Parsing JavaScript
- better lazy than eager?

Marja Hölttä - marja@chromium.org - Twitter: @marjakh

JSConfEU - May 2017

Outline

This talk is about V8, the JavaScript engine of Google Chrome.

The stuff I’m going to talk about is joint work with a ton of people hacking on the Parser.

Thanks to: Adam Klein, Addy Osmani, Caitlin Potter, Camillo Bruni, Daniel Ehrenberg, Daniel Vogelheim, Georg Neis, Jochen Eisinger, Toon Verwaest & others

0. **What** is parsing?

1. **Why** parsing matters?

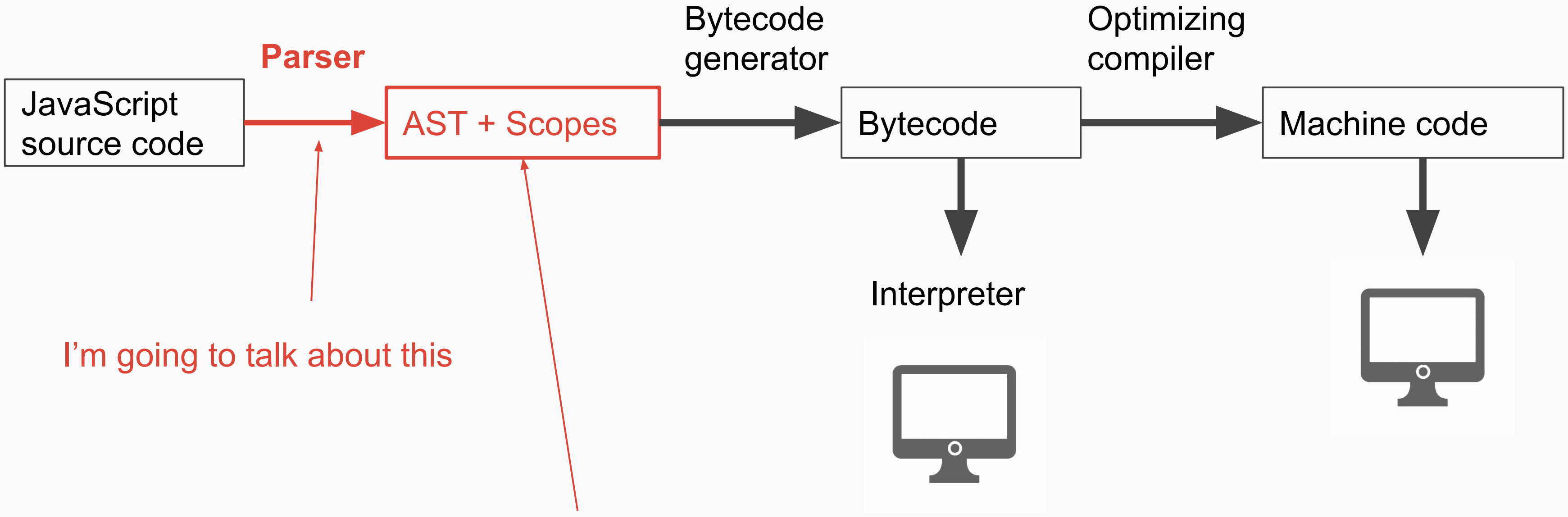
2. **How** V8 parses JavaScript?

3. What **you** (a web developer) can do to help V8 parse better?

WHAT?

0.
WHAT?

V8 overview



Parser

JavaScript source code

AST + Scopes

Bytecode generator

Bytecode

Optimizing compiler

Machine code

Interpreter

I’m going to talk about this

Structured representation of the source code

AST = Abstract Syntax Tree

```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = “bar”;
    return a;
  }
}
```

AST = Abstract Syntax Tree

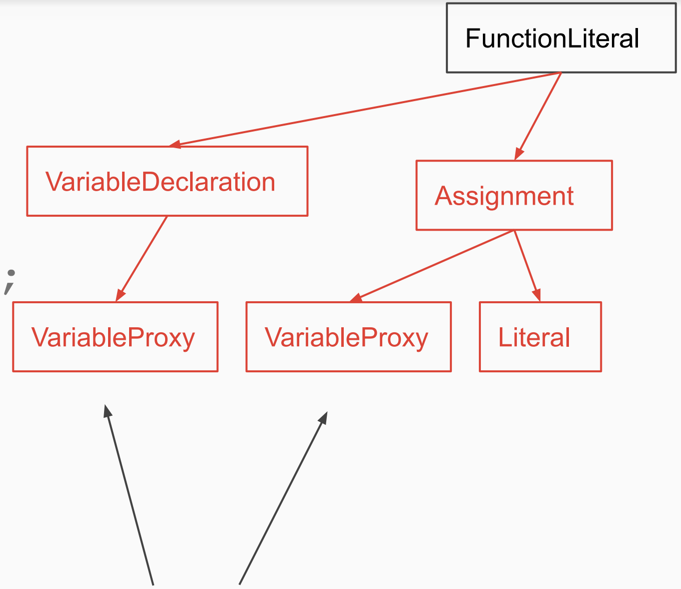
```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = “bar”;
    return a;
  }
}
```

FunctionLiteral

0.
WHAT?

AST = Abstract Syntax Tree

```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = "bar";
    return a;
  }
}
```

