Utility Libraries for Traversing and Manipulating Tree-like Data Structures with Varying Schemas

by

Adam Janicki

S.B., Computer Science and Engineering, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by: Adam Janicki
Department of Electrical Engineering and Computer Science
May 10, 2024

Certified by: David R. Karger
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Utility Libraries for Traversing and Manipulating Tree-like Data Structures with Varying Schemas

by

Adam Janicki

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Tree-like data structures are very commonly used data types found in the wild in a wide array of projects JavaScript projects. A specific example of one of these structures is an abstract syntax tree (AST). However, the lack of good libraries to handle trees has led to many developers and large-scale code bases having to implement their utility functions over and over again. To address these concerns within the JavaScript developer community, we propose Treecle and Vastly: two free open-source libraries that provide utility functions and operations to help developers work with trees and ASTs respectively.

Thesis supervisor: David R. Karger
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank David Karger for his guidance in helping me find a lab and project that I would thoroughly enjoy and be able to make an impact on. I would also like to extend the biggest thanks to Lea Verou, whose guidance and mentorship throughout this project have made this thesis both an enjoyable experience and one that has made me grow as a software engineer, computer scientist, and person.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Challenges and Motivation

Trees are everywhere in the field of software engineering and computer science; JavaScript and web-related projects are no exception. They can present themselves in a variety of ways: in obvious ones like the standard binary tree, and in more subtle ones like JSON [12].

This project stemmed from a desire to refactor and improve the handling of abstract syntax trees (ASTs) in Mavo [22]. Mavo is a low-code web framework for developing user interfaces to lower the barrier to entry of creating high-quality reactive websites. One of the key features of Mavo is its custom expression set, allowing users to use spreadsheet-like formulas to render data. Mavo handles these expressions by parsing them into an AST, performing some manipulations to get the data all in order, and then evaluating the tree. However, the existing code for handling these ASTs is all custom-implemented and could use improvement to clean up the already large code base.

The two libraries proposed in this project, Treecle and Vastly, snowballed from this desire to abstract away commonly used tree operations into separate and modular libraries. In this thesis, we define Treecle as the library built for handling a variety of traversal and manipulation operations on general tree structures which can have varying schemas. In

addition, we define Vastly, which is built on top of Treecle, as the library built for handling several operations on the more specific use case of ASTs. The goal of both libraries is simple: provide developers with a tool that is flexible, powerful, and easy to use.

1.2 Approach

As mentioned, the idea for Vastly originated from abstracting complex AST operations from an already existing large code base. While the original idea was to eliminate some complexity from Mavo's code base, we learned that a lot of the needed functionality could be generalized to many other projects. Therefore, we decided to make it widely accessible to anyone, not just us. We created an NPM [17] package to enable importing Vastly to anyone using ECMAScript modules [6], as well as bundling it so that it could be imported in a standard HTML script element.

After working on Vastly, we came to a second important revelation: while trying to abstract away common AST operations, we realized we were creating many standard tree operations in the process, which suggested that there is a need for a general library for handling any tree of any schema, not just the specific use case of an AST. Upon further investigation, we found that there did not exist a proper utility library for handling general tree-like data structures.

To fix that hole, and abstract many common tree operations that we found ourselves needing in Vastly, we created a more robust library for handling general and flexible operations on trees with customizable and varying schemas, **Treecle**. We realized that while Vastly caters to a niche and specific use case, Treecle could be more impactful given its flexible design, which lets it handle any custom-defined structure a user could want. Because of this, Vastly serves simultaneously as a novel contribution and as a use case of Treecle.

1.3 Contributions

In this thesis, we aim to produce two libraries, Treecle and Vastly, where each can handle a broad array of researched use cases needed by developers. Specifically, we set out to implement over 10 commonly needed operations for each library and validate each function by showing how it could be applied to popular JavaScript frameworks or getting direct feedback from actual developers via interviews, which is discussed in full detail in section 5.

In addition to the functionality in each library, we focus on creating high-quality open-source material, which includes test suites for each of the proposed functions, and rich documentation for each which can be found both in the source code and on our custom-built documentation sites. Due to the scope of this work, we will be focusing on implementation and impact, and less so on testing and documentation. A list of relevant links to Treecle and Vastly's sources are included in appendix A.

This paper proposes and discusses Treecle and Vastly, diving into their designs, purposes, features, implementations, and use cases. This paper is organized as follows: section 2 includes related work and background; section 3 introduces Treecle; section 4 covers Vastly; section 5 includes interview results from JavaScript developers; and section 6 concludes this paper.

Chapter 2

Related Work

2.1 Tree-like Data Structures

First, it is necessary to venture into what trees are, and the different variations and forms they can take in the context of this thesis. In graph theory, a tree is defined as an acyclic graph with n vertices and $n-1$ edges. Typically, we view trees in a top-down manner where there exists one root node, with an arbitrary number of children, where each of those children has an arbitrary number of children, and so on.

Importantly, in this thesis, we refer to trees in a general sense: each node in the tree can have an arbitrary number of children. We call this a directed n-ary tree, where n is the maximum number of children that any node has in the tree over its lifespan. For example, a binary tree is a 2-ary tree, since any node may have a maximum of 2 children. An example of an n-ary tree for $n = 3$ is shown in figure 2.1.

Nodes are described and characterized by a few main factors: their children, their metadata, and sometimes, their parent node. In terms of their children, nodes can either be complete (have the maximum n children), incomplete (have k children for some $0 < k < n$), or a leaf (have 0 children). Metadata refers to any other data stored in a node that does not indicate children or child pointers. For example, nodes might have a name or a value

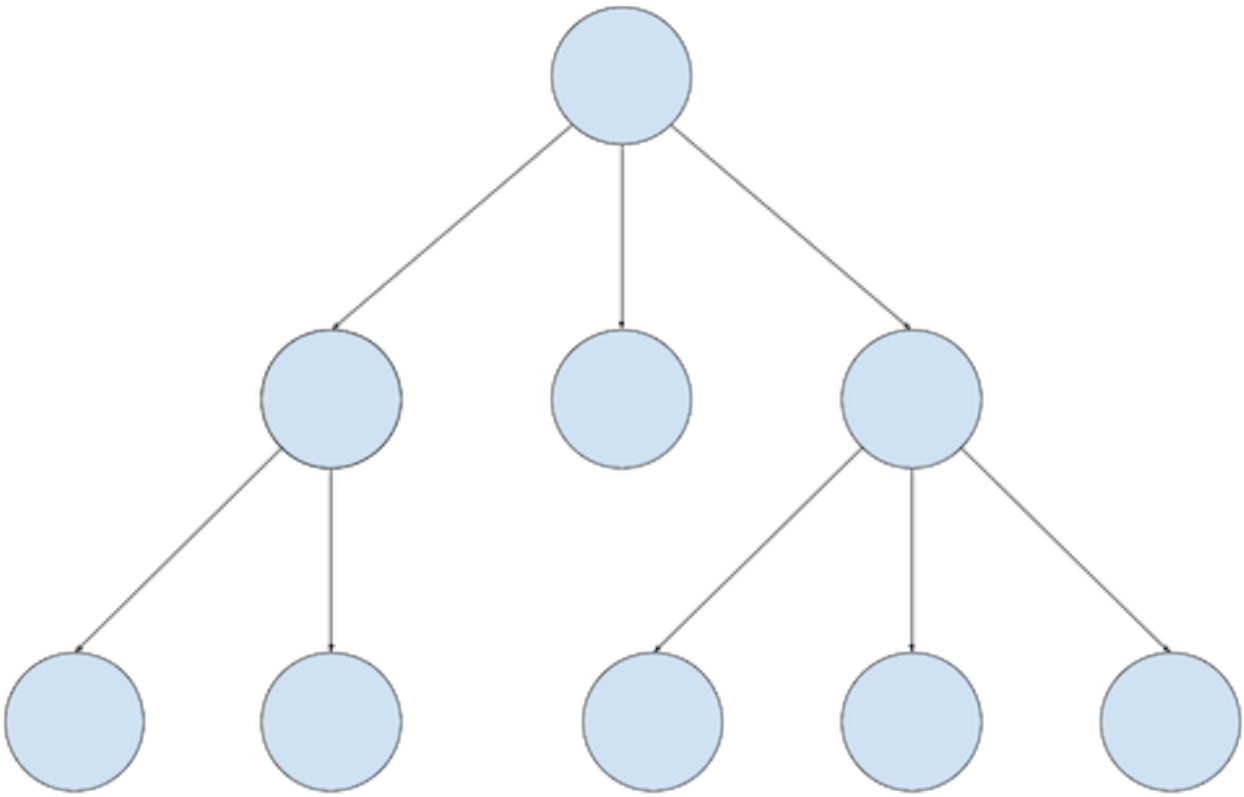

Figure 2.1: Example of a directed n-ary tree where $n = 3$

associated with it.

