So, you want to build your own programming language.

Steve Kinney

Hi, I’m **Steve**.
(@stevekinney)


SendGrid

The Good News: As long as you rein in your ambitions, creating your own language is suspiciously easy.

The evaluator, which determines the meaning of expressions in a programming language, is just another program. — Hal Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs*

Why might you want to do any of this?

You might not want to write your own programming language, but there are elements that are super practical for production code.

Use Cases

- **Domain-specific languages**: Terraform, Gemfiles.
- **Templating languages**: Handlebars, Jade (Er, Pug). This is my use case right now.
- (You have a twisted idea of “fun.”)

Meta point: You can use individual pieces of this talk in your day-to-day work without having to build an entire language.

How do we uses these tools at **Twilio SendGrid**?

- Our drag-and-drop editor takes HTML, parses it into an abstract syntax tree (AST) and manipulates it before serializing it back into HTML.
- We’re working on our own bespoke templating language for dynamic email templates.
- We can sync your position in our side-by-side editor by breaking HTML into an AST and rewriting the elements with information about their position in the text editor.

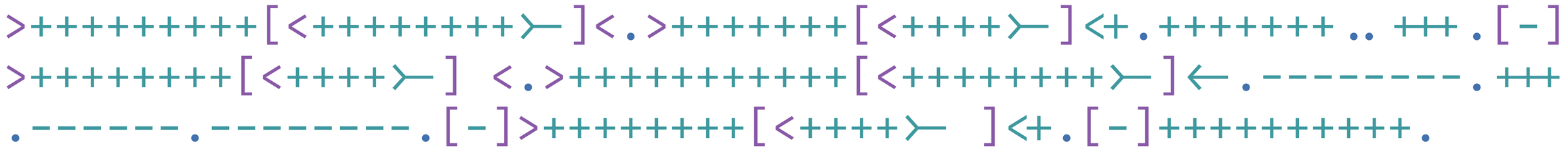
My goal is that you should be able to take pieces of this workshop as inspiration.

So, what even is a compiler?

Answer: Something that turns a higher-level language into a lower-level language*.

Most of us do not write CPU instructions by hand. Most of us do not write Assembly.

Source code is meant to be human readable.



What are we building today?

```
(add 1 2 (subtract 6 3))
```

Tasting Notes

- We’re going to build a Lisp-like language.
- We’re going to leverage the existing JavaScript run-time.

The beauty of Scheme is that the full language only needs 5 keywords and 8 syntactic forms. In comparison, Python has 33 keywords and 110 syntactic forms, and Java has 50 keywords and 133 syntactic forms. — Peter Norvig

Inspiration

- Peter Norvig's Lispy (a Lisp written in Python)
- Eloquent JavaScript, Chapter 12: The Egg programming language.
- Jamie Kyle's Super Tiny Compiler.

The Stages of a Compiler

- **Parsing**: Take the source code and turn it into a representation of that code. (I am going to make believe this is two stages in a bit.)
- **Transformation**: Take that source code and transforms it to do whatever the compiler wants it to do.
- **Generation**: Take the transformed representation and turns it into a new string of code.

Let’s start with parsing…

This is kind of two steps rolled into one.

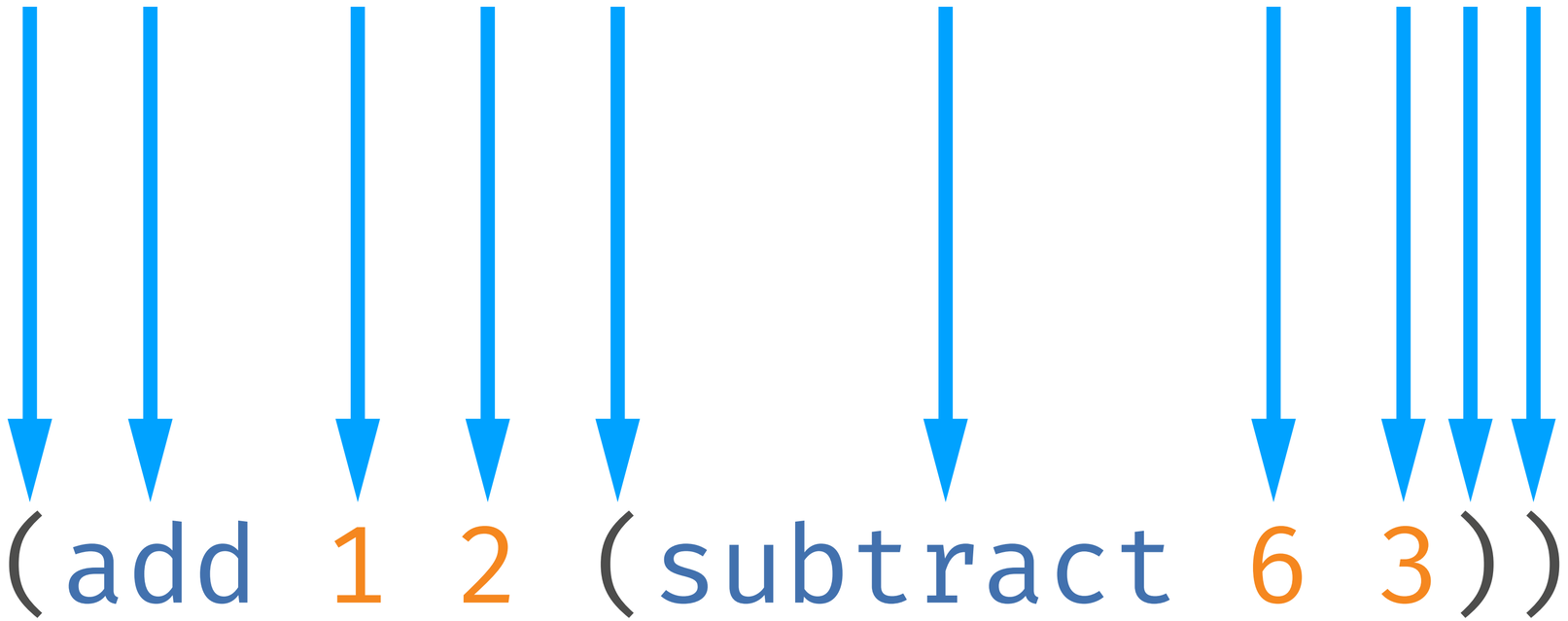
- Lexical analysis
- Syntactic analysis
- (Psychoanalysis)

Lexing

(This is how cool kids say “lexical analysis.”)

