Language types and software development

Prof Rajeev Barua
CS 3101 -- Slide Set 1

PLAKSHA UNIVERSITY

Goal of this course

To understand the practical and scalable theory and methods needed to build a wide variety of program analysis tools, such as:

a. Compilers
b. Debuggers
c. Bug Detection tools
d. Application Performance Monitoring (APM) tools.
e. Memory Leak Detection tools
f. Vulnerability Detection tools
g. Malware Detection tools

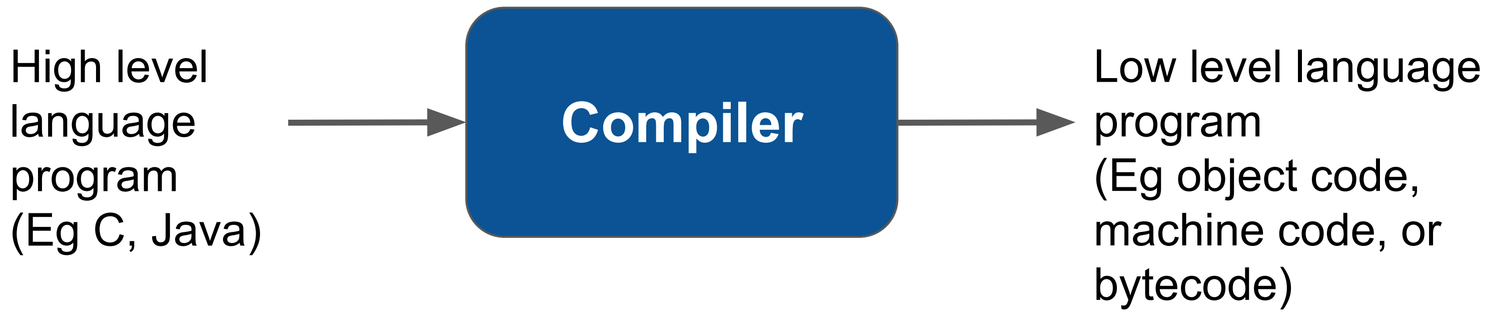
- This course focuses on what is needed to understand the practical theory needed by practitioners to build the tools above.
- Less focus on theory not needed by practitioners, because it is in portions of tools that you can use as a black box.

Topics include

1. Software development process
2. Program analysis needs of non-standard architectures.
3. Lexical analysis
4. Parsing
5. Abstract syntax
6. Bootstrapping
7. Stack memory: layout, usage, and compilation
8. Control flow analysis and optimizations
9. Traditional Dataflow analysis
10. Single Static Assignment (SSA)
11. Alias analysis

We will understand these as the course progresses.

What is a compiler?


High level language program (Eg C, Java)
Compiler
Low level language program (Eg object code, machine code, or bytecode)

- A compiler generally will need expertise in all of the previously mentioned topics to build.
- Other tools listed will need expertise in some of the topics listed, in especially topics 7-11.

Translation and interpretation

To execute a high-level language program, we have two main choices:

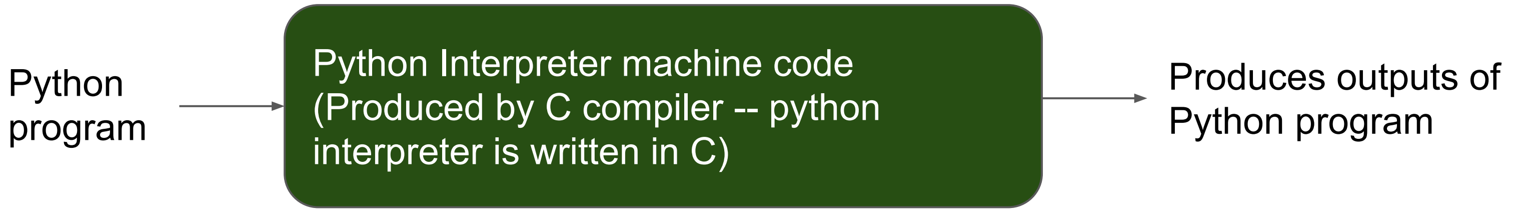
- Translation: to a low-level form (e.g., machine code) and run that directly on the hardware.
- Interpretation: On the fly execution of a high level program using a software program called an interpreter.

