Technical Report: An Asynchronous Call Graph for JavaScript

Dominik Seifert
d01922031@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Michael Wan
b07201003@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Jane Hsu
yjhsu@csie.ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Benson Yeh
pcyeh@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

ABSTRACT

This Technical Report serves as a supplementary document to the "An Asynchronous Call Graph for JavaScript" article. It provides extra background information and showcases results.

CCS CONCEPTS

• **Software and its engineering** → **Concurrent programming structures**.

ACM Reference Format:
Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2021. Technical Report: An Asynchronous Call Graph for JavaScript. In *Proceedings of* . ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

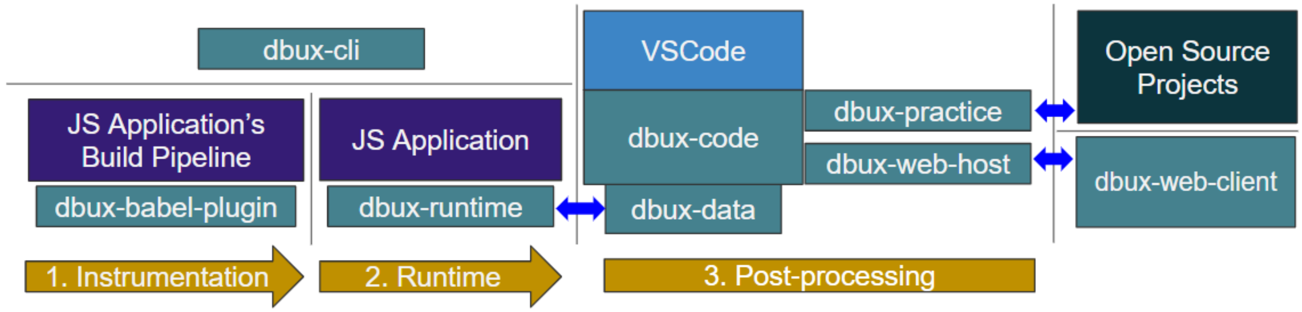

Figure 1: Dbux Architecture.

1 A BRIEF INTRODUCTION TO DBUX

1.1 Architecture

Dbux [3] has four applications and several supplementary modules, as depicted in Fig. 1. Several shared modules, such as the `dbux-common` modules, are not shown.

Dbux's three stages *instrument*, *runtime* and *post-processing* are implemented in four collaborating applications:

- `dbux-babel-plugin` instruments the target application and injects the `dbux-runtime`. It requires to be run with Babel [1].
- `dbux-runtime` records the target application's execution trace and streams it to a server in real time.
- `dbux-code` is a one-click-installable extension to VSCode, available on the VSCode Marketplace, complete with extensive documentation. It also has prepared several real-world projects, bugs and experiments to try it out on. Upon activation, it starts a server to wait for the execution data produced by `dbux-runtime`. When received, it *post-processes* it with the help of the `dbux-data` module before presenting it to the user. Data is processed and presented as soon as it is received, meaning that applications can be debugged while they are still running.
- `dbux-cli` is to Dbux, what "nyc" is to the coverage reporter Istanbul [1], that is: a convenient command line tool that makes it easier for developers to execute a JS application with Dbux enabled, without having to prepare a build pipeline. Instead, it uses a modified version of `@babel/register` [2] to inject `dbux-babel-plugin` on the fly.

1.2 Call Graph Assembly

NOTE: In the following, we refer to the "dynamic call graph" just as "call graph". Dbux does not have a static call graph.

We model the call graph as follows:

(1) We refer to a call graph node, that is the recorded execution of a file or function, as an "executionContext", or **context** for short. Given a function f, we denote the context that represents the i'th execution of some function f as f_i.

(2) Edges represent the caller - callee relationship. For uninterruptible functions: if during it's i'th execution, f calls some function g, then, for some j, g_j is a child —or **callee**— of f_i, and f_i is a parent —or **caller**— of g_j.

(3) Any function execution f_i is considered a Call Graph Root (CGR), if it has no parent caller. This implies that the function was either directly invoked by the JavaScript engine's event queue, or the first invoked function was not recorded.

(4) Traditionally the above rule set was sufficient to build a JavaScript dynamic call graph. However, ES2017 [2] introduced `async` functions, which need special attention due to their property of interruptibility: we refer to the i^{th} execution of some `async` function h as h_i. The context h_i is considered a **real context**. When executed, a **virtual context** h_i^1 is added as a child to h_i. Furthermore, any `await p;` expression tells the scheduler to **interrupt** the current control flow for one tick of the asynchronous queue, or, if `p` is a promise, until that promise has been settled. Once `await` has concluded, that is, after the promise has settled, execution of h_i continues. At this point, our instrumentation adds a new virtual context h_i^k. That context represents the asynchronous continuation of the interrupted real context h_i at a later point in time. We assume that each virtual context $h_i^k, i > 1$ is also a CGR.

In general, all events, contexts and roots are ordered by time of occurrence. Dbux's synchronous call graph implementation renders all CGRs linearly in that order.

[1] https://istanbul.js.org/

[2] https://babeljs.io/docs/en/babel-register

1.3 Developer Survey

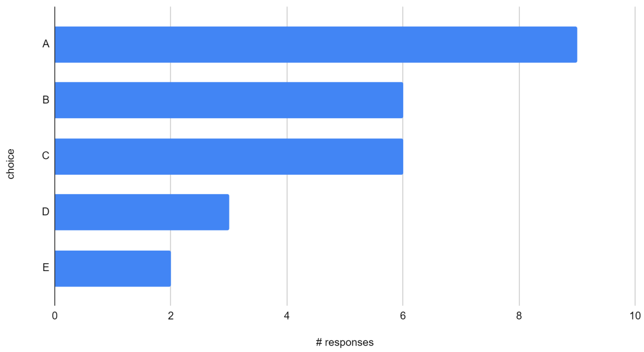

Figure 2: Survey Results: What type of programming problems are the most difficult to deal with? (A) Asynchronous behavior (setTimeout; setInterval; Process.next; promise; async/await etc.) (B) Third-party APIs (e.g. Node API, Browser API, other people's libraries, modules etc.) (C) Programming logic (D) Syntax (E) Events.

