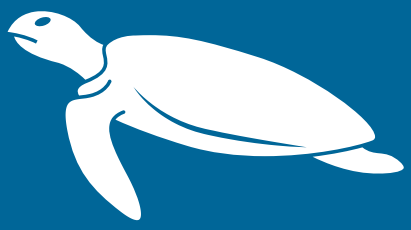
Pocket Guides  Go Make Things

3

VARIABLES, FUNCTIONS & SCOPE

with Vanilla JavaScript



CHRIS FERDINANDI

Variables, Functions, and Scope

By Chris Ferdinandi
Go Make Things, LLC
v1.0.0

Table of Contents

Intro

In this guide, you'll learn:

- The difference between function expressions and function declarations, and which one to use.
- What function hoisting is, and whether or not it's a good thing.
- How to scope your code, and why you want to.
- How to conditionally set variables.
- Useful techniques for passing variables into functions and defining default values.
- The difference between functions and methods.
- How to write ES6 arrow functions.
- How to use ES6 `let` and `const` for defining variables.

Using the code in this guide

Let's get started!

Function expressions vs. function declarations

There have historically been two ways to write a function.

```
// Function declaration
function add(num1, num2) {
    return num1 + num2;
}

// Function expression
var add = function (num1, num2) {
    return num1 + num2;
};
```

The first example, `function add() {}`, is called a *function declaration*. The second example, `var add = function () {}`, is called a *function expression*.

They more-or-less do the same example thing, but there's one subtle yet important difference between them.

***Note:** There's now a third way—ES6 arrow functions—that we'll talk about in a later chapter.*

Hoisting

When a JavaScript file (or HTML document with JavaScript in it) is loaded, *function declarations* are *hoisted* to the top of the code by the browser before any code is executed.

What does that mean, exactly?

Specifically, all of the functions written with function declarations are "known" before any code is run. This allows you to call a function before you declare.

```
/**
 * This works!
 */
function add(num1, num2) {
    return num1 + num2;
}
add(3, 3); // returns 7


/**
 * This does, too!
 */
subtract(7, 4); // returns 3
function subtract(num1, num2) {
    return num1 - num2;
}
```

Function expressions, however, do **not** hoist. If you try to run a function before you've expressed it, you'll get an error.

```
/**
 * This works!
 */
var add = function(num1, num2) {
    return num1 + num2;
};
add(3, 3); // returns 7


/**
 * This does not =(
 */
substract(7, 4); // returns Uncaught TypeErr
or: subtract is not a function
var subtract = function (num1, num2) {
    return num1 - num2;
};
```

Which one should you use?

Which one you chose is almost entirely a matter of personal taste.

I think the more important thing is to pick one style of writing functions and stick with it throughout your script. If you *need* to call a function before it's declared, then, of course, use *function declarations*.

I personally prefer *function expressions*, and use them extensively in all of my code. I like them for one reason: they force more structure into the code base.

With *function expressions*, I cannot call a function before I've expressed it, so my code has a specific, ordered structure to it.

Functions that call other functions are written in a specific order every time, because they won't work otherwise. And any code I run to kick things off happens at the end of the file.

Reading my code is a predictable, structured activity. But again, this is entirely personal preference, and either one works great.

Scope

Scope is the context in which a function or variable is accessible. There are three types of scope: *global*, *local*, and *lexical*.

At a high level, functions have access to variables and other functions set outside themselves, but not variables set inside other functions.

Global Scope

A variable or function in the *global scope* is accessible inside other functions.

```
// this is in the global scope
var sandwich = 'tuna';

var logSandwich = function () {
    // Will log `tuna` in the console
    // It can access sandwich because it's i
n the global scope
    console.log(sandwich);
};
logSandwich();

// Will also log `tuna` in the console
console.log(sandwich);
```

Local Scope

A variable or function that's only accessible in a part of your code base has *local scope*.

```
var logSandwich = function () {
    // this has variable local scope
    var sandwich = 'tuna';

    // Will log `tuna` in the console
    // It can access sandwich because it's s
cope is local to the function
    console.log(sandwich);
};
logSandwich();

// returns "Uncaught ReferenceError: sandwic
h is not defined"
// `sandwich` is local to the logSandwich()
function, and not accessible here
console.log(sandwich);
```

