Everything Is An Interpreter

Every piece of code that we write is an interpreter.

Compiled languages are always interpreted. In the case of compiled languages, there are two interpreters – one that runs at “compile time” and another that runs at “run time”. The compile-time interpreter is an app that pre-calculates some aspects of the program. The pre-calculated result is fed to another interpreter which runs the program to completion. It is assumed that the compile-time interpreter (“the compiler”) runs fewer times than the final app + runtime-interpreter.

The line between compile-time interpretation and run-time interpretation can be thought of as a slider control. We arbitrarily set that slider to a fixed value and CS education tries to teach us how to make the best trade-offs so that the app+runtime-interpreter run as efficiently as possible. The term “efficientcy” has many meanings – the main ones being speed and memory usage. Lately, we (the royal we) have been loading “type safety” into compile-time interpreters. Type checkers are interpreters that run during the phase called compile-time.

Typically, runtime interpretation is done by a hard-wired interpreter called the CPU. Compile-time interpreters reduce apps to something called assembly language. CPUs interpret assembly language,[1]

1 I generalize. I could go down this rabbit-hole deeper – assembler apps are reduced further into bits. CPUs interpret blocks of bits.