Writing a Modern Compiler In Ruby

By Drew Ingebretsen

What is a compiler?

Turns This...

```ruby
def fizzbuzz(number)
  divisibleBy3 = (number % 3 == 0)
  divisibleBy5 = (number % 5 == 0)

  case
  when divisibleBy3 && divisibleBy5
    puts "FizzBuzz"
  when divisibleBy3
    puts "Fizz"
  when divisibleBy5
    puts "Buzz"
  else
    puts number
  end
end

(1..100).each {|n| fizzbuzz n}
```

Into This

```asm
xor    CX, CX

main_loop:
   inc    CX
   cmp    CX, 100
   jg     done

   fizzbuzz_check:
      mov    AX, CX
      mov    BH, 15
      div    BH
      cmp    AH, 0
      je     print_fizzbuzz
      jmp    fizz_check

      print_fizzbuzz:
          PutStr    fizzbuzz_lbl
          nwln
          jmp       main_loop

   fizz_check:
      mov    AX, CX
      mov    BH, 3
      div    BH
      cmp    AH, 0
      je     print_fizz
      jmp    buzz_check
```

What is a compiler?

- A compiler is software which translates a source programming language into a destination programming language.
- An interpreter translates a source programming language into a desired *output.*
- Transpilers are a type of compiler that translates a source programming language to a destination programming language of similar abstraction.

How about Ruby?

- Does the MRI compile or interpret Ruby?
- *Both.* Most modern languages, including MRI Ruby >= 1.9, use a popular hybrid approach known as JIT (Just in Time) Compilation.
- JIT *compiles* the language to an intermediate language called byte code, (using YARV) then *interprets* the byte code.
- Because of the blending of the two concepts over the last 20 years, many people use the term ‘compiler’ to mean software that creates a binary that can be run directly by an operating system. This is not correct
- People say Ruby is not slow because it’s not compiled. This is not correct.

Why write a compiler?

- It’s fun. The engineering challenges you find in compilers are some of the most unique and most fun challengers you’ll run into.
- It’s useful. Several times I’ve had to parse or implement some sort of language to describe a task and knowledge of compilers is useful for these tasks.
- It’s sexy. Everyone knows the most attractive people in software are those who can write a programming language.

Let’s Write a Compiler!

Source Language: Drewby

- A (very) simple programming language and compiler created for this presentation.
- Source code available at http://github.com/drewying/drewby
- JavaScriptish syntax.
- Has a single 64-bit integer variable type, an if statement, a while statement, a print statement, and basic expressions.
- That's it. No functions, objects, arrays, or anything else.
- All memory is allocated on the stack. No heap memory allocation.
- The Drewby compiler has a basic virtual machine which processes an intermediate byte code.

Destination language: x64 Machine Code

- Also commonly known as assembly code.
- The lowest level programming you can do, mimics hardware more than any other programming language.
- Only has jumps. (Gotos). No loops.
- No variables, just memory locations.
- No statements, only operations.
- All operation inputs/ouputs pass through what are called registers.
- Unique to each CPU. The CPU in your laptop has different assembly code from the CPU in your phone from the CPU in your router.

Drewby Example

```
a = 1;
b = 0;
tmp = 0;
count = 0;

while (count < 20) {
 tmp = a;
 a = a + b;
 b = tmp;
 print tmp;
 count = count + 1;
};
```

```
mov [rsp + 0], 1
mov [rsp + 8], 0
mov [rsp + 16], 0
mov [rsp + 24], 0
mov r10, [rsp + 24]
mov r9, 20
cmp r10, r9
jg BB34
mov r14, [rsp + 0]
mov [rsp + 16], r14
mov r13, [rsp + 0]
mov r12, [rsp + 8]
add r13, r12
mov [rsp + 0], r13
mov r10, [rsp + 16]
mov [rsp + 8], r10
```

Stages of Our Compiler

- *Tokenize* the source code into an input stream.
- *Parse* the input stream into an Abstract Syntax Tree
- Write out the Abstract Syntax Tree out to byte code.
- Convert the byte code into machine code.
- Assemble the machine code into a binary.