Eg:

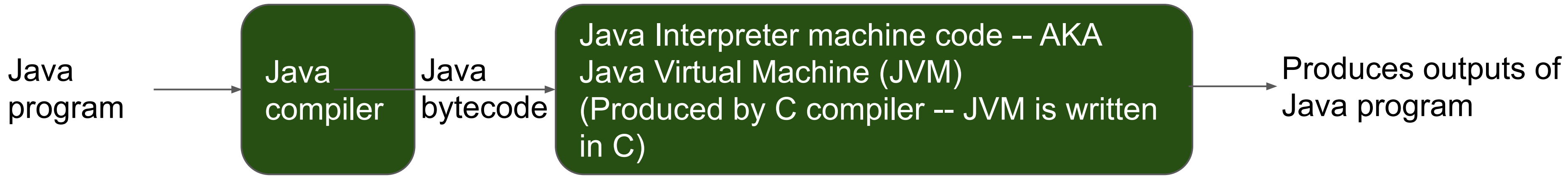
- C is compiled to machine code
- But Python is interpreted using its interpreter
- Java is a hybrid: it is compiled to Java bytecode, which is interpreted.

Example of interpreted languages

Pure interpreted language:


Python program
Python Interpreter machine code (Produced by C compiler -- python interpreter is written in C)
Produces outputs of Python program

Hybrid:


Java program
Java compiler
Java bytecode
Java Interpreter machine code -- AKA Java Virtual Machine (JVM) (Produced by C compiler -- JVM is written in C)
Produces outputs of Java program

No machine code is produced for input program in either case!
But interpreters and compilers are in machine code.

Interpreted vs. compiled languages

Advantages of Interpreted languages:

- Instrumentable
- Portable across platforms. *(In contrast, machine code only runs on a computer if its ISA and its OS match those of the computer.)*

(Used for internal corporate code and SaaS code)

Advantages of Compiled languages:

- Protect intellectual property (IP) by not revealing source code.
- Much faster (especially vs. Python)

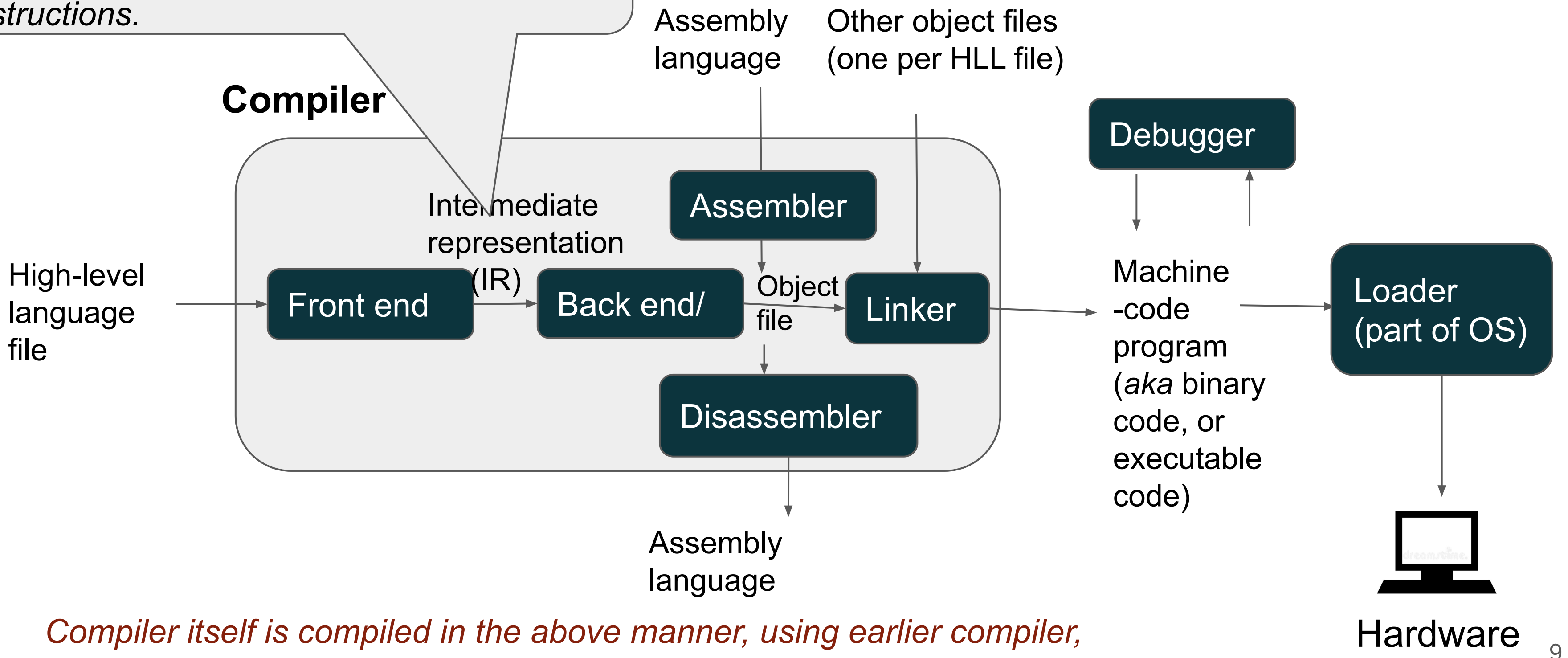
(Used for externally distributed software, or when speed is important.)

