Introduction

CS 1622 Compiler Design

Wonsun Ahn

Instructor Introduction

My Technical Background

- Wonsun Ahn
  - First name is pronounced *one-sun* (if you can manage)
  - Or you can just call me Dr. Ahn (rhymes with *naan*)
- PhD in CPU Design and Compilers
  - University of Illinois at Urbana Champaign
- Current interests
  - Compilation for scripting languages / quantum computing

Why Learn Compilers?

Why Learn Compilers?

- Allows you to write more robust, more efficient programs
  - Deeper understanding of compiler errors / warnings
  - Deeper understanding of code generation / optimization

- Allows you to design your own Domain Specific Language
  - E.g. a new database query language
  - E.g. a new language for musical notations

- Compiler analysis techniques can be used for other purposes
  - E.g. a code analysis tool that finds security vulnerabilities
  - E.g. convert one language to another language

What is a Compiler?

Two Ways to Execute a Program

```
int main()
{
  int x = 1, y = 1, z;
  z = x + y;
  printf(“z = %d\n”, z);
  return 0;
}
```

- Using an interpreter – interpret and execute line by line
- Using a compiler – translate to machine code, and then run

What is an Interpreter?

- Interpreter: A program that executes source code on the target machine by interpreting it **line by line**

```
int main()
{
  int x = 1, y = 1, z;
  z = x + y;
  printf("z = %d\n", z);
  return 0;
}
```

Program State

x	1
y	1
z	2

Interpreters are portable and secure but slow

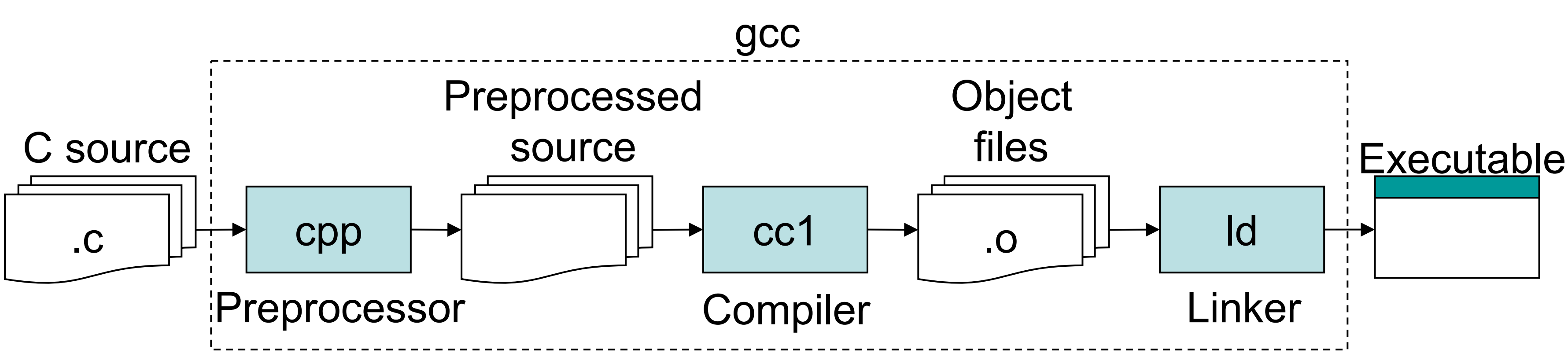
- Interpreter: A program that executes source code on the target machine by interpreting it **line by line**
  - \+ Portable
    - Source code runs anywhere interpreter is installed
  - \+ Secure
    - Interpreters act as a sandboxed virtual environment, where no memory bugs or buffer overflows are possible
  - – Slow execution
    - Source line interpreted on each execution (even in loop)

- Scripting languages are interpreted for portability and security
  - – JavaScript, Python, PHP all started out interpreted

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, source code written in programming language (e.g. C)
- Target code: Output of compiler
  – Typically, binary code written in machine language (e.g. x86)