Step 1 - Tokenization

- Breaking your code into tokens that can be easily parsed by the parser
- A token is an abstract concept of different parts of source code
- Example tokens would be keywords, operation symbols, or the rule that any combination of letters is a variable token
- Tokens defined via regex
- Lex/Rexical

Tokenization Example

```
while b != 0
  if a > b
    a = a - b
  else
    b = b - a
```

WHILE VARIABLE NOT EQUALS CONSTANT

IF VARIABLE GREATER_THAN VARIABLE

VARIABLE EQUALS VARIABLE MINUS VARIABLE

ELSE

VARIABLE EQUALS VARIABLE MINUS VARIABLE

```
WHILE
VARIABLE
NOT
EQUALS
CONSTANT
IF
VARIABLE
GREATER_THAN
VARIABLE
VARIABLE
EQUALS
VARIABLE
MINUS
VARIABLE
ELSE
VARIABLE
EQUALS
VARIABLE
MINUS
VARIABLE
```

Step 2 - Parsing

- Breaking up the stream of tokens into a logical structure which shows the order of instructions.
- This structure is called an Abstract Syntax Tree because it can be navigated as a tree.
- Memory loads and constant variables will be the leaves of the tree. Expressions and statements will be branches. This makes it really easy to parse using a DFS
- Defined using BNF Grammars.
- Lots of different ways to Parse
- YACC/RACC

