How Programs Run and GCC Internals

We can divide how we execute programs into two main categories: **interpreters** and **compilers**. There also exists a compromise between these two approaches that uses **byte-code**.

Interpreters

The **interpreter** itself is some program (compiled from some language like C in the case of CPython).

BASIC IDEA: Take the source code, check it a bit (validating syntax, etc.), then translate it as quickly as possible into an intermediate form on RAM.

Suppose you have this Python source code:

```
def dist(x, y)
    return sqrt(x*x + y*y)
```

One approach for the translated form can be some symbol table with symbols pointing to tree structures, which are representations of the source code divided into elementary parts, like the function name, the arguments, the return expression, etc.

The interpreter will then walk through the data structure and evaluate the nodes. It's as if the source code of the interpreter has some kind of function like:

```
value_t evalexpr(tree_t t) {
  switch (t->type) {
    case PRINT_STMT:
      value_t v = evalexpr(t->printargs);
      printf("%s", v);
    /* ... */
  }
}
```

Notice that the interpreter needs to be highly recursive, evaluating sub-expressions before completing a given expression.

Many interpreters use this approach, but due its recursive nature, it's a lot slower compared to machine code.

- ➕ Usually very flexible and easy to read and write.
- ➕ Very short translation time, allowing you to update and run code inconsequentially.
- ➖ Much slower at runtime compared to byte-code or machine code.
- ➖ No compiler means no compile-time checking, so errors will occur at runtime.

Byte-code

With this approach, the program (typically still called an "interpreter") translates the source code into byte-code form. The `dist` symbol representing the function above then becomes like a pointer to some stream of operations:

```
push x
dup
*
push y
dup
*
*
+
sqrt
```

Because this is simply some byte stream, the interpreter can maintain some instruction pointer and *iterate* through the stream, executing the operations sequentially in similar fashion to true machine code. The C code now looks something like:

```c
switch (*ip++) {
  case DUP:
    top = *sp;
    *--sp = top;
    break;
  /* ... */
}
```

We've seen this exact concept back when we learned about Emacs Lisp byte-code compilation. The tree structure is like the `list` and `cons` model, and the byte-code is a sequential byte stream.

Other languages that use this approach include Python and, although not required by the spec, JavaScript on most browsers. Bash may also be another one.

- ➕ This form also produces *smaller code*, which fits into the data cache better. Tree data structures take up more space because they have a bunch of pointers.
- ➕ This form is also more *performant*. There's no recursion, simply iteratively referencing/derefencing pointers.
- ➖ Byte-code takes longer to produce compared to tree structures in fully interpreted languages.
- ➖ The tree structure approach is also more debuggable because you can trace through the tree to find what went wrong.

Compilers

Abstractly, like a *function* that converts source code to machine code such that if you execute the machine code, the behavior will be identical to what is expected by the source code language spec.

- ➖ Translation time is large.

- ➖ Much less debuggable. Machine code is not human-readable, so we typically have to resort to debuggers like GDB.
- ➕ Much more performant at runtime. Compiled languages skip the necessity of `switch`-like code or even a data stack;they can keep their data in registers instead of plain old RAM.
- ➕ Compilers help catch errors at compile-time via mechanisms like warnings and static checking.

JIT Compilation

A typical optimization that's done these days is like a compromise between byte-code and compilers, the concept of **runtime compilers** (aka **just-in-time (JIT) compilers**).

What it does is produce byte-code and/or possibly even tree structures, but the compiler will also have a subroutine that can compile that byte-code into machine code. The basic idea is to compile only the code that gets run a lot.

- ➕ You get the developer benefits of writing in a more flexible language with the performance benefits close to that of byte-code/machine code.
- ➕ The code "speeds up" as it runs, compiling as it goes. The subroutine can even run in a different thread.
- ➖ You need to write a subroutine for every machine the code will be run on because machine code is machine-dependent.

Why bother with traditional compilers like those for C/C++ if JIT is so awesome?

One catch is, who's going to write the subroutine? Another catch is, who's going to run that code?

JIT interpreters thus have some software overhead because they need to embed a compiler like GCC inside themselves.

Compiler Portability

There are three "platforms" of concern, with specific vocabulary:

1. **"Host":** What platforms can GCC **run** on? e.g. x86-64, ARM, RISC-V, etc.
2. **"Target":** What platforms can GCC generate code **for**? You can run GCC on one architecture to generate code for another platform, so we have this distinction.
3. **"Build":** What platforms do you **build** GCC itself on?

If one wanted to build Emacs/Python, one needs to write source code in C that then translates it to machine code.

At the lower level, GCC can compile itself. The most popular compiler to build GCC is GCC itself.

Chicken or the egg? Circularity?

The first GCC compiler was compiled with an earlier C compiler, which in turn was compiled with assembly language. If you keep going back in time, people bootstrapped from machine code, working at the level of opcodes while toggling physical switches.

GCC Target Assumptions

- Flat address space, where every pointer is the same width and can address any part of memory
- Support for 8-bit bytes and 16-bit words. 32-bit, 64-bit, etc. are technically not required.

GCC Source Code

You want to modularize your compiler. You don't want to have a separate compiler for each platform. The GCC source code can be split up into machine-independent and machine-dependent parts. The latter would have separate modules for RISC-V, ARM, x86-64, etc. The majority of the code is shared code that can be consulted on any architecture.

There's also the concern of language support. GCC can compile C, C++, Java etc., so within the machine-independent part of the code, a portion is modularized into language-dependent code.

The machine-independent, language-independent part of the source code can be thought of as the "heart"/"core" of GCC. Ideally, you want as much of the source code as possible to be in this category.

```
+------+-------+--------+
|  C   |       | RISC-V |
+------+  GCC  +--------+
| C++  | heart |  ARM   |
+------+       +--------+
| Java |       | x86-64 |
+------+-------+--------+
(machine-indep) (machine-dep)
```

Files

The heart of GCC are a bunch of `.cc` files:

```
foo.cc
fob.cc
...
```

Then there a bunch of **machine description files**:

```
x86-64.md
arm.md
...
```

These describe the machine's registers, instructions that operate on the registers, what the instructions do, etc. The descriptions are Lisp-ish but also kind of a hybrid between ASM and C++.

For each platform, there's a `.cc` file like `x86-64.cc` that in effect *implements* the machine description so that the heart of GCC can consult the classes/methods defined by the `.cc` to figure out some hardware details.

The `.cc` specifies a bunch of subroutines to be written. There's a program written by the GCC maintainers that translates the `.md` files to `.cc` files.

This is an example of **project-specific tooling**.