Another key factor is how parents are represented. Some trees are defined in a bottom-up manner where they only store parent pointers and no child pointers. For the scope of this work, we will be focusing on trees with a bounded number of children, the n-ary tree. For trees defined in the standard top-down approach, trees can be augmented to store parent pointers, which enables a node to follow an edge backward to its parent.

This thesis focuses on trees within the context of JavaScript. Given this, it's important to drill down into how trees can look concretely in JavaScript.

2.2 Different Tree Schemas

We define schema to be the way a tree is organized. Every node in the tree stores some information. Broadly, this can be broken up into two categories: metadata (terminal data), or child relations (non-terminal data). When we refer to varying schemas in this paper, we refer to the differences in metadata stored by nodes, and more importantly, how child relations are defined. In this section, we will focus on exploring varying schemas by looking into different ways child relations can be defined, exploring different types of metadata, and finally discussing how nodes are stored in JavaScript.

2.2.1 Children

In theory, children can be easily defined by simply drawing a set of edges from a parent to each of its children, regardless of how many children there are. In practice, an important consideration is how children are stored for a given node. There are 4[1] primary ways of defining children: one special property points to an array of child nodes; one special property points to an object that maps keys to child nodes; each child is referred to by its distinct

[1] A fifth could be storing children in a linked list data structure, which is used in trees like the Fibonacci Heap. However, we consider this case outside the scope of this work, as it's primarily used in specialized cases.

```
{
    name: "1",
    children: [{name: "2"}, {name: "3"}]
}
```

Figure 2.2: Special property points to an array of children

```
{
    name: "root",
    children: {
        "left": {name: "leaf"},
        "right": {name: "leaf"}
    }
}
```

Figure 2.3: Special property points to an object of children

child property; and finally, an unconstrained schema that uses a mix of any of the methods above.

The first case we will examine is the case where one special property points to an array of child nodes. What this means is a node defines a special keyword, for example, `children`, which points to an array of its child nodes. An example is shown in figure 2.2. Importantly, this special property is *arbitrary*. It could be called `children`, `childNodes`, or anything else.

The case where one special property points to an object of children is also fairly common in JavaScript. Similar to the example defined above, a special keyword property, for example, `children`, points to an object which maps keys to children. The benefit of this structure is that children can have names pointing to them instead of array indices. An example is shown in figure 2.3. To reiterate, this special property is also *arbitrary*. It could be called `children`, `childNodes`, or anything else.

There is also the case where special keyword properties point to nodes themselves. There could be multiple properties, for example, a binary tree node might have a `left` and `right` property, each pointing to the corresponding nodes. But perhaps a certain node type might only have a single child, in which case it could have a `child` property pointing to its only child. This would be classified as an incomplete node, where incomplete refers to the fact

```
{
    name: "root",
    left: {name: "leaf"},
    right: {name: "leaf"}
}
```

Figure 2.4: Special properties each point to children

```
{
    name: "root",
    children: [
        {
            name: "internal",
            left: {name: "leaf"},
            right: {name: "leaf"}
        },
        {name: "leaf"}
    ]
}
```

Figure 2.5: Any properties point to any type of children

that the node's degree is not maximal. An example is rendered in figure 2.4.

Finally, there is the case where the schema is unconstrained, and can use any combination of the techniques defined above. In other words, any properties in a node could point to a single node, an array of nodes, or an object of nodes. This is what Treecle assumes under its default configuration, which is further described in section 3.2.1. An example of this mixed schema type is shown in figure 2.5, which uses properties pointing to arrays of children, and properties pointing to a single child.

2.2.2 Metadata

The second main way a tree schema varies is in its metadata, which is data a node stores that is terminal, i.e. it doesn't include child relations. Metadata is simple, trees will want to store different information depending on the context they're in; for example, metadata is often used as the primary way of labeling and classifying nodes. Nodes might have a defined `type` property to identify which type of node they are relative to the kind of tree they're present in. A simple tree might label nodes with either `root`, `internal`, or `leaf`

types. A more complex tree, like the ASTs defined below in section 2.3, may choose a more informative type of node to help identify them, for example, `Identifier`.

Parent pointers are one of the most important pieces of metadata associated with a node. They are augmentations added to trees for two primary reasons: to support operations where walking up the tree is necessary, or to improve performance when walking up the tree is better than traversing down. Many tree operations rely on accessing ancestors of a given node, which makes this an important optimization to add to a tree structure.

2.2.3 Storage

Until now, we've been working under the assumption that all nodes are standard JavaScript objects. However, this is not always the case. There are many instances where developers choose to define their custom node class. In JavaScript, a given instance of a class can have its instance variables accessed the same way as JavaScript objects. That is, using the dot (`.`) operator allows retrieval of a specific property/variable.

As an example, suppose one node is an object and has the metadata property `name`. A second node is an instance of a custom node class and has an instance variable `name`. then, you could access both node's `name` attribute simply by computing `node.name`. For all intents and purposes of this project, we'd view an object node and an instance node containing the same properties/instance variables to be the same. In short, for this thesis, we define trees and their nodes as objects or class instances, but with the restriction that all of their properties must be able to be accessed with the dot (`.`) operator.

2.3 Abstract Syntax Trees

Abstract syntax trees (ASTs) are a specific example of the general tree described above in section 2.1. An AST is generally used as a data structure to store a representation of a program or piece of code [5]. In this project, we will specifically talk about ASTs for

```
// A CallExpression node representing the expression "foo(2 + 2)"
{
    type: "CallExpression",
    arguments: [
        {
            "type": "BinaryExpression",
            "operator": "+",
            "left": {
                "type": "Literal",
                "value": 2,
                "raw": "2"
            },
            "right": {
                "type": "Literal",
                "value": 2,
                "raw": "2"
            }
        }
    ],
    callee: {
        "type": "Identifier",
        "name": "foo"
    }
}
```

Figure 2.6: Example of a CallExpression node type

JavaScript, i.e., ASTs that have been formed from parsing JavaScript.

2.3.1 Node Types

As mentioned in section 2.2.2, ASTs leverage metadata to store useful information about each node, including its type. In this thesis, we will focus on ASTs that match the structure produced by any JavaScript parser. An example of such a parser is jsep [7]. As mentioned, each node in the output AST has a type. Some examples of these types include: `Literal`, `Identifier`, `ThisExpression`, `Compound`, `UnaryExpression`, `Binary Expression`, `ConditionalExpressi` `MemberExpression`, `ArrayExpression`, and `CallExpression`. Figure 2.6 demonstrates what a `CallExpression` node would look like for the expression `foo(2 + 2)`.

2.3.2 Children

As described above in section 2.2.1, there are four ways to store children: in a single array, in a single object, in multiple designated properties, or in a mix. Our definition of an AST uses both the array method, and the designated properties method. For example, a `BinaryExpression` node stores its children in two designated properties: `left` and `right`. Each of those properties points to the left and right child respectively. This is in contrast to a type like `CallExpression`, which uses a single property to store an array of children. A `CallExpression` node will store a property called `arguments`, which is an array of all child nodes representing the arguments to that function. See figure 2.6 for an example of both.

