22 & Sunday 25, 7pm, Upson B7 lab

- CMS access will be set up today. Send mail to cs312-l@cs.cornell.edu if not.