CS 456: Programming Languages

Fall 2005
Tu, Th: 12 - 1:15
Room G066
Administrivia

- Who am I?
- Course Web Page:
- Office Hours:
  - Tu, Th: 3-4pm
  - By appointment
- Main text:
  - Programming Languages: Application and Interpretation
    - S. Krishnamurthi, (online draft)
Course Work

- Lectures
- Scribe
  - Will span two - three classes
  - Summarize and clarify lecture material
  - Elaborate on material (outside research)
- Programming Project
  - Semester-long project
  - Work in small-teams
  - End of semester competition and presentation
- Two midterm examinations
Prerequisites

- Programming experience/maturity
  - exposure to various language constructs
    - Java, ML, Lisp, Prolog, C
  - Undergraduate compilers class
    - CS 352 or equivalent

- Mathematical maturity
  - familiarity with basic concepts in first-order logic, set theory, graph theory, induction

- Most important:
  - Intellectual curiosity and creativity
Resources

- Web page for text:
  - http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs
- Web page for Scheme:
  - http://www.drscheme.org
  - http://www.schemers.org
  - http://www.htdp.org
- Other supplementary texts:
  - Essentials of Programming Languages, Friedman, Wand, Haynes, MIT Press 2001
Background

- Our main goal is to find ways to describe the behavior of programs precisely and concisely

Motivation

- Significant industry and government interest
  - Web, Java
  - Security issues
  - Complexity of modern-day applications
Motivation (cont)

- Foundations
  - Safety or isolation properties
  - Primitive building blocks
  - Need an unambiguous vocabulary

- Understand specific language features
  - Better language design
  - Guide improvements in implementations
Goals

- A more sophisticated appreciation of programs, their structure, and the field as a whole
  - Judge, distinguish, and relate different language features
  - Define and prove rigorous claims about a program’s meaning and behavior
  - Develop sound intuitions to better judge language properties

- Develop tools to be better programmers, designers and computer scientists
Non-goals

- An introduction to advanced programming techniques
- No detailed discussion of machine implementations
  - The course will not be motivated from the perspective of a compiler writer
  - But, impact of design decisions on implementation tractability will be considered when appropriate
- A survey of different languages
Topics

- Part I (Modularity): Names, functions, and data abstraction
- Part II (Control): Recursion, continuations, lazy evaluation
- Part III (Types): Polymorphism, object-oriented programming
- Part IV (Domain-specific programming): Macros
How should we describe languages?

- Four classical approaches:
  - Informal
    - Easy to understand
    - Easy to misunderstand
  - Operational
    - Define programs in terms of rules that apply to a specific virtual machine
    - Useful for implementing a compiler or interpreter
  - Denotational
    - Meaning in terms of functions from syntax (program text) to domains (values)
    - Useful for describing the behavior of programs
  - Axiomatic
    - Logical rules for reasoning about programs
    - Useful for proving program correctness
Language Design

- Designed to fill a void:
  - Enable expression of previously difficult-to-express applications

- Main overhead:
  - Programmer training
    - Languages with many users rarely get replaced (Cobol)
    - Ossification
  - Easy to start in a new niche
Language Design

- Tower of Babel
  - Applications often have distinct (and conflicting) needs
  - AI: (Lisp, Prolog, Scheme)
  - Scientific computing (Fortran)
  - Business (Cobol)
  - Systems programming (C)
  - Scripting (Perl, Javascript)
  - Distributed computation (Java)
  - Special-purpose (.....)

- Important to understand differences and similarities among different language features
Paradigms

- Imperative
  - Fortran, Algol, C, Pascal
  - Designed around a notion of a program store
  - Program behavior expressed in terms of transformations on the store

- Functional
  - Lisp, ML, Scheme, Haskell
  - Programs described in terms of a collection of functions
  - “Pure” functional languages are state-free

- Logic
  - Prolog
  - Programs described in terms of a collection of logical relations

- Concurrent
  - Fortran90, CSP, Linda

- Special purpose
  - TeX, Postscript, HTML, CGI
Metrics

- No universally accepted criteria
- The most popular languages are not necessarily the best ones
  - Consider Cobol or JCL
- General characteristics
  - Simplicity and “elegance” (orthogonality)
  - Readability
  - Safety
  - Programming-in-the-large
  - Efficiency
  - Abstraction
Case studies

- Lisp 1.5
  - Based on λ-calculus
  - Key aspect of the calculus is notion of substitution of free variables:
    - function f (args) = .... x ....
    - Suppose x is not included in args. Where should the binding for x be constructed?
      - At the point where f is defined (lexical scoping)
      - At the points where f is applied (dynamic scoping)
  - Lisp 1.5 (and later dialects) chose dynamic scoping, even though it is widely agreed today that lexical scoping is the more sensible choice.
  - When do these distinctions arise? Why are the differences important?
  - Lack of formal semantics to explore the ramifications of design choice
Case studies

- ML
  - Interaction of types with references
  - Polymorphism: code that works uniformly on all various types of data
    - length: $\alpha$ list -> int
    - hd: $\alpha$ list -> $\alpha$
    - tl: $\alpha$ list -> $\alpha$ list
  - Type inference
    - Assign the most general type to variables based on the contexts in which they occur
Case studies: ML

- References
  - Like updateable pointers in C
  - Expressions
    - ref e: τ -> τ ref
    - !e: τ ref -> τ
    - e1 := e2: τ ref * τ -> τ ref

fun id(x) = x
val c = ref id
fun inc(x) = x + 1

c := inc  
!c (true)

id: α -> α

c: (α -> α) ref

inc: int -> int

Ok if we infer c: (int -> int) ref

Ok if we infer c: (bool -> bool) ref

Type system would fail to prevent a type error
Case studies

- Eiffel
  - Strongly-typed object-oriented language that supports inheritance
  - Uses a notion of covariance (rather than contravariance) in typing functions
    - A function with a restricted set of inputs can be used in a context where a function with a wider set of arguments is expected
    - Type system may fail to prevent runtime type errors
  - Failure to cleanly separate notions of subtyping from subclassing
Lessons

- Language design is as much about safety as it is about efficiency and expressiveness.
- Need tools and frameworks to reason about and compare different language features and designs:
  - We will use a simple core language: the untyped $\lambda$-calculus as a universal computation language. We’ll explore variations and extensions of this language throughout the course
  - We will study how notions such as types, control, state, and objects fit within this language.
Homework

- Reading:
  - Chapter 1
- Familiarize yourself with Dr. Scheme
- Next time:
  - Brief tour of functional programming in Scheme
- Introduction to untyped arithmetic and syntax