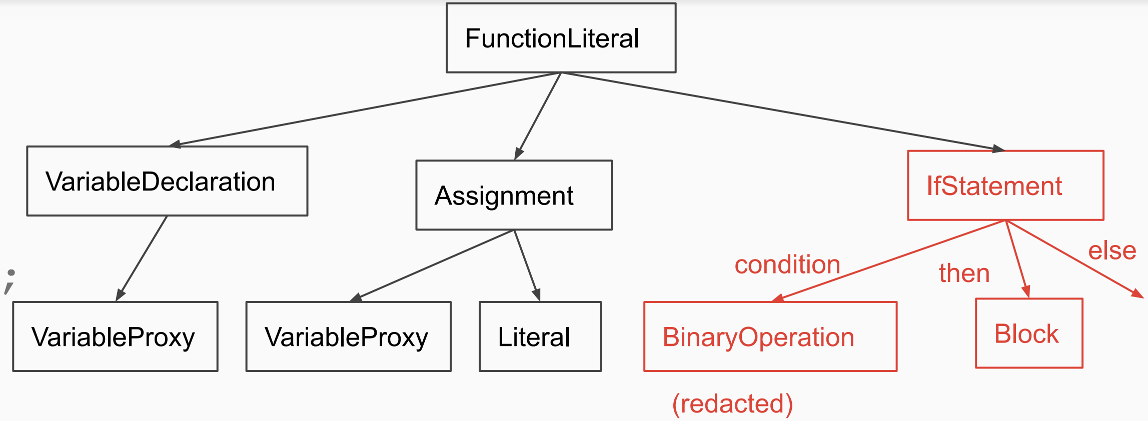


Represent a reference to a variable

0.
WHAT?

AST = Abstract Syntax Tree

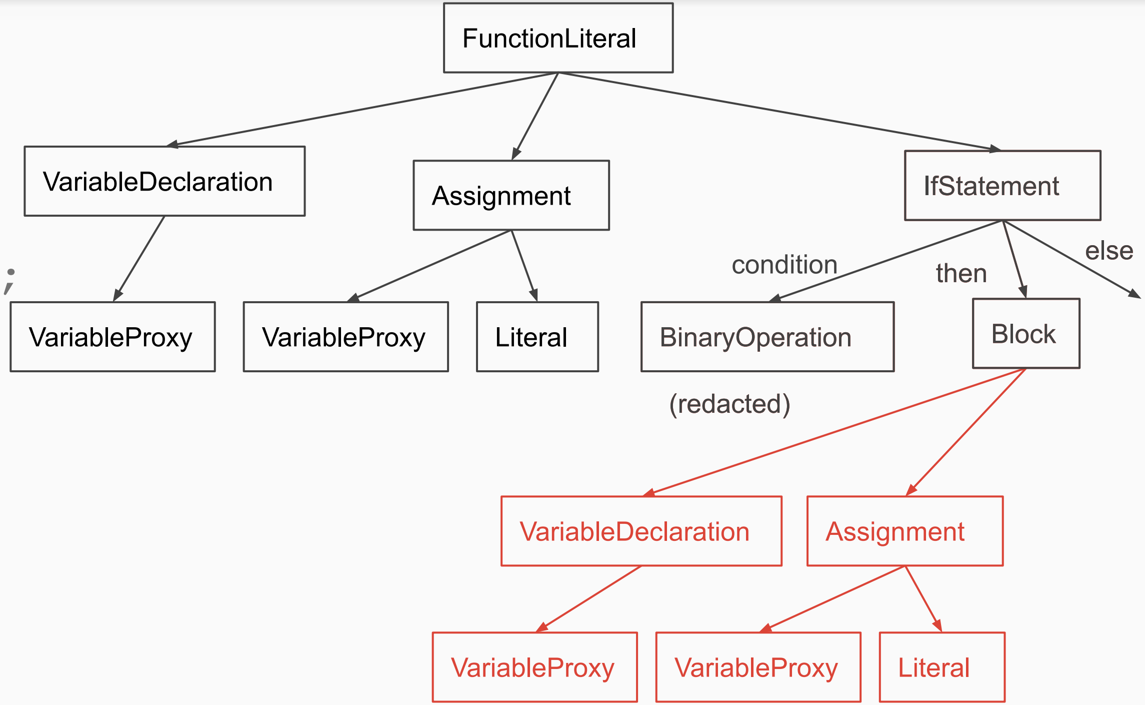
```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = "bar";
    return a;
  }
}
```



