High-level View of a Compiler

Source code ➔ Compiler ➔ Machine code

Implications
- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations
Traditional Two-pass Compiler

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes (better code)

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC

A Common Fallacy

Can we build $n \times m$ compilers with $n+m$ components?
- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Successful in systems with assembly level (or lower) IRs
Traditional Three-part Compiler

Source Code → Front End → IR → Optimizer (Middle End) → IR → Back End → Machine Code → Errors

**Code Improvement (or Optimization)**
- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables

Comp 412, Fall 2010
Run-time Compilation

Systems such as HotSpot, Jalapeno, and Dynamo deploy compiler and optimization techniques \textit{at run-time}

- **Policy chooses between interpreter & compiler**
  - LLVM compiles on 1\textsuperscript{st} call
  - Dynamo optimizes on 50\textsuperscript{th} execution

Source Code \rightarrow \text{Offline Compiler} \rightarrow \text{IR} \rightarrow \text{Code base} \rightarrow \text{JIT Compiler} \rightarrow \text{Run-time environment} \rightarrow \text{Interpreted Code}

Comp 412, Fall 2010