During a workshop in summer 2020 that introduced Dbux to 20 TAs of a local JavaScript Bootcamp provider, we asked the participants what type of bugs they found most difficult to deal with. A total of 10 participants filled out our survey. The top choice for the (multiple choice) "programming problems" was "asynchronous behavior" with 9 votes, while the second place only received 6.

2 ASYNCHRONOUS SEMANTICS PRIMER

Fig. 3 illustrates the three types of Asynchronous Events (AE). In all three cases, the resulting Asynchronous Call Graph (ACG) should feature three nodes, connected by two CHAINs.

Below are several illustrations of asynchronous programs and their expected conceptual ACG.

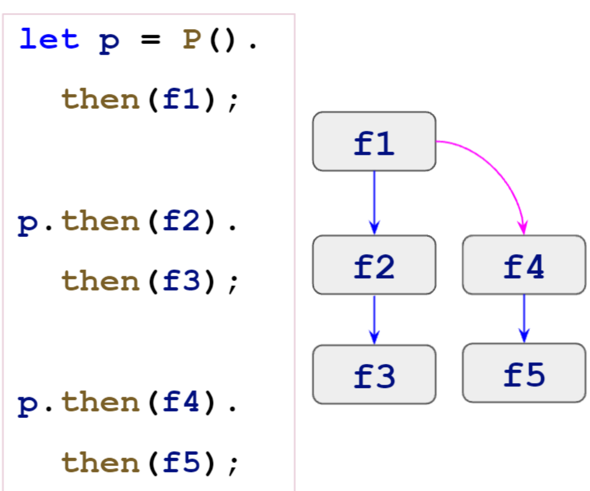

Figure 4: Promises (CHAIN vs. FORK)

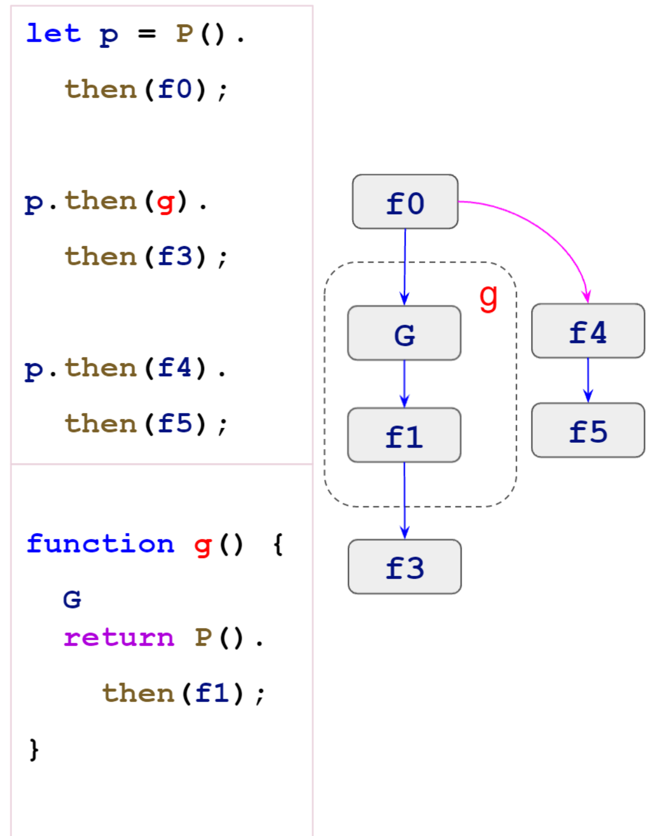

Figure 5: Nested Promises (CHAIN vs. FORK)

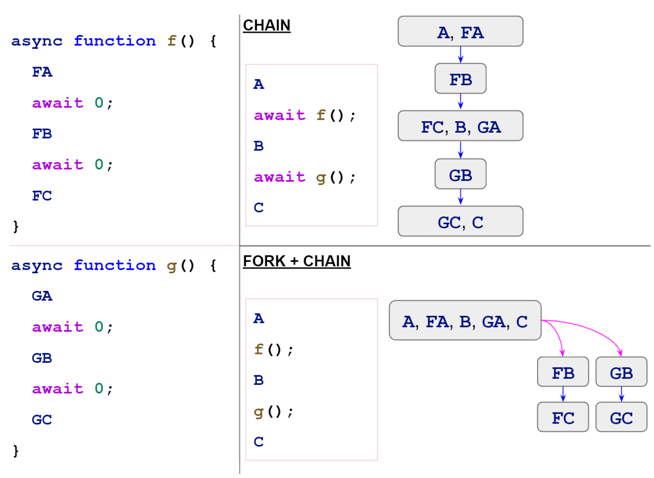

Figure 6: AWAIT (CHAIN vs. FORK)