0.
WHAT?

AST = Abstract Syntax Tree

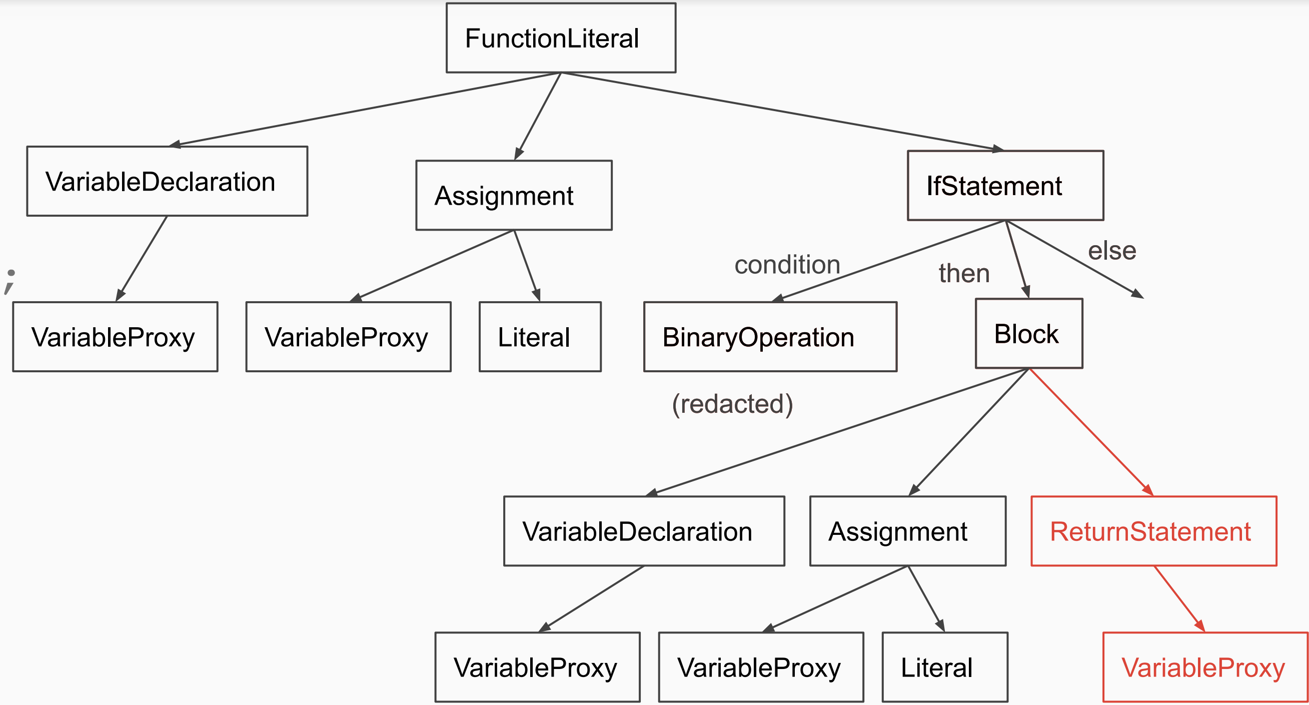
```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = "bar";
    return a;
  }
}
```



0.
WHAT?

AST = Abstract Syntax Tree

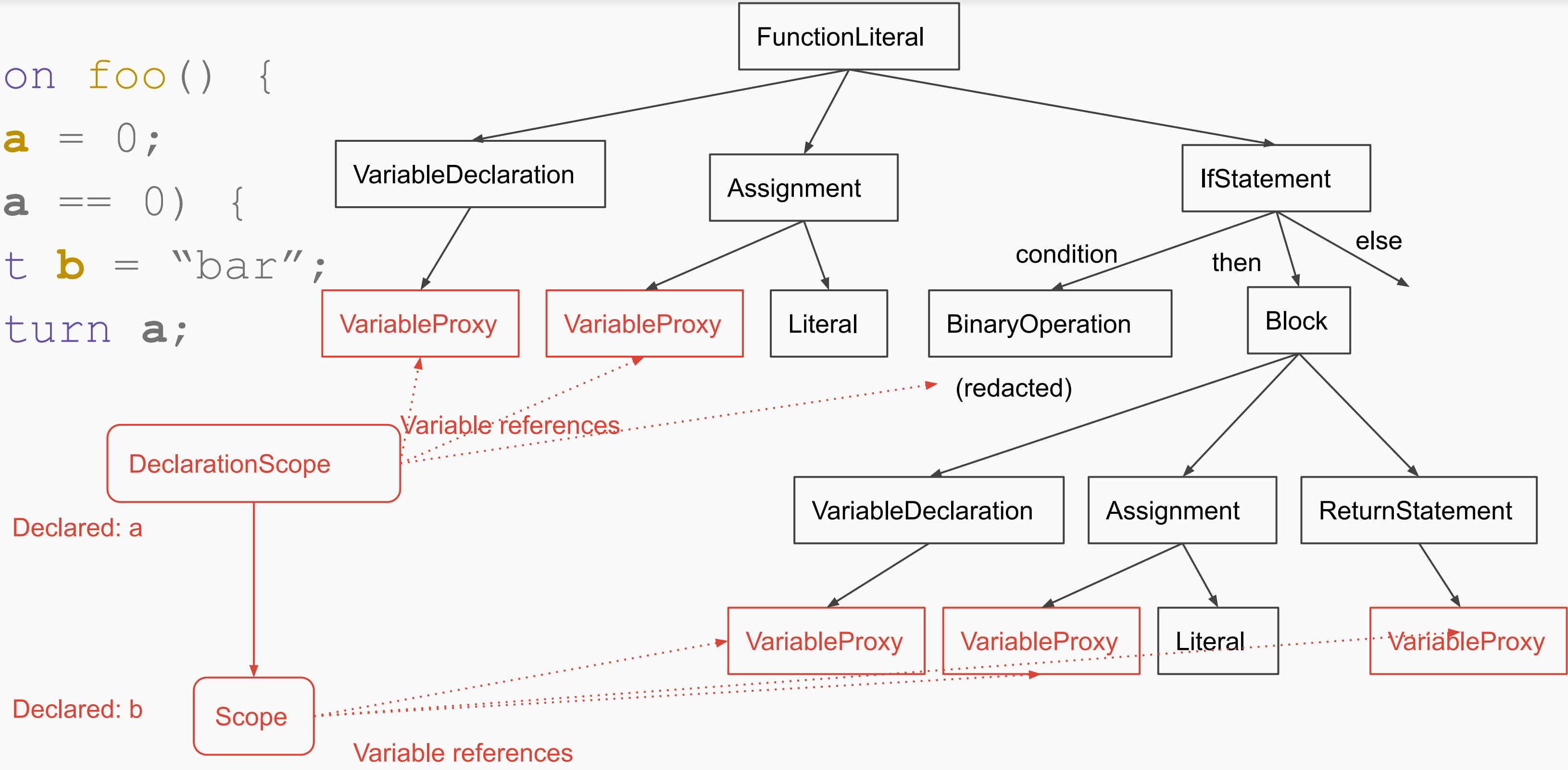
```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = “bar”;
    return a;
  }
}
```