2.4 Existing Libraries

An important step in the motivation and justification for this thesis is an investigation into the existing libraries that attempt to solve the same or similar goals that Treecle and Vastly set out to solve. This section will dive into some relevant libraries for both Treecle and Vastly, and explain why although the libraries possess value, none of them can completely and effectively achieve the goals that Treecle and Vastly set out to solve.

2.4.1 Libraries For General Trees

The main goal of Treecle, as defined in section 1.1, is to provide a library for developers that is flexible, powerful, and easy to use. We will be investigating three tree libraries that are each available for distribution like Treecle: installation via NPM. The three packages are: Tree.js [19], Treeize [13], and TreeModel [11].

Tree.js

The first library, Tree.js [19], is described as a "JavaScript library for creating and manipulating hierarchical tree structures." Tree.js provides a node class and a set of operations as instance methods of that node class. Some of the operations provided include `append()`, `insert()`, `remove()`, `find()`, and `size()`. Tree.js gives a good amount of flexibility and power with its simple set of operations, yet lacks in one key area: developers are forced to use this specific node class built by Tree.js. If a developer already designed their implementation of a tree structure, they would either be forced to fit their design as a subclass of Tree.js's node class or be out of luck.

Treeize

The second library, Treeize [13], is described as a library that "converts row data (in JSON/associative array format or flat array format) to object/tree structure based on simple column naming conventions." Treeize's goal is to enable easy creation of trees, but not much more. The library has a primary operation called `grow()`, which takes in an arbitrary array or object that is tree-like in structure and attempts to turn it into a proper tree, all masked behind a custom tree class. The library is useful because it's very flexible: it can take in a wide array of different data types to create trees, and it's also easy to use. However, what this library lacks is power. Once the tree is created, the library doesn't implement any tree traversal or mutation functionality, which again, renders the user out of luck if they wish to do anything with the tree once it's created.

TreeModel

The third library, TreeModel [11], is described as a library that enables the user to "manipulate and traverse tree-like structures in JavaScript." Perhaps the most robust library of the three, TreeModel implements its custom tree class and supports the creation of empty trees or by passing in a JSON object literal. The library also supports some common tree

operations including `addChild()`, `walk()`, `getPath()`, and `find()`. This makes the library well-rounded but lacks a subtle idea which we intend to solve with Treecle: the library is very strict about tree structure, and only supports nodes having an array of children, as described in figure 2.2. So while this library is high-quality and checks most of the boxes, it's lacking in terms of flexibility.

2.4.2 Libraries For ASTs

Vastly shares the same tenets and major goals as Treecle, to be flexible, powerful, and easy to use. The following three libraries will be investigated to justify the existence of Vastly, since none of the three meet all three goals Vastly sets out to achieve. The three libraries are Acorn [1], AST Types [2], and Estraverse [9].

Acorn

The first library, Acorn [1], is built to parse JavaScript into ASTs. Its purpose is to be a smaller, lightweight alternative to giant libraries like Esprima [8] and others. Acorn's primary job is to handle the complex and intricate logic of being a full-blown JavaScript parser. A subsection is devoted to walking the ASTs but doesn't support any other explicit operations or transformations. While `walk()` is one of the most commonly used and powerful operations for an AST, or any tree, this library is partially lacking features that could be commonly needed by developers. JS parsers like Acorn sometimes provide conveniences for handling ASTs, but oftentimes these operations and functions fall short of a developer's needs, both in terms of power and flexibility.

AST Types

The second library, AST Types [2], provides utilities for initializing ASTs in the same format as Esprima and subsequently provides operations for traversing and manipulating those ASTs. AST Types doesn't provide a parser to parse JavaScript into an AST, but it contains

a builder functionality that allows users to define ASTs by creating building block node instances. Once a user has an instance of this AST class, AST Types has several available functions for these trees, including `visit()` and `eachField()`. Similar to Acorn, this library focuses heavily on traversal and not much else. Developers, like in Acorn, can use these traversal operations to do a wide array of tasks, but the downside is that it makes this library somewhat challenging to use. One example, which is listed in the documentation for this library, is how a developer could make a copy of a tree by creating a new object, calling `eachField()`, and passing in a callback to the function that copies each key/value pair from the existing tree into a new one. This is opposed to a library which might provide a `clone()` or `map()` function to make simple tasks like this far easier for a developer.

Estraverse

The third library, Estaverse [9], is designed for ASTs conforming to the ECMAScript JS syntax and provides traversal functionality for said trees. This library does not contain any parser and depends on the user already having a constructed AST before using it. Like the previous two libraries, this one focuses heavily on the traverse operation, and only exposes 3 operations for developers to import and use: `traverse()`, `replace()`, and `remove()`. These three operations make this library easy to use, but hard for developers to do everything they could want to do with ASTs.

Chapter 3

Treecle

3.1 Motivation and Purpose

The idea to create Treecle arose from our inability to find a library flexible and powerful enough to perform operations on trees that we needed for our other work. Prior research discussed in section 2.4 indicated that existing libraries don't contain enough operations and functionality or don't support commonly used tree schemas defined in section 2.2. Thus, we decided to create Treecle: a utility library for handling tree-like structures that is general enough to handle all common schemas and maintain a level of power by supporting many useful operations.

At a high level, the functions defined in this library can be classified as one of the following three groups: traversal, manipulation, or augmentation. Traversing refers to visiting a subset, or all, of the nodes in the tree and computing some useful information. Manipulation implies performing some arbitrary operation on nodes of the tree that mutates the tree in some way. Finally, augmentation refers to the intentional process of adding metadata to nodes to improve the ease and efficiency of various other operations. Section 3.2 will discuss the functions and operations Treecle provides, and section 3.3 will discuss use cases where Treecle can be most aptly used and applied, which justifies its existence.

3.2 Functions and Operations

3.2.1 Setup

Before discussing Treecle's functions, we must discuss the configuration options necessary to ensure it can support all types of schemas for defining children. Treecle can be used in two ways: by using the default configuration or by providing a custom configuration. By default, a user could import any of Treecle's functions directly into their code, and call them on their tree structures. However, we also provided a way to create custom configuration options by creating an instance of the `Treecle` class and passing in the desired configuration options. This allows the user to define their child relations arbitrarily, which can be useful in cases where the default configuration does not suffice. The configuration options that can be set are: `getChildProperties(node: Node): Array` and `isNode(thing: any): boolean`. The former is a function that takes a node and returns an array of property names that point to children. The latter is a function that takes an object and returns whether or not it is a node. Under the default configuration, all key-value pairs in a node such that the value is an array or object are considered children, and all objects are considered nodes. This is an example of the unconstrained schema type; an example can be found in figure 2.5. However, this can be changed by providing custom configuration options. For example, the configuration shown in 3.1 defines a tree schema where children are stored in a property called `children` and nodes are objects with a property called `type` that can be either `root`, `internal`, or `leaf`. After a user sets up this configuration, they can call Treecle functions and pass in this context as the first argument, which lets Treecle know how to traverse any tree matching the given schema.

```
const treecle = new Treecle({
    getChildProperties: (node) => {
        return ["children"];
    },
    isNode: (obj) => {
        return ["root", "internal", "leaf"].includes(obj?.type);
    }
});
// const rootNode = ...
// Uses the custom configuration to find the first leaf
const firstLeaf = treecle.find(rootNode, (node) => node.type === "leaf");
```

Figure 3.1: Example of a custom configuration

3.2.2 Traversal Functions

The following section details all traversal functions, which visit nodes in the tree but do not perform any mutation on the tree itself. These functions are useful for computing some information about the tree or its nodes and are often used as a precursor to manipulation functions.

The children() Function

The `children()` function is returns all children of a node. Internally, this is implemented by calling `childPaths()` and then getting the children at each path. This is an essential function as it provides a means of traversing the tree by finding descendants of a given node. As a result, it is an important function both internally, and for external users.