2.1 Promise Creation Semantics

To better motivate PromiseLinks, consider the following. In JavaScript, promises can be created in four ways. Somewhat counter-intuitively, (i), (ii) and (iii) do *not* cause an asynchronous event on their own. However, all of them can nest promises:

```
async function send(fpath) {
  const file = await openFile(fpath);

  const cont = await readFile(file);



  await sendFile(cont);



  console.log('File sent!');
}
```

```
function send(fpath) {
  return openFile(fpath).
    then(function (file) {
      return readFile(file);
    }).
    then(function (cont) {
      return sendFile(cont);
    }).
    then(function () {
      console.log('File sent!');
    });
}
```

```
function send(fpath, cb) {
  openFile(fpath, function (file) {

    readFile(file, function (cont) {


      sendFile(cont, function () {
        cb && cb();

        console.log('File sent!');
      });
    });
  });
}
```

Figure 3: Three types of AEs implementing a series of three operations: `openFile` → `readFile` → `sendFile`

The (i) Promise constructor takes an executor function which in turn is provided two parameters: the `resolve` and `reject` functions which are to be called to fulfill the promise. The executor function is called synchronously from the constructor. The Promise constructor is commonly used to wrap asynchronous callbacks into promises. This process is commonly referred to as "promisification".

(ii) `Promise.resolve(x)` and `Promise.reject(x)` are equivalent to using the (i) Promise constructor and synchronously calling `resolve` or `reject` respectively. `Promise.all` and `Promise.race` can further be used to nest multiple promises into one.

When (iii) an async function is called, the runtime environment creates a new promise. Its call expression value is set to that promise. Async functions execute synchronously until the first `await` is encountered. This means that if an async function concluded without explicitly invoking an `await` expression or any of the three other types of events, it does not trigger an asynchronous event. Promises can further be nested by returning them from an async function.

(iv) promise chaining (`then`, `catch`, `finally`).

3 CONCURRENT DATA FLOW: RESULTS

These are the results from the "Concurrent Data Flow" extension on the three producer-consumer problems:

