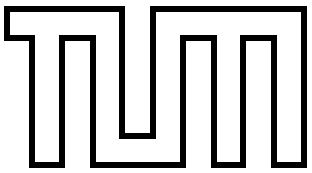


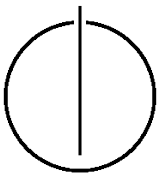
DEPARTMENT OF INFORMATICS

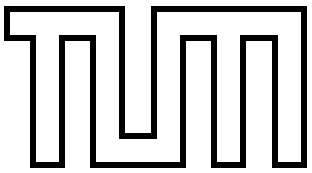
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Control Flow Graphs from JavaScript Programs

Marius Daniel Schulz





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Control Flow Graphs from JavaScript Programs

Herleitung von Kontrollflussgraphen aus JavaScript-Programmen

Author: Marius Daniel Schulz

Supervisor: Prof. Dr. Helmut Seidl

Advisor: Dr. Michael Petter

Submission date: 15. August 2015

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 15. August 2015

_____

Abstract

This thesis contributes to the understanding of how to derive static control flow graphs from JavaScript programs for the purpose of enabling a variety of static data flow analyses. An algorithm is presented that defines how to systematically construct a control flow graph from a program's abstract syntax tree. Furthermore, some limitations of the approach are discussed that are inherent to the dynamic nature of JavaScript. Finally, an exemplary implementation serves as a proof of concept for the applicability of the control flow graph derivation algorithm.

Contents

1 Introduction

1.1 Motivation

When JavaScript[1] first appeared in 1995, it was originally envisioned to be a small scripting language for adding interactivity to web pages, which were mostly static HTML documents back then. The expected number of lines for the average script was probably in the dozens. Nowadays, JavaScript applications with several hundred thousand lines of code are being developed and maintained. The language has been brought to more and more environments and platforms over time, such as was the case with Node.js [**?**] in 2009, and it still continues to grow in usage.

As the size of programs written in JavaScript grows, so does their complexity. Large code bases are considerably harder to understand, debug, and maintain than small ones. Programmers are therefore more likely to introduce bugs into the software as the code base grows. The problem with JavaScript is that it offers little help to programmers during implementation. A dynamically typed language, JavaScript enforces neither consistent type usage nor guaranteed initialization of variables, for example. Moreover, some unfortunate language design choices[2] may lead to unexpected and undesirable behavior at runtime.

The growing ubiquity of JavaScript and its usage in large-scale, critical applications awakens interest in techniques for automatic program verification. Static analysis tools can help developers find erroneous pieces of code by looking for patterns that, according to a given defect model, lead to commonly made mistakes. Data flow analysis, for example, inspects declarations and usages of variables in a program to track the propagation of their values. Using data flow analyses, one can try to answer the following questions:

- Is there a code path that leaves a given variable uninitialized?
- Is the value of a given variable always within a certain range?
- Does a given variable always contain values of the same type?
- Does a given variable ever change after its initial assignment?
- Are there any code parts that can never be reached (dead code)?

At the heart of such data flow analyses lies the inspection of the underlying program's static control flow graph, which must therefore be derived.

---

[1] "JavaScript" is a trademark of Oracle Corporation. The untrademarked name used in the language specification [**?**] as standardized by Ecma International® is "ECMAScript". Following common usage (and for ease of reference), the name "JavaScript" will be used hereinafter to refer to the ECMAScript language.

[2] Outside of *strict mode*, assigning a value to a simple identifier than cannot be resolved leads to the automatic creation of a global variable of that name. This behavior is usually unintended and does more harm than good.

1.2 Related Work

Recently, there has been notable related work in the area of static control flow analysis. For instance, the Flow [**?**] type checker developed at Facebook aims to statically find type errors in JavaScript programs. Flow analyzes plain JavaScript code, which does not have to be explicitly decorated with type annotations. The latter can optionally be included, though, to gradually improve the significance of the type checker.

Additionally, [**?**] presents a static analysis platform for JavaScript based on JSAI, an abstract interpreter that offers user-configurable sensitivity and provably sound analysis. Due to the dynamic nature of the language, however, much work is done in the area of dynamic analyses. [**?**], [**?**], and [**?**], for example, focus on improving the security of JavaScript applications through dynamic information flow control.

2 Capabilities

This thesis outlines the derivation of intraprocedural, static control flow graphs based on a JavaScript program's abstract syntax tree. The goal is to capture the semantics of the main program or any of its functions in the respective control flow graphs, which can be used as the basis for various static data flow analyses as discussed in "Motivation".

2.1 Control Flow Graph Derivation

The derivation of a control flow graph as described in this thesis is based on a purely static analysis of the abstract syntax tree of a program. Consequently, no analysis of actual runtime behavior is performed. This limits the capabilities somewhat, since JavaScript is a highly dynamic programming language with support for language constructs that are difficult to analyze statically. Because of the highly dynamic nature of the language, the control flow graph derivation is subject to the limitations listed in section 3 ("Limitations"). Despite these difficulties, the algorithm analyzing a program's abstract syntax tree must not falsify the program semantics. It must therefore make conservative assumptions about the dynamic parts of the program in order to guarantee sound control flow analysis. Section 4 ("Algorithm") describes the algorithmic approach in detail. Section 5 ("Implementation") presents an exemplary implementation that was developed as part of the work on this thesis.

2.2 Export in Various Formats

In addition to deriving control flow graphs, the exemplary implementation is capable of exporting its resulting data structures. A given control flow graph can be exported as a JavaScript object, which can be analyzed in the JavaScript program using the derivation library. Additionally, control flow graphs can be serialized as JSON for further processing in arbitrary tools, programming languages, and environments. Finally, the DOT export allows for serializing the structure of a control flow graph using the DOT [**?**] graph description language. Visualization tools like Graphviz [**?**] can then layout the graph and render it as an image. The different export formats are explained in sections 5.2.2 ("Object Export"), 5.2.3 ("JSON Export"), and 5.2.4 ("DOT Export"). Appendix A shows examples of serialized exports in JSON and DOT format.

3 Limitations

This section details the limitations that are an inherent part of the process of deriving an accurate control flow graph from a JavaScript program. They mainly revolve around dynamic language features, closure and scope management, and implicit errors.

3.1 Dynamic Language Features

A highly dynamic programming language, JavaScript supports various language constructs that are difficult or even impossible to analyze statically. Two examples are string-based access to object properties and the evaluation of strings as code. If the strings in question depend on environment variables such as time, user input, or external data, it is generally undecidable which piece of code is going to be executed.