Important terms

- Machine code: The only type of code that can directly run on the CPU hardware.
- Instruction Set Architecture: The format of machine code instructions. Examples include:
  - **x86** (including **x86-32** and **x86-64**): used in Intel and AMD desktops, laptops, and servers.
  - **ARM**: Used in iOS and Android mobile devices
  - **MIPS**: Used in embedded devices including routers
  - **Sparc**: Used in SunOS computers
  - **PowerPC**: Used in video game console processors.
  - **z/architecture**: Used in IBM mainframes.
- Object code: the output of the compiler for one file. It has the format of machine code but lacks relocation and external symbol resolution.
- Assembly code: A human-readable ASCII text format representation of machine code.
- Assembler: A piece of software that converts assembly code into machine code.
  - Only needed if a developer wants to hand-write machine code directly without using a compiler.
- Disassembler: A piece of software that converts machine code into assembly language.
  - Needed if a developer wants to read machine code produced by a compiler.

Software development process (compilation

IR is language and ISA independent. I.e., any HL language is compiled to the same IR, and any IR is compiled into any ISA instructions.



Compiler itself is compiled in the above manner, using earlier compiler, or after that, using itself.

Linker and loader functionality

Tasks of the linker: It combines object files into a single executable file. It includes:

- Relocation: Adjusting addresses when files are combined by adding the size of previous files to the addresses in the current file being linked.
- External symbol resolution: Filling in the addresses of external symbols referenced in a file.
- This is also called static linking because any linked routines become part of the executable.

Tasks of the loader.

- Loads a machine code (i.e., binary) file from disk, and stores it in main memory.
- Performs dynamic linking, which fills in the addresses of dynamically linked libraries (aka DLLs) into programs. (E.g., the address of symbol **`printf()`** may be filled in from external library called **`stdio.h`**.)
- Finally, it starts execution of the binary file at its entry point (usually symbol **`_main`**).

Software development component details

Compiler: Convert program files into object code files.

Object code is similar to executable code, but has unresolved external symbol and library references.

Eg usage: `gcc -c file1.c -o file1.o`
//-c means do not link.
//-o means specify output file

Linker: Combines multiple obj files into single executable. Resolves above external references.

Eg usage:
```
gcc file1.o file2.o
                 -o file.exe
```
// Can also combine into one step:
```
gcc file1.c file2.c -o file.exe
```

Loader: Not part of compiler, but part of OS. It loads the executable into memory, allocates memory segments needed, and performs dynamic linking of libraries. In machines without virtual memory, it assigns relocation register.

Debugger: Not part of compiler, but often sold by compiler vendors as part of "Software development Kit" (SDK). SDK= compiler, linker, debugger, and a GUI interface for all of these (aka., an Integrated Development Environment (IDE)).

Needs compiler support to associate machine code entities (registers, memories, hex addresses) with high-level program entities (symbols, line numbers).

Eg: The optional "-g" flag in gcc produces an executable with debug information present. (absent by default).This information maps variables to registers and memory locations. An exe without debug info is called a “stripped binary”