Why Does Bootstrapping Work?

- Fundamentally, because ”data” and “code” are the same.
  - They are just different interpretations applied to bit-strings.

Why Does Bootstrapping Work?

- Fundamentally, because ”data” and “code” are the same.
  - They are just different interpretations applied to bit-strings.
- The compiler is special in the way that it manipulates this data-code duality.
  - When the compiler generates code, it produces and combines bit-strings that become its output data.
  - But hardware interprets these same bit-strings as instructions when executing the generated program.

Why Does Bootstrapping Work?

- Fundamentally, because ”data” and “code” are the same.
  - They are just different interpretations applied to bit-strings.
- The compiler is special in the way that it manipulates this data-code duality.
  - When the compiler generates code, it produces and combines bit-strings that become its output data.
  - But hardware interprets these same bit-strings as instructions when executing the generated program.
- This is a very deep property of the von Neumann architecture that is also the source of many security holes.
- We will study this concept with a concrete example taken from Ken Thompson’s 1983 Turing Award Lecture *Reflections on Trusting Trust*.

Starting Point

- We have a native compiler cc_1 for a subset of C that recognizes only some of the ASCII escape sequences for non-printing characters.
  - We want to extend our language definition to accept the escape `'\v'` for vertical tab.
  - The routine that recognizes ASCII escape sequences and converts them to characters is in the lexer.

```
int convert(void) {
    int c = nextchar();
    if (c != '\\') return c;
    c = nextchar();
    if (c == '\\') return '\\';
    if (c == 'n') return '\n';
    error();
}
```

Attempt #1

```
int convert2(void) {
    int c = nextchar();
    if (c != '\\') return c;
    c = nextchar();
    if (c == '\\') return '\\';
    if (c == 'n') return '\n';
    if (c == 'v') return '\v';
    error();
}
```

- This doesn't work.
- It doesn't even compile with our existing native compiler cc_1.
- Why?

Attempt #2

```
int convert3(void) {
    int c = nextchar();
    if (c != '\\') return c;
    c = nextchar();
    if (c == '\\') return '\\';
    if (c == 'n') return '\n';
    if (c == 'v') return 11;
    error();
}
```

- This works just fine.
- Why?

- We now have a new native compiler cc_2 that compiles programs written in this “extended” C language.

One Final Plot Twist

- The routine `convert2` is written in the extended C language.
  - So we can compile the compiler source with cc_2 but with `convert2` instead of `convert3`.
  - This time, the compilation succeeds.
  - We can now forever discard `convert3`.

One Final Plot Twist

- The routine `convert2` is written in the extended C language.
  - So we can compile the compiler source with cc_2 but with `convert2` instead of `convert3`.
  - This time, the compilation succeeds.
  - We can now forever discard `convert3`.

- In Thompson's words:

  This is a deep concept. The original source code of `convert` *is amazing in that it "knows" in a completely portable way what character code is compiled for a new line in any character set. This act of knowing then allows it to recompile itself, thus perpetuating the knowledge. When we wanted to add the knowledge of the new escape sequence, we had to "train" the compiler. We had to do this once and once it "learned" the concept, we could use the self-referencing definition.*