3.1.1 String-Based Property Access

JavaScript offers a lightweight syntax for creating object literals. Objects consist of properties that map string keys to arbitrary values. This includes functions too, which are first-class citizens in JavaScript:

```
1  var calculator = {
2      increment: function(x) {
3          return x + 1;
4      },
5      square: function(x) {
6          return x * x;
7      }
8  };
```

There are two ways to access an object's properties. The first option is to use *dot notation* as known from other languages with a C-like syntax:

```
1  var two = calculator.increment(1);
```

The other option is to use the property's string key in conjunction with *square bracket notation* that looks for a property with exactly that key:

```
1  var two = calculator["increment"](1);
```

No matter which of the two notations was used to look up the property, the value `undefined` is returned if neither the object itself nor any of the objects in its *prototype chain*[3] define a property with the given key. The above

[3]See section 4.2.1 of the ECMAScript Language Specification. [**?**]

invocations of the `increment` method would have failed with a `TypeError` if the property lookup returned any value that was not a function.

The difficulty of statically analyzing code that accesses object properties using string keys and square bracket notation lies in the observation that it is generally undecidable which (if any) property's value is returned. In the above example, it is trivial for a static analyzer to recognize that the provided key is a string literal with a known constant value. However, the JavaScript grammar allows an arbitrary expression to be placed within the square brackets of the property access; the result of evaluating this expression is then coerced into a string and used as a property key. The following four examples show how different the results of invoking a method accessed this way can be, depending on the value of the property key:

- If `key` contains the value `"increment"`, calling `calculator[key](5)` will return the value `6`. Similarly, if `key` is set to `"square"`, calling `calculator[key](5)` will return the value `25`.
- If `key` is set to `"hasOwnProperty"`, calling `calculator[key](5)` will look for and invoke the `hasOwnProperty` method defined on the `Object` prototype, which returns `false` because the object does not define a property with that key.
- If `key` is set to `"invalid property"`, calling `calculator[key](5)` will trigger a lookup for a method of that name. Neither the object itself nor any other object in its prototype chain defines such a property, and therefore `undefined` is returned. Since `undefined` cannot be invoked as a function, a `TypeError` is thrown.

This illustrates nicely how drastically the control flow behavior can vary depending on the value of the expression within the square brackets. In cases where the key is set to `"increment"`, `"square"`, or the name of any method in the prototype chain, that method will be executed. If, however, the key cannot be found, an exception will be thrown, thereby causing an abrupt completion and a jump in control flow.

3.1.2 Evaluating Strings As Code

In addition to allowing string-based access to object properties, JavaScript provides other language mechanisms that lead to highly dynamic code execution. The most obvious such construct is the global `eval` function that evaluates a given string as code:

```
1  var count = 0;
2  eval("count++;");
```

After running the above code, `count` contains the value `1`. The string passed to the `eval` function can contain any sequence of statements or expressions, including calls to `eval` itself. Generally speaking, that makes it impossible for a static analyzer to understand what statements and expressions are being executed in what order. To further complicate matters, the string that is being evaluated does not have to be a string literal with a known constant value. It can be the result of any expression evaluating to a string, similar to dynamic property keys.

Another difficulty is that in some cases, it is not even possible to statically decide whether a program calls the `eval` function anywhere. The following code snippets illustrates how dynamic property access is used to access the `eval` function without explicitly mentioning its name:

```
1  var obj = window;
2  var key = String.fromCharCode(101, 118, 97, 108);
3  var result = obj[key]("2 + 2");
```

The global object within a browser environment, `window`, is assigned to the variable `obj`. At the same time, `key` is assigned the value `"eval"` by creating a string from the given Unicode values, which represent the four characters *e*, *v*, *a*, and *l*. Finally, the dynamic property access returns the `eval` function defined on the global object and evaluates the string `"2 + 2"`. Therefore, `result` contains the value `4`.

The following example illustrates the unpredictability of evaluating arbitrary JavaScript code using the `eval` function:

```
1  var hasEvalBeenRun = false;
2
3  function runEval(string) {
4      eval(string);
5      hasEvalBeenRun = true;
6  }
7
8  runEval("var hasEvalBeenRun;");
```

Looking at the variable declaration and the `runEval` function, one might assume that `hasEvalBeenRun` is set to `true` after the function has been executed (and given that running `eval` did not throw an exception). This does not have to be the case, though, as the example shows. By evaluating the string `"var hasEvalBeenRun;"` within the function, a new local variable binding is created that *shadows* the outer variable of the same name. Thus, the assignment does not modify the outer variable, but the local one.

3.1.3 Dynamically Creating Functions

Another dynamic language construct is the `Function` constructor, which makes it possible to define new functions at runtime. It accepts an arbitrary number of string parameters, the last of which represents the function body. All other parameters are treated as the names of the function's formal arguments, in the order in which they are passed. [**?**]

The following code snippet shows how a simple `add` function can be created by calling the `Function` constructor with two argument names and a function body calculating and returning the sum of the given values:

```
1  var add = new Function("a", "b", "return a + b;");
```

Similar to the evaluation of strings as code using the `eval` function, the `Function` constructor poses a problem for static analyzers. The string representation of the function body does not have to be a string literal, which would have a constant value. Typically, the `Function` constructor is used to create a function whose body is put together using dynamic string concatenation of code fragments. In general, this makes it impossible to predict statically what the body of the function will look like.

3.2 Implicitly Thrown Errors

In JavaScript, exceptions can be thrown explicitly using the `throw` keyword. When an exception is thrown, none of the statements following the `throw` statement will be executed. Instead, control will be passed to the first `catch` block in the call stack. If no such `catch` block exists, the exception will cause the program to terminate. [**?**]

However, the absence of `throw` statements within a piece of code does not guarantee that no error will be thrown within it. For instance, the following situations (and many others that are not listed below) cause the JavaScript runtime to throw an error:

- A `ReferenceError` is thrown when trying to dereference a variable that has not previously been declared. [**?**]
- A `SyntaxError` is thrown when the JavaScript engine detects syntactically invalid code during its initial parsing phase. [**?**]
- A `TypeError` is thrown when a variable that does not hold a reference to a function is attempted to be called as a function. [**?**]

This list is by far not exhaustive, but it illustrates that many common operations may cause an error to be thrown implicitly. Consequently, the control flow graph for a piece of code would have to include error transitions from every single node that could possibly cause an error to either a `catch` handler or the error exit node of a program. That would result in many additional edges and an even more cluttered control flow graph. It is for this reason that the implementation outlined in section 5 ("Implementation") does not include transitions for implicitly thrown errors.