0.
WHAT?

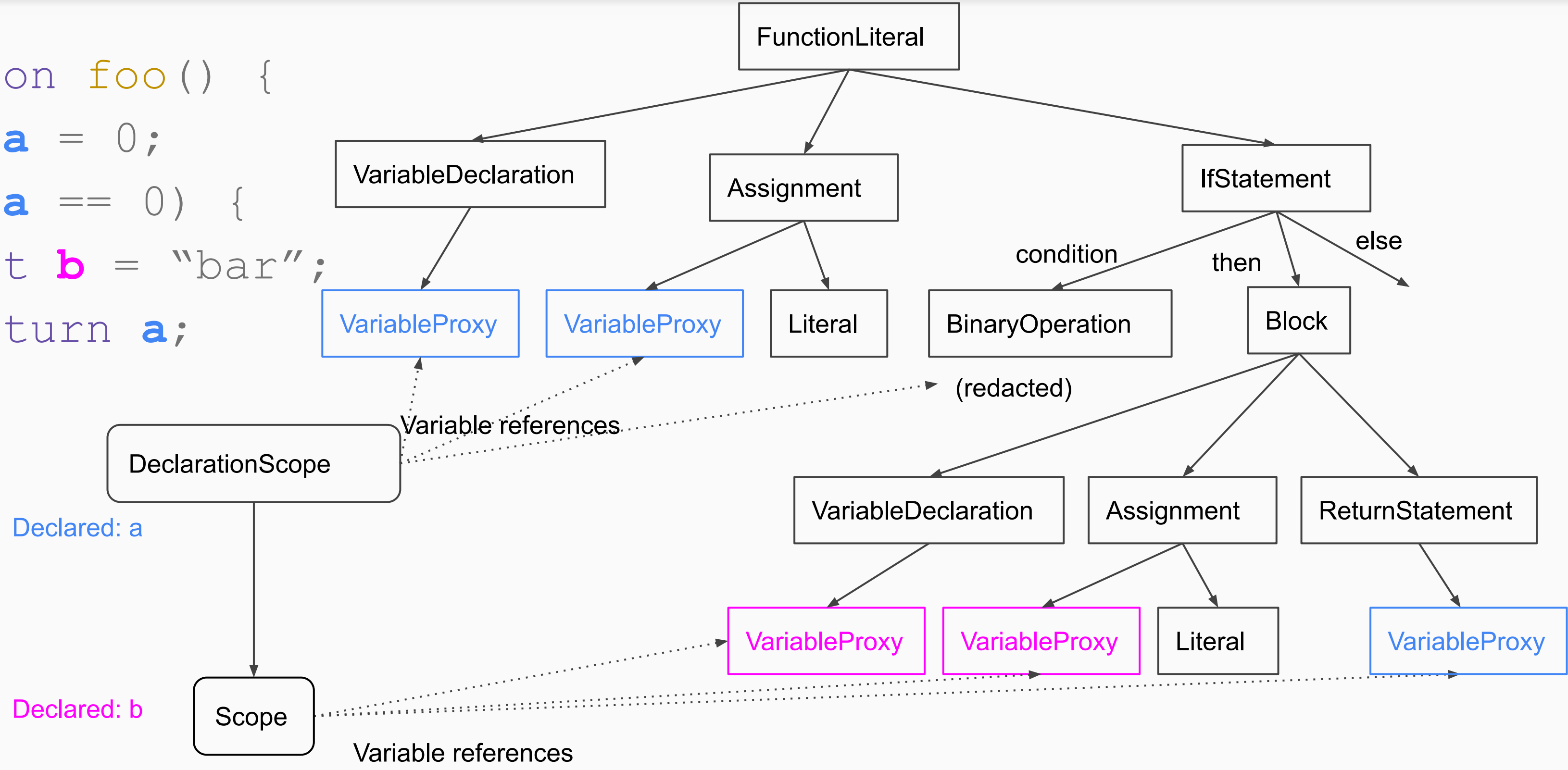
Scopes

```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = “bar”;
    return a;
  }
}
```



Scope analysis

```
function foo() {
  let a = 0;
  if (a == 0) {
    let b = “bar”;
    return a;
  }
}
```