Basically: Take the big string of code and turn it into tokens.

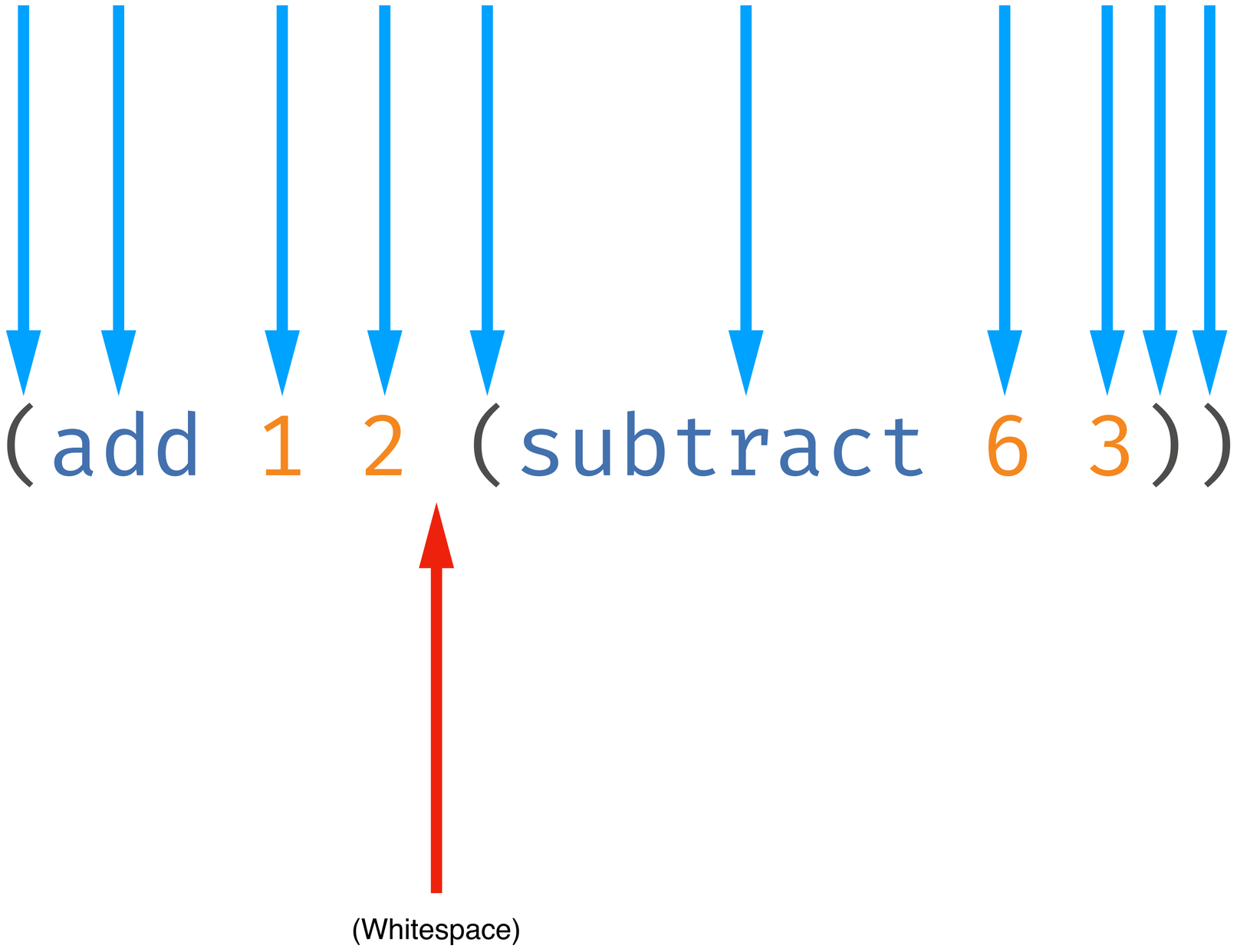
A token is a small unit of the language.


(add 1 2 (subtract 6 3))

How might a lexer work?

- Accept an input string of code.
- Create a variable for tracking our position, like a cursor.
- Make an array of tokens.
- Write a while loop that iterates through the source code input.
- Check each token. See if it matches one of your types.
- Add it to the array of tokens.

First, let’s create some helpers.


(add 1 2 (subtract 6 3))
(Whitespace)

Exercise

- Parsing code is gnarly. Let's try to make it a little bit less gnarly by creating some helpers.
- In `exercise-1.test.js`, there are some tests for helpers for helping us to identify each type of token.
- **Your job**: Un-skip the tests and write some simple functions that get each of the tests passing.

```
const isWhitespace = character ⇒ /\s/.test(character);
const isNumber = character ⇒ /[0-9]/.test(character);
const isOperator = character ⇒ /[\+\-\*\/]/.test(character);
// More identifiers here…
```

How do we break apart this string of text into an array of tokens?

```
const tokenize = (input) => {
  let cursor = 0;
  const tokens = [];

  while (cursor < input.length) {
    // Our logic goes here…
  }

  return tokens;
};
```

Quick disclaimer: Our first attempt at tokenization is going to be *too* simple.

```
if (isOperator(character)) {
  tokens.push(character);

  cursor++;
  continue;
}
```

```
if (isNumber(character)) {
  tokens.push(character);

  cursor++;
  continue;
}
```

```
if (isWhitespace(character)) {
  cursor++;
  continue;
}
```

Exercise

- Let’s write a tokenize function that will iterate through the input string and return an array of tokens.
- Right now, we’ll just worry about identifying single digits, operators, and parenthesis.
- It should skip over any whitespace that it finds.

2 + 3

```
[
  '2',
  '+',
  '3'
]
```

This has a flaw.

22 + 23

```
[
  '2',
  '2',
  '+',
  '2',
  '3'
]
```

Solution: For numbers, identifiers, and strings, we need to collect multiple characters into a single token.

```
if (isNumber(character)) {
  let value = character;

  while (isNumber(input[++cursor])) {
    value += input[cursor];
  }

  tokens.push(value);

  continue;
}
```

Exercise

- When we come across a letter or number, check to see if the *next* character is a number too.
- If so, add it to the token.
- If not, move along.

