CSE341: Programming Languages

Lecture 1
Course Mechanics
ML Variable Bindings

Dan Grossman

Fall 2011

Welcome!

We have 10 weeks to learn *the fundamental concepts* of programming languages

With hard work, patience, and an open mind, this course makes you a much better programmer

- Even in languages we won't use
- Learn the core ideas around which *every* language is built, despite countless surface-level differences and variations
- *Poor* course summary: "Uses SML, Racket, and Ruby"

Today's class:

- Course mechanics
- *[A rain-check on motivation]*
- Dive into ML: Homework 1 due end of next week

Concise to-do list

In the next 24-48 hours:

1. Read course web page:
   http://www.cs.washington.edu/education/courses/cse341/11au/
2. Read all course policies
3. Adjust class email-list settings as necessary
4. Complete Homework 0 (survey worth 0 points)

5. Get set up using emacs and SML
   – Installation/configuration/use instructions on web page
   – Essential; no reason to delay

Who

Course Staff:

Dan Grossman Ben Wood Lydia Duncan Chloe Houvener

Dan: Faculty, 341 my favorite course / area of expertise

Ben: Ph.D. student, also a PL expert

Lydia: Took 341 Spring 2011, helped with new project over summer

Chloe: Took 341 Fall 2010, favorite class so far

A nice large staff: *get to know us!*

Staying in touch

- Course email list: **`cse341a_au11@u.washington.edu`**
  - Students and staff already subscribed
  - You must get announcements sent there
  - Fairly low traffic
- Course staff: **`cse341-staff@cs.washington.edu`**
  plus individual emails
- Message Board
  - For appropriate discussions; TAs will monitor
  - Optional, won't use for important announcements
- Anonymous feedback link on webpage
  - For good and bad: If you don't tell me, I don't know

Lecture: Dan

- Slides, code, and essay summaries posted
  - Often revised after class
    - Summary might not be posted until after class
  - *Take notes*: materials may not describe everything
    - Slides in particular are *visual aids* for me to use
- Ask questions, focus on key ideas
- Engage actively
  - Arrive *punctually* (beginning matters most!) and well-rested
    - Just like you will for the midterm!
  - *Write* down ideas and code as we go
  - If attending and paying attention is a poor use of your time, one of us is doing something wrong

Section: Ben

- Required: will usually cover new material
- Sometimes more language or environment details
- Sometimes main ideas needed for homework
- *Will* meet this week: using SML

Office hours

- Regular hours and locations on course web [soon]
  - Changes as necessary announced on email list

- Use them
  - *Please visit me*
  - Ideally not *just* for homework questions (but that's good too)

Textbooks, or lack thereof

- Will mostly use the “textbooks” as useful references
  - Look up details you want/need to know
- Can provide good second explanations, but (because!) they often take a fairly different approach
- Some topics aren’t in the texts
- Don’t be surprised when I essentially ignore the texts
  - List on web page what sections are most relevant
- *Some but not all of you will do fine without using the texts*

Homework

- Seven total
- To be done individually
- Doing the homework involves:
  1. Understanding the concepts being addressed
  2. Writing code demonstrating understanding of the concepts
  3. Testing your code to ensure you understand and have correct programs
  4. “Playing around” with variations, incorrect answers, etc.

  We grade only (2), but focusing on (2) makes the homework harder
- Challenge problems: Low points/difficulty ratio

Note my writing style

- Homeworks tend to be worded very precisely and concisely
  - I’m a computer scientist and I write like one (a good thing!)
  - Technical issues deserve precise technical writing
  - Conciseness values your time as a reader
  - You should try to be precise too

- *Skimming or not understanding why a word or phrase was chosen can make the homework harder*

- By all means ask if a problem is confusing
  - Being confused is normal and understandable
  - And I may have made a mistake
  - Once you’re unconfused, you might agree the problem wording didn’t cause the confusion

Academic Integrity

- Read the course policy carefully
  - Clearly explains how you can and cannot get/provide help on homework and projects

- Always explain any unconventional action

- I have promoted and enforced academic integrity since I was a freshman
  - Great trust with little sympathy for violations
  - Honest work is the most important feature of a university

Exams

- Midterm: Monday October 31 (spooky ☺), in class
- Final: Tuesday December 13, 2:30-4:20
- Same concepts, but different format from homework
  - More conceptual (but write code too)
  - Will post old exams
  - Closed book/notes, but you bring one sheet with whatever you want on it

Questions?

Anything I forgot about course mechanics before we discuss, you know, programming languages?