Lexical Scope

If you nest your functions, variables and other functions defined in the parent function have *lexical scope* and can be accessed by the inner funtions. The parent function cannot access variables or functions defined within the inner functions.

```
var sandwiches = function () {

    // this is in the lexical scope
```

```
    // chips is in the lexical scope
    var sandwich = 'tuna';

    var logSandwich = function () {

        // Will log `tuna` in the console
        // It can access sandwich because it
's in the lexical scope
        console.log(sandwich);

        // Will log `chips` because it's in
the local scope
        var snack = 'chips';
        console.log(snack);

    };
    logSandwich();

    // Will also log `tuna` in the console
    console.log(sandwich);

    // returns "Uncaught ReferenceError: sna
ck is not defined"
    // `snack` is local to the logSandwich()
 function, and out of the lexical scope
    console.log(snack);

};
sandwiches();
```

Defining and updating variables

Prefixing a variable with `var` defines a new variable. Omitting `var` updates an existing variable.

There are two caveats to this:

1. If a variable is already defined in the current scope, prefixing it with `var` will throw an error.
2. If a variable isn't currently defined, omitting `var` creates a new variable (you should always use `var` to define a new variable, though).

You can define a variable in a function that has the same name as a *global* or *lexical* variable without modifying that variable.

```
var sandwich = 'tuna';

var logSandwich = function () {
    // logs "turkey"
    // Does NOT update the global `sandwich`
 variable
    var sandwich = 'turkey';
    console.log(sandwich);

};
logSandwich();

// logs "tuna"
console.log(sandwich);
```

If you omit the leading `var`, you can update a variable in the *global* or *lexical* scope from within a function.

```
var sandwich = 'tuna';

// logs "tuna"
console.log(sandwich);

var logSandwich = function () {
    // logs "tuna"
    console.log(sandwich);

    // Updates `sandwich` in the global scop
e
    sandwich = 'turkey';

    // logs "turkey"
    console.log(sandwich);

};
logSandwich();

// logs "turkey"
console.log(sandwich);
```

Keeping code out of the global scope

There are times you may want to expose a function or variable to the *global scope* (for example, a lightweight framework you want other scripts to be able to use).

But generally speaking, you want to keep your functions and variables out of the global scope. Otherwise, if another script or developer defines a variable or function that has the same name as the ones in your script, it will override them or introduce conflicts.

You can move your code into a *lexical scope* by wrapping it in a function.

```
// Wrapper for your code
var myScripts = function () {
    // Your codes goes here...
};

// Run your scripts
myScripts();
```

If you want your code to run immediately when the file runs without having to call your function, you can use something called an Immediately Invoked Function Expression (or IIFE)[1]. An IIFE is an anonymous (as in, unnamed) function that runs immediately.

```
;(function (window, document, undefined) {
    // Your code goes here...
})(window, document);
```

Conditional Variables

Sometimes the value of a variable changes based on another value or piece of data.

There are three ways to conditionally set values:

1. `if...else` statements
2. Conditional operators
3. Ternary operators

`if...else` statements

This is the most obvious way to conditionally define a variable.

In the example below, we have a function, `getSandwich()`. If the `useMayo` variable is `true`, we want a turkey sandiwhich. Otherwise, we want peanut butter and jelly.

Here's how to write that with an `if...else` statement.

```
// Use mayonaise?
var useMayo = true;

// Get the sandwich type to make
var getSandwich = function () {
    var sandwich;
    if (useMayo) {
        sandwich = 'turkey';
    } else {
        sandwich = 'peanut butter & jelly';
    }
    return sandwich;
};
```

Conditional operators

You can use a *conditional operator*, `||`, to define a fallback value for a variable if the preferred value doesn't exist.