22 + 23

```
[
  '22',
  '+',
  '23'
]
```

This is what we have so far.

```
['(', 'add', 12, 34, ')']
```

Some thoughts to ponder…

- Do we even need those parentheses?
- What do we do about nested expressions?

```
(add 1 2 (subtract 6 3))
```

Parsing I

Part 1: Basic Processing

How would we go from tokens to expressions?

- If we see an opening parenthesis, start an expression.
- If you see a closing parenthesis, we're done with this expression.
- If you're in the middle of an expression and you see another opening parenthesis, you've got another expression.
- Ditch the parentheses, we don't need them anymore.

Exercise

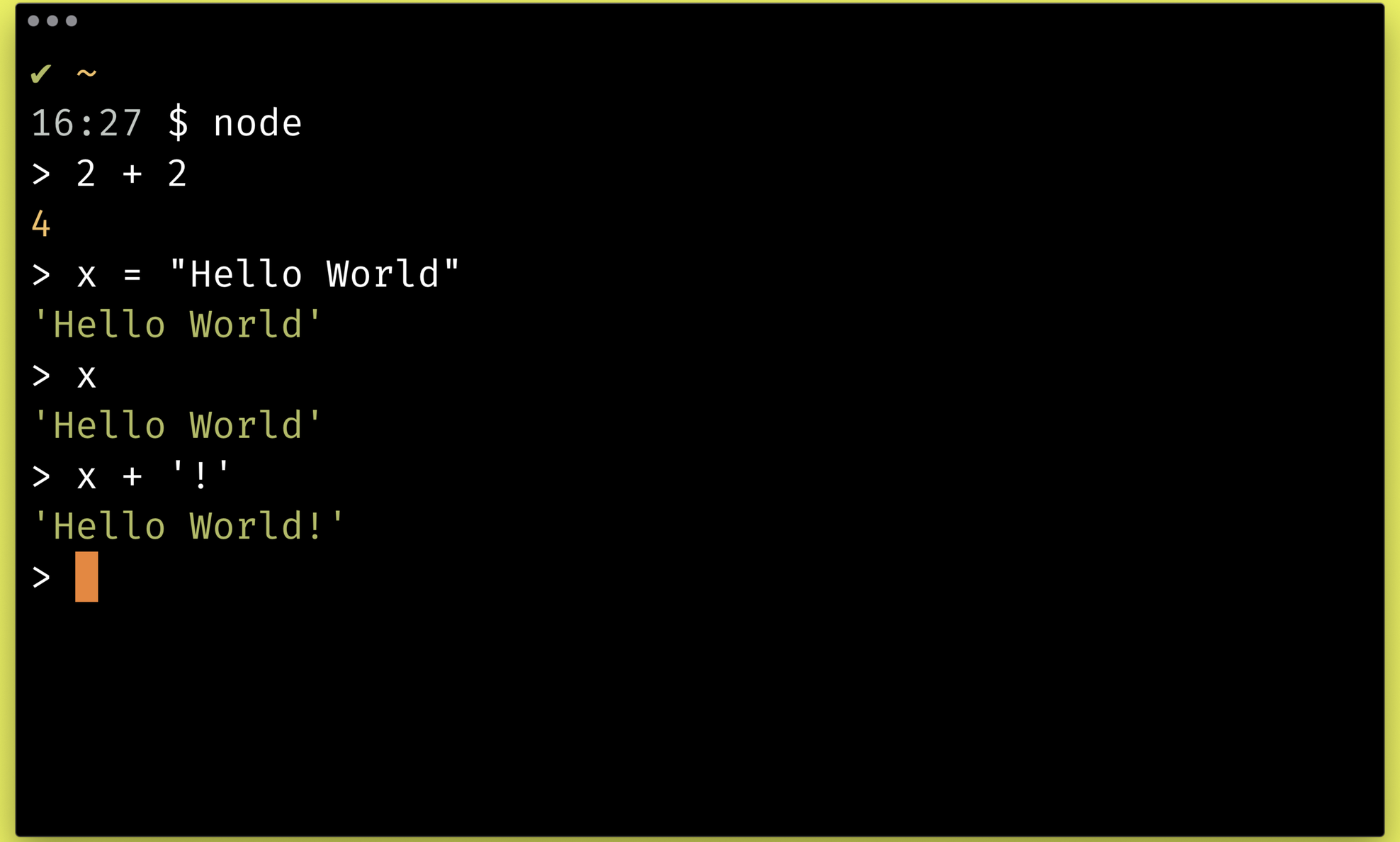
- Create a processTokens() function.
- Start pulling from the front of the array of tokens.
- When you seen an opening parenthesis, start a new expression array.
- Put tokens into that array until you come across the closing parenthesis.
- **Bonus**: Recursively descend into nested pairs of parentheses.

Evaluation

Let’s build a REPL together.

What is a REPL?

It’s a **r**ead-**e**valuate-**p**rint **l**oop. **REPL.**


✔ ~
16:27 $ node
> 2 + 2
4
> x = "Hello World"
'Hello World'
> x
'Hello World'
> x + '!'
'Hello World!'
>

Not so fast...

- We’re been using functions like add and subtract, but what do those mean?
- Above and beyond having a syntax, most languages have some amount of a standard library of built-in functions, objects, and methods.
- We’re leveraging JavaScript’s built-in numbers and arrays, but we’re going to need some functions too.

The Standard Library

- Since JavaScript is our compile target, we'll implement our built-in functions as JavaScript functions.

Exercise

- We need to at least start with the following functions: `add`, `subtract`, `multiply`, `divide`, and `modulo`.
- Because this is a Lisp, these functions should be able to take more than two arguments.

Parsing

(A.k.a. “syntactic analysis.”)

