

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

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 a Compiler?

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

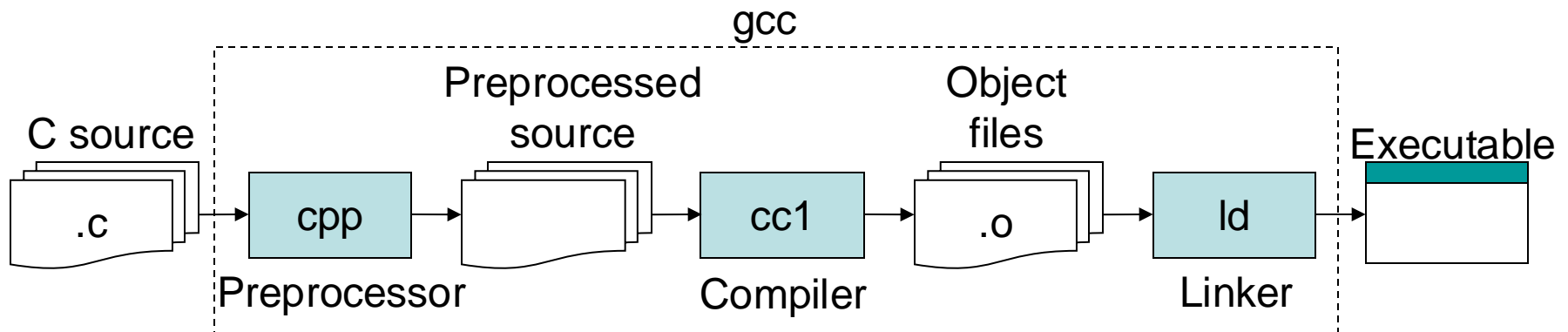
x	1
y	1
z	2

What is an Interpreter?

- Interpreter: A program that executes source code on the target machine by interpreting it **line by line**
 - + Portable
 - Source code runs anywhere where interpreter is installed
 - + Interpreter is simple to implement
 - Popular for languages with small user base
 - **Slow execution**
 - Source line interpreted on each execution (even in loop)

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



What is a Compiler?

- 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
 - Compiler is complex to implement
 - Need a big user base to expend the engineering effort
- + Fast execution
 - No interpretation needed during execution
 - Machine code can be optimized to be even faster

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)

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:



Microsoft
C#.net



python



OpenCL

Just-In-Time (JIT) Compiler

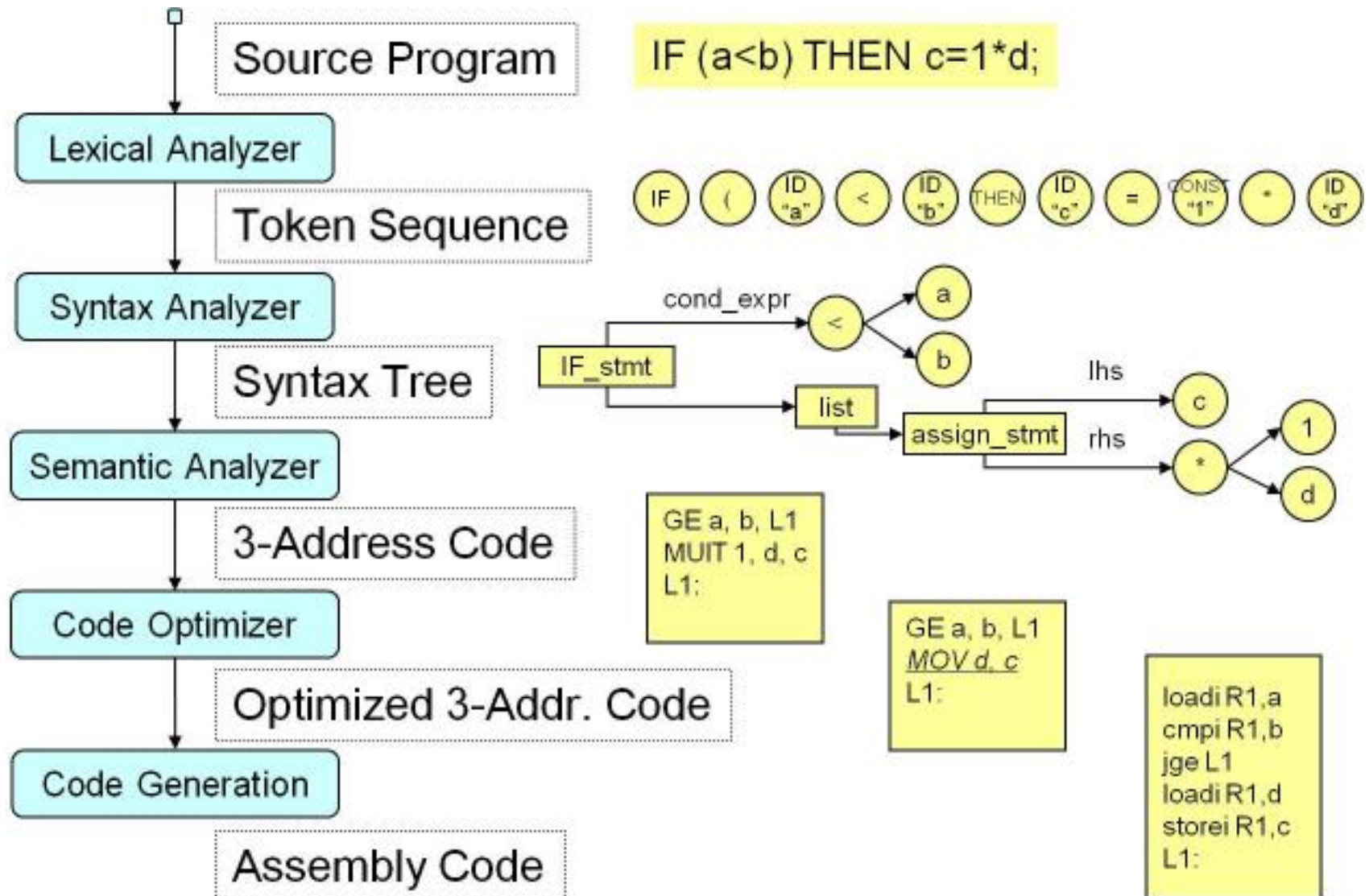
- Just-In-Time (JIT) Compiler: A compiler that performs translation at **runtime** (just in time before execution)
- Pros / Cons compared to AOT compiler
 - + As portable as interpreter
 - Code can run anywhere where there is a JIT compiler
 - + Almost as fast as (sometimes faster) than AOT compiler
 - **Slight overhead due to JIT compilation time**
 - But can sometimes produce better code than AOT
 - By profiling program behavior and optimizing (e.g. profile branch direction and optimize that path)

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, 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)

- **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. 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
 - 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 optimizations can be done in any sequence

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 $M \times N$ 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 that are putting new pressures on the compiler
- Changes in *computer hardware design* is producing new challenges in the back end of the compiler
- Changes in the *programming environment* is producing new challenges in the front end of the compiler

Back End Challenges

- Limits in device miniaturization is driving a sea change in HW
 - CPU frequencies scale no more due to power wall
- 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, Accelerators, etc ...)
 - Challenge: Generating code for this type of architecture
- Memory efficiency must come from **software managed memory**
 - 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

Front End Challenges

- 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
- New emerging applications such as **deep learning**
 - Different behavior compared to traditional apps
 - Challenge: Optimizing app using semantics of deep learning
- Increasingly complex software → increasingly **complex bugs**
 - Most insidious are heisenbugs (particularly data races)
 - 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