Lecture 27 — Wrap-up

- **Review of "big themes" of CS 421**
- **Next steps: follow-up courses and what they cover**
- **Open mic**

From lecture 1: What you will learn this semester

- How to implement programming languages
  - Writing lexical analyzers and parsers
  - Translating programs to machine language
  - Implementing run-time systems
- How to write programs in a functional programming language
- How to formally define languages (including the definitions of type rules and of program execution)
- Key differences between statically-typed languages (e.g. C, Java) and dynamically-typed languages (Python, JavaScript)
- Plus a few other things...

Big themes of CS 421 — # 1

Processing structured data

- Lexing
- Parsing
- Constructing the abstract syntax tree, giving the "deep structure" of the input

Big themes of CS 421 — # 2

Recursive traversal of abstract syntax tree

- Traverse AST to check types
- Traverse AST to evaluate expressions
- Traverse AST to compile code

Big themes of CS 421 — # 3

Defining languages precisely

- “SOS”-style rules for evaluation
- “SOS”-style rules for compilation
- “SOS”-style rules for type-checking/inference
- Rewrite rules used to define machine instruction set
- Proving programs correct using invariants

Big themes of CS 421 — # 4

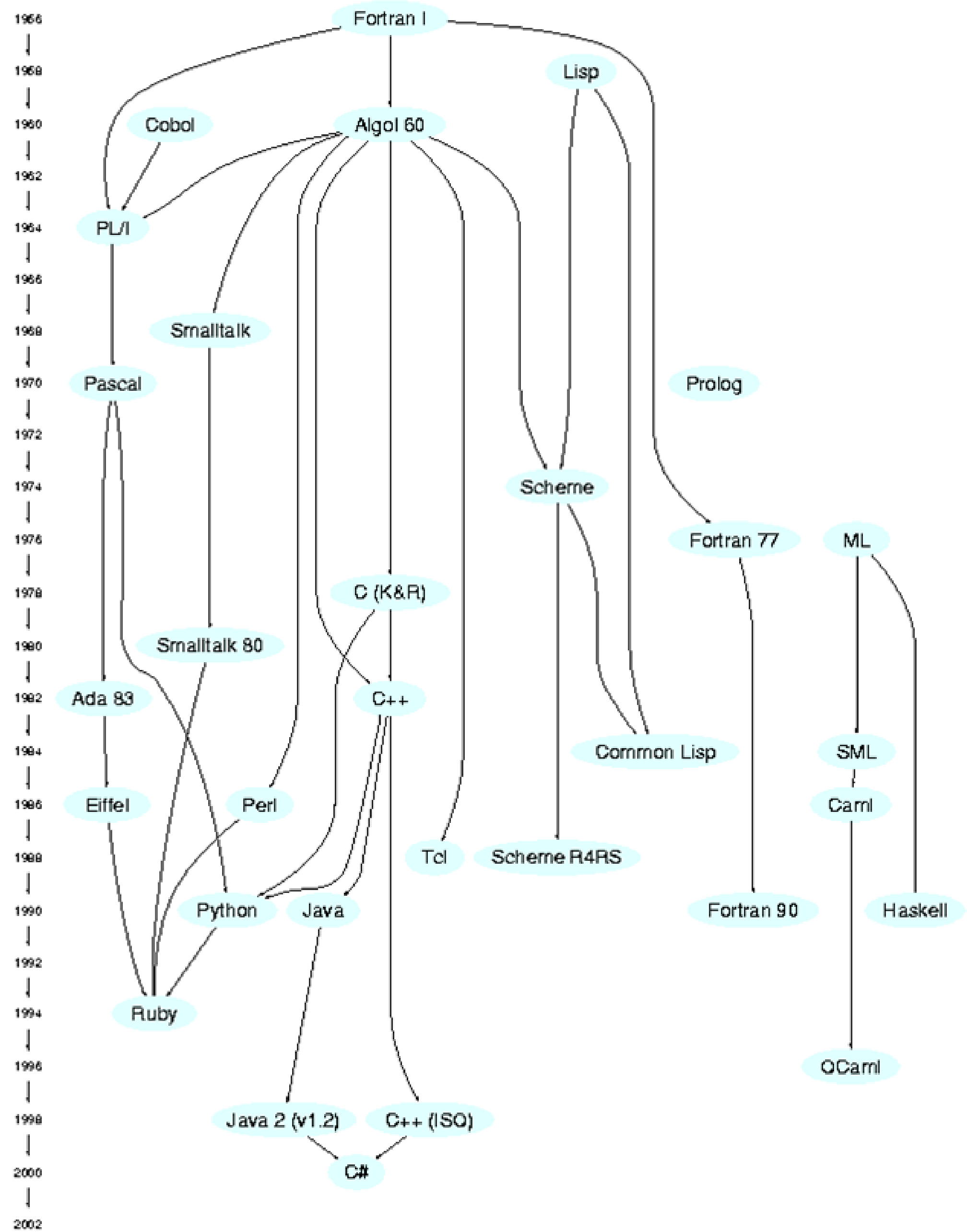
Dynamically-typed vs. statically-typed languages