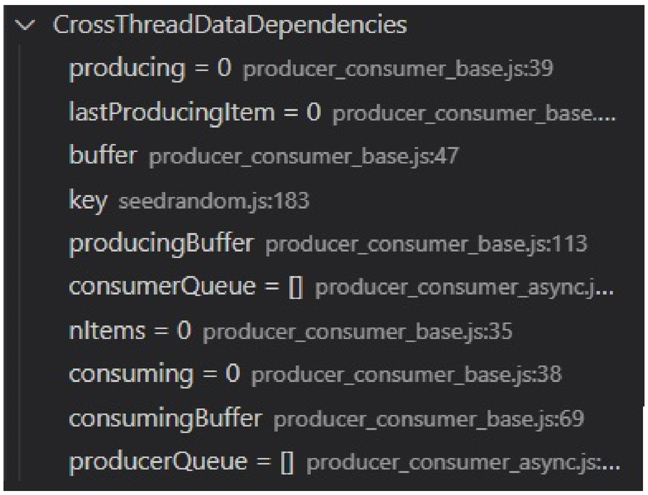

Figure 7: Async Function Implementation

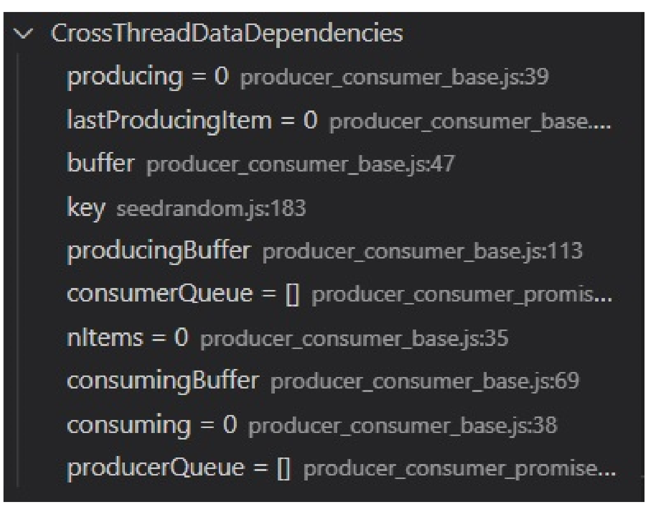

Figure 8: Promise Implementation

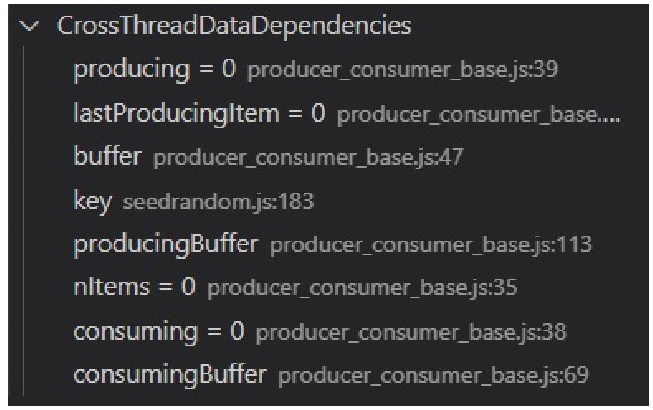

Figure 9: Callback Implementation

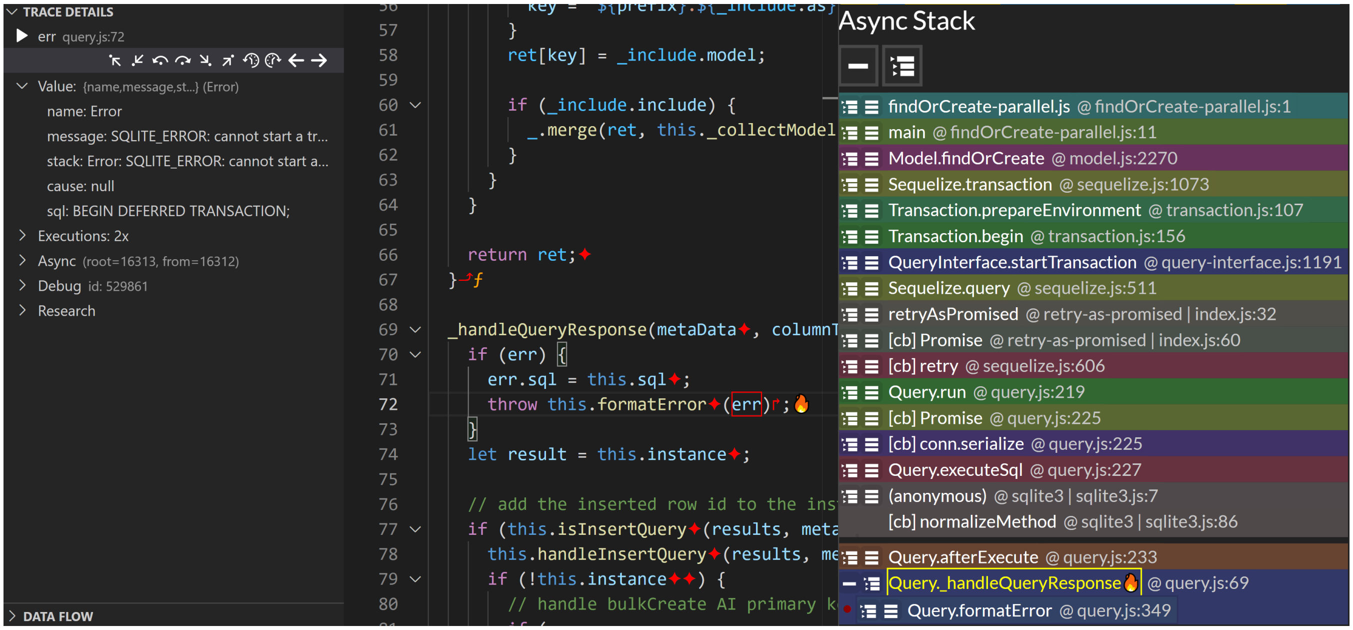

Figure 10: The sequelize ACS when the first thrown error is selected in the code.

4 PROJECT RESULTS

4.1 Project ACS

Fig. 10 shows the asynchronous call stack of the sequelize bug. The stack prominently features the sequelize API call that caused it: `findOrCreateCall`. For contrast: `err.stack` is empty. `formatError` returns a new `Error` object with its `stack` only containing the functions within its current CGR, up to `Query.afterExecute`.

4.2 Project ACGs

In the following, we list the raw ACG results of the nine projects that we had to omit from the article for brevity (enhanced for contrast).

TODO: Editor.md screenshot is missing because the application file importer cannot deal with a special property of the Editor.md log (it created contexts that were never sent to the server; possibly due to erroneous <script> loads). The data file is Ok, but the importer needs a bit more resilience. Aiming to fix within 1-2 weeks.

