Comp 411
Principles of Programming Languages
Lecture 1
Course Overview and Culture

Corky Cartwright

January 9, 2023

Course Facts

- Course web page:

  https://ricecomp411.github.io/Master-2023
  https://ricecomp411.github.io/Master-2023/MainPage.pdf (pdf format)

Comp 411 vs. 511

- No difference.
- A few years ago, Comp 511 included a few extra assignments that were inconsequential.

Why Study Programming Languages?

- Programmers must master the programming languages of importance within the domains in which they are working.
- Program correctness hinges on the meaning of program text.
- New languages are continually being developed. Who knows what languages will be involved in computing 25 years from now?
- Many software applications involve defining and implementing a programming language.
- A deep knowledge of programming languages expands the range of possible solutions available to a software developer. A program design may involve extending the designated implementation language either explicitly (macros, pre-processors & custom class loaders) or implicitly (new libraries, hand-translation).
- The correctness of portable software hinges on program semantics independent of the underlying implementation.

What is Comp 411?

Anatomy (Syntax) and Physiology (Semantics) of Programming Languages

- What is the anatomy of a programming language
  - Parsing and abstract syntax.
  - Lexical nesting and the scope of variables.
  - Static properties of languages (e.g., typing) are syntax-directed.
- What are the conceptual building blocks of programming languages? (Common anatomical structures)
- Use syntactic reduction and high-level interpretation to define meaning of languages (expression evaluators)

What is Comp 411? (cont.)

- Using anatomy to prevent bugs
  - Type systems (syntactic tags with semantic content)
  - Type checking
  - Type inference (and reconstruction)
- Mechanisms for language extension
  - Syntax extension (macros)
  - Reflection
  - Custom class loaders
- Sketch how the interpretive process can be efficiently implemented by machine instructions (intelligent compilation) using good data representations
  - Environment representations
  - CPS transformation
  - Storage management: reference counting and garbage collection

Subtext of Comp 411

- Teach good software engineering practice in Java.
- You have to write a significant number of conceptually challenging lines of code in this course. With good software engineering practices, the workload is reasonable.
- With poor software engineering practices, the workload is unreasonable.
- The assignments in this course leverage abstractions that are not explicit in Java but are easily encoded using the proper design patterns (*e.g.*, composite, interpreter, strategy, visitor).
- In putative successors to Java, notably Scala and Swift, some of these abstractions are built-in to the language. Unfortunately, the semantics of Scala is (are?) hideously complex. Martin Odersky has assured me that a new edition of Scala with a semantically tractable core subset is in the pipeline. I am skeptical. Swift is simpler but the open source version is not as well supported as Java (particularly on Windows) and there is a paucity of open source libraries. Moreover, it is still evolving. I am hopeful that we can use Swift in the future. In the meantime, we will use Java, which is a decent language if it used properly.

Good Software Engineering Practice

- Data-driven design
  - The structure of program text is typically determined by the inductive structure of the data type being processed.
- Test-driven development
  - Unit tests for each non-trivial method written before any method code is written
  - Unit tests are a permanent part of the code base
- Pair programming (no longer feasible with remote instruction and distributed development)
- Continual integration
- Continual refactoring to avoid code duplication
- Conscientious documentation (contracts written at suitable level of abstraction)
- Avoiding mutation unless there is a compelling reason to introduce it

Course Culture

- Approximately 8 programming assignments
  - 7 required
  - 2 extra credit
- Assignments must be done in Generic Java (Java 8 including parameterized types). You an use newer versions of Java to develop programs but verify that you submitted code compiles and runs using Java 8. We encourage you to use DrJava, IntelliJ, or Eclipse. JUnit, JaCoco (a code coverage tool developed by the Eclipse team), and javadoc are built-in to DrJava, IntelliJ, and Eclipse. In addition, they are fully compatible with command line compilation, execution, and testing.
- Late assignments not accepted, but …
  - Every student has 7 slip days to use as he/she sees fit.
  - A maximum of 3 slip days can be used on each of first three assignments.
  - Advice: save as many slip days as possible until late in the term. The last two assignments are the most time-consuming.

Course Culture (cont.)

- Assignments are cumulative.
- Class solutions are provided for the first three assignments three days after they are due.
- After the third assignment, you are on your own except for skeleton test suites which we will provide. Extensive unit testing is important. In most of the projects, you can reuse previous unit tests on subsequent assignments with little or no change.

Course Culture, cont.

- My teaching style
  - Encourage you to develop a passion for the subject and personally digest and master the material.
  - Make the course accessible to students who don't aspire to become language researchers
- Weaknesses:
  - Tendency to digress.
  - Explain concepts at too abstract a level without sufficient examples.
  - Redress: remind me when I have strayed from the course outline; ask questions about examples and tell me if my explanations are too abstract.