The childPaths() Function

The `childPaths()` function is similar to the `children()` function, with the exception that it also returns additional metadata on what path was taken to get from the given node to each of its children. With some schemas, like for that of a binary tree shown in figure 2.4, the paths to the children are trivially length 1, while for others, like for that of a tree with children stored in an object shown in figure 2.3, the paths to the children are more complex.

The closest() Function

The `closest()` function is a utility function that returns the closest ancestor of a node that satisfies a given predicate, or none if no node matches the conditions. Importantly, since this function needs to be able to walk up the tree, this function relies on having parent pointers set, which will be discussed in further detail in section 3.2.4.

The find() Function

`find()` is a function that takes a node and a predicate as a parameter and traverses down the tree from that node and terminates and returns once it finds a node satisfying the given conditions or explores the entire subtree rooted at that node without finding any matches. This function is important for implementing searching functionality.

The map() Function

The `map()` function is a way to create a new tree based on an existing one. The function takes a tree and a mapping function as parameters and returns a new tree where each node is the result of applying the mapping function to the corresponding node in the original tree, in a depth-first manner. For example, if we have a tree where each node has a `value` property which is a number, and we want to create a new tree where each node is the square of the corresponding node in the original tree, we could use the `map()` function to do this.

The reduce() Function

With the `reduce()` function, a user can reduce a tree to a single value by applying a function to each node in the tree. This function is very similar to the standard `reduce()` function in JavaScript, but instead of a simple array, it applies the reducing operation by walking the tree in a depth-first (pre-order) manner.

The walk() Function

Finally, we discuss perhaps the most important operation in Treecle, the `walk()` function. This function is a general-purpose function that can be used to traverse a tree in a depth-first manner, where it takes in a tree and a callback function as parameters, visits each node in the tree, and calls the callback function on each node until the callback function produces a terminating value. This function is incredibly versatile and can be used as a foundation to implement many other tree-related operations and functions.

3.2.3 Manipulation Functions

This section defines and explains the manipulation functions in Treecle, which are defined as functions that apply some mutation or augmentation to the tree in some way.

The replace() Function

The `replace()` function takes in a node and a replacement node, and replaces the subtree rooted at the given node with the subtree rooted at the replacement node.

The transform() Function

The `transform()` function works analogously to the `map()` function; instead of creating and returning a new tree with the given mapping function, it mutates the input tree with the given transforming function. Like `map()`, this function works in a depth-first manner. This function is useful when a user wants to apply some transformation to a tree in place, rather than creating a new tree.

3.2.4 Augmentation Functions

This section defines and explains the augmentation functions in Treecle, which are defined as functions that add some metadata to the nodes in the tree to improve the ease of other

operations.

The Parents Module

Perhaps the most critical and useful augmentation for any tree is the ability to have parent pointers. It was important to us to abstract away all of the complications of handling and maintaining parent pointers, especially concerning storing them inside nodes. We implement parent pointers in Treecle using JavaScript's `WeakMap` data structure [24], which is a way to map arbitrary objects to another object. We use this data structure instead of an ordinary `Map` because it uses weak references, meaning it does not affect garbage collection. In this case, we have a parent map which maps nodes to an object containing their parent and the path that leads from the parent to that node. `parents` is a module that encapsulates this functionality and has three functions: `setPath()`, `getPath()`, and `getParent()`. `setPath()` takes in a node and a path and sets the key-value pair in the parent map. `getPath()` takes in a node and returns the parent along with the path from the parent to that node. Finally, `getParent()` takes in a node and returns the parent of that node. These functions are useful for other operations that require parent pointers, such as `closest()`.

The clearParents() Function

The `clearParents()` function is a utility function that clears all parent pointers from a tree. This function is useful both internally and externally, as it could be used before transforming a tree to ensure that the parent pointers are correct for the transformed tree.

The updateParents() Function

The `updateParents()` function will either set if the tree has no parent pointers or update all parent pointers in the tree. Often, this function is used after a tree has been transformed, to ensure that the parent pointers are correct for the transformed tree.

3.3 Use Cases

The following section details potential use cases for Treecle. We define a use case as a project or space that can use Treecle to simplify its logic in some way, typically in terms of shortening its code footprint. Research was conducted to retrieve a large and diverse set of use cases for Treecle. The goal was to gather a list of projects that varied in a handful of metrics including complexity, size, and popularity. Most of the research was spent trying to find larger JavaScript frameworks that would likely implement their tree function in-house, allowing Treecle to have the biggest impact by reducing lines of code. However, we also wanted to find smaller-scale and less-used GitHub repositories that could use Treecle to full effect. By showing how Treecle could be applied in a diversity of settings, we justify Treecle's existence and purpose as a helpful utility library for handling trees. The use cases, which are discussed in detail below, are ordered in ascending order of project size, in terms of lines of code[1].

3.3.1 Vastly

As mentioned briefly in section 1.2, Vastly is both a novel contribution to this thesis as well as a use case that validates Treecle. After developing Vastly originally, we ported all of the logic about common tree data structures into Treecle, meaning that many of Vastly's functions *already use* various Treecle functions. That means this section is not hypothetical since Treecle is a dependency of Vastly.

As described later in section 4.1, many of Vastly's operations are trivial wrappers around some of Treecle's functions. They are fully enumerated in section 4.1, but to reiterate a few, Vastly's `children()`, `closest()`, `find()`, `walk()`, and more are all just simple wrappers

[1]Throughout this project, any reference made to lines of code is an approximate statistic that is calculated on GitHub repositories by cloning and executing the shell command `git ls-files SRC_FOLDER | grep ".js$" | xargs wc -l`, which is used to count the number of lines in all JavaScript files in the source folder of the project.

```
import "./treecle-setup.js";
import treecleClosest from "../lib/treecle/src/closest.js";

export default function closest (node, type) {
    return treecleClosest(node, n => n.type === type);
}
```

Figure 3.2: Vastly implements `closest` using Treecle's `closest`

around Treecle's functions. Figure 3.2 is Vastly's current implementation of the `closest()` function, which uses Treecle's `closest()` function to make it more specific for ASTs.

3.3.2 LeetCode Problems

LeetCode is a popular platform for practicing coding problems, particularly in the context of technical interviews. The platform has a wide array of exercises, some of which involve trees, in particular, problems revolving around binary trees. LeetCode problems are typically small and bite-sized, so we found that many of them could be solved with ease using Treecle.

We identified a subset of all tree-related questions on LeetCode that could adequately demonstrate the utility of Treecle. These problems include the questions *N-ary Tree Preorder Traversal*, *Evaluate Boolean Binary Tree*, *Search in a Binary Search Tree*, *Maximum Depth of Binary Tree*, *Univalued Binary Tree*, *Leaf Similar Trees*, *Sum of Left Leaves*, and more [21]. These problems are all relatively simple and can be solved with a few lines of code using Treecle's functions. Looking into LeetCode problems was useful in helping to inform which functions in Treecle are more important and generalizable than others. For example, most of these selected problems can be solved using Treecle's `walk()` or `reduce()`. An example is shown in figure 3.3 for how Treecle could be used to solve *N-ary Tree Preorder Traversal*.

3.3.3 Cheerio

Cheerio is a library for parsing, manipulating, and traversing HTML and XML documents using a jQuery-like API. Cheerio is a popular library with nearly 28,000 stars, 2,000 forks,

```
// A potential implementation of LeetCode's N-ary Tree Preorder Traversal
   problem using Treecle's reduce
// Assumes treecle instance has been properly configured with this tree's
   custom settings
/* specs */
function preorder(root) {
    return treecle.reduce(root, (acc, node) => {
        acc.push(node.val);
        return acc;
    }, []);
}
```

Figure 3.3: Using Treecle's `reduce` in LeetCode's N-ary Tree Preorder Traversal Problem

```
// A potential implementation of Cheerio's contains function using Treecle
   's closest
// Assumes treecle instance has been properly configured with this tree's
   custom settings
/* specs */
export function contains(container, contained): boolean {
    if (container === contained) {
        return false;
    }
    return Boolean(treecle.closest(contained, (node) => {
        return node === container;
    }));
}
```

Figure 3.4: Using Treecle's `closest` in Cheerio

and approximately 10,000 lines of code [4]. HTML documents are tree-like, meaning Cheerio parses them into a tree data structure to perform all operations in its API.