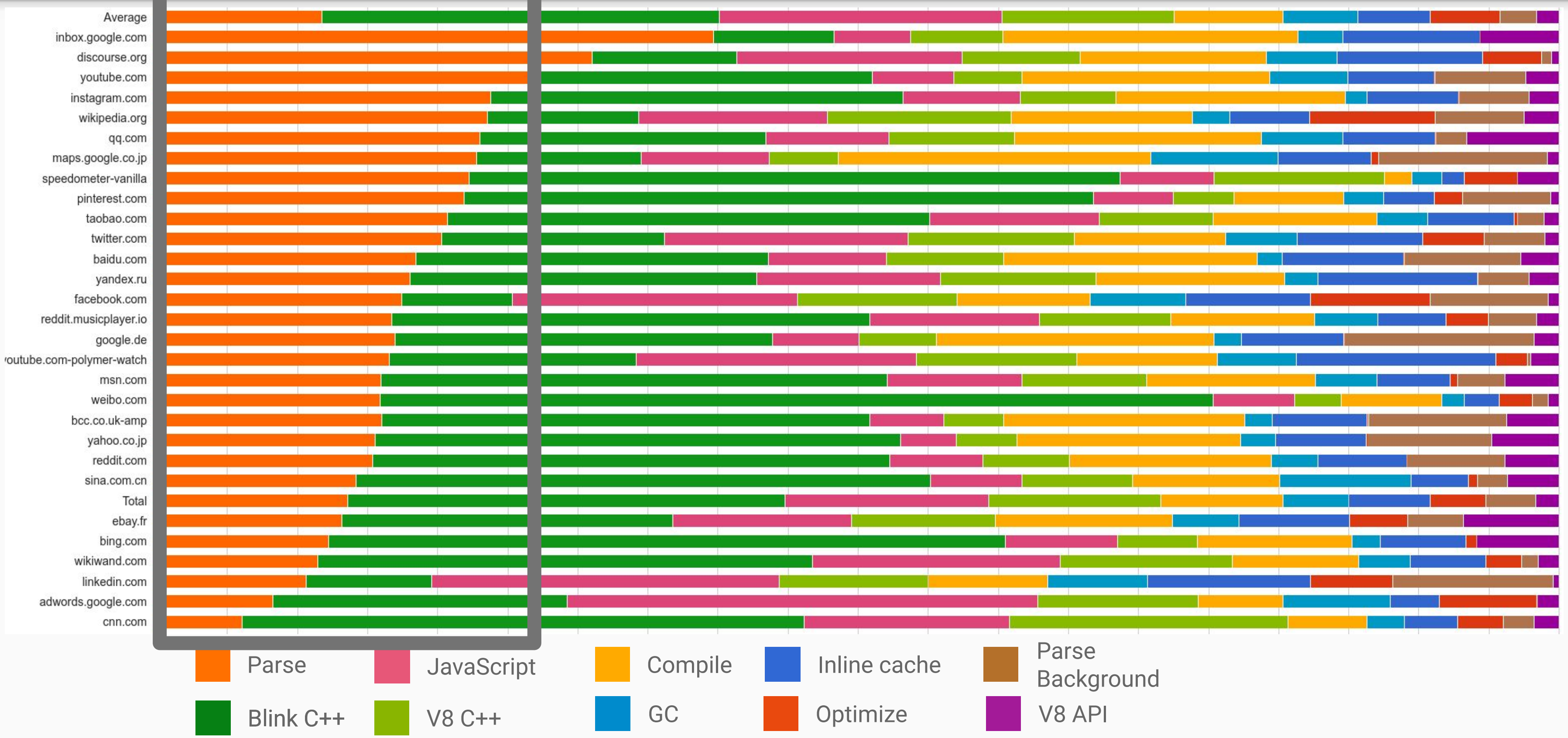
Now look at this sloth...

https://commons.wikimedia.org/wiki/File:Bradypus.jpg

WHY?

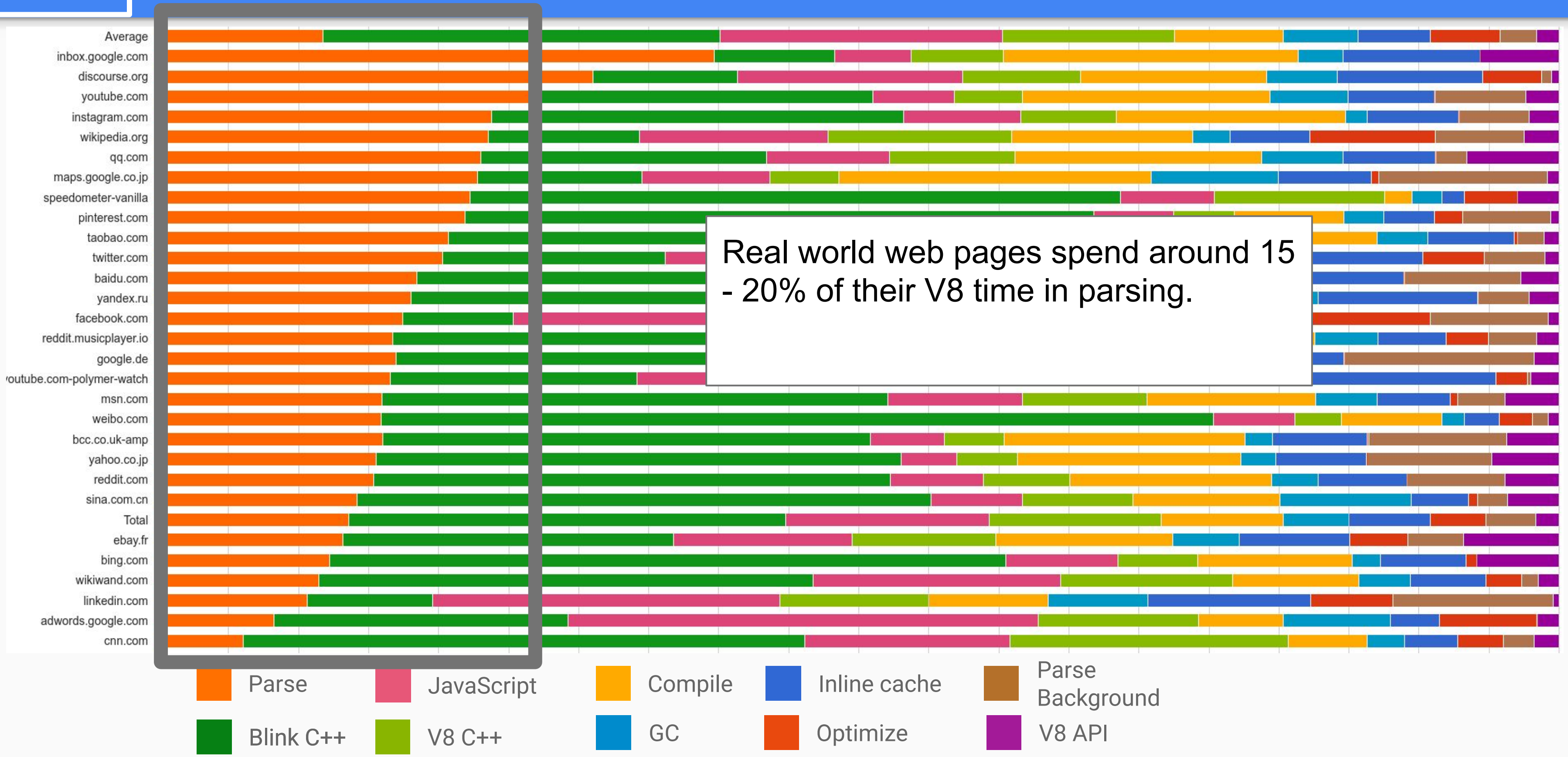
1.
WHY?

