Lecture 1 — August 3, 2015

- Call me Mac. I'm your lecturer for this course.
  - Research Area: Software Engineering; Specialty: Refactoring
  - Office hours as-posted on the course page, and by appointment
- Previously:
  - Assistant Professor at the University of Texas at San Antonio
  - Affiliate Assistant Professor with the University of Washington, Bothell
  - Sun Microsystems* (2000–2003) — *(*the makers of Java, now owned by Oracle)* I worked on a parallel debugger called Prism
  - Basho Technologies (2013) — I worked on a NoSQL database
  - Google Inc (2014) — I used Javascript for front-end programming, and Java for back-end services
  - Independent contractor (various times) — I once worked on porting Microsoft Word/Office 98 from Windows to the Mac OS (PowerPC days)

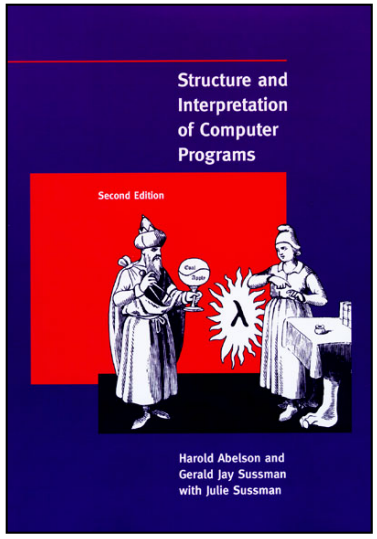
Why Study Programming Languages?

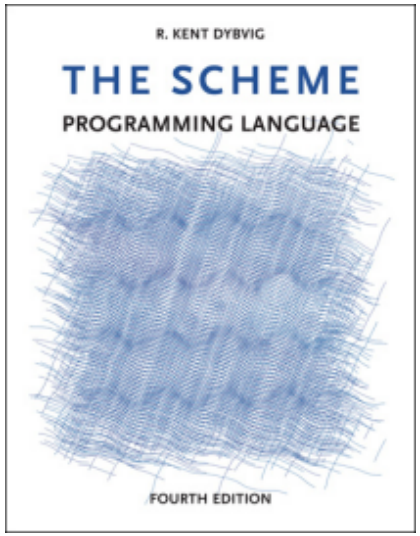
- *"Language shapes the way we think, and determines what we can think about." —Benjamin Whorf*
- *"Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought." —Kenneth E. Iverson*

- Hypothesis: ***Programming language*** shapes ***programming thought***
- Characteristics of a language affect how ideas can be expressed in the language

CSE130 Summer Session II, 2015

- Today:
  - Course Overview
  - Specifying Syntax





- Resources:
  - Text: *Structure and Interpretation of Computer Programs,* by Abelson, Sussman, and Sussman, Second Edition. A classic! Full text: https://mitpress.mit.edu/sicp/full-text/book/book.html
  - Videos: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/video-lectures/
  - Reference: *The Scheme Programming Language,* Fourth Edition, by R. Kent Dybvig. Full text: http://www.scheme.com/tspl4/

Course Goals

"Free your mind" —Morpheus

- You will learn
  - several new languages and constructs
  - ways to describe and organize computation
  - understand the limits of computation
- Will learn the anatomy of a PL
- Fundamental building blocks of languages reappear in different guises in different languages and different settings
- Buried inside any extensible system is a programming language
  - Emacs: E-Lisp; Word: WordBasic; Quake: QuakeC…
  - SQL, Renderman, LaTeX…

Primary languages in this course:

- Scheme
- Java 8
- JavaScript (Node.js/ Harmony)
- Prolog and prolog-like languages
- Lambda Calculus

CSE130: Programming Languages

"But I can only show you the door. You're the one that has to walk through it." —Morpheus

- Our site: **http://cse130.instructures.org**
- Evaluation:
  - 10% — Written Assessments; 4 total, take home, prep for the final
  - 30% — Programming Exercises
  - 30% — Interpreter Project
  - 30% — Final

- First reading:
  - Read Sections 1.1 and 1.2 of SICP (pages 1–56)

Academic Integrity

- All work must be your own
- If you are in doubt about what constitutes plagiarism, please contact me
- If you are aware of academic misconduct, you can either report it to me or the academic integrity office:
  - https://students.ucsd.edu/academics/academic-integrity/reporting.html
  - There is also an anonymous Whistleblower Hotline: (877) 319-0265
- Because you will be given access to the autograder tool, be mindful to use this access responsibly
  - Echoing the expected outputs without property analyzing the inputs may make a test “PASS” the autograder, but that is gaming the system
  - Any submissions that falsify results like this will get a zero
- All submissions will be inspected by the course staff and automated tools to detect plagiarism and gaming-the-system

Syntax vs. Semantics

```
c = a + b;
```

Specifying Syntax

- Grammars are used to precisely specify the syntax of a language.
  - A ***grammar*** is a set of rules that describe a language's syntax
  - ***Tokens*** (or ***terminals***) are the "alphabet" grammars are made from
  - ***Non-terminals*** are named entities in the grammar, one step above tokens
- ***Rules*** relate non-terminals to a sequence of tokens and non-terminals:

$$\textit{non-terminal} \rightarrow \text{[sequence of tokens, non-terminals]}$$

  - Where "→" can be pronounced as "can have the form of"
  - Sequence can possibly be empty, in which case an "ε" is used as a placeholder

Syntax Example: Java Identifiers

JavaIdentifier → *Letter*

JavaIdentifier → *JavaIdentifier* *LetterOrDigit*

Letter → `_` | `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`

LetterOrDigit → *Letter* | *Digit*

Digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

- Things to note:
  - **Input alphabet** is ASCII
  - A non-terminal can be defined by multiple rules
  - Rules can be "recursive"
  - "A → B | C" is shorthand to mean "A → B" and "A → C"

Syntax Example: Java Identifiers

JavaIdentifier → *Letter*

JavaIdentifier → *JavaIdentifier* *LetterOrDigit*

Letter → `_` | `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`

LetterOrDigit → *Letter* | *Digit*

Digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

`i2k`

Syntax Example: Java `import` statement

- Tokens don't have to be only ASCII.
- Instead, we can define grammars from any set of symbols.
- An identifier is one kind of token in Java

- The tokens are in blue:

  ImportDeclaration → **`import`** JavaIdentifier *DottedNames OptStar* **`;`**

  DottedNames → ε | *DottedNames* **`.`** JavaIdentifier

  OptStar → ε | **`.*`**

EBNF Example: Java `import` statement

- Extended Backus-Naur form notation (EBNF):
  - [x] means zero or one occurrences of x
  - { x } means zero or more occurrences of x

ImportDeclaration → **`import`** JavaIdentifier {**`.`** *JavaIdentifier*} [**`. *`**] **`;`**

- Concise, easier to read, less need for ε
- Compare to non EBNF version:

ImportDeclaration → **`import`** JavaIdentifier *DottedNames OptStar* **`;`**

DottedNames → ε | *DottedNames* **`.`** JavaIdentifier

OptStar → ε | **`. *`**