Ways to Study Programming Languages

- The *sampler approach*
- The *universals approach*
- The *semantics approach*
- The *implementation approach*

The Sampler Approach

- Aka the *comparative approach*, the *survey approach*, the *language-of-the-week approach*
- Examine the “look and feel” of a number of different programming languages
  - Perhaps write the same programs in all the different programming languages you look at
- An approach with merit, but rather outdated
  - Often ends up spending too much time on relatively minor matters like syntax, quirks of specific compilers, etc.
  - Analogy: philology vs. linguistics

The Universals Approach

- Identify a set of key issues that most/all programming languages address
- When studying a particular issue, see how it's dealt with in a variety of programming languages
- Essentially a somewhat updated version of the sampler approach, grouping topics by property rather than by language
- Not a bad approach, but still primarily descriptive

The Semantics Approach

- Develop a framework/model for formalizing what happens when programs execute and how that relates to the constructs provided by a programming language
- Can be done mathematicologically with extreme rigor in terms of functions, sets, predicate logic, etc. and then called
  - *Denotational semantics*
  - *Mathematical semantics*
  - *Formal semantics*

The Semantics Approach (cont.)

- Can also be done in terms of various kinds of formal automata (e.g., Turing machines):
  - *Operational semantics*
- A more explanatory and analytical approach that downplays surface issues of syntax, etc.

The Implementation Approach

- Show how languages work by implementing *language processors* for programming languages
- Customarily done by implementing *definitional interpreters* for programming languages
- Done carefully, combines the best features of the universals approach and the semantics approach

Definitional Interpreters

- An interpreter that shows what each construct in programming language *X* does during program execution
- Designed to work on a simplified, idealized representation of programs in programming language *X*, a representation that ignores issues of pure surface-form (i.e., syntax, precedence, punctuation, etc.)

Definitional Interpreters (cont.)

- Such interpreters are often built incrementally:
  - Start with an interpreter for a stripped-down version of programming language *X*
  - “Grow” the interpreter by adding additional features of programming language *X*, one at a time
- The incremental approach can be especially illuminating and useful when done using *meta-circular interpreters*

Meta-Circular Interpreters

- A *circular interpreter* for a programming language *P* is
  - An interpreter for *P* programs
  - The interpreter is itself written in *P*
- A *meta-circular interpreter* for a programming language *P* is
  - An interpreter for programs that use only a subset of the full *P* programming language
  - The interpreter is written, as nearly as possible, in the same subset of *P* that the interpreter interprets

The Payoff

- If
  - *P* is a programming language based on sound mathematicological principles
  - You incrementally develop a (meta-)circular interpreter for *P*
  - *P* contains features representative of those found in many/most programming languages

  then you end up knowing a lot about both the formal and implementational semantics of lots of programming languages, even though you've worked on a definitional interpreter for just a single programing language

Typically

- When using the meta-circular definitional interpreters approach, you try to choose a programming language for interpretation that's small, expressive, and flexible:
  - For us, that means we'll work on implementing Scheme interpreters written in Scheme
- Additionally, we'll see how matters related to
  - Types and type-checking
  - Object-oriented features
  - Other things, time permitting

  can be added to a Scheme-like framework