Language Implementations

Language Choice

By now you have worked with (at least) one functional language and (I assume) a couple of imperative languages. There are hundreds of others, some very weird.

Our next major topic: what are the important differences between them?

The question I'll be thinking of: if you need to choose a programming language for a project, how can you make that decision?

To answer that, you need to know what some of the features of the candidate languages are, and what they mean for the programs you write. What are the tradeoffs?

Some important things we won't talk much about:

- Popularity: how many people use the language? What is the community surrounding the language?
- Tools: is there a Windows IDE, etc.
- Libraries/toolkits: Rails vs Django vs Express, Unity vs Unreal vs Godot.
- Syntax: curly braces or indenting, required semicolons, etc.
- Implementation: how to write compilers.

Important things we will consider:

- How likely is it that a programmer's solutions are actually correct?
- How easy/fast is it for a programmer to solve problems?
- Do the resulting programs execute quickly?

Balancing those factors depends on the problem at hand. The last is easy to quantify, and therefore tempting to focus too much attention on.

Compilers & Interpreters

A *compiler* is a program that translates a program from one representation to another. Often, this is source code (the code the programmer types) to machine code.

Machine code: instructions for the processor. The target could be something other than machine code.

Assembly language: more human-readable but almost one-to-one with machine code.

The word “compiler” usually refers to translation from a higher- to lower-level representation.

Higher-level: more like what people want to write/understand.

Lower-level: more like what the computer understands/executes.

Converting between higher-level representations is *source-to-source translation* or *transpiling*.

This is common when converting to JavaScript for execution in a web browser, e.g. CoffeeScript, TypeScript, Dart.

Maybe also another language to C, to take advantage of the C compilers' optimization and portability.

Often languages are compiled to instructions for a *virtual machine*, a platform independent runtime environment like the Java Virtual Machine (*JVM*), .NET Common Language Runtime (*CLR*), etc.

Once compiled for the VM, it can be interpreted by the VM implementation. Compare: machine code which is sent directly to the processor.

Process virtual machine

An *interpreter* is a piece of software that takes a program in some language (other than machine code) and executes it.

This *could* be original source written by the programmer, but that's very unusual. More likely, some intermediate form for a virtual machine.

The speed of execution will depend on the implementation and form of input.

Most programs that appear to be interpreters are actually both a compiler and interpreter.

The source code is first compiled for a VM in memory, and that is then executed.

Whether the intermediate form is stored on disk or not doesn't really matter: compilation still happened.

i.e. There is no sense in which Java and C# are compiled, but Haskell, JavaScript, Python, or Ruby are not.

Virtual Machines

When compiling to a virtual machine, the result is generally called *bytecode*.

Bytecode is typically a very low-level representation of a program's logic designed to be interpreted quickly. It looks more like assembly/machine code than a high-level language.

In other words: a virtual machine is an interpreter for bytecode.

e.g. a Java `.class` file is bytecode for the Java Virtual Machine and can be disassembled.

e.g. Python bytecode can be inspected as well.

Bytecode interpreters (virtual machines) are generally slower than direct execution of machine code.

Any interpreter has some overhead when converting bytecode to actual machine instructions. This must be done throughout the execution, so there's some overhead and we expect slower execution.

Compiler Optimization

Compilers can do many things to *optimize* the code they generate. They generally optimize for speed, but maybe also for size/memory.

Optimization can be done when producing either machine code or bytecode. Both can be made better by analysis of the program.

Wikipedia: Optimizing compiler

Optimization is often off by default: check your compiler's command line options.

A compiler can do many things to optimize. e.g.

- Move local variables from memory to registers.
- Move logic out of a loop: loop-invariant code motion (code example).
- Remove dead code (code example).
- Eliminate tail calls.
- Reorder logic for the processor pipeline.

C compilers tend to have received more attention, so have more optimizations implemented.

We saw GHC do some optimizations on Haskell code, and it can do others.

- Strictly-evaluate expressions when results are definitely needed.
- Eliminate tail calls.

