CS 360: Programming Languages

Lecture 1
Course Mechanics
Scheme Variable Bindings

Phil Kirlin

Fall 2012

(*material adapted from Dan Grossman, U. Washington)*

Welcome!

We have 14 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: "We learned Scheme and Java"

Today's class:

- Course mechanics
- *[A rain-check on motivation]*
- Dive into Scheme

Concise to-do list

In the next 24-48 hours:

1. Find course web page: go to http://www.rhodes.edu/kirlin and click on our class.
2. Read all course policies
3. Sign up for Piazza

4. Get set up using DrRacket
   - Installation/configuration/use instructions on web page (soon)
   - Essential; no reason to delay

Staying in touch

- Piazza – a message board for college classes
  - For appropriate discussions and announcements
  - Use to get help – everyone can post and respond to questions. Can even post and respond anonymously!
  - Just don't post any code that gives away homework answers or parts of answers.

Office hours

- Regular hours and locations on syllabus
  - 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 "textbook" as a useful reference
  - Look up details you want/need to know
  - Lots of additional online resources we will use to cover Scheme and Java.
- Some topics aren't in the texts
- Don't be surprised when I essentially ignore the texts
- *Some but not all of you will do fine without using the texts*

Homework

- Roughly 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

Academic Integrity

- Rule of thumb – don't look at anyone else's code, correct or incorrect.
- You are to complete assignments individually.
- You may discuss assignments in general terms with other students including a discussion of how to approach a problem, but the code you write must be your own.
- You may get help when you are stuck, but this help should be limited and should never involve details of how to code a solution.
- You may not have another person (current student, former student, tutor, friend, anyone) \walk you through" how to solve an assignment.

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 Scheme and Java (possibly others) because:
  - They let many of the concepts "shine"
  - Using multiple languages shows how the same concept can "look different" or actually be slightly different
- 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, *"let go of C++ and Python"*
  - Learn Scheme 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 C++/Python" 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 using Python or C++
  - Understand why functional programming is increasingly important in the "real world" even if Scheme isn't a widely popular language in industry.

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

A strange environment

- The next few weeks will use
  - The Scheme language
  - The DrRacket editing environment
  - A read-eval-print-loop (REPL) for evaluating programs
- *You* need to get things installed, configured, and usable
  - On your own machine
  - Instructions are online (read carefully; ask questions)
- Working in strange environments is a CS life skill

Scheme from the beginning

- A program is a sequence of *definitions* and *expressions*
  - A definition defines a variable. (This is sometimes called a variable binding.)
  - An expression is something that can be evaluated (the result of the expression is computed [and sometimes printed]).
  - Expressions always evaluate to a *value* (definitions never do).
- Each definition or expression is evaluated in the order they appear, using the *environment* produced by any previous definitions and expressions.
  - Environment holds variables and their values (bindings).
  - A value is the result of evaluating an expression.

A very simple Scheme program

```
; My first Scheme program

(define x 3)

(define y 7)

(define z (- x y))

(define q (* (+ x 2) (- z 3)))

(define abs-of-z (if (< z 0) (- z) z))

(define abs-of-z-simpler (abs z))
```


A variable definition

```
(define q (* (+ x 2) (- z 3)))
```

More generally:

```
(define x e)
```

- *Syntax*:
  - *Keyword* **`define`**
  - *Variable* **`x`**
  - *Expression* **`e`**
    - many forms of these, most containing subexpressions

Expressions

- We have seen many kinds of expressions:

  `3` **`4`** **`#f`** **`#t`** **`x`** ***`(+ e1 e2)`*** ***`(* e1 e2)`***

  `(if test e1 e2)`
- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.
- Every kind of expression has
  1. Syntax
  2. Type checking (implicitly) at run-time, not compile-time
     - Variables do have types (e.g., integer, float, boolean)
     - Any type mismatches aren't detected until you try to do something illegal. (like Python; not like C++)
  3. Evaluation rules
     - Produces a value (or error message or infinite loop)

Variables

- Syntax:

  sequence of letters, digits, hyphen, underscore, not starting with digit
- Evaluation:

  Look up value in current environment

Addition

- Syntax:

  `(+ e1 e2)` where ***`e1`*** and ***`e2`*** are expressions
- Type-checking:

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

  then ***`(+ e1 e2)`*** has type **`integer`**

  `(`also **`rational, real, complex)`**
- 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 **`integer`**
  - **`#t`**, **`#f`** have type **`boolean`**

A slightly tougher one

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

The foundation we need

We have many more expressions to learn before we can write "anything interesting"

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

Preview of what's to come:

- Mutation (a.k.a. assignment): use new variables instead
- Statements: everything is an expression (well, except definitions)
- 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: `(define x = 7)` instead of `(define x 7)`

Work on developing resilience to mistakes

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