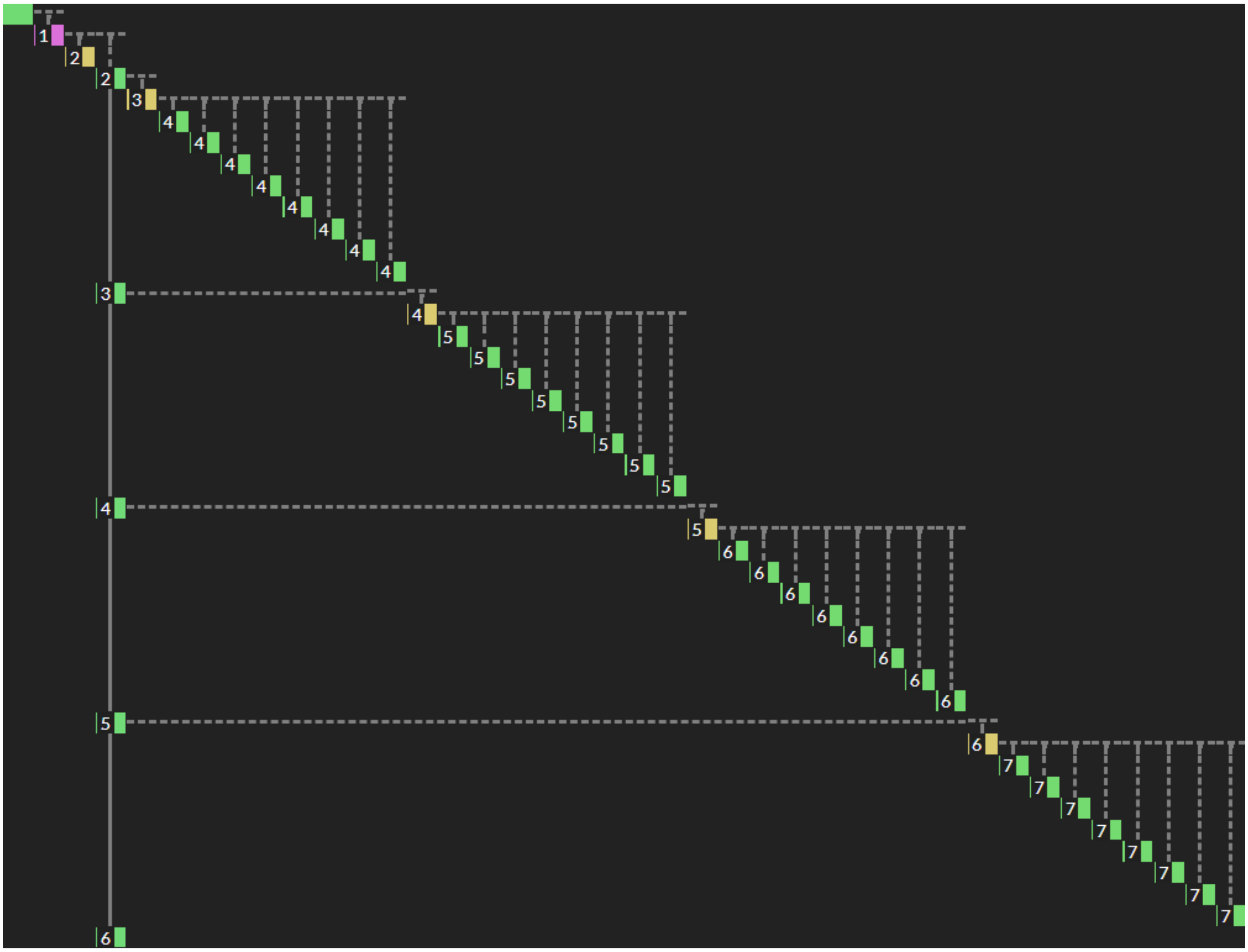

Figure 11: 2048

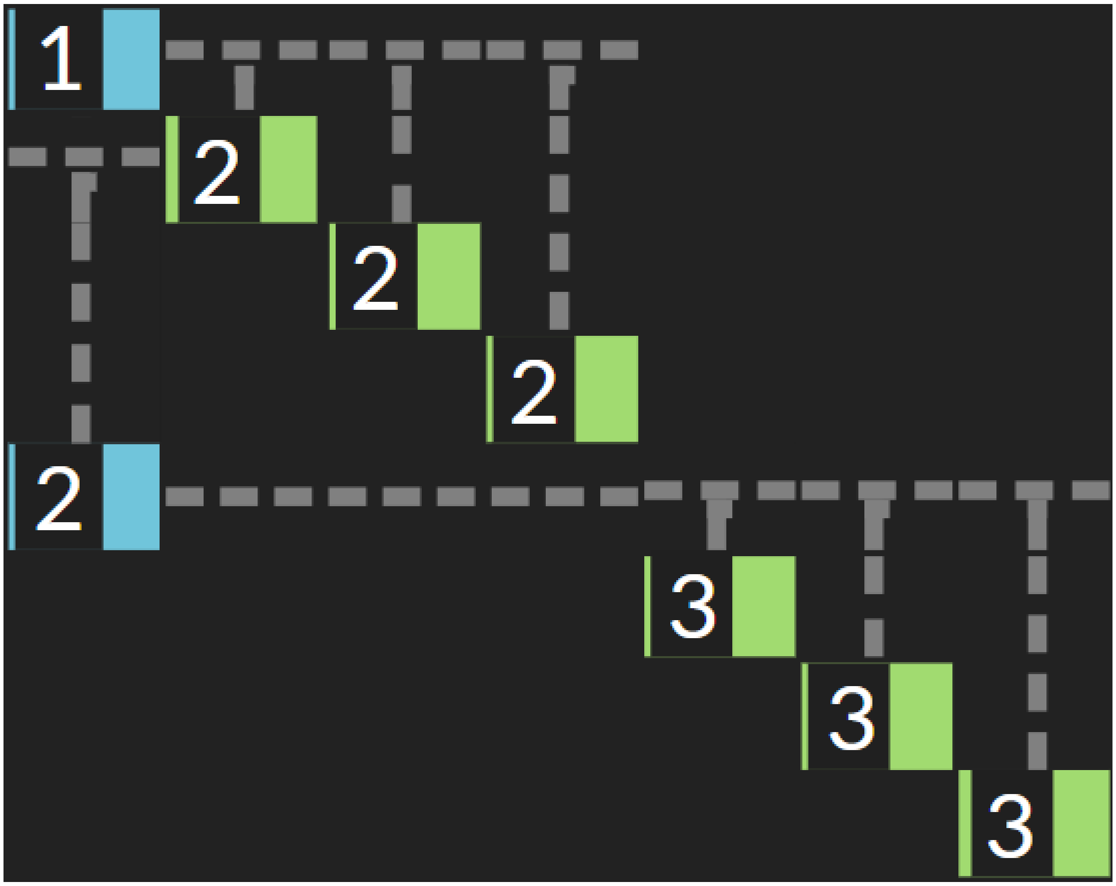

Figure 12: Bluebird

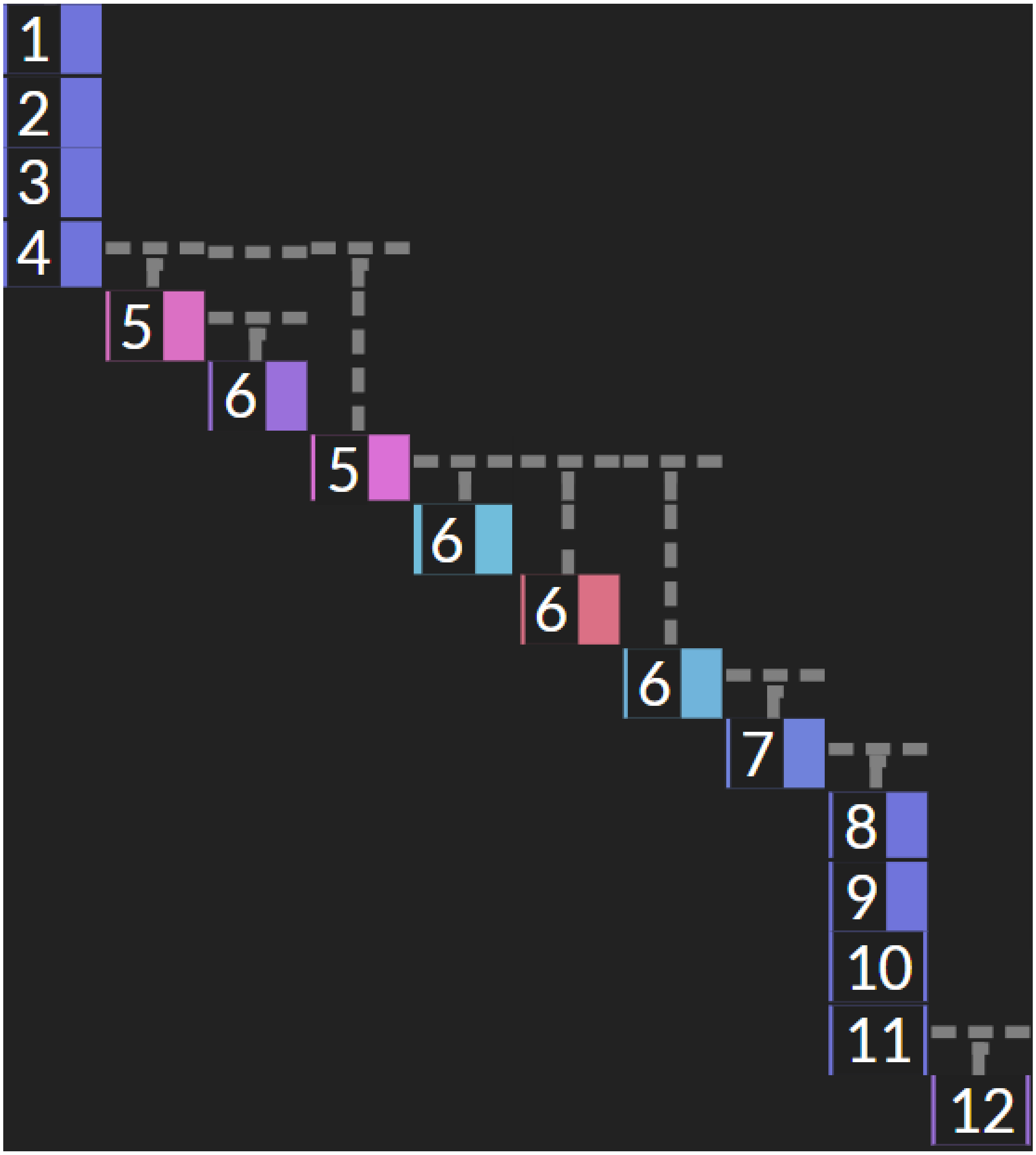

Figure 13: Express