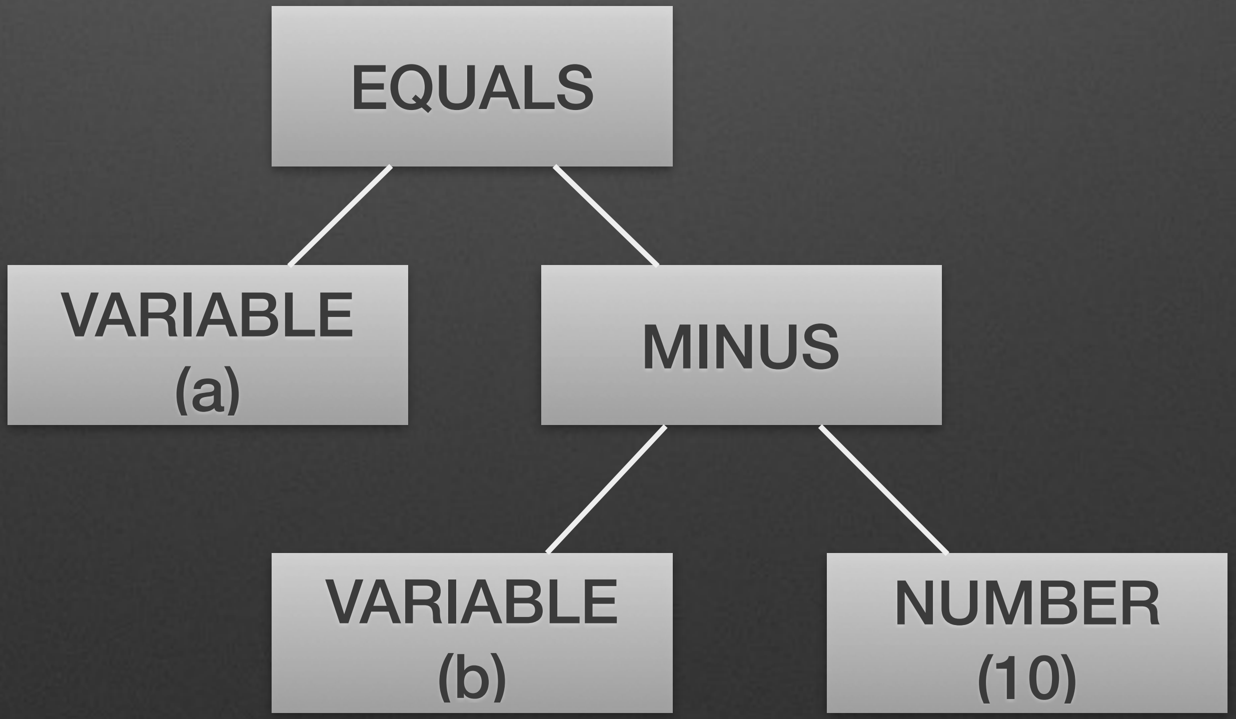
Parsing Example

a = b - 10

VARIABLE
EQUALS
VARIABLE
MINUS
NUMBER

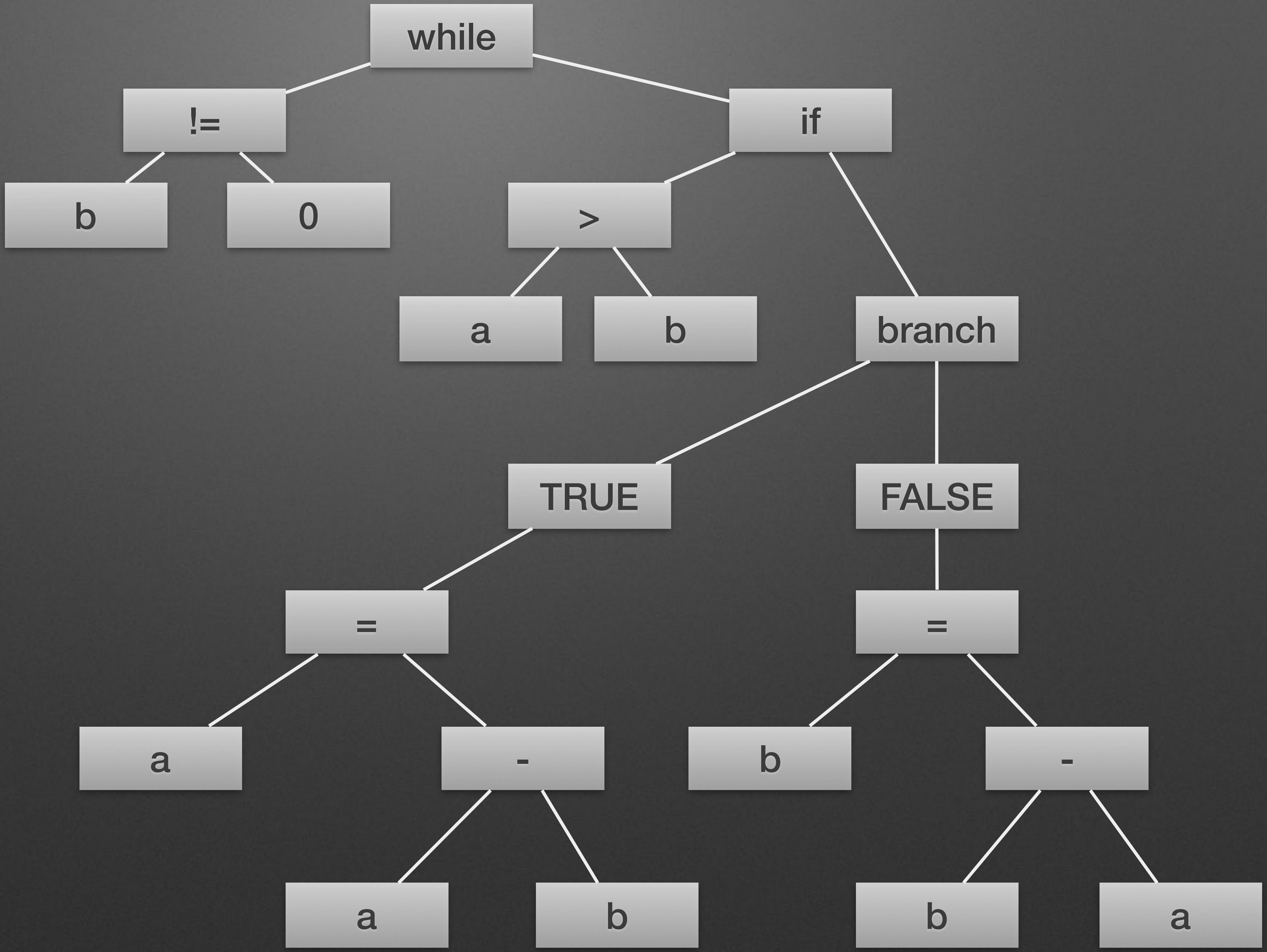
```
statement : VARIABLE EQUALS expression

expression :  expression MINUS expression
            | VARIABLE
            | NUMBER
```