In our `getSandwich()` function, let's check to see if `useMayo` exists, and if not, provide a default value.

```
var getSandwich = function (useMayo) {

    // Provide a default value for useMayo
    useMayo = useMayo || true;

    // Get the sandwich type
    var sandwich;
    if (useMayo) {
        sandwich = 'turkey';
    } else {
        sandwich = 'peanut butter & jelly';
    }

    return sandwich;

};
```

Ternary operators

A *ternary operator* provides a shorter way to write `if...else` statements. It has three parts:

```
var someVar = [the condition] ? [the value if true] : [the value if false];
```

It's the equivalent of this.

```
var someVar;

if ([the condition]) {
    someVar = [the value if true];
} else {
    someVar = [the value if false];
}
```

We can rewrite our `getSandwich()` function to define `sandwich` using a ternary operator.

```
var getSandwich = function (useMayo) {

    // Provide a default value for useMayo
    useMayo = useMayo || true;

    // Get the sandwich type
    var sandwich = useMayo ? 'turkey' : 'pea
nut butter & jelly';

    return sandwich;

};
```

Browser Compatibility

All three approaches work in all modern browsers, and back to at least IE6.

Arguments

You can pass information into a function as variables. These variables are called *arguments*.

For example, in the `add()` function below, `num1` and `num2` are arguments.

```
var add = function (num1, num2) {
    return num1 + num2;
};
```

Then you would pass in two numbers to add like this.

```
add(3, 4);
// returns "7"
```

Default values

If you don't pass in a value for an argument, your script will use `undefined`. This can result in some unexpected results.

```
add(3);
// returns "NaN"
```

You can handle this one of two ways:

1. Make sure an argument exists before using it.
2. Set a default value for an argument.

To check that an argument exists, you'd set up an `if` statement, and use a bang (`!`) to test of the variable exists. If it fails, call `return` to end the function.

```
var add = function (num1, num2) {

    // If num1 or num2 aren't defined, bail
    if (!num1 || !num2) return;

    // Add the numbers
    return num1 + num2;

};
```

To set a default value for an argument, you can redefine it (without the `var` prefix). For ease, we'll use a ternary or conditional operator.

```
var add = function (num1, num2) {

    // If num1 or num2 aren't defined, set t
hem to 0
    num1 = num1 || 0; // conditional operato
r
    num2 = num2 ? num2 : 0; // ternary opera
tor

    // Add the numbers
    return num1 + num2;

};
```

Get all arguments passed in to a function

Within any function, you can use the `arguments` variable to get an array-like list of all of the arguments passed into the function.

You don't need to define it ahead of time. It's a native JavaScript object.

```
var add = function (num1, num2) {

    // returns the value of `num1`
    console.log(arguments[0]);

    // returns the value of `num2`
    console.log(arguments[1]);

    // ...

};
```

This is particularly useful if you would rather allow an unlimited number of arguments to be passed in to your function.

Let's say you wanted to be able to pass an unlimited amount of numbers into `add()` and add them together. The `arguments` variable is perfect for this!

```
var add = function () {

    // Set a starting total
    var total = 0;

    // Add each number to the total
    for (var i = 0; i < arguments.length; i+
+) {
        total += arguments[i];
    }

    // Return to the total
    return total;

};
```

In the example above, we're defining a default variable `total` with a value `0`. We use a `for` loop to iterate over each argument and add it to the total. Then we return the total.

If someone passes in no arguments, it returns `0`. Otherwise, they can add one or more numbers together.

Browser Compatibility

The `arguments` variable works in all modern browsers, and back to at least IE6.

Functions and Methods

You know what a function is, but you may have also heard the term *method* used before and wondered what it mean.

So what's the difference between a function and a method? Nothing. They're the same thing.

In languages other than JavaScript, the two terms have specific meanings. And JavaScript developers will sometimes apply their own definition to the two terms. The one I see most often is that a method is a function that's inside an object like this.

```
var helperLibrar = {
    method1: function () {
        return 'This is a method, supposedly
';
    }
};
```

The problem with these definitions is that no one agrees on them, and many, many JavaScript developers use *function* and *method* interchangeably.

In my opinion, they're the same thing.

Arrow Functions

Arrow functions were introduced to JavaScript in ES6.

Their intent was to provide a shorter syntax for writing functions and eliminate some of the confusion that exists around `this`. Because they look so dramatically different from traditional functions, though, they often make scripts *more* confusing.

However, more and more scripts and tutorials are being written with arrow functions, so it's important that you at the very least understand how they work and can read code written with them.