Parsing for real world web pages: 15-20% (V8)



1.
WHY?

Parsing for real world web pages: 15-20% (V8)



Real world web pages spend around 15 - 20% of their V8 time in parsing.

Parsing speed

- “Production Web Apps Performance Study”
- A typical single page web app
  - loads 0.4 MB of JavaScript
  - spends ~**370 ms** parsing it (on mobile - Moto G4)
- -> parsing speed ~ 1 MB / s

HOW?

2.
HOW?

How does V8 parse JavaScript?

- Two parsing modes: eager and lazy
- Why is parsing hard?
- Why is benchmarking parsing hard?

Two parsers

- Parser: full, “eager”
  - Used for parsing functions we want to compile
  - Builds AST
  - Builds Scopes
  - Finds all syntax errors
- PreParser: fast, “lazy”
  - Used for skipping over functions which we don’t want to compile
  - Doesn’t build an AST; builds Scopes but doesn’t put variable references or declarations in them
  - ~ twice as fast as Parser
  - Finds a restricted set of errors (doesn’t comply with the JavaScript spec!)

Lazy or eager?

```
let a = 0; // Top level code is eager

// IIFE = immediately invoked function expression
(function eager() { ... })(); // Body is eager

// Top level function but not IIFE
function lazy() { ... } // Body is lazy

   // Later:
   ... lazy();
   // -> eager parsed and compiled now!
```

Lazy or eager?

```
// Other heuristics:
!function eager2() { ... },function eager3() { ... }
```

Trickier lazy vs eager cases

```
let f1 = function lazy() { ... }; // Body is lazy. All OK!
         ^^ lazy parsing decision made here

let f2 = function lazy() { ... }(); // Oh no!
// We already lazy parsed, and now we need to eager parse
// right after.
```

Lazy or eager?

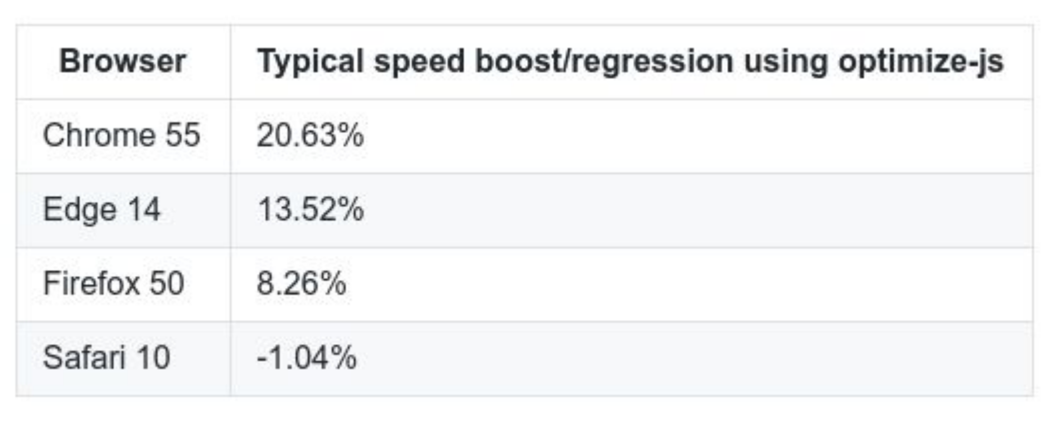
- Rules not specified in the JavaScript spec
- Each engine is free to implement lazy and eager as they see fit (or not implement them at all)
- V8 tries to guess which functions are probably called and eager parse them.

Why is lazy vs eager important?

- We need lazy parsing, since web pages ship code they don’t need.
- Which one to use?
  - If we eager parse something that is not needed, we’re wasting time.
  - If we lazy parse something that **is** needed, we pay the cost of preparse + parse
  - 0.5 * parse + 1 * parse = 1.5 * parse
- If we only knew which code is executed on startup!

Forcing eager parsing

- Optimize.js wraps functions it thinks will be executed in parens!



Browser	Typical speed boost/regression using optimize-js
Chrome 55	20.63%
Edge 14	13.52%
Firefox 50	8.26%
Safari 10	-1.04%

from https://github.com/nolanlawson/optimize-js

Change

```
function foo() { … }
```

to

```
(function foo() { … })
```

- But really, we should just
  - parse / compile the right functions
  - minimize the cost for cases where we fail to do so
- Working on it!