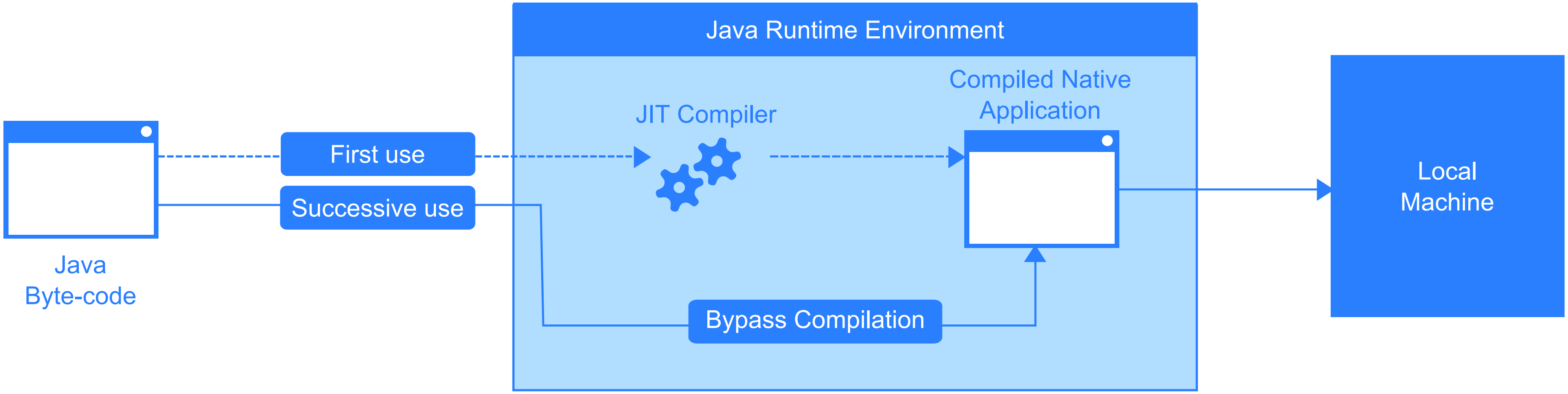
Compiled code is less portable and secure but fast

- Compiler: A program that **translates** **source code** written in one language to a **target code** written in another language
  - – Less portable
    - Machine code is tied to a specific machine architecture
  - – Less secure
    - Machine code runs on bare machine w/ no sandboxing
  - \+ Fast execution
    - No interpretation needed during execution
    - Machine code can be optimized to be even faster

- HPC (High-Performance Computing) / system codes are compiled
  - – C/C++, Fortran

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)
  - Javac that converts Java code to byte-code is AOT

JIT-Compiled code is portable, secure *and* fast

- Pros / Cons
  - \+ Portable: code runs anywhere JIT compiler is installed
  - \+ Secure: also runs in a sandboxed virtual environment
  - \+ Sometimes faster than AOT compiled code
    - By profiling program behavior and optimizing
      (e.g. profile branch direction and optimize that path)
  - – Overhead due to JIT compilation time
    - Overhead mitigated over prolonged reuse of code

- With larger user-base, interpreters move on to JIT compilers
  - JavaScript (Chrome V8) , Python (PyPy), PHP (Hiphop)

AOT vs. JIT Compilation

Ahead-Of-Time (AOT) Compilation

Compile

Binary → Distribute → Binary / Hardware

Languages:

C/C++

Fortran SOURCE

Just-In-Time (JIT) Compilation

Translate/ Minify

Bytecode/ Source → Distribute → Bytecode/ Source / JIT Compiler / Hardware

Languages:

