CSE 5343: Introduction

Aho, Lam, Sethi, Ullman:
Chapter 1, Sections 1.1, 1.2, 1.5
Implementation Methods

- Compilation (C, C++, ML)

- Interpretation (Lisp)

- Hybrid systems (Java)
The Entire Toolchain (1/2)

• Preprocessor: source to source translation
  – E.g. GNU C/C++ macro preprocessor `cpp`
    • Inlines `#include`, evaluates `#ifdef`, expands `#define`
    • Produces valid C or C++ source code

• Compiler: source to assembly code
  – E.g. GNU C/C++/... compiler `gcc`
  – Produces assembly language for the target processor
  – Assembly is easier to generate/debug than object code

• Assembler: assembly to relocatable object code
  – E.g. GNU assembler `as`
  – Translates mnemonics (e.g. ADD) to opcodes; resolves symbolic names for memory locations
The Entire Toolchain (2/2)

• Linker: **relocatable object code** from several modules (including libraries) to **single executable program**
  – E.g. GNU linker **ld**
  – Resolves inter-module symbol references; relocates the code (recomputes addresses)

• Example: **gcc** from Unix command line is a **driver program** that invokes the entire toolchain
  – **gcc -E test.c**: preprocessor (output: C code)
  – **gcc -S test.c**: preprocessor+compiler (output: assembly)
  – **gcc -c test.c**: preprocessor+compiler+assembler (output: object code for this compilation unit)
Inside the Compiler: **Front End**

- **Lexical analyzer (aka scanner)**
  - Converts ASCII or Unicode to a *stream of tokens*

- **Syntax analyzer (aka parser)**
  - Creates a *parse tree* from the token stream

- **Semantic analyzer**
  - Type checking and conversions; other semantic checks

- **Generator of intermediate code**
  - A parse tree is too high-level for code generation & optimization
  - Create *lower-level intermediate representation (IR)*:
    - e.g., three-address code
Inside the Compiler: Middle Part

• **Analysis** of intermediate code
  – Additional IRs: control-flow graph (CFG), static single-assignment form (SSA), def-use graph, etc.
  – Control-flow analysis, dataflow analysis, pointer analysis, side-effect analysis, polyhedral analysis, ...

• **Machine-independent optimization** of intermediate code: better three-address code
  – Copy propagation, dead code elimination, code motion, constant propagation, redundancy elimination, parallelization, data locality optimizations, ...

• Currently, this is where most of **compiler research** is focused
Inside the Compiler: **Back End**

- **Code generator:** from *intermediate code* to *assembly code*
  - Instruction selection, instruction ordering, register allocation, storage allocation, ...

- **Machine-dependent code optimizations**
  - Elimination of redundant loads and stores, elimination of unreachable code, use of machine idioms (e.g., specialized instructions)

- **A symbol table** maintains information about names, types, and scopes
  - Used by all phases of the compiler
The Bigger Picture: Compiler Technology

• Very strong connections with language design and with computer architecture
  – Compiler technology had direct impact on the success of C, C++, Java, and C#
  – Intel has its own compiler
  – How to take advantage of multi-core/GPU/TPU?

• Strong connection with software engineering tools
  – IDEs have many code analyses and transformations
  – Static verification tools – heavy use of compiler analyses (real-world impact)
  – Automated debugging tools
  – Test coverage tools & test generation tools