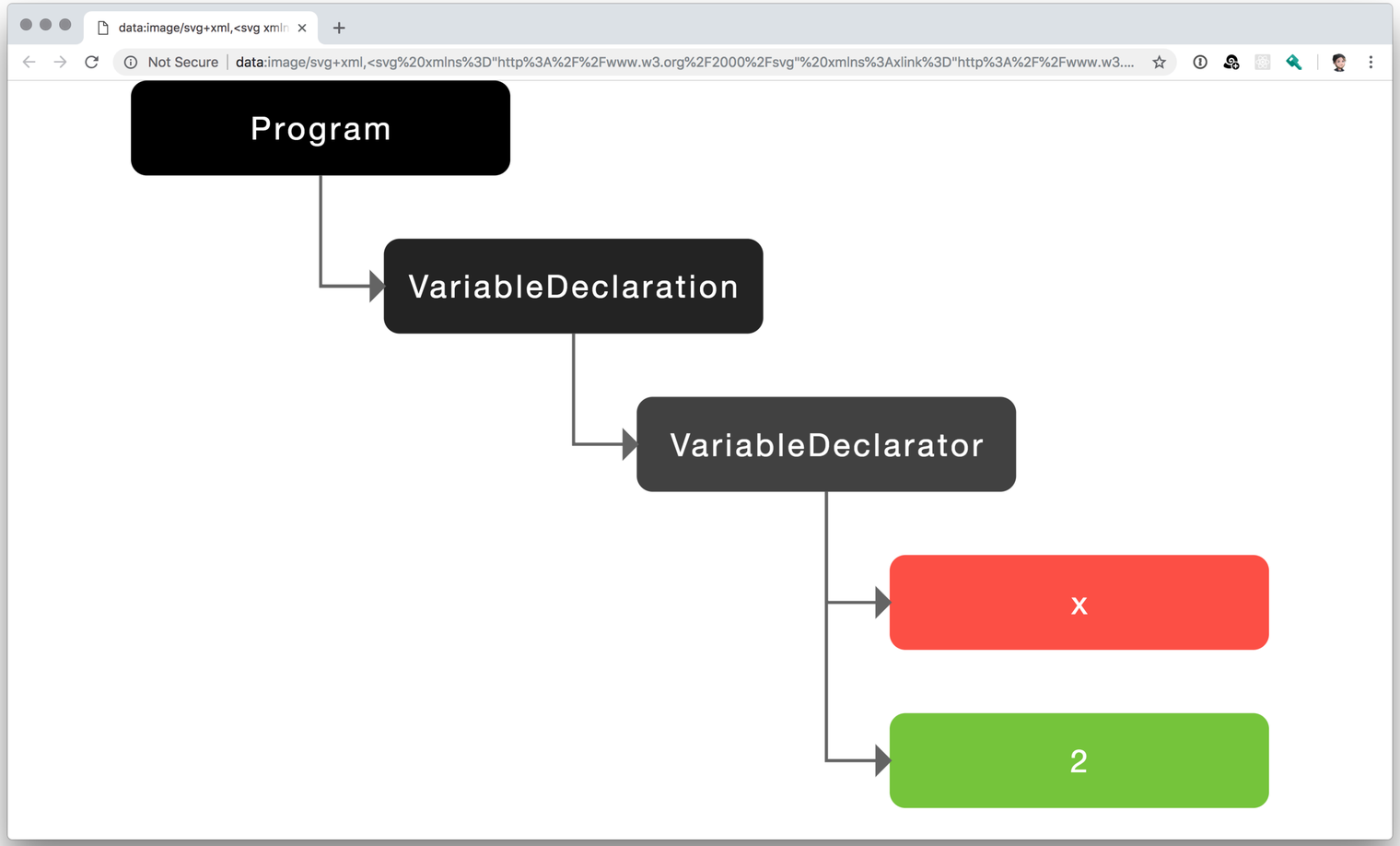
Okay, so you've broken it into tokens, next you need to figure out how arrange those tokens in a way that means something.

Syntactic analysis: turn the tokens into an intermediate representation or abstract syntax tree.

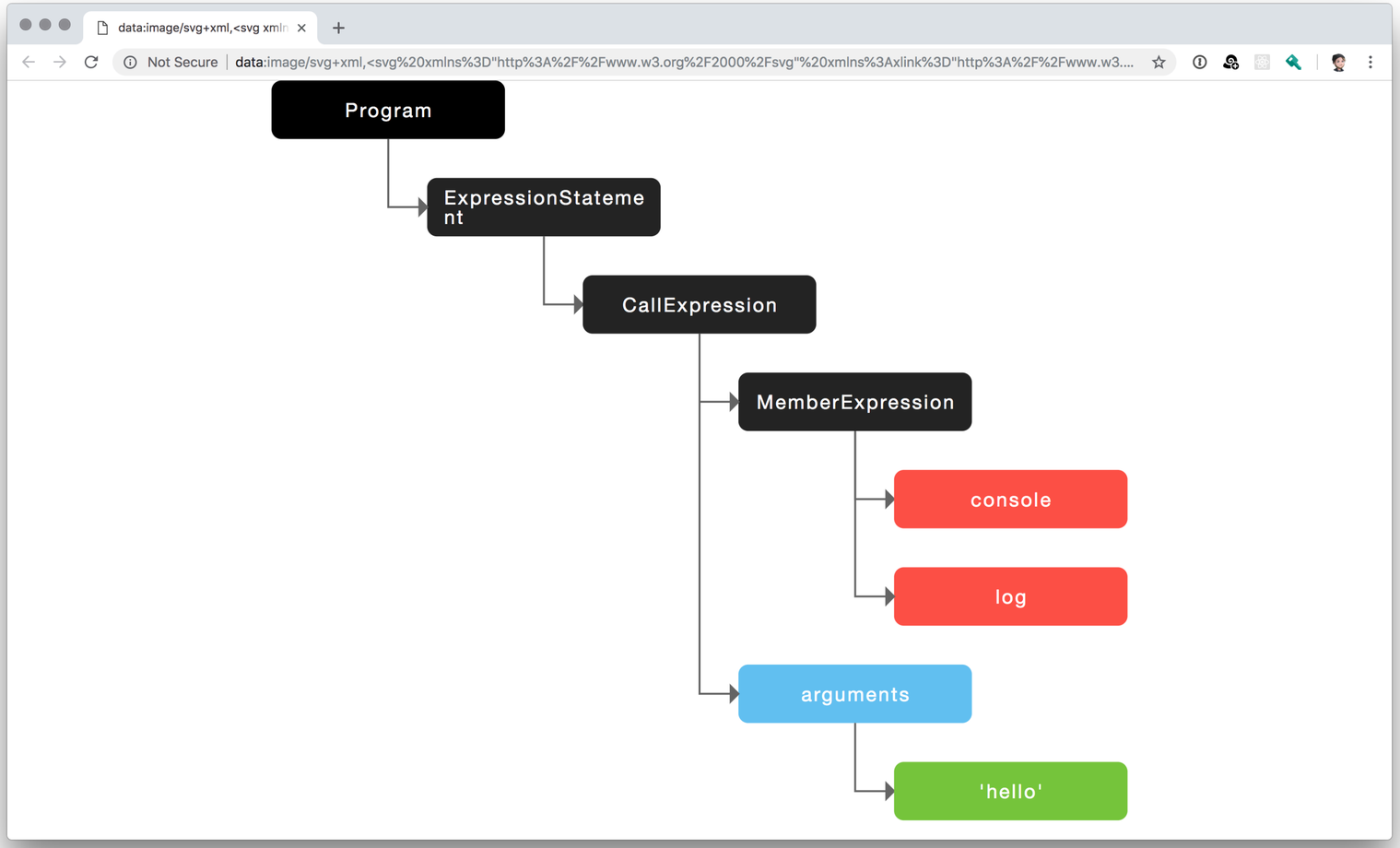
AST Explorer is your bud.