Given its use of a DOM-like tree structure, Cheerio is a prime example of a library that could benefit from using Treecle. Cheerio implements the following tree-related functions: `contains()`, `find()`, and `replaceWith()`. The `contains()` function checks whether a given node is a descendant of another node, `find()` searches for nodes that match a given predicate, and `replaceWith()` replaces all nodes matching a predicate with another node. These functions could be implemented using Treecle's `closest()`, `find()`, and `replace()` functions respectively. An example of how Treecle's `closest()` could be used to implement Cheerio's `contains()` is shown in figure 3.4.

```
// A potential implementation of one of Vue.js's traversal functions using
    Treecle's walk
// Assumes treecle instance has been properly configured with this tree's
    custom settings
/* specs */
function walk(vnode, children) {
    treecle.walk(vnode, (node) => {
        if (node.component) {
            children.push(node.component.proxy)
        }
    });
}
```

Figure 3.5: Using Treecle's `walk` in Vue.js Core

3.3.4 Vue.js Core

Vue.js is a popular JavaScript framework for building reactive web applications in a modular, component-based manner. Their `core` repository is where much of their behind-the-scenes logic and computations are implemented, which includes a wide array of things such as their reactivity engine, compiler, and runtime system. The `core` repository is a large and complex codebase, with nearly 45,000 stars, 8,000 forks, and approximately 120,000 lines of code [23]. The repository contains multiple types of trees used in distinct locations, with the prime examples being in the compiler and runtime system modules.

The primary way trees are used is a Virtual DOM tree, which is a tree representation of what the DOM should look like with the applied Vue components. Vue is such a large and complex framework that it's hard to extract all locations where Treecle could come be used. We found that this package implements a custom tree traversal function on more than three separate occasions. These traversals' implementations could be replaced with Treecle's `walk()` function to reduce the chance of bugs in the current implementation which contains more than three variants of a similar function. An example of how this might be implemented is shown in figure 3.5.

3.3.5 Summary of Use Cases

Each of the above use cases provides evidence that Treecle is a useful utility library that can be used to improve existing codebases by simplifying all tree-related operations. The benefits of using Treecle over implementing a custom solution are numerous, including reducing the chances of bugs, reducing the amount of work and code that must be written, and increasing the readability and maintainability of the code. In addition, the diversity of each of these use cases demonstrates Treecle's ability to perform well regardless of the size of the project or the complexity of the tree operations that need to be performed. Our research uncovered many other examples of potential use cases for Treecle, but due to scope, we can't cover all of them in full detail like the ones listed above. Instead, table 3.1 is provided that gives a short overview of the remaining use cases. In short, these use cases demonstrate Treecle's versatility and ability to be used in many projects.

Table 3.1: Overview of additional Treecle use cases

Name	**Stars**	**Lines of Code**	**Number of Use Cases**	**URL**
Mavo	2.8k	12k	9	https://github.com/mavoweb/mavo
hTest	19	2k	3	https://github.com/leaverou/htest/
estraverse	921	1k	2	https://github.com/estools/estraverse

Continued on next page

Table 3.1 – continued from previous page

Name	Stars	Lines of Code	Number of Use Cases	URL
eslint	24.2k	82k	2	https://github.com/eslint/eslint
pdf.js	46.2k	110k	2	https://github.com/mozilla/pdf.js
virtual-dom	11.6k	2k	1	https://github.com/Matt-Esch/virtual-dom
formatjs	14.1k	355k	1	https://github.com/formatjs/formatjs
tailwindcss	78.2k	29k	1	https://github.com/tailwindlabs/tailwindcss
uikit	18.1k	13k	3	https://github.com/uikit/uikit
vscode	158k	1.2m	3	https://github.com/microsoft/vscode

Continued on next page

Table 3.1 – continued from previous page

Name	Stars	Lines of Code	Number of Use Cases	URL
treejs	52	600	1	https://github.com/m-thalmann/treejs
obj-traverse	102	500	2	https://github.com/brojd/obj-traverse
dom-handler	320	900	1	https://github.com/fb55/domhandler
scour	305	1k	2	https://github.com/rstacruz/scour
react-tree-walker	305	400	2	https://github.com/ctrlplusb/react-tree-walker

Chapter 4

Vastly

4.1 Motivation and Purpose

As mentioned briefly in section 1.1, the original motivation to create Vastly stemmed from the desire to simplify the codebase of a web framework called Mavo [22]. Mavo is a declarative web framework that allows users to build reactive web apps simply by annotating HTML elements with custom expressions to populate them with custom data. As a result of these expressions, Mavo's source code implements a variety of AST operations from scratch, which clog the codebase with relevant yet tangential code that could be abstracted away. Originally, the plan was to modularize this functionality to simplify Mavo, but we then realized that these operations are common for ASTs, and could prove useful to other developers. Vastly was created as a public, open-source library to abstract away these AST operations and provide a simple, easy-to-use API. Now, most of the operations we created in Vastly have been abstracted into Treecle, meaning Vastly is both a novel library and a use case for Treecle.

Recall from section 2.3 that ASTs are a specific type of tree that are typically used to represent a piece of logic in an expression. For the scope of this thesis, we will be only considering ASTs with the structure defined precisely in section 2.3.1, where each node has

a predefined node type. Throughout this section, when we refer to an AST, we will be assuming an AST that conforms to that structure.

Given that Vastly now uses Treecle and many of its functions, we will discuss only the ones novel to Vastly in this section. The list of functions that are implemented in Vastly as a trivial wrapper around a Treecle function is as follows: `childPaths()`, `children()`, `clearParents()`, `closest()`, `map()`, `parents()`, `replace()`, `transform()`, `updateParents()`, and `walk()`. Similar to Treecle, each of Vastly's novel functions can be classified into three categories, which differ slightly: creation, traversal, and manipulation. Creation refers to functions that produce or build an AST, traversal refers to functions that visit nodes in the AST but do not perform any mutation, and manipulation refers to functions that apply some mutation and or augmentation to the AST in some way.

4.2 Functions and Operations

4.2.1 Creation Functions

The parse() Function

Importantly, Vastly does not implement its own JavaScript expression parser from scratch. The parse function provides a wrapper around a user-chosen parser, or if none is provided, uses the jsep parser [7]. The parse function takes an expression as a string and returns an AST representing that expression. An example AST is shown in figure 4.1 for the expression $2 + 5$. The parse function gives users flexibility in choosing their parser or omitting its use entirely by using the default parser.