4 Algorithm

This section illustrates the algorithmic approach used to derive a static control flow graph from a JavaScript program. It describes the graph format and how to parse statements and expressions. This section also discusses techniques for simplifying negated conditions where possible.

4.1 Obtaining the Abstract Syntax Tree

The algorithm for deriving static control flow graphs does not operate on the JavaScript program's raw source code, but on its abstract syntax tree. Therefore, a JavaScript parser is required to parse the program and return its abstract syntax tree, which then serves as the basis for control flow graph derivation. While the format of the abstract syntax tree can vary between parsers, the following sections assume the ESTree [**?**] format (without loss of generality). ESTree describes a simple format that covers the entire ECMAScript 5.1 grammar, and it is implemented by popular open-source JavaScript parsers such as Acorn [**?**] or Esprima [**?**]. The control flow graph derivation library described in section 5 ("Implementation"), for instance, uses Esprima to obtain an abstract syntax tree in ESTree format.

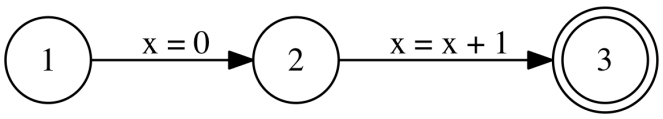
Appendix A shows an exemplary JavaScript program and its abstract syntax tree as generated by Esprima.

4.2 Deriving a Control Flow Graph

Once the abstract syntax tree of a program has been generated, it can be traversed to successively construct the control flow graph.

4.2.1 Graph Format

The control flow graph is a directed graph that is generally not acyclic because of cycles created by loop structures. Its nodes represent states of the program, whereas its edges represent transitions between those states. Edges can be annotated with statements or expressions that change the state of the program. Here is an example of a very simple control flow graph:



It represents the following simple JavaScript program:

```
1  var x = 0;
2  x = x + 1;
```

The abstract syntax tree in ESTree format looks like this:

```
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "x"
          },
          "init": {
            "type": "Literal",
            "value": 0
          }
        }
      ],
      "kind": "var"
    },
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "AssignmentExpression",
        "operator": "=",
        "left": {
          "type": "Identifier",
          "name": "x"
        },
        "right": {
          "type": "BinaryExpression",
          "operator": "+",
          "left": {
            "type": "Identifier",
            "name": "x"
          },
          "right": {
            "type": "Literal",
            "value": 1
          }
        }
      }
    }
  ]
}
```

4.2.2 Parsing Statements and Expressions

Every JavaScript program's abstract syntax tree in ESTree format has a root node of type `Program` with a `body` property that contains an array of all the main program's statements. The simple JavaScript program at the end of the previous section consists of two statements: a variable declaration and an expression statement. `x = x + 1` is an assignment expression and therefore not a statement on its own, but since it appears in statement position, the expression is treated as a statement of type `ExpressionStatement`.

The process of traversing the abstract syntax tree and deriving a control flow graph from the statements and expressions encountered can be nicely broken up into many functions, each of which knows how to parse a specific node type. One such function can call another to parse the various components of the node it knows how to parse, thereby creating a hierarchy of nested function calls. For example, the aforementioned program could be parsed by a function call hierarchy similar to the following pseudocode:

```
- parseProgram()
    - parseStatements()
        - parseStatement()
            - parseVariableDeclaration()
                - parseVariableDeclarators()
                    - parseVariableDeclarator()
                        - parseInit()
                            - parseExpression()
                                - parseLiteral()
                        - parseIdentifier()
        - parseStatement()
            - parseExpressionStatement()
                - parseExpression()
                    - parseAssignmentExpression()
                        - parseRightHandSide()
                            - parseBinaryExpression()
                                - parseLeftHandSide()
                                    - parseIdentifier()
                                - parseRightHandSide()
                                    - parseLiteral()
                                - parseOperator()
                        - parseLeftHandSide()
                            - parseIdentifier()
                        - parseOperator()
```

In most cases, every expression statement adds to the control flow graph a new node with an incoming edge that is annotated with the expression. An

exception to that rule, however, are expression statements that wrap expressions of type `SequenceExpression`.[4] Rather than adding a single node, an individual node is added for each expression in the sequence, which slightly reduces the complexity of those edge annotations in the control flow graph.

4.2.3 Composing Structural Patterns

A control flow graph is created by composing various primitive patterns for statements and expressions. The following figures show a couple of examples for common control structures like conditions and loops:

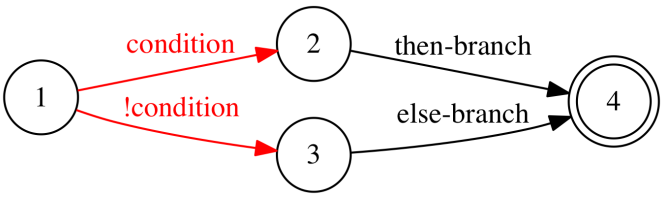


Figure 1: An `if`-statement with an `else`-block

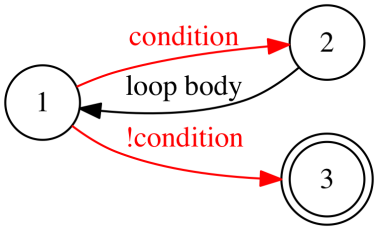


Figure 2: A `while`-loop

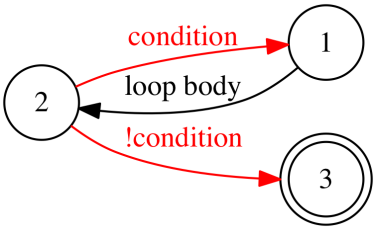


Figure 3: A `do-while`-loop

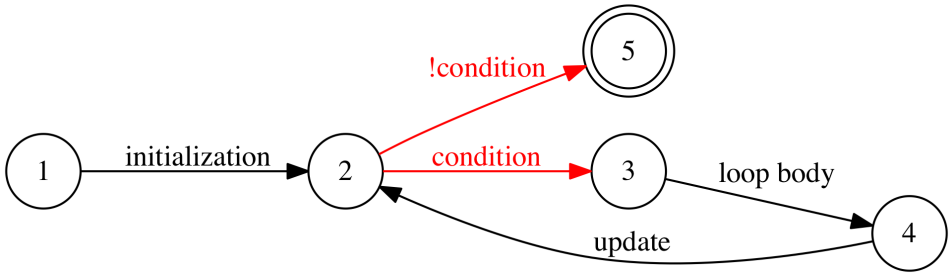


Figure 4: A `for`-loop

[4] A sequence expression as specified by ESTree [**?**] consists of two or more expressions, separated by the comma operator, and evaluates to the value of its last expression.