http://bit.ly/ast-fun

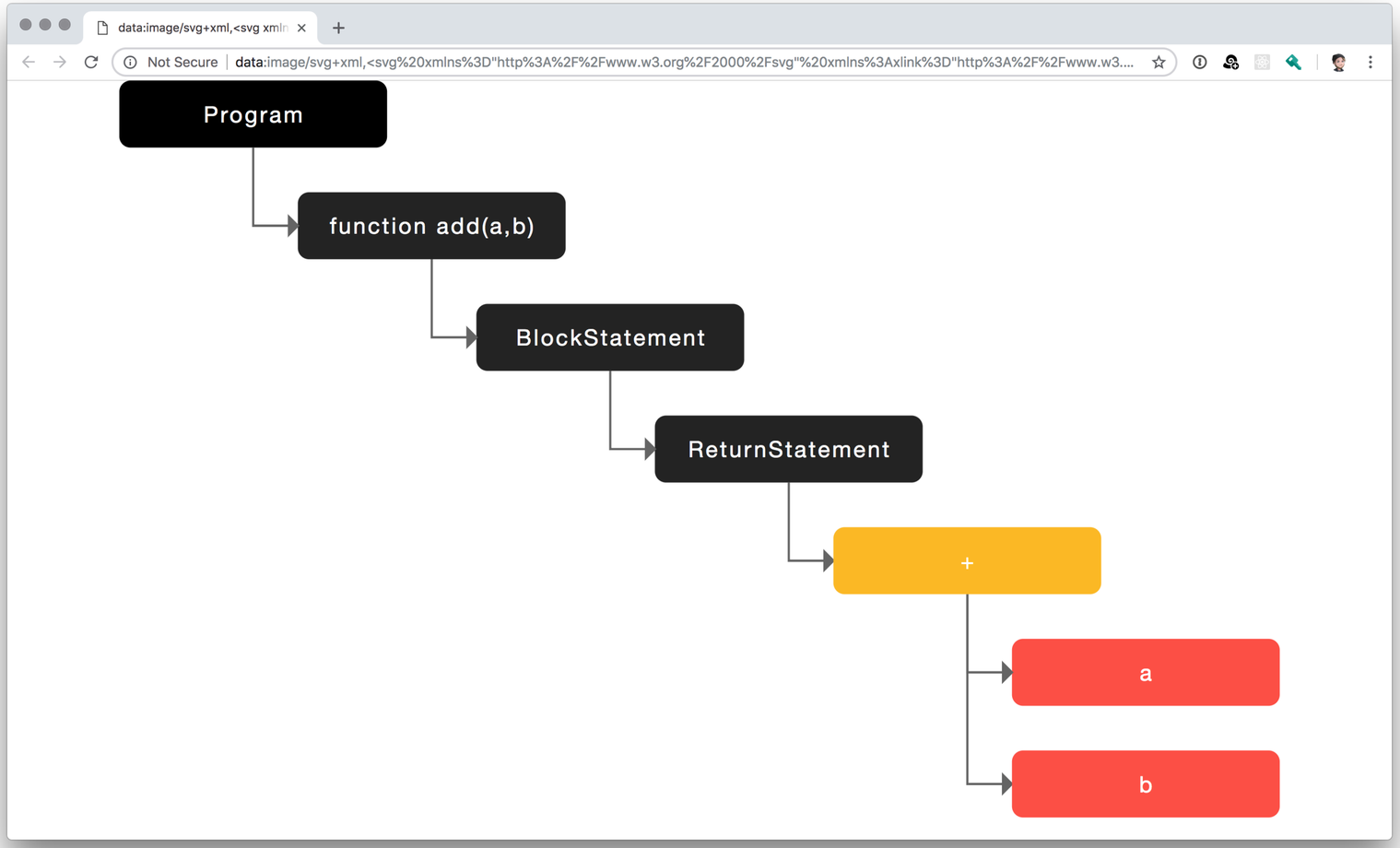
```
var x = 2;
```

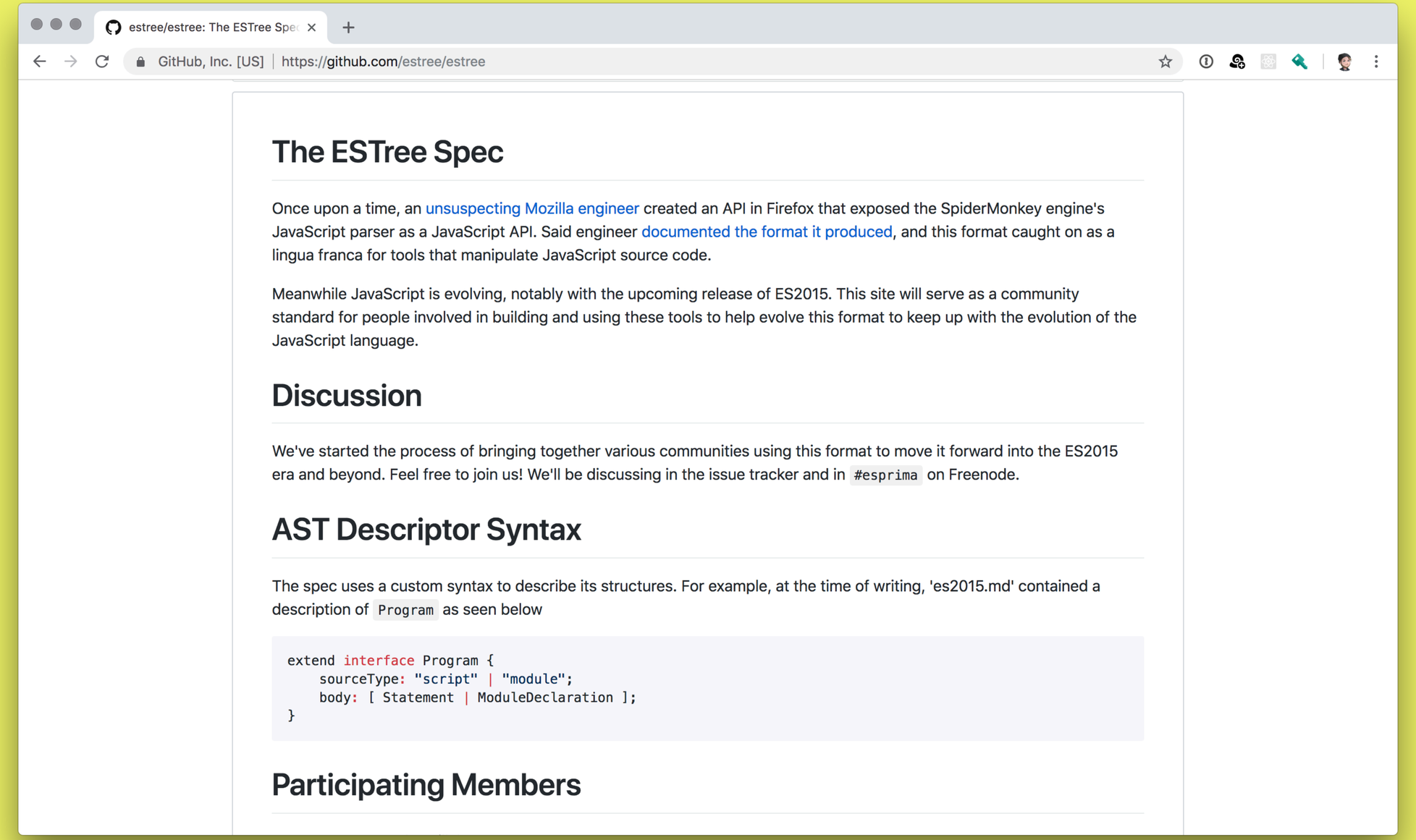

data:image/svg+xml,<svg xml
Not Secure
data:image/svg+xml,<svg%20xmlns%3D"http%3A%2F%2Fwww.w3.org%2F2000%2Fsvg"%20xmlns%3Axlink%3D"http%3A%2F%2Fwww.w3....
Program
VariableDeclaration
VariableDeclarator
x
2

console.log('hello');



```
function (a, b) { return a + b; }
```





The ESTree Spec

Once upon a time, an unsuspecting Mozilla engineer created an API in Firefox that exposed the SpiderMonkey engine's JavaScript parser as a JavaScript API. Said engineer documented the format it produced, and this format caught on as a lingua franca for tools that manipulate JavaScript source code.

Meanwhile JavaScript is evolving, notably with the upcoming release of ES2015. This site will serve as a community standard for people involved in building and using these tools to help evolve this format to keep up with the evolution of the JavaScript language.

Discussion

We've started the process of bringing together various communities using this format to move it forward into the ES2015 era and beyond. Feel free to join us! We'll be discussing in the issue tracker and in `#esprima` on Freenode.

AST Descriptor Syntax

The spec uses a custom syntax to describe its structures. For example, at the time of writing, 'es2015.md' contained a description of `Program` as seen below

```
extend interface Program {
    sourceType: "script" | "module";
    body: [ Statement | ModuleDeclaration ];
}
```

Participating Members

How might we build an AST?