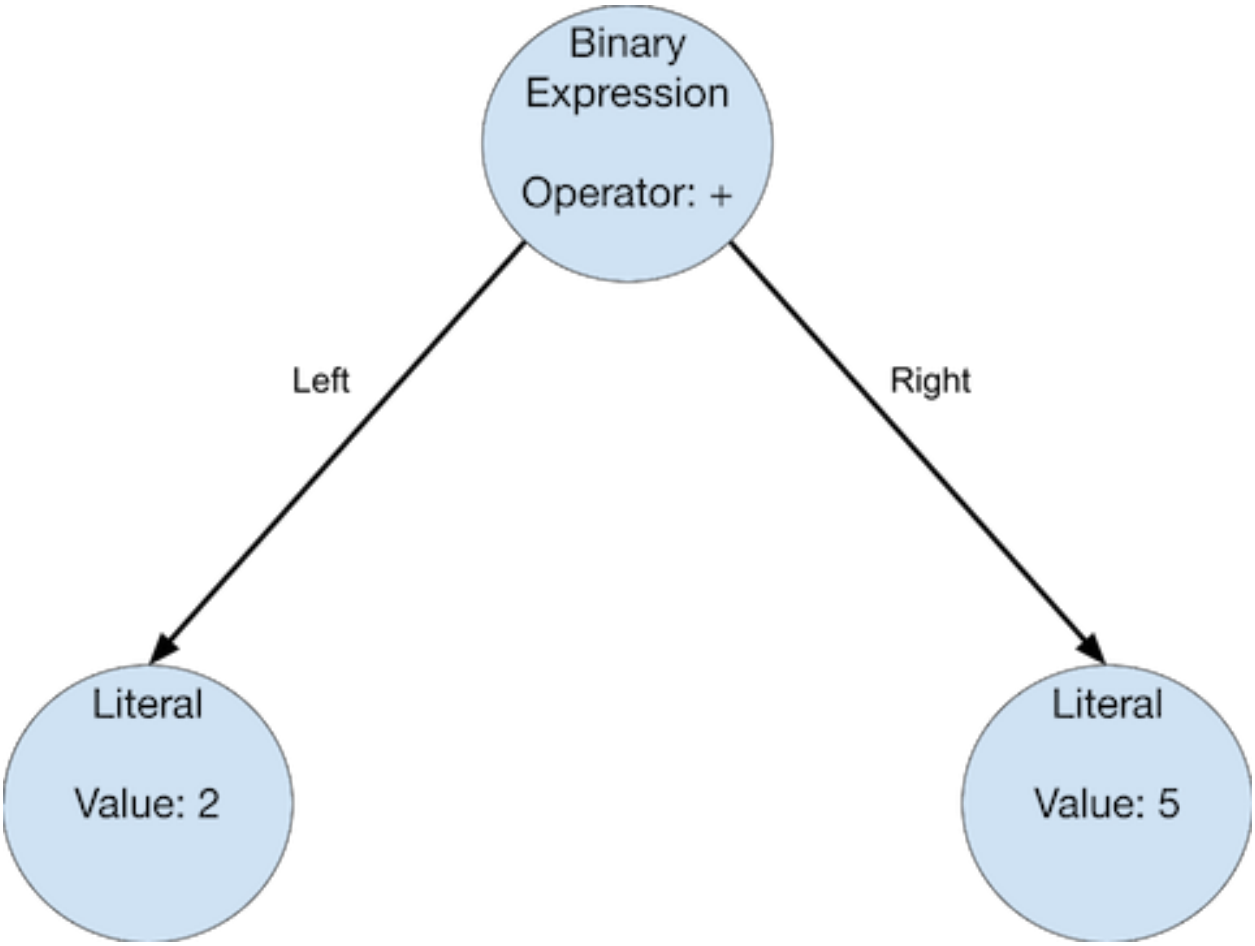

Figure 4.1: Example of a parsed AST for the formula $2 + 5$

4.2.2 Traversal Functions

The evaluate() function

The `evaluate()` function is a function that takes in an AST and a list of contexts in ascending order of precedence and evaluates the AST in terms of those contexts. A context is a mapping of identifiers to values, and the function recursively traverses the AST and continuously computes the value of subexpressions until it reaches leaf nodes, such as identifiers, when it looks up the value of the identifier in the context. In addition, the way each node type is evaluated can be customized.

The serialize() Function

`serialize()` is simply a function that takes in an AST and converts it back into a string expression. Besides it is nice to have a function to represent a complex data structure as a string, making it easier to read a human, one case in particular where this function could prove useful is getting an updated expression value after the AST is mutated in some way.

The variables() Function

The `variables()` function is a function that returns a list of all top-level identifier nodes in the AST. This function was created within Mavo as part of the refactoring of the expression evaluation pipeline but would prove useful in other contexts. Importantly, the function doesn't simply return all identifiers, but only the top-level ones, which include all simple identifiers, e.g. `x`; the root of member expressions, e.g. `a` for the expression `a.b`; and function names as well, e.g. `foo` for `foo(1)`. Mavo will use this function with the `prepend()` function to prefix all top-level identifiers with `\$data`.

4.2.3 Manipulation Functions

The prepend() Function

The `prepend()` function is perhaps the most specific and complex in the codebase besides `evaluate()`. This function is titled `prepend()` because it takes in as input two ASTs, and returns a new AST where the second AST is prepended to the first AST. For example, if we had one AST, `ast1`, representing the expression `bar`, and another, `ast2`, representing the expression `foo`, then the output of `prepend(ast1, ast2)` would yield a brand new AST corresponding to the expression `foo[bar]`. This function was built with Mavo in mind, where to refactor the expression evaluation pipeline, we needed to be able to prepend the data root `$data` to all identifiers, e.g. `x` to `$data.x`.

4.3 Use Cases

The following section details potential use cases for Vastly, or in other words, projects and spaces where Vastly could be used to simplify a piece of code in some meaningful way. The goal was to find use cases that vary in metrics like complexity, size, and popularity. For example, we tried to find popular JavaScript frameworks that used ASTs, and other smaller-scale and less-used GitHub repositories that could use Vastly to full effect. By showing how Vastly could be applied in a diversity of settings, we justify Vastly's existence and purpose as a helpful utility library for handling ASTs. The projects will be discussed in ascending order of project size, in terms of lines of code in the codebase.

4.3.1 simple-eval

`simple-eval` [20] is a package for evaluating simple JavaScript expressions. The library utilizes ASTs in the expression evaluation pipeline by converting expressions to an AST, and then evaluating that AST to produce a value. This package already has a relatively large

```
// A potential implementation of simple-eval's reduce function using
   Vastly's evaluate
/* specs */
export default function reduce(node, ctx) {
    return Vastly.evaluate(node, ctx);
}
```

Figure 4.2: Using Vastly's `evaluate` in simple-eval

amount of users, accumulating almost 500,000 weekly downloads on NPM. The package is relatively small, containing approximately 1,000 lines of code [18].

The package defines two functions that could be perfectly adapted to using Vastly. First, this library defines a function called `parse()`, which takes in a string expression and returns an AST. This function could be replaced with Vastly's `parse()` function. Second, the library defines a function called `reduce()`, which takes as input an AST conforming to the same structure used in this paper, and evaluates the AST given a context. This function's behavior is identical to that of `evaluate()` which Vastly implements. By swapping these two functions with Vastly's, this library reduces its size by an estimated 100 lines of code, which would improve the maintainability and readability of the codebase. An example is pictured in figure 4.2 of how `reduce()` might be implemented to use Vastly's `evaluate()` function.

4.3.2 Mavo

As mentioned briefly in 4.1, Mavo is a new web framework that allows users to build reactive web apps simply by annotating HTML elements with custom expressions to populate them with custom data. Mavo represents a medium-sized use case; it has 2,800 stars and 181 forks on GitHub, and its source code is on the order of 10,000 lines [15]. We'll take look at relevant areas within Mavo where any of Vastly's functions defined in section 4.2 could be applied.

Within Mavo's source code is file called `Mavoscript.js` [15]. This module contains various functions related to the expression language that Mavo uses. This file is a perfect candidate for Vastly, as it includes many AST operations that could be abstracted away

```
// A potential implementation of Mavo's walk function using Vastly's walk
/* specs */
walk: function(node, callback, o = {}) {
  return Vastly.walk(node, callback, {only: o.type, except: o.ignore});
}
```

Figure 4.3: Using Vastly's `walk` in Mavo

into Vastly's API. For example, the module contains functions called `walk()`, `serialize()`, `parse()`, and `closest()`, all of which are functions within Vastly. Because of this, each of these functions could be replaced with Vastly's, which would save an estimated 150 lines of code in the module. This would significantly cut down the complexity of the module, which would in turn increase its maintainability and readiness for change. An example is pictured in figure 4.3 of how Mavo's `walk()` might be implemented to use Vastly's `walk()` function.

4.3.3 CesiumJS

CesiumJS [3] is a visualization library for building 3D globes and 2D maps in-browser. It is another large-scale project with nearly 12,000 stars, 4,000 forks, 40,000 commits, and a codebase size of approximately 680,000 lines of code [3]. To aid in creating these visual elements and scenes, CesiumJS implements its own expression language to define attributes in scenes. It can be used to define and perform calculations on data like temperature, color, and other metadata about the scene [10]. As a result, CesiumJS needs to handle the ASTs that come from parsing these custom expressions, which makes it a reasonable candidate to use Vastly given its already monumental size.