Java

Microsoft C#.net

JavaScript

python

php

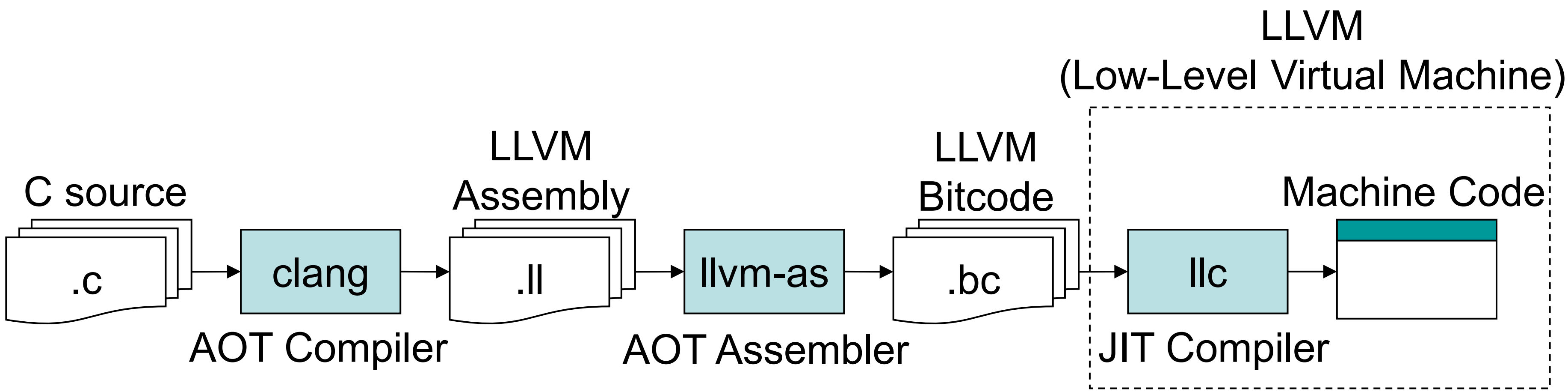
Lua

R

OpenCL

Can C/C++ be JIT-Compiled? Yes, of course!

- Clang (clang.llvm.org): Open-source compiler using JIT-compilation
  - Used by Apple in all their products



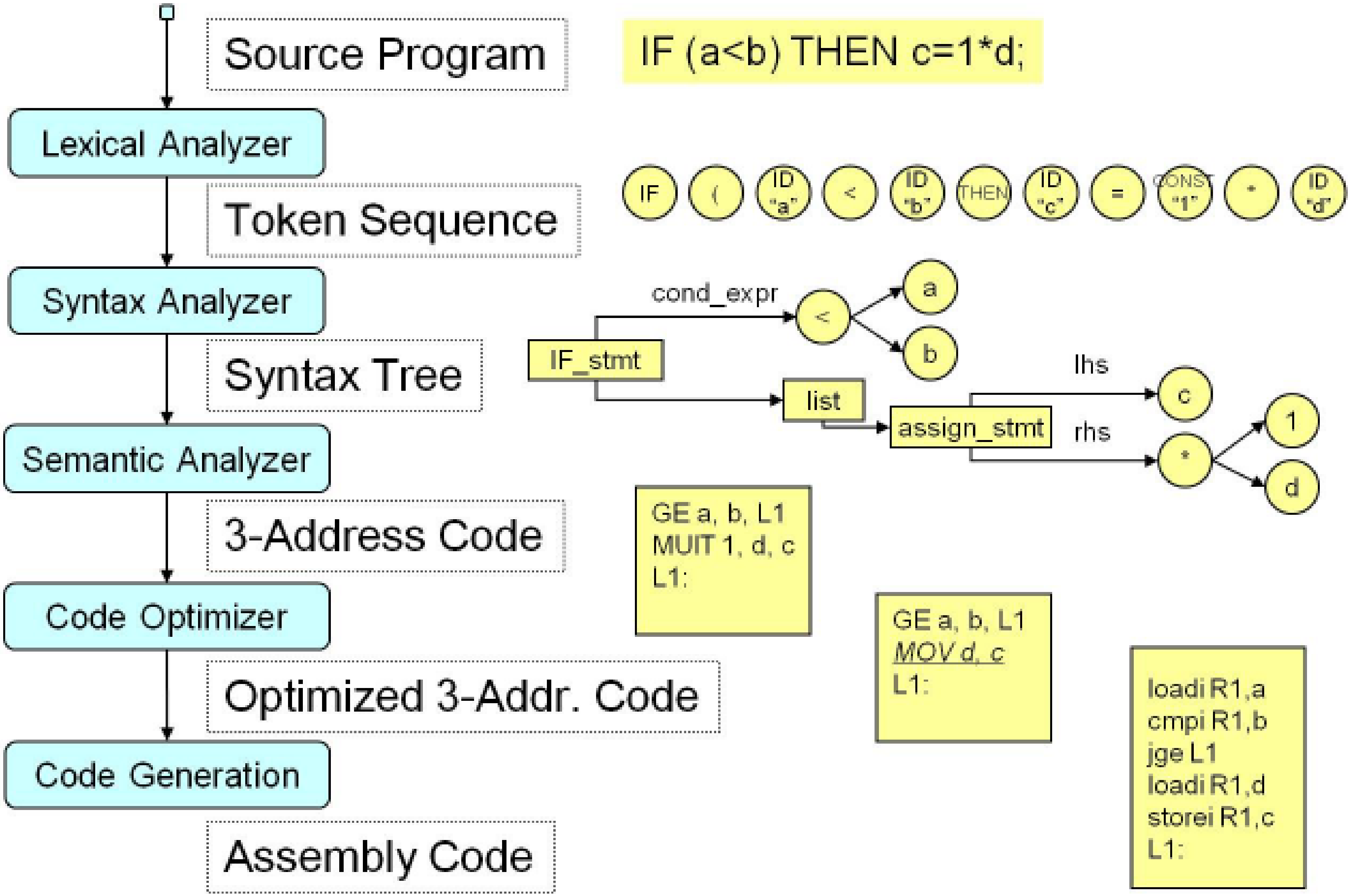
- LLVM bitcode can run anywhere an LLVM is installed
- JIT Compiler can profile & optimize code at runtime just like Java
  - To avoid compile overhead, machine code can be stored as a file (On next run, executable file can be run without compilation)