- Iterate through the array of tokens.
- For each number, string, etc., add that token to same level of the tree.
- For each CallExpression (e.g. function) collect the parameters and then recurse down into the function body.

```
const parser = (tokens) ⇒ {
  let cursor = 0;

  const walk = () ⇒ {
    let token = tokens[cursor];
    // Do stuff with your tokens here…
  }
}
```

```
if (token.type === 'Number') {
  current++;
  return {
    type: 'NumberLiteral',
    value: token.value,
  };
}
```

```
while (
  (token.type !== 'paren') ||
  (token.type === 'paren' && token.value !== ')')
) {
  node.params.push(walk());
  token = tokens[current];
}
```

PSA: ASTs aren’t just for programming languages.

A word or two on **parser generators.**

Somewhat more controversial, I wouldn't bother wasting time with lexer or parser generators and other so-called "compiler compilers."

***They're a waste of time**. Writing a lexer and parser is a tiny percentage of the job of writing a compiler. Using a generator will take up about as much time as writing one by hand, and it will marry you to the generator (which matters when porting the compiler to a new platform). And generators also have the unfortunate reputation of emitting lousy error messages.* — Walter Bright.

Transformation

(More than meets the eye.)

Breaking apart our source code into an internal representation is cool and everything, but we theoretically did all of this work in order to create some kind of output, right?

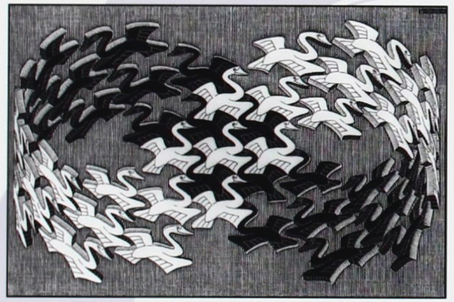
TL;DR: Manipulate the AST
and do your thing.

