Programming Languages and Techniques (CIS120)

Lecture 2

January 17, 2013

Value-Oriented Programming

If you are joining us today…

- See Wed's slides/screencast on course website
- Read the course syllabus/lecture notes on the website
- Sign yourself up for Piazza http://www.piazza.com/
- Install OCaml/Eclipse on your laptop; ask if you have questions http://www.seas.upenn.edu/~cis120/current/ocaml_setup.shtml
- No laptops, tablets, smart phones, etc., during lecture

Registration

- If you are not registered, add your name to the waiting list
  - There are spots available, but we want to make sure that those who need to take the course this semester can register first
- Need a different recitation?
  - If the want you want is open, switch online
  - Recitations 204, 208 and 209 have the most space
  - If you need to attend a closed recitation, add your name to the recitation change request form
  - Go to the recitation you want, even if not registered

Announcements

- Please *read:*
  - Chapter 2 of the course notes
  - OCaml style guide on the course website (http://www.seas.upenn.edu/~cis120/current/programming_style.shtml)
- Homework 1: OCaml Finger Exercises
  - Practice using OCaml to write simple programs
  - Start with first 4 problems (lists next week!)
  - Due: Tuesday, January 28th at 11:59:59pm (midnight)
  - Start early!
- Guest lecturer (Peter-Michael Osera) next week
  - Prof. Weirich OH Monday, then away Tuesday-Saturday

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
  - Some assignments will also include style points, added later
  - 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 *few* submissions: no penalty
  - Each submission after the first few will be penalized
  - Your final grade is determined by the *best* raw score
- Late submissions
  - 10 point penalty if less than 24 hours late
  - 20 point penalty if 24-48 hours late
  - *Submissions not accepted after 48 hours past the deadline*

Recitations / Lab Sections

- First recitations start Wednesday and Thursday
  - Bring your laptops
  - Install tools (OCaml, eclipse) on your laptop *before* recitation next week
  - http://www.seas.upenn.edu/~cis120/current/ocaml_setup.shtml
- Goals of first meeting:
  - Meet your TAs and classmates
  - Debug tool (OCaml, eclipse) installation problems
  - Practice with OCaml before your first homework is due
  - If you are eager to get started, first lab material already available

Important Dates

- Homework:
  - Homework due dates listed on course calendar
  - Mostly Tuesdays, some Fridays
- Exams:
  - 12% First midterm: Friday, February 21st, in class
  - 12% Second midterm: Friday, April 4th, in class
  - 18% Final exam: Wednesday, May 7th, 9-11 AM
  - Contact me *well in advance* if you have a conflict

Where to ask questions

- Course material
  - **Piazza Discussion Boards**
  - TA office hours, on webpage calendar
  - Tutoring, Sunday and Monday evenings
  - Prof office hours: Mondays from 1 to 3 PM, or by appointment (changes will be announced on Piazza)
- HW/Exam Grading: see webpage
- About the CIS majors
  - Ms. Jackie Caliman, CIS Undergraduate coordinator

Clickers

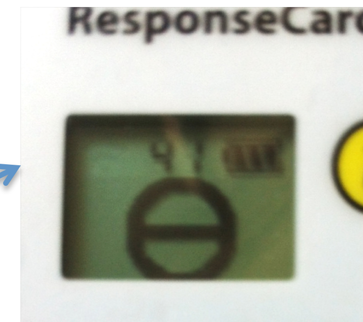
Clicker Basics

- Beginning today, we'll use clickers in each lecture
  - Grade recording starts 1/28
- Any kind of TurningPoint ResponseCard is fine
  - Doesn't have to be the exact model sold in the bookstore
- Use the link on the course website to register your device ID with the course database

6-character device ID

Test Drive

- Clickers out!
- Press any of the number buttons
  - Make sure the display looks like this:



- If it looks like this…

  - … first check that the channel is set to 41
    - If not, try pressing Channel, then 41, then Channel again to reset the channel
  - If this doesn't work come to office hours

Have you successfully installed OCaml on your laptop?

```
1) Yes
2) No
```

In what language do you have the most significant programming experience?

```
1) Java C#
2) C, C#, C++ or Objective-C
3) Python, Ruby, or MATLAB
4) Clojure, Scheme, or LISP
5) OCaML, Haskell, or Scala
6) Other
```

Programming in OCaml

Read Chapter 2 of the CIS 120 lecture notes, available from the course web page

Course goal

Strive for beautiful code.

- Beautiful code
  - is *simple*
  - is easy to understand
  - is likely to be correct
  - is easy to maintain
  - takes skill to develop

Value-Oriented Programming

- Java, C, C#, C++, Python, Perl, etc. are tuned for an **imperative** programming style
  - Programs are full of *commands*
    - *"Change x to 5!"*
    - *"Increment z!"*
    - *"Make this point to that!"*
- Ocaml, on the other hand, promotes a **value-oriented** style
  - We've seen that there are a few commands…
    - e.g., `print_endline, run_test`

    … but these are used rarely
  - Most of what we write is *expressions* denoting *values*

Metaphorically, we might say that

imperative programming is about **doing**

while

value-oriented programming is about **being**

Simplification vs. Execution

- We can think of an OCaml expression as just a way of writing down a *value*
- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to this value
- By contrast, a running Java program performs a sequence of *actions* or *events*
  - *… a variable named x gets created*
  - *… then we put the value 3 in x*
  - *… then we test whether y is greater than z*
  - *… the answer is true, so we put the value 4 in x*
  - *… etc.*

What is an OCaml module?

```
open Assert

let attendees (price:int) :int =
  (-15 * price) / 10 + 870

let test () : bool =
   attendees 500 = 120
;; run_test "attendees at 5.00" test

let x = attendees 500

;; print_int x
;; print_endline "end of demo"
```

module import

function declarations

let declarations

commands

(Top-level) Let Declarations

A let declaration gives a *name* (a.k.a. an *identifier*) to the value of some expression*

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

There is no way of *assigning* a new value to an identifier after it is declared.