Parsing Example 2

```
while b != 0
  if a > b
    a = a - b
  else
    b = b - a
```



Symbols and Tables Oh My!

- While parsing and creating the syntax tree it’s very common to create a symbol table, also known as a variable table.
- In machine code there are no variables, only locations in memory. The symbol table tracks variable names with their location in memory.
- In advanced compilers the symbol table can also be responsible for things like managing memory allocations (garbage collection) and custom types.
- Other popular things done at this step are function tables, string tables, object graphs, type graphs, etc.

Symbol Table Example

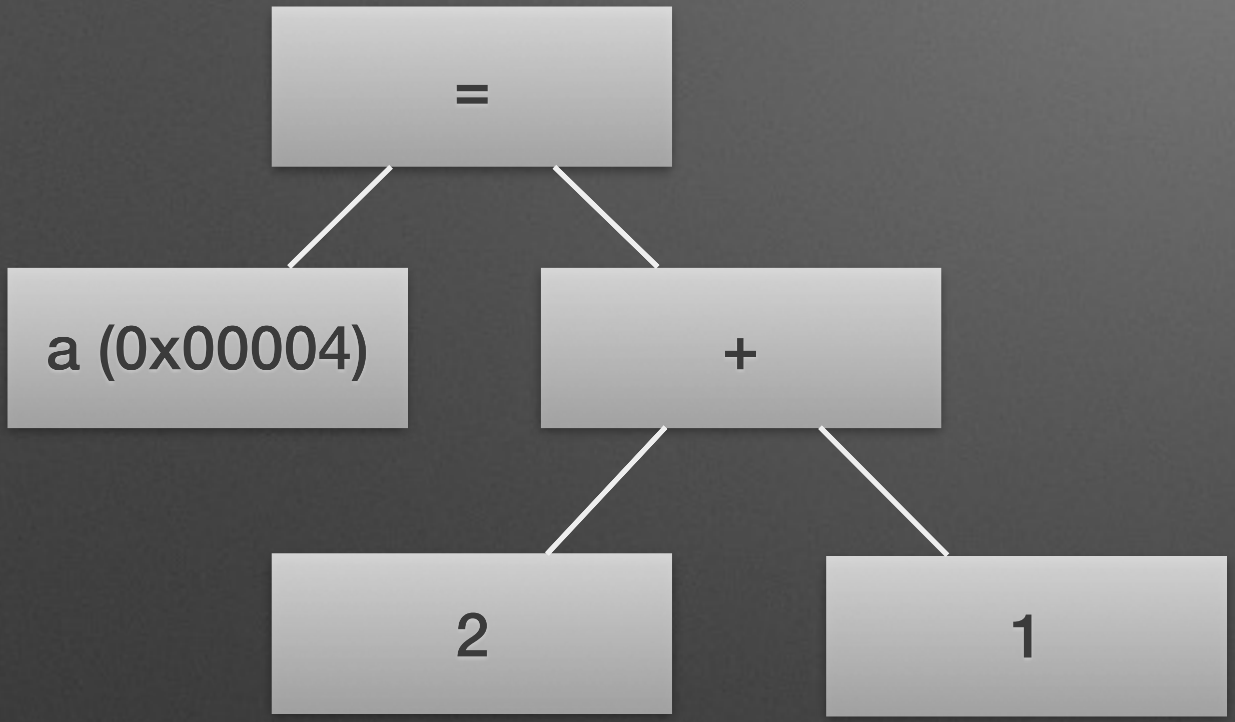
VARIABLE NAME	MEMORY LOCATION	TYPE	SIZE
FOO	0x00024H	INTEGER	4
BAR	0x00028H	FLOAT	4
FOO_ARRY	0x00032H	ARRAY(INTEGER)	24
BAR_S	0x00056H	STRUCT	12