The Visitor Pattern

Design Patterns

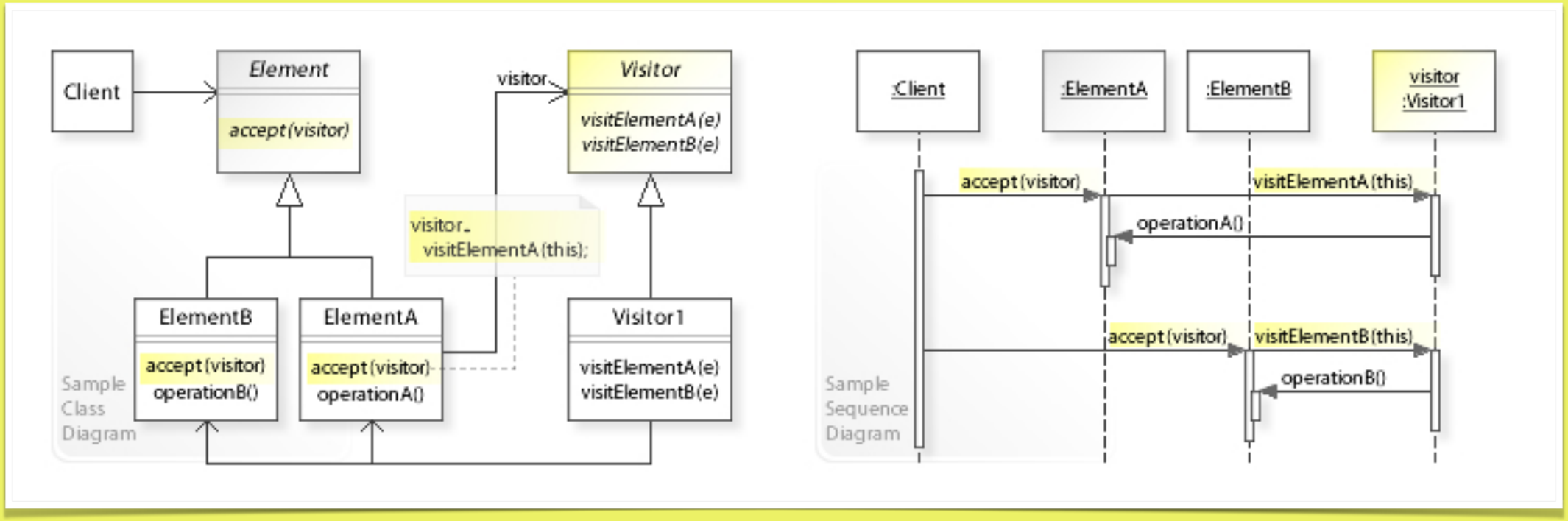
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



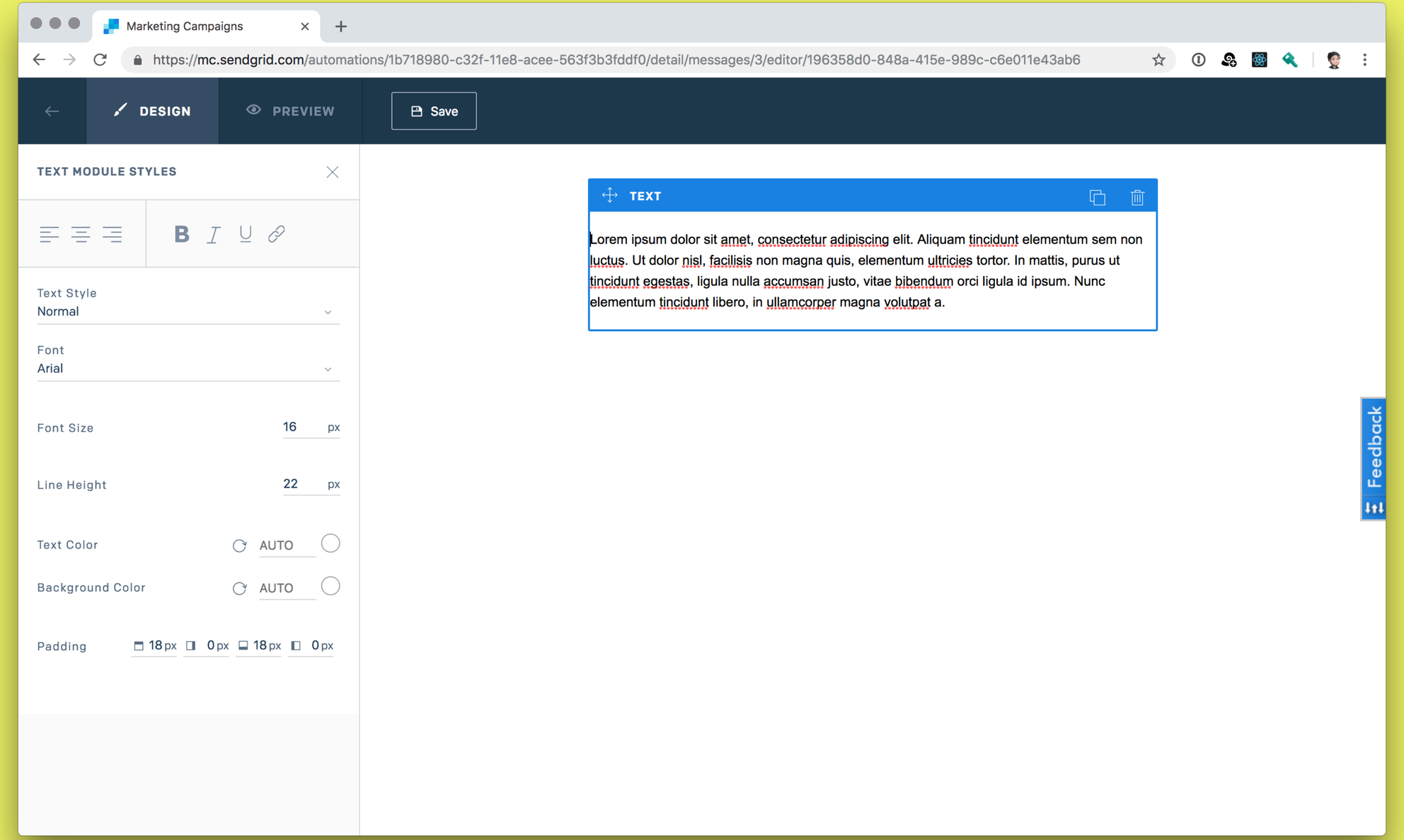
By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63201110

Basically, we do a depth-first search through our tree.