Each pattern starts with node #1 and ends with the node that has a double border. In the above figures, those final nodes only have a double border for the purposes of clarification; they are regular nodes in the control flow graph. When two consecutive patterns are combined, the final node of the first becomes the entry node of the second. The black edges that are labeled *then-branch*, *else-branch*, and *loop body* represent the bodies of the control structures. They can consist of arbitrarily many statements, which can contain further nested structures themselves. Similarly, *initialization* and *update* can hold arbitrarily complex expressions. For example, a `for`-loop can have a sequence expression like `i++, j--` as its update component.

The red edges represent conditional transitions. A node that is not a final node must have either exactly one unconditional outgoing edge or exactly two conditional outgoing edges. For a pair of conditional edges, the condition is evaluated only once. Depending on whether the result is truthy or falsy[5], the corresponding conditional edge is taken.

Appendix A presents a larger program and its control flow graph.

4.2.4 Simplifying Negated Conditions

As Figure 1 in section 4.2.3 ("Composing Structural Patterns") shows, an `if`-statement with an `else`-block is represented by a control flow structure that has two conditional edges. One is taken when the condition is truthy (the *then-branch*), the other when it is falsy (the *else-branch*). The following listing illustrates this situation:

```
1  if (x < 0) {
2      // ...
3  } else {
4      // ...
5  }
```

The then-branch is taken when `x < 0` returns a truthy value. It might therefore seem intuitive to assume the else-branch to be taken if and only if `x >= 0` is truthy. However, the precise semantics are slightly different. The else-branch is taken if and only if `x < 0` returns a falsy value. In general, the correct negation of `x < 0` is not `x >= 0`, but `!(x < 0)`. The reason for this seemingly odd behavior is that JavaScript defines some special values for which no reasonable comparison can be done using relational operators. `NaN` and `undefined` are two such values for which no sound order relation is defined.

[5] JavaScript has six values that are considered falsy: `undefined`, `null`, `false`, `0` (zero), `NaN` ("not a number"), and `""` (empty string). All other values are considered truthy.

The following relational comparisons against `NaN` and `undefined` illustrate why the condition cannot be safely inverted (without changing the semantics of the program) by swapping the operator accordingly:

```
NaN <  0   // false
NaN >  0   // false
NaN <= 0   // false
NaN >= 0   // false

undefined <  0   // false
undefined >  0   // false
undefined <= 0   // false
undefined >= 0   // false
```

This observation prevents some simplification of conditional edges in a control flow graph. To stay true to the program semantics, the conditional annotation of the edge transitioning into the else-branch must be `!(x < 0)` (rather than the simpler `x >= 0`) to account for those special values.

However, some simplification of expressions is possible. Both the equality comparison operators (`==` and `!=`) and the identity comparison operators (`===` and `!==`) can safely be negated by flipping the first character from an equal sign to an exclamation mark, and vice-versa. Also, two consecutive applications of the unary negation operator (`!`) cancel each other out and do not affect the truthiness of a value. Therefore, the following simplifications are safe and do not affect the semantics of the program:

```
// Simplifying a negated identity comparison
!(x === 0)
x !== 0

// Simplifying a negated inequality comparison
!(x != 0)
x == 0

// Canceling out a double negative
!!(x)
x
```

Nota bene: The above simplifications are only semantically correct w.r.t. the evaluation of the expression to a truthy or falsy value, as is the case when evaluating the condition of an `if`-statement. The simplifications do not necessarily result in an identical value or even a value of the same type. For instance, the unary negation operator coerces its operand into a boolean value. Therefore, `!!0`, which evaluates to the boolean value `false` (because `!0` evaluates to `true`), is not identical to `0`, which is a numeric value. Both are falsy values, though, and thus direct control flow correctly.

5 Implementation

As part of this thesis, the outlined algorithm for deriving control flow graphs has been implemented. The resulting JavaScript program consists of two main parts: the *Styx*[6] core library that implements the control flow graph derivation (discussed in section 5.2) and a command-line application that provides a wrapper around the core library (discussed in section 5.3).

5.1 Technology Stack

The following sections give an overview of the programming languages and build tools that were used to implement and build the Styx core library and the wrapping command-line application.

5.1.1 Programming Languages

When choosing the technology stack to build upon, the goal was to develop a piece of software that could run in any modern JavaScript environment, including both Node.js and all major browsers' JavaScript engines. Taking that into account, an adequate language version to target is ECMAScript 5, which was standardized [**?**] in 2009. Now in 2015, it is widely supported in various environments and therefore a suitable language target.

However, the Styx core library is not written in JavaScript directly, but in TypeScript [**?**], a superset of JavaScript that adds optional static typing to the language. After the TypeScript compiler has found a given program to be type-correct, it emits an equivalent JavaScript program with all type annotations removed. Therefore, the type information is a purely compile-time artifact without any runtime manifestation. The decision in favor of TypeScript was made to take advantage of the following benefits:

1. Static typing helps detect a certain class of errors at compile-time by checking that a program is written in a type-correct manner. This is especially helpful in a dynamic language like JavaScript, which allows any function to be called with an arbitrary number of arguments.

2. TypeScript supports the new ECMAScript 2015 language version that was standardized [**?**] in June of 2015. It adds a plethora of features to JavaScript, such as a native module system, lexical scoping, arrow functions, destructuring, and more. The TypeScript transpiler is capable of down-leveling most new language features such that engines implementing ECMAScript 5 can understand them. This allows for using next-generation JavaScript without sacrificing compatibility.

[6] The library was named after the river Styx, which in Greek mythology is believed to form the boundary between Earth and the Underworld.

5.1.2 Build Tools

To build Styx from source, both the TypeScript compiler and the Browserify module bundler [?] must be run as discussed in the following paragraphs.

TypeScript Compiler. Since the entire code base for the Styx core library is written in TypeScript, it needs to be transpiled to plain JavaScript before it can be executed by a JavaScript engine. The official TypeScript compiler, *tsc*, is used to accomplish that. It ships as part of the npm [?] package `typescript`, which can be installed by running the following command:

```
$ npm install --global typescript
```

The `src` directory of the code base contains a special `tsconfig.json` file that specifies configuration settings for the TypeScript compiler, such as the list of files to include and which compiler flags to set. During development, the compiler is run in *watch* mode via the `-w` flag: every time a `.ts` file changes on disk, the TypeScript compiler will type-check and transpile it. The generated JavaScript code is then written to a `.js` file located within the destination folder that is specified in `tsconfig.json`.