Lazy parsing inner functions - ctxt allocation

```
let f = (function outer() { // Eager
    let a = 20;
    function inner() { return a; }
    return inner;
})();
console.log(f()); // Where is the 20 coming from?
```

2.
HOW?

Lazy parsing inner functions

- If we lazy parse inner functions, we need to know which variables they refer to
- Normally lazy parsing doesn’t care about variable names
- We need lazy parsing w/ names (speed between Parser & PreParser)
- Lazy parsing inner functions will always be heavier than lazy parsing top level functions (because of JavaScript semantics).
- Modern JavaScript is heavily nested :/

```
let f = (function outer() {
    let a = 20;
    function inner() { return a; }
    return inner;
})();
console.log(f());
```

2.
HOW?

Reparsing

```
function lazy_outer() { // Lazy parse this
    function inner() { ... } // This too
}

... lazy_outer(); // Lazy parsing inner again!
```

Reparsing cont.

```
function lazy_outer() { // Lazy parse this
    function inner() {
        function inner2() { ... }
    }
}

... lazy_outer(); // Lazy parsing inner & inner2
... inner(); // Lazy parsing inner2 (3rd time!)

// FIXME(marja): just skip them.
```

Why is parsing hard? “Ambiguities”

- Arrow functions: we don’t know what we are parsing up front - arrow function parameter list or just a comma separated expression list?

```
(a, b, c) => { return a; } // OK: arrow function
(a, b, c) // OK: comma expression
(a, 1, 2) // OK: comma expression
(a, 1, 2) => { return a; } // Not OK
(a, ...b) => { return b; } // OK: arrow func + rest param
(a, ...b) // Not OK
```

- V8 parser never rewinds!

Why is parsing hard?

- High feature complexity + more language features are added all the time
- A typical parser bug: Eager parsing fails with

```
var g = ({x}, g = () => eval("x")) => { var x = 2;
return g(); };
assertEquals(1, g({x: 1}));
```

- Features involved
  - Lazy vs eager
  - Destructuring: {x} (turned out not to be relevant)
  - Default parameters
  - Arrow functions (arrow function as default parameter to another arrow function)
  - Eval (eval in the body of an arrow function which is a default parameter)

Why is benchmarking parsing hard?

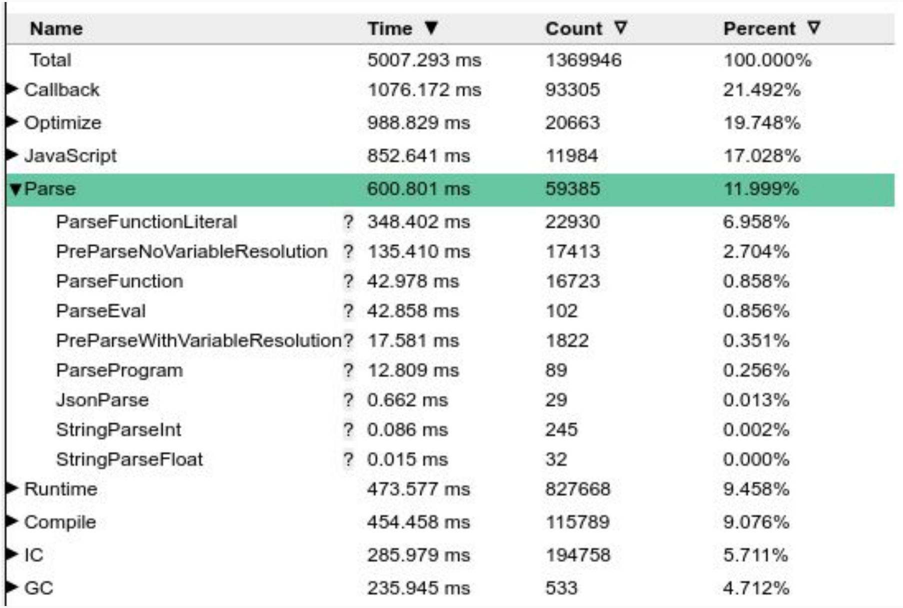
```
// Benchmark 1 (not a parsing benchmark)
function big() { ... } // Lazy (no paren before function)
(function benchmark() {
   start_timer(); big(); measure();
})();

// Benchmark 2 (parsing benchmark)
(function benchmark() {
   start_timer(); eval(“lots of code”); measure();
})();
```

YOU?

3.
YOU?

Web developers: Ship less code!

- JavaScript Start-up Performance
- Ship less JavaScript
    - Chrome Dev Tools now has code coverage!
- Measure the parse cost of your code and dependencies
    - chrome://tracing and v8.runtime_stats



Name		Time ▼	Count ∇	Percent ∇
Total		5007.293 ms	1369946	100.000%
▶ Callback		1076.172 ms	93305	21.492%
▶ Optimize		988.829 ms	20663	19.748%
▶ JavaScript		852.641 ms	11984	17.028%
▼Parse		600.801 ms	59385	11.999%
ParseFunctionLiteral	?	348.402 ms	22930	6.958%
PreParseNoVariableResolution	?	135.410 ms	17413	2.704%
ParseFunction	?	42.978 ms	16723	0.858%
ParseEval	?	42.858 ms	102	0.856%
PreParseWithVariableResolution	?	17.581 ms	1822	0.351%
ParseProgram	?	12.809 ms	89	0.256%
JsonParse	?	0.662 ms	29	0.013%
StringParseInt	?	0.086 ms	245	0.002%
StringParseFloat	?	0.015 ms	32	0.000%
▶ Runtime		473.577 ms	827668	9.458%
▶ Compile		454.458 ms	115789	9.076%
▶ IC		285.979 ms	194758	5.711%
▶ GC		235.945 ms	533	4.712%