Compiler optimization is only limited by the program analysis that can be done.

The basic promise of the compiler: it will produce the *same result* as what you specified, not that it will do exactly what your code says.

Wikipedia: Program analysis

An optimizer has to unravel the details of your code to determine the “true” result you've asked for. Sometimes, it might be best to think of your code as a description of the results you want, not as a strictly imperative description of behaviour.

How good can optimizers be? Compare `gcc` and `clang` on summing integers with `-O2` optimization.

Conclusion: compiler writers think of your code as a specification of the result you want, not necessarily as a literal sequence of steps to follow.

Just-In-Time Compilation

Interpreting (byte)code at run-time always comes with a speed penalty. Even the most clever bytecode needs to be somehow processed while it's running: that takes instructions and therefore time.

To overcome this, some interpreters/VMs include a *just-in-time compiler*.

Wikipedia: Just-in-time compilation

A Just-In-Time compiler starts with bytecode and either…

1. interprets but when it decides that some piece of logic will be used frequently, it…
2. for all code…

… during execution, compiles that code *to machine code* and stores it in memory.

Then, when that code needs to execute, it can use the the machine code version.

Should be faster: often much faster. The cost: some memory and compilation work at run-time.

Languages traditionally thought of as “slow” are getting a lot of JIT-related attention. e.g. PyPy for Python, V8 and SpiderMonkey for

MS Research talk on JIT compilation and speed:

JavaScript.

Why Aren't More Users More Happy With Our VMs?

JITs have some advantages over Ahead-Of-Time (AOT) compilation. JITs can…

- optimize machine code for the specific CPU architecture;
- collect usage stats and optimize output for the way code is actually being used;
- create type-specific versions of functions for the types of arguments they actually receive (see “dynamic binding”, later).

But, JITs have to do their work during program execution.

That will slow things down (at least momentarily). Performance may be less predictable.

Language Implementations

Any programming language* could be compiled to machine code, or to a VM, or interpreted directly. Don't confuse the *usual* way a language gets treated with a rule about how it *must* be executed.

Compiling/interpreting/JIT-compiling are questions about the *tools*, not the *language*. Some examples…

* except some problematic features in some

To run C and/or C++ code, we could:

- Compile to machine code with GCC or Clang or ICC.
- Compile to LLVM bytecode with Clang and then interpret/JIT compile with GraalVM.
- Interpret with PicoC.

Java can be executed by:

- Oracle Java or OpenJDK bytecode compilers and VM implementations.
- Compiled to machine code with GCJ (now defunct).
- Compiled to (JVM? LLVM?) bytecode and interpreted by GraalVM.

Python:

- Bytecode compiled and interpreted with the standard CPython implementation.
- Interpreted and JIT compiled with PyPy.
- Selectively JIT compiled with Parakeet, Numba.
- Ahead-of-time compiled to machine code with Pythran or Cython or Nuitka.
- Compiled to .NET CLR bytecode with IronPython or JVM bytecode with Jython.

Different implementations of a specific programming language can vary widely in performance. There are language design choices that can affect performance (more later), but tools have more of an effect than most people realize.

A Silly Benchmark

I have spent way too long on an example to illustrate this: Mandelbrot Set Language Shootout.

The code does a numeric calculation in many languages and compares the runtimes for many implementation.

Some lessons from that comparison:

- Some languages have *many* implementations beyond the default tools you'll find as a beginner. Do they all support exactly the same language features? Maybe. Maybe not.
- Different compilation paths and/or runtime environments have a huge effect on the way they execute.

- JIT compiler creators are doing amazing work. Meta-tools like GraalVM, LLVM are helping.
- There's no clear distinction between fast and slow languages: it seems like any language can be “fast” with enough compiler cleverness.
- There seem to be some things about languages that make them slower *by default*.

Don't read too much into one benchmark. This is a tight loop with floating-point calculations: no strings, no integers, no arrays, etc.

Compare the PyPy benchmarks which is a much more comprehensive benchmark suite.