Overview

Lecture 3 provided a general overview of the main steps (or *phases*) of a compiler and an interpreter. We also went over the three different types of interpreters, and the basic difference among compilers, interpreters, and transpilers. Before the next lecture, do the following. Note that we will go over in more detail what many of the steps/phases do in class over the semester.

1. ***Finish HW-0.*** Ensure your programming environment is set up and working (Java and Maven), obtain the HW-0 starter code, and work on HW-0. Note that the goal of HW-0 is for you to be ready to start on HW-1 as soon as it is posted (on Friday).
2. ***Know the MyPL Syntax from Lecture 2.*** Carrying over from Lecture 2, ensure you know the MyPL syntax and constructs. You will be tested on it!
3. ***Learn the phases.*** Be sure you can recreate the compilation and interpretation phases discussed in class. You should be able to recreate the diagram from class including what is passed between each step/phase.
4. ***Understand the ideas and tools below.*** More information than what was covered in the lecture are presented below.
5. ***Research language implementations.*** For each of the following languages, determine whether their standard implementations are compilation or interpretation based: JavaScript (V8), Python, Java, Kotlin, Swift, Go, Lua, R, C#, Haskell, OCaml, GNU C++, CLang C++, and Rust. Note that some language implementations support both interpretation and compilation (and even transpilers).

Additional Concepts and Terms

Write Once, Run Anywhere. In traditional compilation, the last step of machine-code generation is tightly coupled to the hardware (CPU) and the operating system. For example, `g++` and similar C/C++ compilers output executables that can only be run on the specific OS and hardware that the program is compiled on. This means that the program must be recompiled for each hardware/OS combination (Windows vs Linux vs Mac, Arm vs Intel, etc.). Often, to get programs to run correctly across different environments, small (or sometimes large) modifications must also be made to the program. This is often the case for larger-scale systems (e.g., web servers, databases, etc.). Alternatively, a selling point for many VM-based languages is that they do not require "recompilation" to be run on different machines. For example, running a program in Java involves two steps. The first step "compiles" a program to Java's bytecode format (via `javac`). The bytecode format, which is machine/OS independent, is then run using the JVM (via `java`). By implementing the JVM on many different hardware/OS combinations, it is possible to take the bytecode generated on one environment (hardware/OS combo) and then run it in a completely dif-

ferent environment. This ability is often referred to as *write once, run anywhere*. It was a major selling point for Java, and continues to be an advantage (especially as its optimization approaches have advanced and matured). Another popular approach is to use LLVM for back-end compilation (see below), in part, for similar reasons. Just-in-Time (JIT) compilation, as discussed in class, shares these same "write once, run anywhere" capabilities, and today, are largely used in place of a pure VM-based approach.

The LLVM compiler infrastructure. LLVM is a set of "tools" (libraries) that can be used as part of back-end programming language implementation. The LLVM infrastructure primarily consists of an intermediate representation (IR), various optimizers, target (machine) code generators (for many CPUs), and a debugger. Languages that leverage LLVM as part of their implementation (see above) typically write all of the front-end steps and then implement a specific intermediate code generation step that then generates LLVM IR (from which various LLVM tools/libraries can be used). Thus, the backend steps of the language implementation are handled (and controlled) via LLVM libraries. CLang is a mature C/C++ implementation that uses LLVM (as opposed to the GNU compiler collection). More information about LLVM can be found here: https://llvm.org/. One of the features of LLVM is that once a program is represented in the IR, it is possible (via LLVM library calls) to perform JIT compilation. Because of the support provided by LLVM for its IR, LLVM also has become popular as an approach for supporting "write once, run anywhere" features.