3.
YOU?

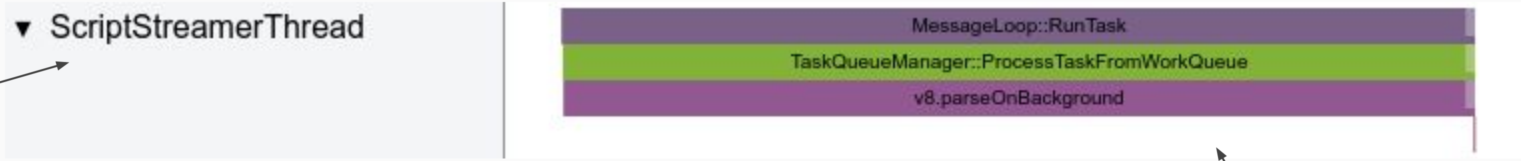
Web developers: Code caching + bundling

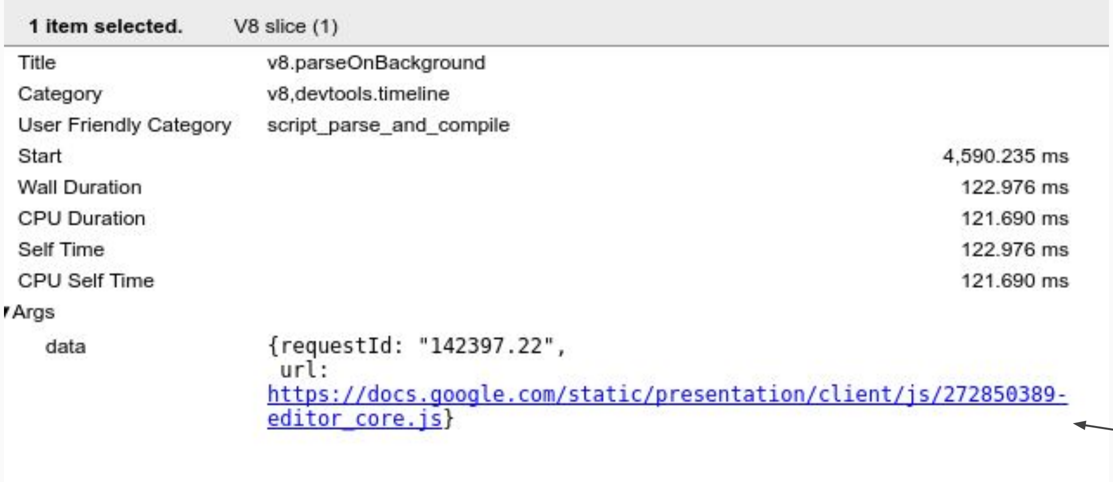
- Code caching: V8 caches the bytecode of frequently used scripts
- Bundling: if you update one part of the bundle, you lose the code cache for the full bundle.

3.
YOU?

Web developers: Use streaming

- Script streaming: downloading and parsing in parallel
- Big scripts
  - Load as early as possible and async
  - Make sure script streaming kicks in (chrome://tracing)





```
1 item selected.     V8 slice (1)
Title                    v8.parseOnBackground
Category                 v8,devtools.timeline
User Friendly Category   script_parse_and_compile
Start                                                        4,590.235 ms
Wall Duration                                                  122.976 ms
CPU Duration                                                   121.690 ms
Self Time                                                      122.976 ms
CPU Self Time                                                  121.690 ms
▾Args
   data                  {requestId: "142397.22",
                          url:
                         https://docs.google.com/static/presentation/client/js/272850389-
                         editor_core.js}
```

3.
YOU?

Web developers: Avoid eval

- Avoid `eval(“lots of code”)`: no streaming, no code cache

3.
YOU?

Web developers: Parens hack

- Use the parens hack selectively to force eager parsing & compilation of the critical path
  - Older Chrome versions
  - Across browsers
  - Need performance right now (can’t wait for our fixes)

Bonus content

- V8 parser is a hand-written recursive descent parser
- ~ 15 000 LoC (C++) and another ~ 7 000 LoC for the AST + Scopes

```
typename ParserBase<Impl>::StatementT ParserBase<Impl>::ParseIfStatement(ZoneList<const
AstRawString*>* labels, bool* ok) {
  int pos = peek_position();
  Expect(Token::IF, CHECK_OK); Expect(Token::LPAREN, CHECK_OK);
  ExpressionT condition = ParseExpression(true, CHECK_OK);
  Expect(Token::RPAREN, CHECK_OK);
  StatementT then_statement = ParseScopedStatement(labels, false, CHECK_OK);
  StatementT else_statement = impl()->NullStatement();
  if (Check(Token::ELSE)) {
    else_statement = ParseScopedStatement(labels, false, CHECK_OK);
  } else {
    else_statement = factory()->NewEmptyStatement(kNoSourcePosition);
  }
  return factory()->NewIfStatement(condition, then_statement, else_statement,
                                   pos);
}
```

THE
END

What you should remember from this talk

- 370 ms
- Check your parse time in chrome://tracing

```
(function eager() { ... })();
```

- Ship less code

If you have further questions / comments, please get in touch (marja@chromium.org, Twitter: @marjakh).