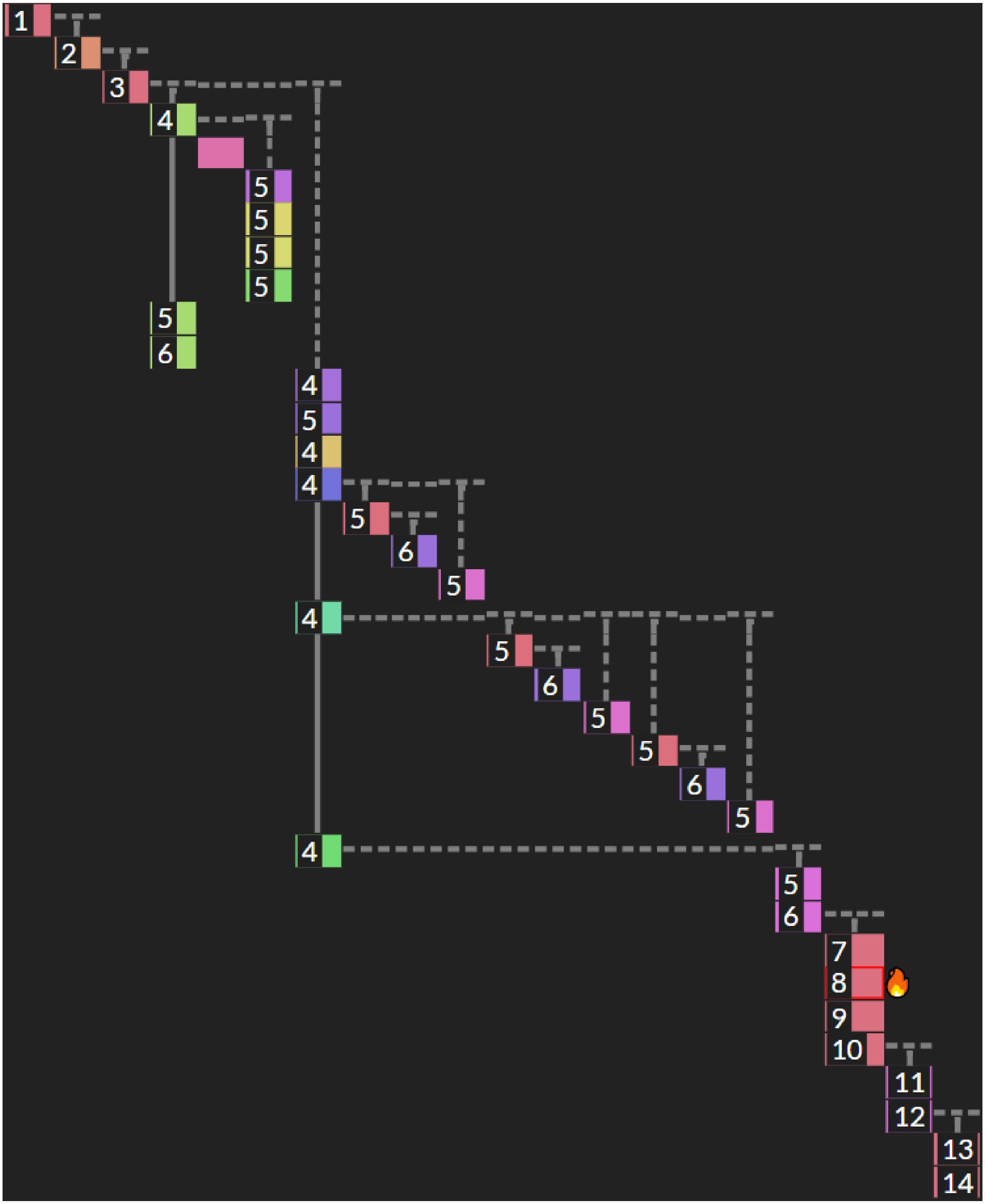

Figure 14: Hexo

Figure 15: node-fetch

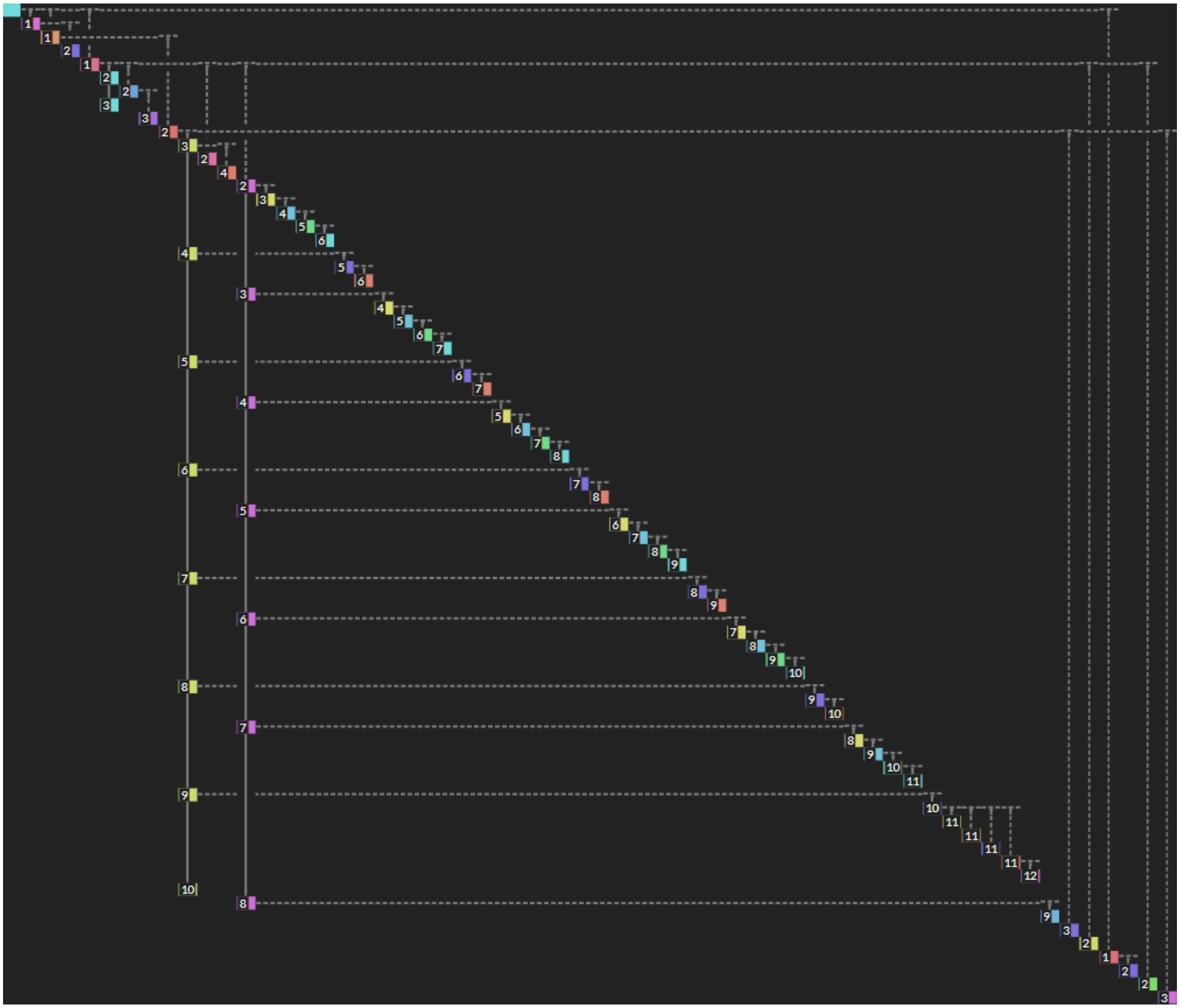

Figure 16: socket.io

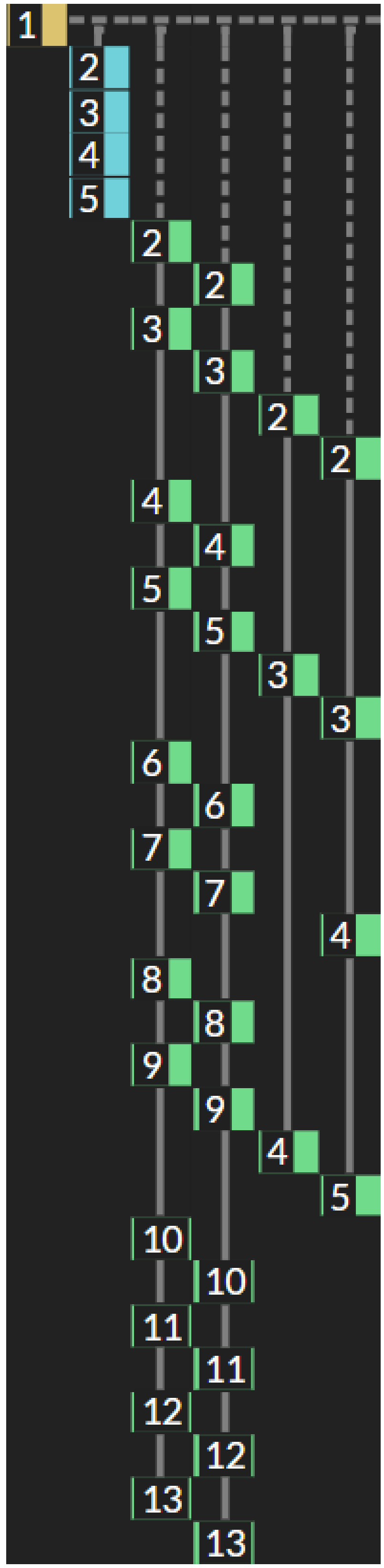

Figure 17: todomvc