Step 3 - Byte Code

- Once we have the abstract syntax tree we then write it out to byte code.
- Byte code is a special type of programming language code which is made to resemble a generic assembly language.
- Or you could call it machine code for a machine that isn’t real, aka a virtual machine.
- Byte code is typically organized into control units called Basic Blocks.
- JVM, YARV
- Easily parsed from an AST doing a DFS.

Byte Code Example

a = 1 + 2



YARV ByteCode

```
[ 4] a
0000 trace
0002 putobject 1
0004 putobject 2
0006 opt_plus <ic:1>
0008 dup
0009 setlocal a
```

DVM ByteCode

```
load_constant 0 1
load_constant 1 2
add 2 0 1
write_memory 0x00004 2
```

Step 4 - From Byte Code to Machine Code

- Once we have byte code we then convert it to machine code.
- This is typically a really simple process, since byte code is so close to machine code.
- The majority of compiler optimizations also happen at this step. Common compiler optimizations include loop unrolling, function inlining, local value numbering, common subexpression, superblock cloning, and MANY MANY more. (Seriously your code looks nothing like what you wrote).
- JIT languages will often interpret the byte code (using a “Virtual Machine”) instead of writing out machine code.

Assembly Code Example

```
load_constant 0 1             MOV RAX, 1
load_constant 1 2             MOV RBX, 2
add 2 0 1                     ADD RAX, RBX
write_memory 0x00004 2        MOV RSP + 4, RAX
```

Step 5 - Assembling/Linking

- Once we have the assembly code we “assemble” it.
- Assembling is basically a dictionary lookup of op codes/ register addresses to the binary numbers equivalent.
- Once we have a binary stream of our assembly, we link the binary. This wraps the binary into a file that contains information the OS needs to run the file.
- Information such as virtual memory mappings, dynamic library hooks, etc.
- Done with the commands nasm/ld

Assembling Example

```
mov [rsp + 0], 1
mov [rsp + 8], 0
mov [rsp + 16], 0
mov [rsp + 24], 0
mov r10, [rsp + 24]
mov r9, 20
cmp r10, r9
jg BB34
mov r14, [rsp + 0]
mov [rsp + 16], r14
mov r13, [rsp + 0]
mov r12, [rsp + 8]
add r13, r12
mov [rsp + 0], r13
mov r10, [rsp + 16]
mov [rsp + 8], r10
```

```
0000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
0000010 0c 00 00 00 e0 02 00 00 85 00 00 00 00 00 00 00
0000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
0000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 19 00 00 00 98 00 00 00
0000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
0000080 00 10 00 00 00 00 00 00 00 10 00 00 00 00 00 00
0000090 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
00000a0 07 00 00 00 05 00 00 00 01 00 00 00 00 00 00 00
00000b0 5f 5f 74 65 78 74 00 00 00 00 00 00 00 00 00 00
00000c0 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
00000d0 ff 1e 00 00 00 00 00 00 01 01 00 00 00 00 00 00
00000e0 ff 0e 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000f0 00 04 00 80 00 00 00 00 00 00 00 00 00 00 00 00
0000100 19 00 00 00 48 00 00 00 5f 5f 4c 49 4e 4b 45 44
0000110 49 54 00 00 00 00 00 00 00 20 00 00 00 00 00 00
0000120 00 10 00 00 00 00 00 00 00 10 00 00 00 00 00 00
0000130 00 02 00 00 00 00 00 00 07 00 00 00 01 00 00 00
0000140 00 00 00 00 00 00 00 00 22 00 00 80 30 00 00 00
0000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Questions?