Modules. To structure the code base, the Styx core library is broken up into native JavaScript modules that were introduced as part of ECMAScript 2015. As of version 1.5, the TypeScript compiler understands this syntax and is able to transpile the modules into a different format. Styx targets CommonJS modules [?], which is specified within `tsconfig.json` by setting the `module` property to `"commonjs"`. Following the CommonJS format, the rewritten modules declare their imports via `require` and their exported values via `exports`. Once all modules have been transpiled this way, the Styx core library is now ready to be loaded from within a Node.js application or any other JavaScript environment that supports CommonJS modules.

Browserify. While a Node.js application is able to resolve CommonJS dependencies that are imported via the `require` function, browsers cannot do that natively. One option to work around this issue is to use a module loader, which dynamically resolves CommonJS dependencies. The disadvantage of such module loaders is that they have to make additional HTTP requests to resolve the declared dependencies. Depending on the nesting level of `require` calls and the performance goals of an application, dynamically fetching CommonJS modules might not be a viable option. This is why the build process of the Styx core library includes feeding the transpiled CommonJS modules to Browserify [?], a popular module bundler. Starting at a given entry file, Browserify will recursively analyze all the `require` calls and bundle all dependencies found. The result is a single JavaScript file that can easily be requested by a browser.

5.2 Core Library

The key part of the implementation is the Styx core library. It implements the outlined algorithm for deriving the control flow graph from a program or function's abstract syntax tree. Styx itself does not have any dependencies. However, it requires that the abstract syntax tree it is handed follow the ESTree [**?**] specification. The two JavaScript parsers Acorn [**?**] and Esprima [**?**], for instance, return an abstract syntax tree in ESTree format that can directly be passed along to the Styx parser.

5.2.1 Public Interface

Styx exposes a public interface that offers a `parse` method and several other methods for exporting a given control flow graph in various formats.

parse. The `parse` method accepts as its parameters an abstract syntax tree and a parser configuration. Using these two inputs, it derives a control flow graph for each of the program's functions and also for the main program itself. It then returns all control flow graphs within a wrapping data structure that is called a *flow program*. The first parameter of the `parse` method, `ast`, represents the abstract syntax tree to be parsed. It is a required parameter that must comply with the `Program` format as specified by ESTree. Additionally, the optional second parameter `options` can be provided to configure which optimization passes to apply to each control flow graph of the flow program.

exportAsObject. The `exportAsObject` method accepts a single parameter named `flowProgram` that has been returned from the `parse` method. It exports the given program and all its functions as a JavaScript object structure that holds all nodes and edges in flat arrays rather than nested object structures, thus allowing for simpler serialization and deserialization. The `exportAsObject` method is discussed in detail in section 5.2.2.

exportAsJson. Similar to the `exportAsObject` method, `exportAsJson` takes a flow program and converts it to a JavaScript object with a different structure. However, `exportAsJson` does not return the new object itself, but a string containing its JSON representation. An optional settings parameter allows for customization of the JSON indentation. `exportAsJson` is explained in section 5.2.3.

exportAsDot. Lastly, the `exportAsDot` method exports a single control flow graph in DOT format. The resulting graph description can be passed along to a visualization tool to render an image of the given control flow graph. `exportAsDot` is detailed in section 5.2.4.

5.2.2 Object Export

When a flow program is exported as a JavaScript object, the returned value adheres to the following *FlowProgram* schema:

```
interface FlowProgram {
    program:   MainProgram;
    functions: FlowFunction[];
}

interface MainProgram {
    flowGraph: FlowGraph;
}

interface FlowEdge {
    from:  number;
    to:    number;
    type:  string;
    label: string;
    data:  Object;
}

interface FlowFunction {
    id:        number;
    name:      string;
    flowGraph: FlowGraph;
}

interface FlowGraph {
    nodes: FlowNode[];
    edges: FlowEdge[];
}

interface FlowNode {
    id:   number;
    type: string;
}
```

FlowProgram. A flow program that has been exported as a JavaScript object follows the *FlowProgram* schema, which defines the two properties `program` and `functions`. The `program` property holds a single descriptor object for the main program. Similarly, the `functions` property holds an array of function descriptors, each of which stores both an `id` and a `name` property to uniquely identify the function. Both *MainProgram* and *FlowFunction* descriptors define a `flowGraph` property containing the associated control flow graph.

FlowGraph. The *FlowGraph* schema defines the two properties `nodes` and `edges` that are both flat arrays. Representing a control flow graph this way has the advantage that every node and every edge only appears once in the serialized export. If, however, a node were to define an `outgoingEdges` property, every edge of that array would have to be part of another node's `incomingEdges` property as well. In that case, all edges would have been serialized twice, thus complicating the deserialization process.

FlowNode. Within the `nodes` array, every element follows the *FlowNode* schema and therefore has a numeric `id` property. In addition to that, the `type` property is set to one of the following string values:

1. `"Entry"` for the single entry node of a control flow graph,
2. `"SuccessExit"` for its single success exit node,
3. `"ErrorExit"` for its single error exit node, or
4. `"Normal"` for all other nodes.

FlowEdge. Finally, the `edges` array contains all edges of the control flow graph. An edge's `from` and `to` properties describe which source and target node it connects. Similar to a node, an edge also has a `type` property that has one of the following string values:

1. `"AbruptCompletion"` for an edge representing a jump in control flow caused by an abrupt completion (`break`, `continue`, `return`, or `throw`),
2. `"Conditional"` for an edge that is part of a pair of edges representing the truthy and falsy paths of a condition,
3. `"Epsilon"` for an edge that has neither a label nor attached data and exists purely for connecting nodes in the control flow graph, or
4. `"Normal"` for all other edges.

The `label` property contains a string representation of the statement or expression for which the edge has been created. Additionally, the `data` property holds the part of the abstract syntax tree that Esprima parsed for the given statement or expression. That includes the precise type of the syntax tree node and all of its properties. The `data` property is useful when a deeper understanding of the current part of the control flow graph is required than can be obtained from the edge's string representation alone.

5.2.3 JSON Export

The JSON export is a thin wrapper around the Object export described in the previous section. Instead of returning the raw JavaScript object itself, the `exportAsJson` method serializes it as JSON and returns the resulting string. `exportAsJson` accepts an optional second parameter that can specify a `pretty` property. The resulting JSON string will be indented for better readability if `pretty` has a truthy value; otherwise, it will be minified.[7]

Appendix A lists an example of a flow programs exported as JSON.