Within CesiumJS's expression module are some functions that operate on ASTs including `getVariables()` and the `Expression` constructor. For simplicity, these are the two that will be discussed, as the entire module is thousands of lines long and has other functions that could be non-trivially adapted to use Vastly. The `getVariables()` function returns all variables used in the AST, meaning it could be replaced with Vastly's `variables()` function. The `Expression` constructor takes in a string expression and parses it into an

```
// A potential implementation of CesiumJS's Expression datatype's
    getVariables method using Vastly's variables
/* specs */
Expression.prototype.getVariables = function () {
  return Vastly.variables(this._runtimeAst);
};
```

Figure 4.4: Using Vastly's `variables` in CesiumJS

AST, meaning it could use Vastly's `parse()` function. An example is pictured in figure 4.4 of how CesiumJS's `getVariables()` might be implemented to use Vastly's `variables()` function.

4.3.4 Summary of Use Cases

All of the aforementioned use cases are examples of how Vastly could be used to simplify and improve the maintainability of a codebase. By replacing the AST operations in these projects with Vastly's, the codebase would be simplified, and the maintainability and readability of the code would be improved. These three projects also show Vastly's ability to operate at different scales, being able to fit in with a project like simple-eval with about 1,000 lines of code, Mavo with more than 10,000 lines of code, and CesiumJS with more than 100,000 lines of code. Importantly, this is a small set of use cases that was identified. In short, these use cases demonstrate Vastly's versatility and ability to be used in many projects. Additional use cases that weren't elaborated upon can be found in table 4.1.

Table 4.1: Overview of additional Vastly use cases

Name	Stars	Lines of Code	Number of Use Cases	URL
expression-to-mql	2	600	2	https://github.com/mongodb-js/expression-to-mql

Continued on next page

Table 4.1 – continued from previous page

Name	Stars	Lines of Code	Number of Use Cases	URL
Vue.js Core	44.5k	120k	3	https://github.com/vuejs/core
estraverse	921	1k	2	https://github.com/estools/estraverse

Chapter 5

Developer Interviews

5.1 Purpose of Interviews

The subsequent sections contain the findings of two interviews conducted with software engineers. The purpose was to gather more information on the potential use cases for Treecle, and more generally the problems real developers have revolving around trees. The interviews were semi-structured [14], and the questions were designed to gather information on the interviewee's background, their past and current use cases with trees, and what they look for and don't look for in a library. The interviews were conducted over Zoom, and consent was given from both interviewees to have their responses and information included in this thesis. They have both had their names anonymized to Interviewee 1 and Interviewee 2 out of respect for privacy. The two candidates were chosen given their experience in the software industry combined with the fact that they are both currently working on tree-related projects. The insights and takeaways from these interviews are discussed in the following sections.

5.2 Interview 1

5.2.1 Introduction of Interviewee

Interviewee 1 is an experienced software engineer and developer. They have been in the software industry for roughly 15-20 years, and have had experience working at a range of companies, including being a principal architect for Salesforce. Currently, they are working on a startup where they are a co-founder and CEO. The company's platform is a tool for handling building data, such as floors and rooms, and the hierarchical structure that the data takes made them a relevant candidate to interview given that they have a use case in solving problems with tree-like data.

5.2.2 Summary of Findings

The goal of the interview was to dig into Interviewee 1's past and current use cases with trees, and what sorts of problems they're trying to solve. In addition, we tried to get a sense of what types of tree-related functions they had to build in-house, particularly, if they had any problems or issues.

Starting with their past use cases, in a previous role, Interviewee 1 mentioned that their team had to build a large-scale UI for a website completely from the ground up. Over the years, they dealt with many tree-related problems, mostly involving walkers and traversals for the DOM or Virtual DOM. After asking a follow-up question about why they chose to repeatedly implement everything in-house, and multiple times at that, they responded with insight into adopting libraries. They noted issues with previous experiences with libraries mainly included lack of documentation, lack of examples or a playground, and difficulty to start using. They said that 90% of libraries suffer from these issues, and also mentioned that documentation is the cornerstone to building a library that is easily adoptable.

Moving onto their current use case, Interviewee 1, as mentioned above, is the founder

and CEO of a startup working with building data. After asking how their building data is structured, they replied that the root of their data is a portfolio, where a portfolio can contain any number of buildings as children in addition to some other metadata. Moving down a level, each building can have any number of floors as children, as well as other metadata, for example, the name of the building. Each floor is comprised of any number of rooms as children, and similarly has associated metadata. This hierarchical structure keeps going a couple more levels, with other nodes including sensors and other physical objects as leaf nodes.

With this building data, Interviewee 1 described the various end goals of working with this data. The main one being a way to render a 2D or 3D view of the building. The process of rendering the building involved traversing the tree structure and using it to recursively build an SVG drawing. Because of this, traversal operations are fundamental to their system.

5.2.3 Insights and Takeaways

Interviewee 1 presented a use case that validates the general design of Treecle. The way the building data is structured means there exist around 10 distinct node types, where each node type is either a leaf node or stores its children in an array in its `children` property, just like described in section 2.2. This means the structure can be described using Treecle's configuration options with ease.

One important point brought up during the interview was efficiency and performance. I asked what the approximate node count would be in typical buildings in a typical dataset, and they replied that there wouldn't usually be more than 10,000. They mentioned that they hadn't needed to make significant performance optimizations yet, but added that it is an important consideration to keep in mind when developing a library like Treecle. We discussed that typically performance is a tradeoff with the simplicity of a piece of code and that it's usually better to focus on performance last. Nevertheless, the tradeoff is an important consideration for future work.

Overall, this interview was incredibly useful to our research. The use case described by Interviewee 1 reinforces how trees are defined and structured in JavaScript and supports Treecle's goal and ability to support varying schemas, including this one. In addition, the emphasis Interviewee 1 put on documentation and the performance versus simplicity tradeoff helped to influence the plans for future work for this project, which is discussed in detail in section 6.2.2.

5.3 Interview 2

5.3.1 Introduction of Interviewee

Interviewee 2 is a similarly experienced software engineer and developer to Interviewee 2, having nearly 15 years in the field. Their background, however, is unique given that they were originally an artist and designer, and turned into a self-taught software engineer after getting into web programming as a hobby. Interviewee 2's unique background allows them to give a great perspective on design, as their work with companies typically focuses on front-end development and emphasizes the end-user experience side of things. Interviewee 2 was deemed an appropriate candidate to interview for this project given their current personal project developing a visual editor for building websites.

5.3.2 Summary of Findings

As mentioned, Interviewee 2's primary use case involves a project they are currently developing. They described this project as a "cross between Vim [26] and Webflow [25]." It's supposed to be a visual editor for creating websites but with an extensive set of keyboard shortcuts to do so. Currently, Interviewee 2's implementation of this editor involves manipulating the DOM directly and using its built-in functions for nodes, including `appendChild()`, `replaceChild()`, and `removeChild()` [16]. The shape of the data they are working with is

inherently hierarchical in nature given that the DOM is a tree itself. Interviewee 2 went on to describe the various types of nodes that they planned to use. They described the root node as a "site" node, which could contain any number of intermediate nodes like "pages" or "sections". Finally, the leaves of the tree are text, images, or shapes.

The purpose of this project is to be able to use the editor to create websites visually, and then export it into code. However, the grand vision of Interviewee 2 is to create an intermediate syntax that can be used to describe the websites, and then create some transpilers to convert that intermediate representation into the various front-end frameworks, such as Vue, React, Angular, and plain HTML. Because of this, they are looking to make their own custom virtual DOM implementation to handle the intermediate representation instead of using the native DOM and its functions. This made them a good candidate because they are looking for libraries to aid in the implementation of this custom tree structure and its operations. While the structure is undetermined completely, Interviewee 2 described that the plan would be to have nodes with their corresponding types, along with children stored in an array, meaning this schema again could be supported by Treecle.