- Dynamically-typed languages
  - More flexible; less safe
  - Less efficient (tag-checking; boxing)
  - Usually implemented by interpretation (easy)
- Statically-typed languages
  - More efficient (no tag-checking)
  - Usually implemented by compilation (more efficient, but hard)

Big themes of CS 421 — # 5

Traditional vs. functional programming

- Traditional (imperative, object-oriented)
  - Program by side-effect, i.e. assignment
  - Better matches machine architecture
  - Minimize data movement; don't (necessarily) require garbage collection; more efficient
- Functional
  - Program by calculating values
  - Recursion over lists and trees
  - Functions as values
  - More concise; may be less efficient


1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
Fortran I
Lisp
Cobol
Algol 60
PL/I
Smalltalk
Pascal
Prolog
Scheme
Fortran 77
ML
C (K&R)
Smalltalk 80
Ada 83
C++
Common Lisp
SML
Eiffel
Perl
Caml
Tcl
Scheme R4RS
Python
Java
Fortran 90
Haskell
Ruby
OCaml
Java 2 (v1.2)
C++ (ISO)
C#

Next steps

- Theory — CS 422
  - Type systems for more languages (e.g. functional languages with class hierarchies)
  - Proofs of correctness of type systems and compilers (relative to SOS semantics)
  - Semantics of concurrent/parallel languages
- Implementation — CS 426
  - More on parsing: building LR parsers; error-correction
  - Static analysis for optimization — e.g. given call $e.f(\ldots)$, determine which f this is (if possible)
  - Code generation for real machines
- Program verification — CS 476, 477

Outline for final

- Recursion on lists and trees; ASTs
- Methods of parsing and lexing
  - Manual lexing; lexer generators
  - Bottom-up parsing; parser actions and precedence decls
  - Top-down parsing; LL(1); left-factoring
  - Expression grammars
  - Translation to abstract syntax
- Methods of execution
  - Interpretation of AST (SOS rules)
  - Compilation to native code (including machine-dependent optimizations) (Compilation schemes)

- Compilation to abstract machine, followed by:
  - Emulation of abstact machine, *or*
  - Compilation to native code at run time ("just-in-time")

- Run-time environments
  - "Raw" machine — no automatic memory management; no reflection; no standardized data layouts; OS "service calls"
  - Virtual machine (e.g. Java virtual machine, Common Language Runtime) — garbage collection; defined data layouts; reflection; higher-level services — e.g. threads — provided by run-time system

- Statically-typed vs. dynamically-typed langauges
  - Tagged values
  - Advantages/disadvantages of static typing

- **Higher-order functions**
  - **map, fold, curry, uncurry, etc.**
  - **Combinator-style programming (e.g. parser combinators; picture combinators)**
  - **Interpretation via substitution model and environment model (closures)**
  - **Using higher-order functions in non-functional languages (function objects)**
  - **Lazy evaluation; the $\Downarrow_\ell$ rules**
- **Type systems for OCaml**
  - **Monomorphic & polymorphic systems; value restriction**
- **Program verification — loop invariants**

List of terms you should know (in no particular order)

Curried vs. uncurried functions
Recursion fairy
Compilers
- Front-end
- Abstract syntax trees (ASTs)
- Type-checking, symbol table
- Back-end
- Intermediate representation
- Machine-independent optimization
- Code generation (machine-dependent optimization)

Lexer
- Token

- DFA
- lexer-generator, lex/ocamllex, regular expressions

Parsing

- Parse trees
- Sentences, sentential forms
- Epsilon production
- Ambiguous grammar
- Nullable non-terminal
- A-sentence, A-sentential form
- Extended cfg
- Stratified expression grammar
- Shift-reduce parsing
- Ocamlyacc precedence declarations
- Recursive descent
- LL(1), FIRST set, FOLLOW set

- Left-recursive (or right-recursive) grammar
- Left-factoring

Static vs. dynamic typing; tagged values; type errors vs. run-time errors

Proof system - judgment axiom, rule of inference

Structured operational semantics (SOS)

Side effects; "threaded" store

Inheritance

Compilers

- Interpretation vs. compilation
- Virtual machine; bytecode; just-in-time compilation
- Short-circuit evaluation
- L-values vs r-values
- V-tables
- Substitution model vs. environment model; closure
- Two-level store

Automatic memory management

- Reachable cells; garbage cells; free list
- Reference-counting
- Mark-and-sweep garbage collection
- Stop-and-copy garbage collection

Hoare logic; loop invariants; termination conditions; Hoare triples

Anonymous functions

Higher-order functions; map; fold_right

Parser combinators; picture combinators (MP 11)

Monomorphic vs. polymorphic types; let-polymorphism

Type checking vs. type inference

Type scheme; generalization and instantiation

Ocaml references; the value restriction

Lazy evaluation; lambda calculus; beta-reduction

Function objects