[7]Note that the JSON string is only indented and formatted to help a human reader understand the object structure. Outside of string literals, whitespace is insignificant and is generally ignored by JSON parsers.

5.2.4 DOT Export

Flow programs can be exported in DOT [?] format as well. DOT is a graph description language that can be consumed by visualization tools such as Graphviz [?] to render various sorts of graphs. In the case of Styx, the DOT export is a directed and generally not acyclic graph. It describes a finite state machine representing the control flow in a program or function.

In contrast to the JSON export, the DOT export does not include the entire flow program, but only a single control flow graph. Since Styx works on an *intra*procedural rather than an *inter*procedural level, the control flow graphs of the program and its functions are parsed separately from each other and can therefore be visualized independently.

Appendix A shows an example of a control flow graph exported as DOT and its respective Graphviz rendering.

5.2.5 Installation via npm

In order to make the Styx library easily accessible to JavaScript developers, it has been published as a package to the npm Registry [?], a public collection of open-source JavaScript packages. On a machine with npm set up, the library can be installed by running the following command:

```
$ npm install --save styx
```

Once the `styx` package has been installed, the library can be loaded through `require` calls like any npm package. A very simple example program (that also requires the `esprima` package to be installed) can look like this:

```
1  // Import Esprima and Styx
2  var esprima = require("esprima");
3  var styx = require("styx");
4
5  // Create a string with demo code
6  var code = "console.log('Hello World!');";
7
8  // Generate the abstract syntax tree
9  var ast = esprima.parse(code);
10
11 // Derive the control flow graph
12 var flowProgram = styx.parse(ast);
13
14 // Export the control flow graph as JSON
15 var jsonExport = styx.exportAsJson(flowProgram);
16 console.log(jsonExport);
```

5.2.6 Building from Source

The source code for Styx is published under the permissive MIT License [**?**] and is publicly available on GitHub [**?**]. To build Styx from source, the Git repository needs to be cloned on a local development machine. By running the following commands, the repository is going to be cloned into the `styx` folder, which is then made the current working directory:

```
$ git clone https://github.com/mariusschulz/styx.git
$ cd styx
```

Now that the repository has been set up, the npm dependencies declared in the special `package.json` file need to be restored. They can be installed via npm by running the `install` command without any arguments:

```
$ npm install
```

As described in the "Build Tools" section, the TypeScript code must be transpiled into plain JavaScript. That is most easily done by calling the TypeScript compiler with `-p` argument pointing to the directory containing the `tsconfig.json` configuration file (assuming that the current working directory is the root of the Git repository):

```
$ tsc -p src
```

To simplify the process of bundling modules using Browserify as much as possible, the Gulp [**?**] build system has been set up. It defines a `browserify` task in its `gulpfile.js` configuration file that is found in the repository root. Additionally, the `package.json` file defines an npm script called `browserify` so that the module bundling can be triggered by running the following command:

```
$ npm run browserify
```

During development, it is handy to have both the TypeScript compiler and Browserify listen for file changes. When a `.ts` file is saved within the `src` folder, the TypeScript compiler will transpile it immediately and save the corresponding `.js` file within the destination directory. That, in turn, will be picked up by the Browserify listener, which will trigger the module bundling again.

To run the TypeScript compiler in watch mode, call it with the `-w` flag:

```
$ tsc -p src -w
```

The browserify listener can be started by running another npm script:

```
$ npm run browserify-watch
```

5.3 Command-Line Interface

The *Styx CLI* project provides a simple command-line interface that exposes the functionality of the Styx core library. It imports the Esprima [**?**] parser and is therefore capable of loading a JavaScript file, generating its abstract syntax tree, and deriving a control flow graph from it.

5.3.1 Installation via npm

Similar to the Styx core library, the Styx command-line interface is available via npm as well. It can be installed by running the following command:

```
$ npm install --global styx-cli
```

As part of the global installation, the Styx CLI has been added to the system's PATH environment variable. It can then be accessed by calling `styx` within a terminal. The available arguments are listed in detail in section 5.3.3 ("Command-Line Arguments").

5.3.2 Building from Source

The source code for the Styx CLI project is available on GitHub [**?**] under the MIT License, just like the core library. The process of building it from source is similar as well:

```
$ git clone https://github.com/mariusschulz/styx-cli.git
$ cd styx-cli
$ npm install
```

Unlike the core library, the Styx command-line interface is not written in TypeScript. It does, however, use ECMAScript 2015 language features that need to be transpiled to ECMAScript 5 in order to be widely supported by current Node.js installations. The Babel [**?**] transpiler, which is installed as part of the `gulp-babel` npm package, is used to do that. It is invoked by the Gulp build system when executing the `babel` task defined in the `gulpfile.js` configuration file. The Gulp `babel` task can be triggered by running this npm script:

```
$ npm run babel
```

Another npm script has been defined for setting up a file watcher that re-triggers the Babel transpilation process whenever a `.js` file changes:

```
$ npm run babel-watch
```

After the transpilation is done, the `built` directory contains the source files making up the Styx command-line interface. All JavaScript files have been down-leveled from ECMAScript 2015 to ECMAScript 5, ready to be executed by any current Node.js version.

5.3.3 Command-Line Arguments

Using the Styx command-line interface installed via npm, a JavaScript file can be analyzed by running the following command:

```
$ styx input.js
```

This simple usage of the `styx` executable provides a single argument, the JavaScript file to analyze, which is the only argument that is required. The following command-line arguments are accepted:

- The `--format` argument (or its `-f` alias) determines the target output format. If specified, this argument must be either `json` or `dot`. If not specified, the export defaults to the JSON format.
- The `--graph` argument (or its `-g` alias) specifies the id of the function whose control flow graph should be exported. This argument only applies when `--format` is set to `dot`. If not present or set to a falsy value, the main program's control flow graph will be exported.
- The `--minify-json` (or its `-m` alias) argument can be provided to export minified (rather than indented) JSON to minimize the size of the output. This argument only applies when `--format` is set to `json`.
- The `--output` argument (or its `-o` alias) can be used to specify a path to a file to which the output should be redirected. If not present, the serialized export will be written to standard output.
- The `--help` argument displays a help screen that lists instructions for command-line usage and all available arguments. If the `--help` argument is present, no export will happen, no matter what other arguments are specified; instead, only the help screen will be shown.