In addition to asking about their specific use case, and how Treecle may lend itself to that tree schema, we also discussed what makes good and bad libraries, or more specifically, what a library would have to do to get adopted. Interviewee 2 mentioned the following traits, which are similar to the ones mentioned by Interviewee 1: ease of integration; ease of setting up the development environment; active maintenance and contributions; and great documentation including specifications, examples, and tutorials. When asked about past experiences with good and bad libraries, Interviewee 2 identified that overall the most important element is trust. They mentioned how thorough documentation and testing coupled with active maintenance and contributions are the most important factors in building trust with a library. They also added that they're extremely unlikely to use a library if it hasn't been updated in over 2 years.

5.3.3 Insights and Takeaways

Interviewee 2's use case would also lend itself well to Treecle given that it defines a schema in a way supported by Treecle, and requires operations that Treecle currently provides. Similarly to Interviewee 1's use case, this use case would also choose to store children in an array property, meaning the schema could be easily defined using Treecle's configuration options.

The unique background of Interviewee 2 also allowed them to give an interesting insight into the user experience of a library, particularly related to the trust essential to building a widely and easily adoptable library. The main point they made regarding building trust is including rich documentation, specifically, types of documentation for every skill level. What they meant by that is having specifications and technical API documentation for someone experienced who needs to find what a particular function does; having examples of uses for someone starting to familiarize themself with the library, and finally, having tutorials for someone completely new to the library. This insight was particularly useful in shaping the future of this project and is discussed in more detail in section 6.2.2.

Chapter 6

Conclusion

6.1 Summary of Contributions

This thesis introduces and proposes Treecle and Vastly: two JavaScript libraries that aim to provide simple, easy-to-use APIs for tree and AST operations respectively. The contributions of this project include the design, implementation, testing, and documentation for both libraries, as well as research into the justification of each one.

We have demonstrated via the use cases shown in sections 3.3 and 4.3 that both libraries have the potential to simplify codebases, which has additional benefits that include making the clients of these libraries safer from bugs, more readable, and more maintainable. In addition, through the interviews conducted and discussed in sections 5.2 and 5.3, we saw two more real-life use cases that Treecle could, in theory, be applied to. In short, Treecle and Vastly serve as useful utility libraries that can improve the quality of any codebase with trees or ASTs.

6.2 Future Work

Although both libraries boast a variety of functions already, the libraries are still currently in an alpha state, meaning there is still much work to be done. Listed below, and discussed

in more detail, are areas for future improvement for both libraries in general.

6.2.1 Expanding Functionality

The most significant area of improvement for future work in both libraries is expanding the set of offered operations to include more functions that have been justified as needed via the use cases that were researched. Researching potential use cases for Treecle uncovered a lot of typically-used procedures related to trees. The following is an incomplete list of the functions that could be added to Treecle in the future.

1. `clone()`: Takes in a tree and returns a deep copy of that tree.
2. `deepEquals()`: Takes in two trees and returns whether they are equal in structure and values.
3. `filter()`: Takes in a tree and a predicate and returns a new tree with only the nodes that satisfy the predicate.
4. `stringify()`: Takes in a tree and returns a string representation of that tree, which can be customized from use case to use case.
5. `remove()`: Takes in a node and removes itself from the tree, adjusting parent pointers as necessary.

6.2.2 Testing and Documentation

One area of improvement both libraries could benefit from significantly is testing and documentation. Currently, both libraries contain test suites that cover the majority of functions in the library, but there are a minority of functions left uncovered, as well as some more complex functions that could use more robust testing strategies. In addition, the documentation for both libraries is pretty well organized but could be improved by adding more examples and such for each function, as the documentation is currently sparse. Specifically,

more robust descriptions of functions could be added, and more importantly, examples and a playground. Interview 1, which was discussed in section 5.2, gave the important insight that having thorough documentation increases the chances that a developer would choose to adopt the library. Interview 2, from section 5.3 had a concurring opinion, emphasizing the important role documentation plays in the trust relationship between a library and a developer looking to use it.

6.2.3 Performance

Performance was considered a relatively low priority at each stage of the design process when implementing both Treecle and Vastly. It was considered loosely only in terms of asymptotic runtime, but we intentionally added constant factors to the runtime of some functions if it resulted in cleaner code. Now that the codebases are stable and have implemented most of the functions we wanted at the offset, it could prove useful to go back and optimize each function for performance, wherever possible.

Appendix A

Relevant Links

The following is a list of links relevant to Treecle and Vastly:

A.1 Treecle

1. Source code: https://github.com/mavoweb/treecle
2. Documentation and deployment: https://treecle.mavo.io/
3. NPM package: https://www.npmjs.com/package/treecle

A.2 Vastly

1. Source code: https://github.com/mavoweb/vastly
2. Documentation and deployment: https://vastly.mavo.io/
3. NPM package: https://www.npmjs.com/package/vastly

References

[1] *Acornjs/acorn: A small, fast, javascript-based javascript parser*, https://github.com/acornjs/acorn.

[2] *Benjamn/ast-types: Esprima-compatible implementation of the mozilla js parser api*, https://github.com/benjamn/ast-types.

[3] *Cesiumgs/cesium: An open-source javascript library for world-class 3d globes and maps :earth$_a$mericas* :, https://github.com/CesiumGS/cesium.

[4] *Cheeriojs/cheerio: The fast, flexible, and elegant library for parsing and manipulating html and xml.* https://github.com/cheeriojs/cheerio.

[5] X. Chen, Y. Chen, Z. Chen, *et al.*, "Abstract syntax tree for programming language understanding and representation: How far are we?," Dec. 2023.

[6] *Ecma-262 - ecma international*, https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[7] *Ericsmekens/jsep: Javascript expression parser*, https://github.com/EricSmekens/jsep.

[8] *Esprima*, https://esprima.org/.

[9] *Estools/estraverse: Ecmascript js ast traversal functions*, https://github.com/estools/estraverse.

[10] *Expression - cesium documentation*, https://cesium.com/learn/ion-sdk/ref-doc/Expression.html.

[11] *Joaonuno/tree-model-js: Manipulate and traverse tree-like structures in javascript.* https://github.com/joaonuno/tree-model-js.

[12] *Json*, https://www.json.org/json-en.html.

[13] *Kwhitley/treeize: Converts row data (in json/associative array format) to tree structure based on column naming conventions.* https://github.com/kwhitley/treeize.

[14] S. Mashuri, M. Sarib, F. Alhabsyi, H. Syam, and R. Ruslin, "Semi-structured interview: A methodological reflection on the development of a qualitative research instrument in educational studies," Feb. 2022.

[15] *Mavoweb/mavo: Create web applications entirely by writing html and css!* https://github.com/mavoweb/mavo.

[16] *Node - web apis | mdn*, https://developer.mozilla.org/en-US/docs/Web/API/Node.

[17] *Npm | home*, https://npmjs.com.

[18] *P0lip/simple-eval*, https://github.com/P0lip/simple-eval.

[19] *Scttnlsn/tree.js: Javascript library for creating and manipulating hierarchical tree structures.* https://github.com/scttnlsn/tree.js.

[20] *Simple-eval - npm*, https://www.npmjs.com/package/simple-eval.

[21] *Tree - leetcode*, https://leetcode.com/tag/tree/.

[22] L. Verou, A. X. Zhang, and D. Karger, *Mavo: Creating interactive data-driven web applications by authoring html*, 2016. DOI: 10.1145/2984511.2984551. URL: https://dl-acm-org.libproxy.mit.edu/doi/pdf/10.1145/2984511.2984551.

[23] *Vuejs/core: vue.js is a progressive, incrementally-adoptable javascript framework for building ui on the web.* https://github.com/vuejs/core.

[24] *Weakmap - javascript | mdn*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap.

[25] *Webflow: Create a custom website | visual website builder*, https://webflow.com/.

[26] *Welcome home : Vim online*, https://vim.org/.