CSE 505: Programming Languages (Fall '95)

Instructor: Craig Chambers

- office hours: TuTh 12:00-1:00 (Sieg 309)

TAs: Jeff Dean and Vass Litvinov

- Jeff Dean's office hours: W 11:30-12:30 (Chateau 110)
- Vass Litvinov's office hours: M 11:30-12:30 (Sieg 423)

Goals:

- Learn to use formal semantics as a tool for understanding and describing language semantics precisely
- Gain some experience using advanced features of modern programming languages
- Become aware of current open problems and active research areas in the design of modern programming languages
- Understand trade-offs in language design

Readings:

- *The Structure of Typed Programming Languages*, David Schmidt
- copies of overviews of languages being studied
- important papers from the literature

Grading:

- weekly homework assignments: 20%
- course project: 20%
- two tests: 18% each
- final exam: 24%

Course outline

Formal semantics

- Chapter 1
  - abstract vs. concrete syntax; EBNF
  - static typing; type inference rules
  - denotational semantics; domains; stores
  - operational semantics
  - axiomatic semantics
- Chapter 2
  - named abstractions; environments
  - lazy and eager abstraction evaluation
  - recursive abstractions
  - intro to domain theory
  - variables, records, modules
- Chapter 3
  - parameters to abstractions
  - substitution rule; free variables; name capture
  - lazy and eager parameter evaluation
  - type equivalence
  - classes
- Chapter 4
  - local declarations
  - scoping; lexical vs. dynamic
  - extent
- Chapter 5
  - expression-oriented language; record and lambda abstractions
  - first-class, higher-order abstractions
  - orthogonality
- Chapter 6
  - (untyped) lambda calculus
  - bound vs. free variables; closed expressions
  - α, β, η reductions; normal forms
  - confluence (Church-Rosser property)
  - simply-typed lambda calculus; subject reduction; strong normalization
- Chapter 8
  - higher-order typed lambda calculi (maybe)

Functional languages

- Scheme
  - atoms, symbols, lists, cons, car, cdr; define, let, let*, cond
  - higher-order first-class functions
  - tail recursion
  - eval
  - continuations (maybe)
- ML

  - lists, tuples; val, fun, if, let
  - pattern matching
  - type inference
  - polymorphic types
  - higher-order functions
  - datatypes, polymorphic datatypes
  - exceptions
  - references
  - modules; signatures and structures
- Haskell
  - lazy evaluation
  - array & list comprehensions
  - type classes

Object-Oriented Languages

- Smalltalk
  - objects, classes, instances, methods, instance variables
  - inheritance
  - dynamic dispatching
  - first-class classes; meta-classes
- Cecil
  - multi-methods
  - multiple inheritance
  - subtyping; subtype polymorphism
  - inheritance vs. subtyping
  - covariance vs. contravariance
  - bounded quantification; F-bounded quantification; matching