CS 603: Organization of Programming Languages
What should a programming language do?

Express computations
- precisely
- at a high level
- in a way we can reason about them
What’s it all for?

Make it easier to write programs that really work
Why study programming languages?

New ways of thinking about programming

Writing programs is fundamental

Language can help or hinder

Become a sophisticated, skeptical consumer
Why do people actually like this stuff?

Language is the way to write code
  • write code that’s elegant
  • write code that’s cool
CS 603 Agenda

Intellectual tools to understand & evaluate languages

- Language features
- Questions with answers

Learn the notations of the trade

- Precise way to model languages
- Foundation for further study

Learn by doing

- Write lots of (mostly short) programs
- Many difficult programs (thought required)
  (High difficulty per line of code)
Study of Language $\equiv$ Study of Features

Language features influence code

*Choose* abstractions (languages) to fit needs

Build your vocabulary (add to your toolbox)
- Higher-order functions
- Polymorphism (reuse)
- Pattern matching for symbolic computing
- Data for symbolic computing: lists, tables, sets
- Abstract datatypes, encapsulation
- Objects and subtyping
- Modules, parameterization
- Searching and backtracking
How to use Features

From the definition of Scheme:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Larry Wall (inventor of Perl) and Bjarne Stroustrup (inventor of C++) might not agree.
Why study “weird” features?

Some languages more powerful than others

Mistake to use any but the most powerful
   Except for: compatibility, libraries

Problem: habit blinds us to power

Happy user of Blub: beats Cobol, machine code
   Use Haskell, Lisp, or Icon? No! Equivalent to Blub, plus weird stuff nobody uses

Blub looks good enough because I think in Blub

CS 603: tour of power in languages
Course orientation: Language Design

Expressive Power

Language Design

Programmer’s Productivity  Implementation
The search for expressive power

Functions
Types
Pattern matching
Context-free grammars
Making the programmer more productive

Programming methodologies, software engineering

Goals:
- fewer bugs, more easily isolated
- reuse code

Techniques:
- Abstract data types
- Modules (including “generics”)
- Objects and inheritance (reuse)
- Separate compilation/smart recompilation
Influence of Implementation

Techniques
- Parser generators
- Attribute-grammar systems
- Memory allocation
- Garbage collection
- Runtime typing/tagging

Efficiency concerns
- fast execution
- fast compilation
- fast program construction

(Compiler implementation is primarily a topic for CS 614)
Describing it all precisely

Formal semantics:
  Operational semantics (tool of the trade)
  Denotational semantics (for mathematicians)
  Axiomatic semantics
  Predicate transformers
Some design dimensions

Typing
  strong vs. weak (ill-defined terms)
  static vs. dynamic
  monomorphic vs. polymorphic

First-class values
  structures?
  procedures? (funarg problem)
  are built-in types different?
More design dimensions

Safety
    no *unexplained* core dumps (and more…)

Control flow
    stack-based
    heap-based, closures & continuations
    logic programming (Prolog, unification)
    backtracking (Icon)

*Non-dimensions:*
    “Simplicity,” “Orthogonality,” “Readability”
    …
Administrivia — Grading

Weight of grades:

- 3 midterm exams: 45% to 60%
- Final exam: 20% to 35%
- Projects: 10%
- Homework: 10%
Collaborate! (Up to a point)

• what professionals do
• vital to your success
• discuss problems, techniques, ideas
• for team assignments: collaboration is fine with members of your team
• for individual assignments: discussion only, no sharing of code or written answers
Administrivia — Policies, procedures

Policies and procedures
  • handed out in class
  • on the web

Know what is expected
Prerequisites

You must enjoy programming

You must also like math

C (or C++ or Java)

Ability to learn new languages quickly: Scheme, ML, Smalltalk, Prolog

Windows and/or Unix

Algorithms, data structures

Basic mathematics (set theory, logic, induction)
Course of Study

Work hard, learn a lot

Focus on

• **semantics**, not syntax
• **the unusual**, not the common (**weird but powerful**)  
• **answerable** questions
Methods of study

Case studies of interpreters

- Learn foundations of languages by studying and modifying implementations
- Study abstracted “essentials” of languages
- (Mostly) uniform implementation framework

Supplement by

- Descriptive tools of the professionals: operational semantics type systems

- Work with real languages
Readings

Ramsey and Kamin

Programming Languages: An Interpreter-Based Approach

distilled essence of languages
uniform syntax, implementation framework
available in Ferguson Center supply store
## Topics

<table>
<thead>
<tr>
<th>Unit/Language</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative Core</td>
<td>environments, bindings, ASTs, operational semantics</td>
</tr>
<tr>
<td>Scheme</td>
<td>S-expressions, recursion and lists, programs as data, first-class &amp; higher-order functions</td>
</tr>
<tr>
<td>ML</td>
<td>type inference</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>object-oriented programming</td>
</tr>
<tr>
<td>Prolog</td>
<td>logic programming, unification</td>
</tr>
</tbody>
</table>