Understanding V8's Bytecode

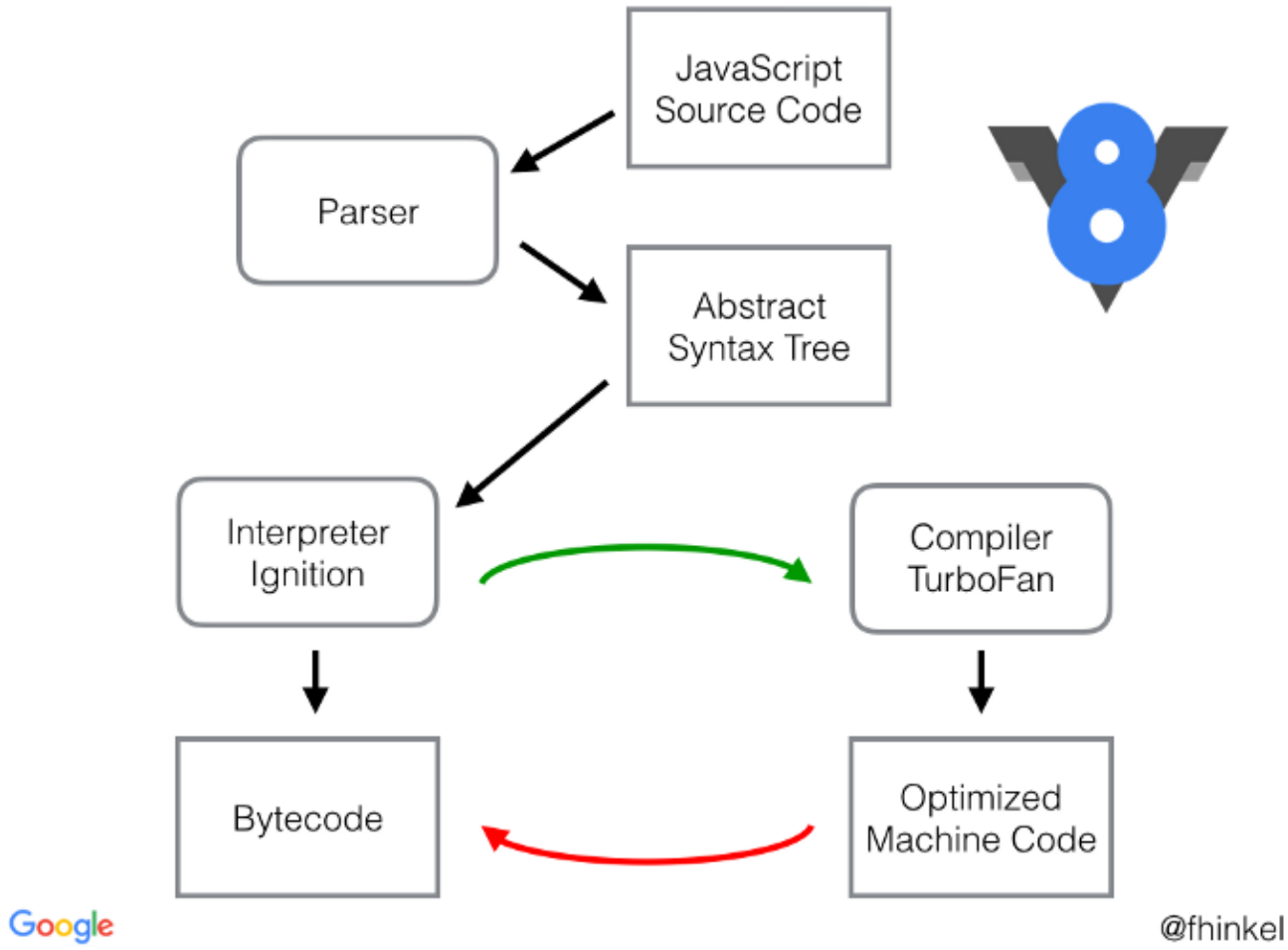
Franziska Hinkelmann
Aug 16, 2017 · 5 min read

V8 is Google's open source JavaScript engine. Chrome, Node.js, and many other applications use V8. This article explains V8's bytecode format — which is actually easy to read once you understand some basic concepts.

This post is available in Chinese, translated by justjavac.

Ignition! We have lift-off! Interpreter Ignition is part of our compiler pipeline since 2016.

When V8 compiles JavaScript code, the parser generates an abstract syntax tree. A syntax tree is a tree representation of the syntactic structure of the JavaScript code. Ignition, the interpreter, generates bytecode from this syntax tree. TurboFan, the optimizing compiler, eventually takes the bytecode and generates optimized machine code from it.



