Introduction

CS 2210 Compiler Design
Wonsun Ahn
What is a Compiler?

- Compiler: A program that **translates source code** written in one language to a **target code** written in another language
- Source code: Input to compiler
  - Typically a program written in human readable language (e.g. C)
- Target code: Output of compiler
  - Typically binary code written in machine language
    - Native machines: x86, ARM, MIPS code
    - Virtual machines: Java bytecode, LLVM bitcode
  - Binary code can later be executed on the given machine
- Target code for one compiler can be source code for another
  - Javac compiler: Java source code ➔ Java bytecode
  - Java Virtual Machine compiler: Java bytecode ➔ native code
  - Cython: Python source code ➔ C source code
- Invariant: original semantics must stay the same
What is an Interpreter?

• Interpreter: A program that reads in source code and executes it on the target machine by interpreting it line by line
  + Software becomes more portable
    • Binary code is tied to a specific machine architecture
    • Source code can run anywhere with correct interpreter
  + Software becomes more secure and safe
    • Interpreter can perform runtime checks (e.g. to prevent reads of illegal memory locations)
  + Interpreter is relatively simpler to implement than compiler (Popular for languages with small user base)
    – Interpretation much slower compared to binary execution (Source line has to be interpreted each time it’s executed)
Just-In-Time (JIT) Compiler

• Just-In-Time (JIT) Compiler: A compiler that performs translation at runtime (just in time before execution)
  – Traditional compilers are called Ahead-Of-Time (AOT)
  – Typically implemented when an interpreted language gains enough of a user base to warrant its development
• Pros / Cons compared to AOT compiler
  + Software can be run in a portable, secure, safe way
  – Slight performance overhead due to JIT compilation time (But much less than interpretation since compiled code can be cached and reused)
  + Opportunities for better performance
    • By profiling program behavior and optimizing against it
    • By tailoring target code to details of underlying machine
AOT vs. JIT Compilation

**Ahead-Of-Time (AOT) Compilation**

- **Compile**
- **Binary** → **Distribute**

Languages: C/C++, Fortran

**Just-In-Time (JIT) Compilation**

- **Translate/Minify**
- **Bytecode/Source** → **Distribute**

Languages: Java, C#.net, JavaScript, Python, PHP, Lua, R, OpenCL

**Hardware**
Topics Covered
Compiler Phases

Source Program

Lexical Analyzer

Token Sequence

Syntax Analyzer

Syntax Tree

Semantic Analyzer

3-Address Code

Code Optimizer

Optimized 3-Addr. Code

Code Generation

Assembly Code

IF (a < b) THEN c = 1 * d;

IF (ID "a" < ID "b") THEN ID "c" = CONS "1" * ID "d";

GE a, b, L1
MUIT 1, d, c
L1:

GE a, b, L1
MOV d, c
L1:

loadi R1, a
cmpi R1, b
jge L1
loadi R1, d
storei R1, c
L1:
Front End

• Group of phases that **analyzes** the source code and builds one or more internal representations (IRs) out of that analysis
  – Comprised of lexical, syntax, and semantic analyses
  – IRs can be syntax trees, 3-address codes, etc.

• Lexical Analysis,
  – Input: Source code text
  – Output: Sequence of tokens (smallest unit with meaning)
    • Tokens: Identifiers, keywords, constants, operators …
  – Scans text left to right and generates tokens one by one, using language token definitions
  – Also checks for illegal tokens (e.g. malformed identifier)
Front End

• Syntax Analysis
  – Input: Sequence of tokens
  – Output: Syntax tree
  – Adds tokens to a hierarchical structure called a syntax tree that represents program, using language grammar rules
  – Also checks for syntax errors (e.g. malformed if statement)
Front End

• Semantic Analysis
  – Input: Syntax tree
  – Output: Low-level IR (e.g. 3-address code)
  – Allocates memory location for each variable definition
  – Associates variable uses with variable definitions
  – Generates a pseudo-code IR easily translatable to machine code
  – Also checks for semantic errors (e.g. undefined variable)
Back End

• Group of phases that **synthesizes** machine code from the internal representations (IRs) generated by the front end
  – Comprised of code optimization and code generation

• Code Optimization
  – Input: Low-level IR (e.g. 3-address code)
  – Output: Optimized low-level IR
  – Modifications to IR such that code runs faster and/or consumes less memory
  – Comprised of multiple optimizations applied in sequence
  – Typically does not change the IR format
Back End

• Code Generation
  – Input: (Optimized) low-level IR
  – Output: Target Code
  – Perform following tasks to transform IR to target code:
    • Instruction Selection – select actual machine ISA instructions to implement computation in IR
    • Register Allocation – allocate frequently used locations in processor registers
  • An additional code optimization phase may follow code generation to apply target machine specific optimizations
Multiple Front Ends and Back Ends

• Modern compilers typically have multiple front ends
  – E.g. GCC (GNU Compiler Collection) has front ends for C, C++, Fortran, and Java among others
  – Means all front ends generate the same IR format that can be passed to the back end for code generation
• Modern compilers typically have multiple back ends
  – E.g. GCC has back ends for x86, ARM, SPARC, etc.
  – Means same IR can be translated to multiple targets by the code generator
• A common IR is central in enabling this diversity
  – Instead of M X N implementations for M languages and N machines, only need M + N implementations
Why Learn Compilers?
Why Learn Compilers?

• Will allow you to write more robust, more efficient programs
  – You will have a deep understanding of how your program will execute on the actual machine

• Compiler analysis techniques can be used for other purposes
  – Can be used to analyze any kind of structured text data
  – Can be used to translate any format to another format

• Your research may involve compilers
Why Compilers is Still a Research Issue

• You may think…
  – The C language has been around since the 1970s
  – We are still using the C language
  – There is probably not much left to do in terms of compilers

• But you may be wrong
  – Changes in programming environment is producing new challenges in the front end of the compiler
  – Changes in computer hardware design is producing new pressures in the back end of the compiler
Popularity of Programming Languages

* Number of programmers coding in given language over time (Reference: www.ohloh.net)

- C/C++ dominated in 2005
- Now, JavaScript and Python are the most used languages
Front End Challenges

• Prevalence of **scripting languages** (JavaScript, Python, …)
  – Need for fast prototyping in a fast changing industry
  – Need for computing in non-CS fields (physics, finance, …)
  – Very flexible by design. E.g. Dynamic Typing:
    ```javascript
    var x = 1, y, z;
    if (...) y = 2; else y = “2”;
    z = x + y; // z == 3 or z == “12”
    ```
  – Challenge: Generating efficient code for such languages

• Increasingly complex software ➞ increasingly **complex bugs**
  – Most insidious are heisenbugs (particularly data races)
  – Challenge: Removing or detecting bugs through code analysis

• Increasing sophisticated **security exploits**
  – Side-channel attacks (e.g. Spectre / Meltdown exploit)
  – Challenge: Removing information leakage from generated code
Back End Challenges

• Limits in device miniaturization is driving a sea change in HW
  – CPU frequencies scale no more due to power wall
  – CPUs are less reliable due to transistor doping variability
• Performance must come from **parallelism**
  – Must run code in parallel to make up for lack of freq. scaling
  – Challenge: Automatically parallelize or vectorize code
• Processor efficiency must come from **heterogeneous** architectures
  – Different types of processors on chip for different types of code
    (e.g. CPUs, GPGPUs, Accelerators, etc …)
  – Challenge: Generating code for this type of architecture
• Memory efficiency must come from **software cache coherence**
  – HW manages caches in a reactive manner: not efficient enough
  – Challenge: Generating code that manages caches in software using knowledge of program provided by programmer or analysis