*We sometimes call these identifiers *variables*, but the terminology is a bit confusing because in languages like Java and C a variable is something that can be modified over the course of a program. In OCaml, like in mathematics, once a variable's value is determined, it can never be modified… As a reminder of this difference, for the purposes of OCaml we'll try to use the word "identifier" when talking about the name bound by a let.

Programming with Values

- Programming without mutable variables requires a shift of perspective that can be challenging at first!

- But, in the end, it leads to code that is simpler to understand

(Top-level) Function Declarations

function name parameter names parameter types

```
let total_secs (hours:int)
               (minutes:int)
               (seconds:int)
             : int =
   (hours * 60 + minutes) * 60 + seconds
```

result type

function body (an expression)

Commands

```
;; run_test "Attendees at $5.00" test

;; print_endline "Attendees at $5.00"
;; print_int (attendees 500)
```

- Top-level commands run tests and print to the console
- Such commands are the *only* places that semicolons should appear in your programs (so far)

What does an OCaml program do?

```
open Assert

let attendees (price:int) :int =
  (-15 * price) / 10 + 870

let test () : bool =
  attendees 500 = 120
;; run_test "attendees at 5.00" test

let x = attendees 500

;; print_int x
```

To know if the test will pass, we need to know whether this expression is true or false

To know what will be printed we need to know the value of this expression

To know what an OCaml program will do, you need to know what the value of each expression is.

Calculating Expression Values

Calculating with Expressions

OCaml programs mostly consist of *expressions*.

We understand programs by *simplifying* expressions to values:

```
3 ⇒ 3
3 + 4 ⇒ 7
2 * (4 + 5) ⇒ 18
attendees 500 ⇒ 120
```

(values compute to themselves)

The notation <exp> ⇒ <val> means that the expression <exp> computes to the value <val>.

Note that the symbol '⇒' is *not* OCaml syntax. It's a convenient way to *talk* about the way OCaml programs behave.

Step-wise Calculation

- We can understand $\Rightarrow$ in terms of single step calculations written '$\longmapsto$'
- For example:

```
 (2+3) * (5-2)
⟼ 5 * (5-2)        because 2+3 ⟼ 5
⟼ 5 * 3            because 5-2 ⟼ 3
⟼ 15               because 5*3 ⟼ 15
```

- *Every* form of expression can be simplified with $\longmapsto$

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 3 > 0 then 2 else -1) * 100
```

```
if x > y then "x is bigger"
else if x < y then "y is bigger"
else "same"
```

Simplifying Conditional Expressions

- A conditional expression yields the value of either its ‘then’-expression or its ‘else’-expression, depending on whether the test is ‘true’ or ‘false’.
- For example:

```
    (if 3 > 0 then 2 else -1) * 100
⟼  (if true then 2 else -1) * 100
⟼  2 * 100
⟼  200
```

- It doesn’t make sense to leave out the ‘else’ branch in an ‘if’. (What would be the result if the test was ‘false’?)

Function Calls

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

```
total_secs 5 30 22
```

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

Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the functions.

```
   total_secs (2 + 3) 12 17
⟼ total_secs 5 12 17
⟼ (5 * 60 + 12) * 60 + 17    subst. the args
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

```
let total_secs (hours:int)
               (minutes:int)
               (seconds:int)
             : int =
(hours * 60 + minutes) * 60 + seconds
```

Local Let Declarations

Let declarations can appear both at top-level and *nested* within other expressions.

```
let f (x:int) : int =
  let y = x * 10 in
  y * y

let test () : bool =
   (f 3) = 900
;; run_test "test f" test
```

scope of x is the body of f

scope of y is nested within the body of f

scope of f is the rest of the program

Nested let declarations are followed by "in"

Top-level let declarations are not

- Every local 'let...in...' is an *expression*
- So these are legal OCaml expressions:

```
(let x = 1 in x + x) * 2
```

```
if 1 > 0 then (let x = 1 in x + x) else 3
```

```
let x = (let y = 1 in y + 0) in x+x
```

```
if (let x = 0 in x < 1) then "foo" else "bar"
```

Scope

Multiple declarations of the same variable or function name are allowed. The later declaration *shadows* the earlier one for the rest of the program.

```
let x = 1 in
let y = x + 1 in
let x = 1000 in
let z = x + 2 in
x + y + z
```

scope of x

scope of y

scope of x (shadows earlier x)

scope of z

Summary

- To read:
  - Chapter 2 of lecture notes
  - OCaml style guide (http://www.seas.upenn.edu/~cis120/current/programming_style.shtml)
- To do:
  - Look at lab material in preparation for recitation
  - Start first four problems of HW 1