Programming Languages and Techniques (CIS120e)

Lecture 1
Sep 8, 2010

Course Overview and Logistics
Introduction to Program Design
What’s “e” For?

An “experimental” track of CIS120

- ... where we’ll try out some new ideas about how to organize and present the material
  - taught by Pierce and Zdancewic
- ... in parallel with a “120 classic” track
  - taught by Marcus
Course Philosophy

• Programming
  – ... is *fun, useful*, and *rewarding* in its own right
  – ... is also a conceptual foundation for all of computer science
  – ... involves tools (languages, libraries, IDEs, etc.) that are often large and complex, for good reasons!
  – ... takes lots of practice to master

• Teaching introductory computer science
  – Start with basic skills of “algorithmic thinking” (AP/110)
  – Develop more systematic design and analysis skills in the context of larger and more challenging problems (120)
  – Come to grips with industrial-strength tools and design processes, e.g. Java (120, 121, and beyond)

• About CIS120e
  – Teach Java *after* setting up the context needed to understand why Java and OO programming are good tools
  – Concentrate on the process of *designing* programs
  – Give a taste of the breadth and depth of CS
CIS120 vs. CIS120e

• Same fundamental concepts, slightly different emphasis:
  – 120: greater focus on Java details; first part of course in Java; last part in Python
  – 120e: greater focus on program design skills; first part of course in OCaml; last part in Java
• Equally good preparation for 121 and later courses
• Same prerequisites: programming experience equivalent to AP computer science or CIS110
• Similar difficulty level and grading standards
• No distinction on transcript
  – The “e” in 120e is just an informal designation; formally, this is just section 002 of CIS120
Switching Tracks

If you are currently signed up for the 120e track but prefer to switch to “120 classic,” please speak to one of the instructors.

Questions?
Introductions

• Instructors:
  – Benjamin Pierce
  – Steve Zdancewic

• TAs:
  – Lu Chen
  – Michael Greenberg
  – Ryan Menezes

• ... you?
Administrivia

• Web site: [www.cis.upenn.edu/~cis120e](http://www.cis.upenn.edu/~cis120e)
  – This is the place to go for up-to-date information on office hours, etc.

• Lab section:
  – Thursdays 4:00-5:00pm
  – Room: Moore 100A (for this week, at least)

• Course textbook: None required

• Mailing list: [CIS120e-Fall10@lists.seas.upenn.edu](mailto:CIS120e-Fall10@lists.seas.upenn.edu)
  – You will receive a notification when you’ve been subscribed to this list
  – We’ll use it for announcements. You should also feel free to use it for discussions of general interest (e.g., questions about lectures, clarifications of homework problems).
  – If you are not currently registered for Lab Section 206, please email bcpierce@cis so we know to add you
Grades

• 20% Homework
  – 10 programming projects
• 10% Labs
  – Including participation
• 20% First midterm
• 20% Second midterm
• 30% Final exam
Homework Policies

• Projects will be (mostly) automatically graded
  – We’ll give you some tests, as part of the assignment
  – You’ll write your own tests to supplement these
  – Our grading script will apply additional tests
  – Your score is based on how many of these you pass
  – Your code must compile to get any credit

• You will be given your score (on the automatically graded portion of the assignment) immediately

• Multiple submissions are allowed
  – First \( n \) submissions: no penalty
  – Each submission after the first \( n \) costs \( k \) points (out of 100)
  – For the first assignment, \( n = 7 \) and \( k = 1 \)

• Late submissions
  – 10% penalty if less than 24 hours late
  – 20% penalty if 24-48 hours late
  – Submissions not accepted after 48 hours past the deadline
Academic Integrity

• All submitted homework must be individual work
  
  Not OK:
  - Copying / sharing of code
  - Discussions of specific homework problems with other students

  OK / encouraged:
  - “High level” discussions of concepts from lecture

• Course staff will check for copying. Violations will be treated seriously.