Basic Syntax

A basic arrow function isn't all that different from a traditional function. The word `function` gets dropped, an a fat arrow (=>) is added between the parentheses and brackets (`()` and `{}`, respectively).

```
// A traditional function
var add = function (num1, num2) {
    return num1 + num2;
};

// The arrow function version
var add = (num1, num2) => {
    return num1 + num2;
};
```

Note: *Named arrow functions have to be written as a function declaration. There's no way to write one as a function expression.*

A simpler way to return a value

If your function is only returning a value, as is the case with our `add()` function, you can simplify the function even further by dropping the brackets and `return`.

```
// returns `num1 + num2`
var add = (num1, num2) => num1 + num2;
```

This only works if the only thing you're doing is returning a value. If you need to do more stuff with your function, you have to use brackets.

Specifying argument defaults

In the last lesson, we specified defaults for `num1` and `num2` if they weren't passed in. Arrow functions make this even easier to do.

You can set a default value for each argument at the time that you give it a name with `= value`.

```
// Setting defaults with a traditional funct
ion
var add = function (num1, num2) {

    // If num1 or num2 aren't defined, set t
hem to 0
    num1 = num1 || 0; // conditional operato
r
    num2 = num2 ? num2 : 0; // ternary opera
tor

    // Add the numbers
    return num1 + num2;

};

// Setting defaults with an arrow function
var add = (num1 = 0, num2 = 0) => num1 + num
2;
```

`arguments` doesn't work

In the last lesson, we used the `arguments` variable to get all of the arguments passed in to our function and add them together.

```
var add = function () {

    // Set a starting total
    var total = 0;

    // Add each number to the total
    for (var i = 0; i < arguments.length; i+
+) {
        total += arguments[i];
    }

    // Return to the total
    return total;

};
```

In arrow functions, `arguments` doesn't exist.

It's been replaced by *rest parameters*. They work a lot like `arguments`, but with two notable advantages.

1. You can assign them to any variable name you'd like.
2. You can start at any argument you want.

You define *rest parameters* by passing in an argument prefixed with `...`.

```
var logStuff = (arg1, arg2, ...moreArgs) =>
{

    // Logs arg1
    console.log(arg1);

    // Logs arg2
    console.log(arg2);

    // Logs an array of any other arguments
you pass in after arg2
    console.log(moreArgs);

};

// In this example...
// arg1 = 'chicken'
// arg2 = 'tuna'
// moreArgs = ['chips', 'cookie', 'soda', 'd
elicious']
logStuff('chicken', 'tuna', 'chips', 'cookie
', 'soda', 'delicious');
```

Here's our `add()` function rewritten as an arrow function.

```
var add = (...args) => {

    // Set a starting total
    var total = 0;

    // Add each number to the total
    for (var i = 0; i < args.length; i++) {
        total += args[i];
    }

    // Return to the total
    return total;

};
```

Browser Compatibility

Arrow functions work in all modern browsers, Safari 10 and up, and Mobile Chrome and Android 45 and up. They have no IE support.

Since support for them is pretty limited at this time, you would need to use a compiler like Babel[2] to convert them to traditional functions for broader browser support. Unfortunately, they cannot be polyfilled.

Babel does actually have an “in the browser” version you can load with a script tag, *but*… it requires you to inline your entire script, so it’s not really a good solution for production sites.

`var`, `let`, and `const` (and when to use which)

ES6 introduced two new ways to define variables: `let` and `const`.

They've been the source of a fair bit of confusion, particularly around when to use which. Let's clear that up.

`let`

`let` does the *almost* the same exact thing as `var`.

The big difference between `let` and `var` is that you can't redefine a variable set with `let` in the same scope.

```
// The value of `sandwich` is "tuna"
var sandwich = 'tuna';

// The value of `sandwich` is now "chicken"
var sandwich = 'chicken';

// The value of `chips` is "Cape Cod"
let chips = 'Cape Cod';

// Throws an error: "Uncaught SyntaxError: I
dentifier 'chips' has already been declared"
let chips = 'Lays';
```

You can still change the value of `chips`. You just can't define it as a new variable once it's already been defined *within the current scope*. You *can* use `let` to define a new variable with the same name in a different scope, though.

```
// The value of `chips` is "Cape Cod"
let chips = 'Cape Cod';

// The value of `chips` is now "Lays"
chips = 'Lays';

var getChips = function () {

    // This works because it's a different s
cope
    let chips = 'Wise';

    // Returns "Wise"
    return chips;

};

// Logs "Lays" in the console
console.log(chips);
```

const

Unlike `var` and `let`, if you define a variable with `const`, it cannot be given a new value. It is, as the term implies, constant.

```
// The value of sandwich is "tuna"
const sandwich = 'tuna';

// Throws an error: "Uncaught TypeError: Ass
ignment to constant variable."
sandwich = 'chicken';
```

Browser Compatibility

`let` and `const` work in all modern browsers, and IE11 and up. They cannot be polyfilled.