In this Class …

- We will learn about how compilers are built in general
- Concepts we will learn applies to both AOT and JIT compilers

Topics Covered

Compiler Phases



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 units with meaning)
    - Tokens: Identifiers, keywords, constants, operators …
  - Scans text left to right and generates tokens one by one
    - Scanned using regular expression pattern detection

Front End

- **Syntax Analysis**
  - Input: Sequence of tokens
  - Output: Syntax tree
  - Adds tokens to a hierarchical structure called a syntax tree
    - Tree built using language grammar rules
  - Also checks for syntax errors (e.g. missing semicolons)
- **Semantic Analysis**
  - Input: Syntax tree
  - Output: Low-level IR (e.g. 3-address code)
  - Associates variable uses with variable definitions
  - Generates an IR easily translatable to machine code
  - Also checks for semantic errors (e.g. type errors)

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
  - Modifies IR so that code runs faster and uses less memory
  - Comprised of multiple optimizations applied in sequence
  - Typically same IR format kept before and after
    - So that users can pick and choose from optimizations

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 CPU registers to speedup access
- 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, Objective-C/C++, and Rust among others
  - All front ends generate the same *common IR* format
- Modern compilers typically have multiple back ends
  - E.g. GCC has back ends for x86, ARM, SPARC, etc.
  - Each back end translate *common IR* to respective target
- 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

Challenges Facing Compilers

Challenges Facing Compilers Today

- Fundamental changes are happening in the computing stack
  - Putting new pressures on the compiler

- Changes in **languages** and **applications**
  - Putting pressure on the **front end** of the compiler

- Changes in **computer hardware design**
  - Putting pressure on the **back end** of the compiler

Front End Challenges

- Increasingly complex software ➔ increasingly **complex bugs**
  - Most insidious are heisenbugs (data races, memory errors, …)
  - Challenge: Removing or detecting bugs through code analysis
- Increasingly sophisticated **security exploits**
  - Side-channel attacks (e.g. Spectre / Meltdown exploit)
  - Challenge: Protecting generated code from vulnerabilities
- New emerging applications such as **deep learning**
  - Different behavior compared to traditional apps
  - Challenge: Optimizing app using semantics of deep learning
- Explosive use of **scripting languages** (JavaScript, Python, R…)
  - Very flexible by design. E.g. Dynamic Typing:

    ```
    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

Back End Challenges

- Device physics are putting insurmountable barriers to performance
  – Power Wall: Barrier on CPU performance due to overheating
  – Memory Wall: Barrier on memory performance due to bandwidth
- 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, GPUs, TPUs, NPUs, Accelerators, etc …)
  – Challenge: Generating code for this type of architecture
- Memory efficiency must come from **software managed caches**
  – CPU caches lack understanding of software memory access patterns
  – Challenge: Generating code that manages caches in software using knowledge of program provided by programmer or analysis