• If in doubt, ask us!

Penn’s code of academic integrity:
http://www.vpul.upenn.edu/osl/acadint.html
Prerequisites

• We assume you can already write 10-100-line programs in some imperative or OO language
  – Not necessarily Java or any other particular language
Course Goals

• Ability to write larger (~1000 lines) programs
  – increased independence ("working without a recipe")

• Fluency in program design
  – test-driven programming
  – modular decomposition
  – working with different programming idioms
    • declarative / collection-oriented / stateful / object-oriented / event-driven, etc.

• Firm grasp of fundamental principles of OO programming

• Fluency in core Java
Tools

• OCaml
  – Well-known functional programming language
  – Mostly used in symbol processing applications (compilers, theorem provers) and the financial industry
  – Lightweight, approachable setting for learning about program design
  – Levels the playing field at the start of the course

• Java
  – Industry standard OO language / development platform

• Eclipse
  – Popular open-source integrated development environment (IDE)

• We’ll get these set up in the Lab this week
  – See course web pages for additional information
Homework 1

• Available by this evening
  – we’ll let you know by email
• Due: 15 Sept. 2010 at 11:59:59pm
• See course web pages for details
Design is the process of translating informal specifications (“word problems”) into running code.

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each
Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($0.10) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner $180. Each attendee costs another four cents ($0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.
(Interactive Interlude)
Essential OCaml Cheatsheet
OCaml’s built-in primitive types include...

- **int**
  
  0, 1, 42, -1, 999

- **string**
  
  “hello world”

- **char**
  
  ‘h’

- **bool**
  
  true, false
Expressions

Numeric expressions:

1 + 2    addition
1 – 2    subtraction
2 * 3    multiplication
10 / 3   integer division
10 mod 3 modulus (remainder)

Standard rules of operator precedence:

1 + (2 * 3) = 7
1 + 2 * 3 = 7
(1 + 2) * 3 = 9
Expressions

Comparisons:

=  equality
<>  inequality
<  less than
>=  greater than or equal

Boolean (logical) operators:

not  logical negation
&&  and
||  or

(these can be used with any type of data – numbers, strings, characters, etc.)
let total_secs (hours:int), (minutes:int), (seconds:int) : int =
(hours * 60 + minutes) * 60 + seconds
Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments.

```
total_secs 5 30 22
```

(Note that the list of arguments is not parenthesized.)
Conditional Expressions

if s = “positive” then 1 else -1
if day >= 6 && day <= 7 then “weekend” else “weekday”

OCaml conditionals are expressions: they can be used inside of other expressions:

(if x > 0 then x else –x) * 100

if x > y then “x is bigger”
else if x < y then “y is bigger”
else “same”
A variable declaration gives a name to the result of some expression.

```
let pi = 3.14159
let seconds_per_day = 60 * 60 * 24
```

Variable declarations can appear both at top-level and within other expressions.

```
let f (x:int) : int =
  let y = x * 2000 in
  y * y
```

Variable declarations within other expressions are followed by “in”. Top-level variable declarations are not.
An OCaml program is a set of *modules*. Each module lives in its own file. A module named `Foo` lives in a file named `foo.ml`.

Each module consists of a sequence of *declarations*. A declaration can be either a *variable declaration* or a *function declaration*.

```ocaml
let x = 100
let s = "hello world"
let f (k:int):int = k * 5 + x
```
If you are writing a module M and you need to refer to a name x from another module N, either prefix it with the module name and a dot (N.x) or else write “open N” at the top of m.foo and then write just x to refer to N.x.
Tests

By convention, tests are placed in a separate module: the tests for a module named `Foo` will live in a module named `FooTests` (in a file named `fooTests.ml`).

Tests are just calls to the `assert_eq` function:

```ml
assert_eq "Attendees at $5.00" 120 (attendees 500);
```

(Note that tests are followed by a semicolon.)