To push support back further, you would need to use a compiler like Babel[3]. Babel does actually have an "in the browser" version you can load with a script tag, *but*... it requires you to inline your entire script, so it's not really a good solution for production sites.

Putting it all together

To make this all tangible, let's work on a project together. We'll take a small plugin that lazy loads images and clean it up a bit. We want to make it backwards compatible, and help eliminate a few potential bugs.

The starter template and complete project code are included in the source code[4] on GitHub.

Getting Setup

A working version of the plugin is in the template file.

Throughout the page copy, I've included empty `<figure>` elements that will eventually contain our lazy loaded images. Each one has the `.lazy-load` class on it, which the script uses as a hook to get the images that should be lazy loaded.

The `[data-image]` attribute includes a URL to the image to be lazy loaded, and the `[data-caption]` attribute (if included) is the caption we'd like to include with our image.

```
<figure class="lazy-load" data-caption="A li
feguard station on a deserted beach" data-im
age="img/beach.jpg"></figure>
```

The template also includes some lightweight CSS to make our images responsive, and our captions look good.

```
figure {
    margin: 0 0 1.4em;
}

caption {
    color: #808080;
    display: block;
    font-size: 0.8em;
    font-style: italic;
    padding: 0.25em 0 0.5em;
    text-align: center;
}

img {
    height: auto;
    max-width: 100%;
}
```

And finally, here's the JavaScript that makes it all work.

```
// Get our lazy load images
var images = document.querySelectorAll('.laz
y-load');

// Listen for scroll events
window.addEventListener('scroll', loadImages
```

```
, false);

// Determine if an element is in the viewpor
t
var isInViewport = (elem) => {
    var distance = elem.getBoundingClientRec
t();
    return (
        distance.top >= 0 &&
        distance.left >= 0 &&
        distance.bottom <= (window.innerHeig
ht || document.documentElement.clientHeight)
 &&
        distance.right <= (window.innerWidth
 || document.documentElement.clientWidth)
    );
};

// Insert caption into the DOM
var addCaption = (img, text) => {
    var caption = document.createElement('ca
ption');
    caption.innerHTML = text;
    img.parentNode.insertBefore(caption, img
.nextSibling);
};

// Load our images
function loadImages () {
```

```
    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++)
{

        // Check if the image is in the view
port
        if (isInViewport(images[i])) {

            // Make sure the image has a dat
a-image attribute and hasn't already been lo
aded
            if (!images[i].hasAttribute('dat
a-image') || images[i].classList.contains('l
oaded')) continue;

            // Load the image
            images[i].innerHTML = '<img src=
"' + images[i].getAttribute('data-image') +
'">';

            // Add a caption
            var img = images[i].querySelecto
r('img');
            var text = images[i].getAttribut
e('data-caption');
            addCaption(img, text);

            // Add a .loaded class to our im
age placeholder
            images[i].classList.add('loaded'
```

```
                images[i].classList.add('loaded
);

            }

        }

}

// Load images on page load
loadImages();
```

As you can see, there's a mix of traditional functions and arrow functions, function declarations and function expressions, and all of our code is in the global scope.

Alright, let's clean this script up!

Converting to traditional functions

Arrow functions have poor backwards compatibility, and we're not using a compiling tool like Babel to convert this script. Let's convert our two arrow functions—`isInViewport()` and `addCaption()`—to traditional functions for broader browser support.

To do this, we'll add the word `function` before the arguments, and remove the fat arrow.

```
// Determine if an element is in the viewpor
t
var isInViewport = function (elem) {
    var distance = elem.getBoundingClientRec
t();
    return (
        distance.top >= 0 &&
        distance.left >= 0 &&
        distance.bottom <= (window.innerHeig
ht || document.documentElement.clientHeight)
 &&
        distance.right <= (window.innerWidth
 || document.documentElement.clientWidth)
    );
};

// Insert caption into the DOM
var addCaption = function (img, text) {
    var caption = document.createElement('ca
ption');
    caption.innerHTML = text;
    img.parentNode.insertBefore(caption, img
.nextSibling);
};
```

Function Declarations or Function Expressions: Pick One

The script currently uses a mix of function declarations and function expressions. We should pick one style and stick with it.