What this course is about

- Many essential concepts relevant in any programming language
  - And how these pieces fit together
- Use the languages ML, Racket, and Ruby because:
  - They let many of the concepts “shine”
  - Using multiple languages shows how the same concept can “look different” or actually be slightly different
  - In many ways simpler than Java
- A big focus on *functional programming*
  - Not using *mutation* (assignment statements) (!)
  - Using *first-class functions* (can’t explain that yet)

Let’s start over

- For at least the next two weeks, *please “let go of Java”*
  - Learn ML as a “totally new way of programming”
  - Later we’ll contrast with what you know
  - Saying “oh that is kind of like that thing in Java” will confuse you, slow you down, and make you learn less

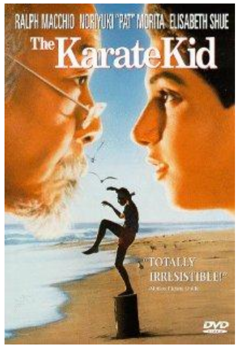
- In a few weeks, we’ll have the background to
  - Intelligently motivate the course
  - Understand how functional programming is often simple, powerful, and good style – even when “stuck” in Java or C
  - Understand why functional programming is increasingly important in the “real world” even if ML and Racket aren’t widely popular languages

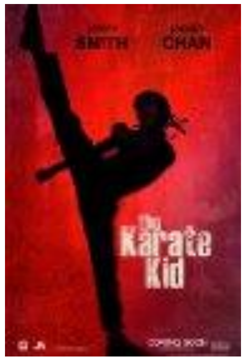
My claim

Learning to think about software in this "PL" way will make you a better programmer even if/when you go back to old ways

It will also give you the mental tools and experience you need for a lifetime of confidently picking up new languages and ideas

[Somewhat in the style of *The Karate Kid* movies (1984, 2010)]





A strange environment

- The next 4-5 weeks will use
  - The ML language
  - The emacs editor
  - A read-eval-print-loop (REPL) for evaluating programs
- *You* need to get things installed, configured, and usable
  - Either in the department labs or your own machine
  - We’ve written instructions (read carefully; ask questions)
- Only then can you focus on the content of Homework 1
- Working in strange environments is a CS life skill

ML from the beginning

- A program is a sequence of *bindings*
- *Type-check* each binding in order using the *static environment* produced by the previous bindings
- *Evaluate* each binding in order using the *dynamic environment* produced by the previous bindings
  - Dynamic environment holds *values*, the results of evaluating expressions
- Today the only kind of binding is a *variable binding*

A very simple ML program

```
(* My first ML program *)

val x = 34;

val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 - z else z;

val abs_of_z_simpler = abs z
```

A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

More generally:

```
val x = e;
```

- *Syntax*:
  - *Keyword* **`val`** and *punctuation* **=** and **`;`**
  - *Variable* ***`x`***
  - *Expression* ***`e`***
    - many forms of these, most containing subexpressions

Expressions

- We have seen many kinds of expressions:

  `34   true    false   x   `*`e1`*`+`*`e2`*`  `*`e1`*`<`*`e2`*

  `if `*`e1`*` then `*`e2`*` else `*`e3`*
- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.

- Every kind of expression has
  1. Syntax
  2. Type-checking rules
     - Produces a type or fails (with a bad error message ☹)
     - Types so far: **`int  bool  unit`**
  3. Evaluation rules (used only on things that type-check)
     - Produces a value (or exception or infinite-loop)

Variables

- Syntax:

  sequence of letters, digits, _, not starting with digit

- Type-checking:

  Look up type in current static environment

  – If not there fail

- Evaluation:

  Look up value in current dynamic environment

Addition

- Syntax:

  `e1 + e2` where ***`e1`*** and ***`e2`*** are expressions

- Type-checking:

  If ***`e1`*** and ***`e2`*** have type **`int`**,

  then ***`e1 + e2`*** has type **`int`**

- Evaluation:

  If ***`e1`*** evaluates to **`v1`** and ***`e2`*** evaluates to **`v2`**,

  then ***`e1 + e2`*** evaluates to sum of **`v1`** and **`v2`**

Values

- All values are expressions
- Not all expressions are values
- A value "evaluates to itself" in "zero steps"
- Examples:
  - **`34`**, **`17`**, **`42`** have type **`int`**
  - **`true`**, **`false`** have type **`bool`**
  - **`()`** has type **`unit`**

A slightly tougher one

What are the syntax, typing rules, and evaluation rules for conditional expressions?

use

`use "foo.sml"` is an unusual expression

It enters bindings from the file **`foo.sml`**

Result is **`()`** bound to variable **`it`**

- Ignorable

The foundation we need

We have many more types, expression forms, and binding forms to learn before we can write “anything interesting”

Syntax, typing rules, evaluation rules will guide us the whole way!

For homework 1: functions, pairs, conditionals, lists, options, and local bindings

- Earlier problems require less

Will not add (or need):

- Mutation (a.k.a. assignment): use new bindings instead
- Statements: everything is an expression
- Loops: use recursion instead

Pragmatics

Lecture has emphasized building up from simple pieces

But in practice you make mistakes and get inscrutable messages

Example gotcha: **`x = 7`** instead of **`val x = 7`**

Work on developing resilience to mistakes

- Slow down
- Don't panic
- Read what you wrote very carefully