V8's compiler pipeline

If you want to know why we have two execution modes, you can check out my video from JSConfEU:

Franziska Hinkelmann: JavaScript engines - how do they even? | JSConf E...

Bytecode is an abstraction of machine code. Compiling bytecode to machine code is easier if the bytecode was designed with the same computational model as the physical CPU. This is why interpreters are often register or stack machines. **Ignition is a register machine with an accumulator register.**

You can think of V8's **bytecodes as small building blocks** that make up any JavaScript functionality when composed together. V8 has several hundred bytecodes. There are bytecodes for operators like `Add` or `TypeOf`, or for property loads like `LdaNamedProperty`. V8 also has some pretty specific bytecodes like `CreateObjectLiteral` or `SuspendGenerator`. The header file bytecodes.h defines the complete list of V8's bytecodes.

Each bytecode specifies its inputs and outputs as register operands. Ignition uses registers `r0, r1, r2, ...` and an accumulator register. Almost all bytecodes use the accumulator register. It is like a regular register, except that the bytecodes don't specify it. For example, `Add r1` adds the value in register `r1` to the value in the accumulator. This keeps bytecodes shorter and saves memory.

Many of the bytecodes begin with `Lda` or `Sta`. The **a** in `Lda`**a** and `St`**a** stands for **a**ccumulator. For example, `LdaSmi [42]` loads the Small Integer (Smi) `42` into the

accumulator register. `Star r0` stores the value currently in the accumulator in register `r0`.

So far the basics, time to look at the bytecode for an actual function.

```
function incrementX(obj) {
  return 1 + obj.x;
}

incrementX({x: 42});  // V8's compiler is lazy, if you don't run a
function, it won't interpret it.
```

> *If you want to see **V8's bytecode of JavaScript code**, you can print it by calling D8 or Node.js (8.3 or higher) with the flag `--print-bytecode`. For Chrome, start Chrome from the command line with `--js-flags="--print-bytecode"`, see Run Chromium with flags.*

```
$ node --print-bytecode incrementX.js
...
[generating bytecode for function: incrementX]
Parameter count 2
Frame size 8
  12 E> 0x2ddf8802cf6e @    StackCheck
  19 S> 0x2ddf8802cf6f @    LdaSmi [1]
        0x2ddf8802cf71 @    Star r0
  34 E> 0x2ddf8802cf73 @    LdaNamedProperty a0, [0], [4]
  28 E> 0x2ddf8802cf77 @    Add r0, [6]
  36 S> 0x2ddf8802cf7a @    Return
Constant pool (size = 1)
0x2ddf8802cf21: [FixedArray] in OldSpace
 - map = 0x2ddfb2d02309 <Map(HOLEY_ELEMENTS)>
 - length: 1
           0: 0x2ddf8db91611 <String[1]: x>
Handler Table (size = 16)
```

We can ignore most of the output and focus on the actual bytecodes. Here is what each bytecode means, line by line.

LdaSmi [1]

`LdaSmi [1]` loads the constant value `1` in the accumulator.

Star r0

Next, `Star r0` stores the value that is currently in the accumulator, `1`, in the register `r0`.

```
LdaNamedProperty a0, [0], [4]
```

`LdaNamedProperty` loads a named property of `a0` into the accumulator. `ai` refers to the i-th argument of `incrementX()`. In this example, we look up a named property on `a0`, the first argument of `incrementX()`. The name is determined by the constant `0`. `LdaNamedProperty` uses `0` to look up the name in a separate table:

```
- length: 1
           0: 0x2ddf8db91611 <String[1]: x>
```

Here, `0` maps to `x`. So this bytecode loads `obj.x`.

What is the operand with value `4` used for? It is an index of the so-called *feedback vector* of the function `incrementX()`. The feedback vector contains runtime information that is used for performance optimizations.

Now the registers look like this:

Add r0, [6]

The last instruction adds `r0` to the accumulator, resulting in `43`. `6` is another index of the feedback vector.

Return

`Return` returns the value in the accumulator. That is the end of the function `incrementX()`. The caller of `incrementX()` starts off with `43` in the accumulator and can further work with this value.

At a first glance, V8's bytecode might look rather cryptic, especially with all the extra information printed. But once you know that Ignition is a register machine with an accumulator register, you can figure out what most bytecodes do.

Learned something? Clap your 👏 to say "thanks!" and help others find this article.

> *Note: The bytecode described here is from V8 version 6.2, Chrome 62, and a (not yet released) version of Node 9. We always work on V8 to improve performance and memory consumption. In other V8 versions, the details might be different.*

Check out my blog for more things V8 and Node.js ✨🚅🚀✨

Thanks to Andreas Haas.

JavaScript V8 Compilers Bytecode Interpreters