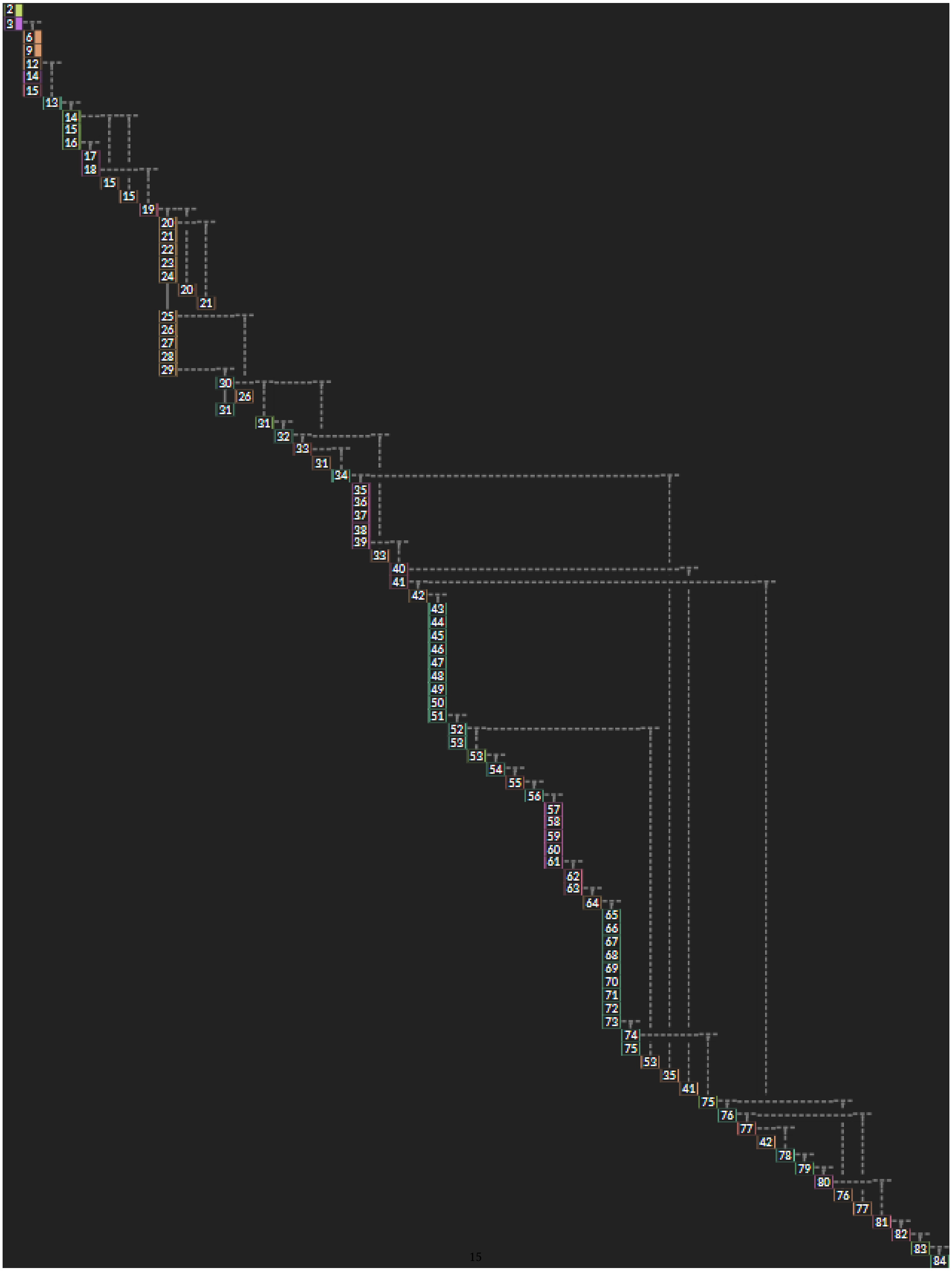

Figure 18: webpack

REFERENCES

[1] (accessed in 12/2020). *Babel*. https://babeljs.io/docs/en/index.html

[2] Ecma International. 2021. *ECMAScript® 2017 Language Specification*. Retrieved 10/2021 from https://262.ecma-international.org/8.0/

[3] Dominik Seifert and Michael Wan. 2019. *Dbux*. Retrieved 10/2021 from https://github.com/Domiii/dbux