The Visitor Pattern™ allows us define different types of actions for each node visited as it walks the tree.

```
import traverse from '@babel/traverse';
```

```
traverse(ast, {
  enter(path) {
    if (path.node.type === 'VariableDeclaration' && path.node.kind === 'var') {
      path.node.kind = 'let';
    }
  },
});
```


Marketing Campaigns
https://mc.sendgrid.com/automations/1b718980-c32f-11e8-acee-563f3b3fddf0/detail/messages/3/editor/196358d0-848a-415e-989c-c6e011e43ab6
DESIGN
PREVIEW
Save
TEXT MODULE STYLES
Text Style
Normal
Font
Arial
Font Size
16
px
Line Height
22
px
Text Color
AUTO
Background Color
AUTO
Padding
18 px
0 px
18 px
0 px
TEXT
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam tincidunt elementum sem non luctus. Ut dolor nisl, facilisis non magna quis, elementum ultricies tortor. In mattis, purus ut tincidunt egestas, ligula nulla accumsan justo, vitae bibendum orci ligula id ipsum. Nunc elementum tincidunt libero, in ullamcorper magna volutpat a.
Feedback

HTML 👉 AST 👉 HTML

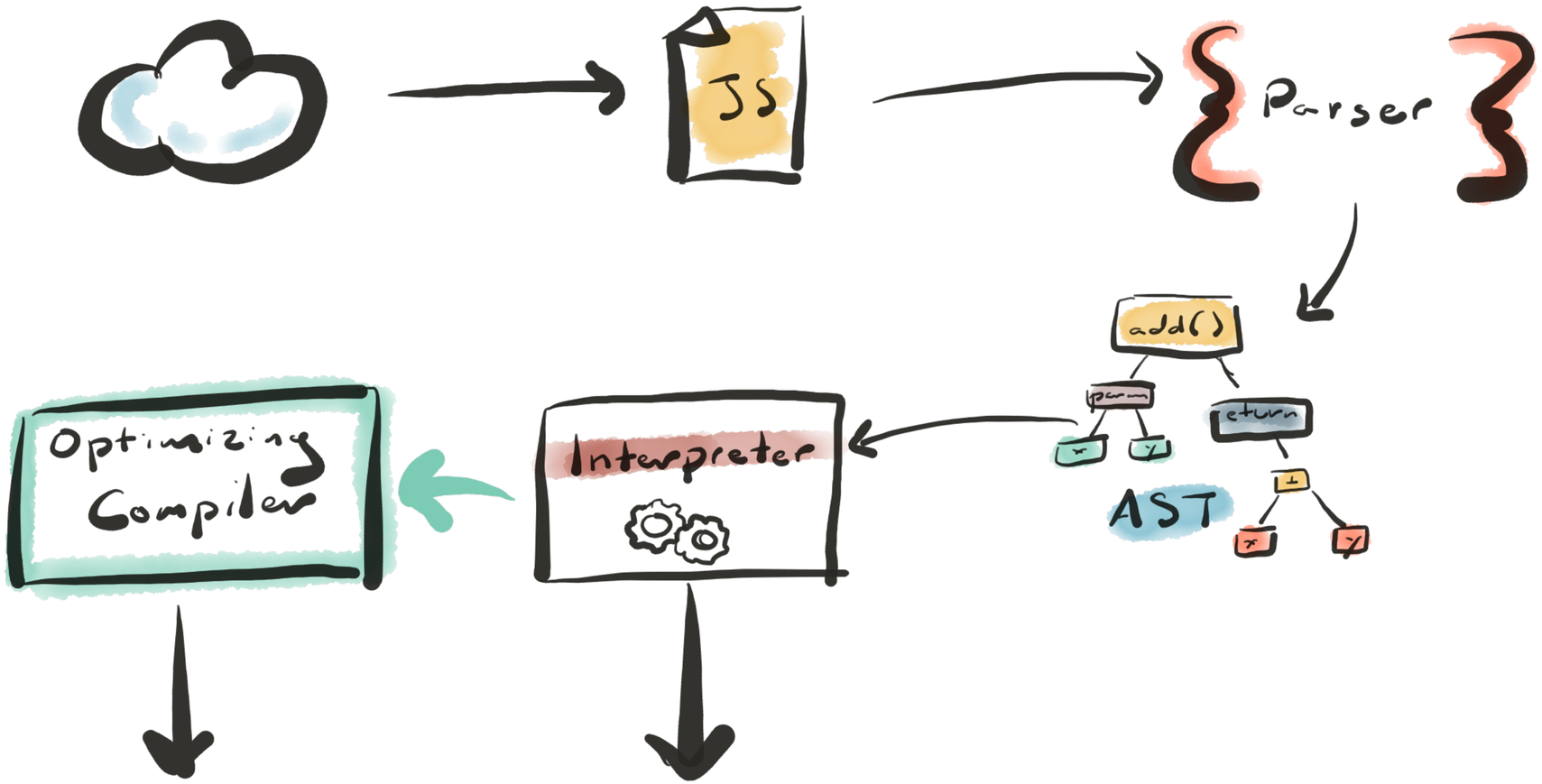
Runtime Model

(A Brief Interlude of Hand-Waving)

How we represent objects, methods, types, methods and structure in memory.

In this course, we're totally cheating because we're transpiling to another language that is going to handle all of that.

How is JavaScript compiled?
Let’s take a look at what V8 does.


JS
Parser
add()
AST
Interpreter
Optimizing Compiler
Machine Code
x86 ARM
10111000111 0011
BYTECODE

Generation

(Or, parsing in reverse.)

Code Generation Options

- Write your own low-level CPU-instruction compiler. (This is probably a terrible idea.)
- Use a compiler framework like LLVM.
- Target the JVM.
- Transpile (because it’s 2019).

If you can get yourself to a Babel-compliant AST, then you can use a tool off the shelf, otherwise, you'll have to do it yourself.

```
import generate from '@babel/generator';
```

```
generate(ast, options, code);
```

Some Other Bonus Terms

(That I couldn’t find a non-awkward way to include.)

Homoiconicity: A language that can modify its own underlying data structure.

Self-Hosting: The language is written in the language itself.

Questions?