For learning purposes, let's go with function expressions, since they force a bit more structure into the code.

We'll conver `loadImages()` by removing the leading `function`, adding `var` before the function name and `= function` between the name and the parentheses. We also need to add a semicolon at the end.

```
// Load our images
var loadImages = function () {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++)
{

        // Check if the image is in the view
port
        if (isInViewport(images[i])) {

            // Make sure the image has a dat
a-image attribute and hasn't already been lo
aded
            if (!images[i].hasAttribute('dat
```

```
a-image') || images[i].classList.contains('l
oaded')) continue;

            // Load the image
            images[i].innerHTML = '<img src=
"' + images[i].getAttribute('data-image') +
'">';

            // Add a caption
            var img = images[i].querySelecto
r('img');
            var text = images[i].getAttribut
e('data-caption');
            addCaption(img, text);

            // Add a .loaded class to our im
age placeholder
            images[i].classList.add('loaded'
);

        }

    }

};
```

Checking for argument values

In the `addCaption()` function, the `text` argument is the caption text to use. Not every image has a caption, though, and those images have an empty `<caption></caption>` underneath them.

Let's make sure `text` has a value, and if not, end the function early.

```
// Insert caption into the DOM
var addCaption = function (img, text) {
    if (!text) return;
    var caption = document.createElement('ca
ption');
    caption.innerHTML = text;
    img.parentNode.insertBefore(caption, img
.nextSibling);
};
```

Scoping our code

All of our code sits in the global scope. Lets wrap it in an IIFE to give it lexical scope.

```
;(function (window, document, undefined) {

    // Get our lazy load images
    var images = document.querySelectorAll('
```

```
.lazy-load');

    // Determine if an element is in the vie
wport
    var isInViewport = function (elem) {
        var distance = elem.getBoundingClien
tRect();
        return (
            distance.top >= 0 &&
            distance.left >= 0 &&
            distance.bottom <= (window.inner
Height || document.documentElement.clientHei
ght) &&
            distance.right <= (window.innerW
idth || document.documentElement.clientWidth
)
        );
    };

    // Insert caption into the DOM
    var addCaption = function (img, text) {
        if (!text) return;
        var caption = document.createElement
('caption');
        caption.innerHTML = text;
        img.parentNode.insertBefore(caption,
 img.nextSibling);
    };

    // Load our images
```

```
    var loadImages = function () {

        // Loop through each lazy load image
        for (var i = 0; i < images.length; i
++) {

            // Check if the image is in the
viewport
            if (isInViewport(images[i])) {

                // Make sure the image has a
 data-image attribute and hasn't already bee
n loaded
                if (!images[i].hasAttribute(
'data-image') || images[i].classList.contain
s('loaded')) continue;

                // Load the image
                images[i].innerHTML = '<img
src="' + images[i].getAttribute('data-image'
) + '">';

                // Add a caption
                var img = images[i].querySel
ector('img');
                var text = images[i].getAttr
ibute('data-caption');
                addCaption(img, text);
```

```
                    // Add a .loaded class to ou
r image placeholder
                    images[i].classList.add('loa
ded');

                }

            }

        };

        // Listen for scroll events
        window.addEventListener('scroll', loadIm
ages, false);

        // Load images on page load
        loadImages();

})(window, document);
```

Now our code is out of the global scope.

Congratulations! You just cleaned up a script with consistent function style, better backwards compatibility, and proper scope.

About the Author

Hi, I'm Chris Ferdinandi. I help people learn JavaScript.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts. I run Go Make Things with Bailey Puppy, a lab-mix from Tennessee.

You can find me:

- On my website at GoMakeThings.com.
- By email at chris@gomakethings.com.
- On Twitter at @ChrisFerdinandi.

1. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/boilerplates/iife.js↩
2. https://babeljs.io↩
3. https://babeljs.io↩
4. https://github.com/cferdinandi/variable-function-scope-source-code/↩