As an alternative to the `--output` argument, the Unix output redirection operator (`>`) can be used to write the serialized export to a file. The following two commands are therefore semantically equivalent:

```
$ styx input.js --output output.json
$ styx input.js > output.json
```

Finally, the following command exports the control flow graph of the function with the id 42 in DOT format:

```
$ styx input.js --format dot --graph 42 > output.gv
```

6 Future Work

The exemplary implementation that was done as part of this thesis is mainly a proof of concept. It demonstrates the capabilities of the outlined approach for statically analyzing JavaScript programs to derive control flow graphs. Some of the limitations described in section 3 ("Limitations"), such as the highly dynamic nature of the language, are inherent to JavaScript and therefore cannot be altered. Other aspects, however, have simply not been implemented in the Styx core library, mostly due to lack of time.

One such area that would benefit from further future work is the proper tracking of identifiers across scopes. When a function declares a variable or function with the name of a variable or function that is already used in an outer scope, the new local declaration *shadows* the outer one. Within the inner function scope, all references to the shadowed identifier will be resolved to the new local binding. The Styx core library could keep track of variable and function declarations across all scopes and assign a unique name to each identifier. Currently, the caller of the library must manage variable declarations itself to detect shadowing in nested scopes. Unique identifier names would simplify disambiguation of bindings in the control flow graph.

Another area that might become increasingly interesting in the future is support for ECMAScript 2015 [**?**] language constructs. When work on the Styx core library started, ECMAScript 5.1 [**?**] was the most recent language version standardized by Ecma International and thus became the targeted version for control flow analysis. New language features, such as default values or destructuring, must be supported in order to reflect program semantics correctly. This would require rewriting and extending the Styx core library. Esprima [**?**], the JavaScript parser, and ESTree [**?**], the specification of the abstract syntax tree format, already support ECMAScript 2015.

Finally, the Styx core library could be extended to deal with implicitly thrown errors as well. The control flow graphs currently only contain edges for explicitly thrown exceptions, as is discussed in section 3.2 ("Implicitly Thrown Errors"). Some operations might throw errors in certain cases, for instance when attempting to access a property of an object that is `null`. In contrast, other operations never throw an error, e.g. negating an existing value using the unary `!` operator. For some data flow analyses, it might be helpful to explicitly add all possible error edges to the control flow graph. It would then be possible, for example, to analyze whether a function is guaranteed to never result in an error (apart from unforeseeable runtime errors like program termination due to lack of memory). In general, the Styx core library could be extended such that it annotates the control flow graphs derived with additional attributes (if that data is available) to enable further data flow analyses.

7 References

8 Appendix A

8.1 Example Program

In the following sections, a simple JavaScript program is shown together with its control flow graph as derived by the Styx core library. Additionally, exports in both JSON and DOT format are presented, as is a rendering of the control flow graph. Finally, the program's abstract syntax tree as generated by Esprima is shown.

8.1.1 Source Code

The program below implements a function that returns the Collatz sequence for a given natural number n. The `collatzSequence` function has been deliberately kept simple for the sake of brevity. Therefore, it does not perform extensive argument validation, for example.

```
1  function collatzSequence(n) {
2      var sequence = [n];
3
4      if (n <= 0) {
5          return [];
6      }
7
8      while (n !== 1) {
9          if (n % 2 === 0) {
10             n = n / 2;
11         } else {
12             n = 3 * n + 1;
13         }
14
15         sequence.push(n);
16     }
17
18     return sequence;
19 }
```

8.1.2 DOT Export

When the control flow graph of the `collatzSequence` function is exported in DOT format, the Styx core library returns the following graph description:

```
1  // collatzSequence
2  digraph control_flow_graph {
3      node [shape = doublecircle] 6
4      node [shape = circle]
5
6      // Unconditional edges
7       1 -> 2  [label = "n = $$params[0]"]
8       2 -> 3  [label = "sequence = [n]"]
9       4 -> 6  [label = "return []"]
10      7 -> 6  [label = "return sequence"]
11      9 -> 11 [label = "n = n / 2"]
12     10 -> 11 [label = "n = (3 * n) + 1"]
13     11 -> 5  [label = "sequence.push(n)"]
14
15     // Conditional edges
16     edge [color = red, fontcolor = red]
17     3 -> 4  [label = "n <= 0"]
18     3 -> 5  [label = "!(n <= 0)"]
19     5 -> 8  [label = "n !== 1"]
20     5 -> 7  [label = "n === 1"]
21     8 -> 9  [label = "(n % 2) === 0"]
22     8 -> 10 [label = "(n % 2) !== 0"]
23 }
```

8.1.3 Graphviz Rendering

Using the *dot* renderer that is part of the Graphviz [?] visualization suite, a rendering of the control flow graph described above can look as follows:

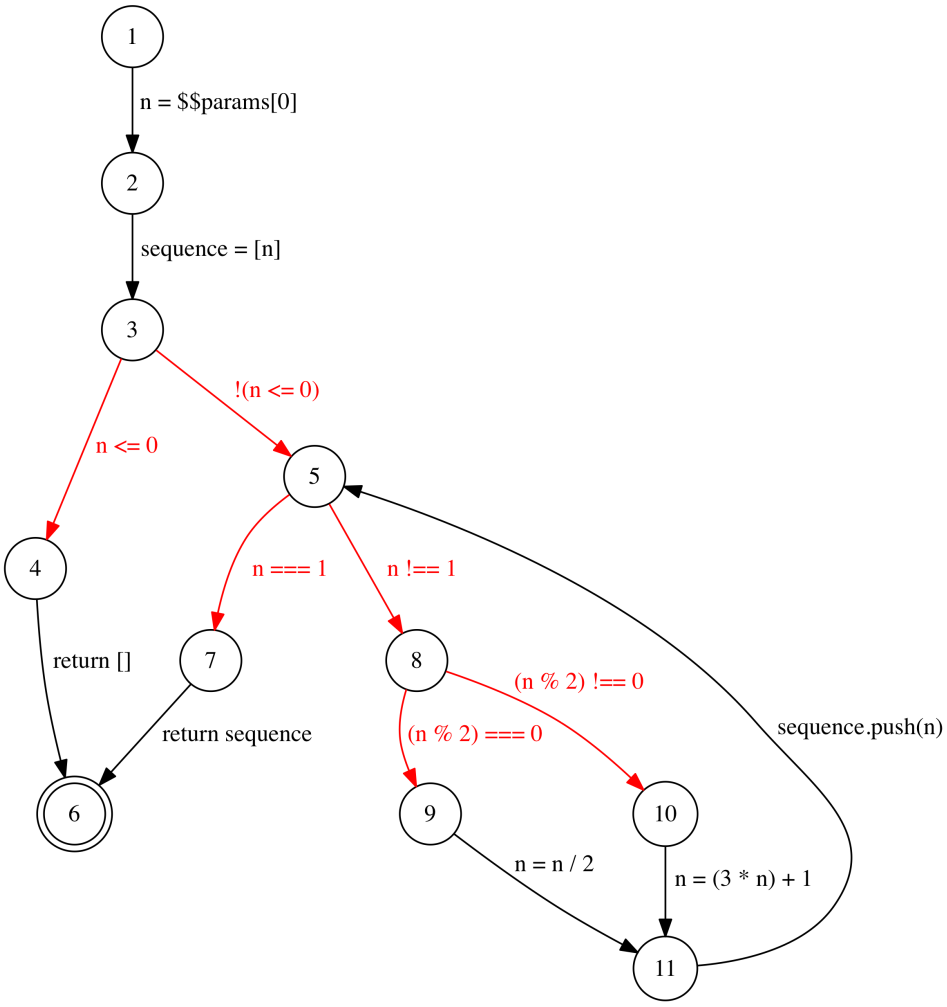


Figure 5: The control flow graph of the `collatzSequence` function

8.1.4 JSON Export

Exporting the control flow graph as JSON yields the following result:

```
{
  "program": {
    "flowGraph": {
      "nodes": [
        {
          "id": 1,
          "type": "Entry"
        },
        {
          "id": 2,
          "type": "SuccessExit"
        }
      ],
      "edges": [
        {
          "from": 1,
          "to": 2,
          "type": "Epsilon",
          "label": "",
          "data": null
        }
      ]
    }
  },
  "functions": [
    {
      "id": 1,
      "name": "collatzSequence",
      "flowGraph": {
        "nodes": [
          {
            "id": 4,
            "type": "Entry"
          },
          {
            "id": 5,
            "type": "SuccessExit"
          },
          {
            "id": 7,
            "type": "Normal"
          },
          {
```

```
            "id": 8,
            "type": "Normal"
          },
          {
            "id": 9,
            "type": "Normal"
          },
          {
            "id": 10,
            "type": "Normal"
          },
          {
            "id": 11,
            "type": "Normal"
          },
          {
            "id": 13,
            "type": "Normal"
          },
          {
            "id": 15,
            "type": "Normal"
          }
        ],
        "edges": [
          {
            "from": 4,
            "to": 7,
            "type": "Normal",
            "label": "n = $$params[0]",
            "data": {
              "type": "AssignmentExpression",
              "operator": "=",
              "left": {
                "type": "Identifier",
                "name": "n"
              },
              "right": {
                "type": "MemberExpression",
                "computed": true,
                "object": {
                  "type": "Identifier",
                  "name": "$$params"
                },
                "property": {
                  "type": "Literal",
```

```
                    "raw": "0",
                    "value": 0
                  }
                }
              }
            },
            {
              "from": 7,
              "to": 8,
              "type": "Normal",
              "label": "sequence = [n]",
              "data": {
                "type": "VariableDeclarator",
                "id": {
                  "type": "Identifier",
                  "name": "sequence"
                },
                "init": {
                  "type": "ArrayExpression",
                  "elements": [
                    {
                      "type": "Identifier",
                      "name": "n"
                    }
                  ]
                }
              }
            },
            {
              "from": 8,
              "to": 9,
              "type": "Conditional",
              "label": "n !== 1",
              "data": {
                "type": "BinaryExpression",
                "operator": "!==",
                "left": {
                  "type": "Identifier",
                  "name": "n"
                },
                "right": {
                  "type": "Literal",
                  "value": 1,
                  "raw": "1"
                }
              }
```

```
            },
            {
              "from": 8,
              "to": 10,
              "type": "Conditional",
              "label": "n === 1",
              "data": {
                "type": "BinaryExpression",
                "operator": "===",
                "left": {
                  "type": "Identifier",
                  "name": "n"
                },
                "right": {
                  "type": "Literal",
                  "value": 1,
                  "raw": "1"
                }
              }
            },
            {
              "from": 9,
              "to": 11,
              "type": "Conditional",
              "label": "(n % 2) === 0",
              "data": {
                "type": "BinaryExpression",
                "operator": "===",
                "left": {
                  "type": "BinaryExpression",
                  "operator": "%",
                  "left": {
                    "type": "Identifier",
                    "name": "n"
                  },
                  "right": {
                    "type": "Literal",
                    "value": 2,
                    "raw": "2"
                  }
                },
                "right": {
                  "type": "Literal",
                  "value": 0,
                  "raw": "0"
                }
```

```
      }
    },
    {
      "from": 9,
      "to": 13,
      "type": "Conditional",
      "label": "(n % 2) !== 0",
      "data": {
        "type": "BinaryExpression",
        "operator": "!==",
        "left": {
          "type": "BinaryExpression",
          "operator": "%",
          "left": {
            "type": "Identifier",
            "name": "n"
          },
          "right": {
            "type": "Literal",
            "value": 2,
            "raw": "2"
          }
        },
        "right": {
          "type": "Literal",
          "value": 0,
          "raw": "0"
        }
      }
    },
    {
      "from": 10,
      "to": 5,
      "type": "AbruptCompletion",
      "label": "return sequence",
      "data": {
        "type": "ReturnStatement",
        "argument": {
          "type": "Identifier",
          "name": "sequence"
        }
      }
    },
    {
      "from": 11,
      "to": 15,
```

```
            "type": "Normal",
            "label": "n = n / 2",
            "data": {
              "type": "AssignmentExpression",
              "operator": "=",
              "left": {
                "type": "Identifier",
                "name": "n"
              },
              "right": {
                "type": "BinaryExpression",
                "operator": "/",
                "left": {
                  "type": "Identifier",
                  "name": "n"
                },
                "right": {
                  "type": "Literal",
                  "value": 2,
                  "raw": "2"
                }
              }
            }
          },
          {
            "from": 13,
            "to": 15,
            "type": "Normal",
            "label": "n = (3 * n) + 1",
            "data": {
              "type": "AssignmentExpression",
              "operator": "=",
              "left": {
                "type": "Identifier",
                "name": "n"
              },
              "right": {
                "type": "BinaryExpression",
                "operator": "+",
                "left": {
                  "type": "BinaryExpression",
                  "operator": "*",
                  "left": {
                    "type": "Literal",
                    "value": 3,
                    "raw": "3"
```

```
                    },
                    "right": {
                      "type": "Identifier",
                      "name": "n"
                    }
                  },
                  "right": {
                    "type": "Literal",
                    "value": 1,
                    "raw": "1"
                  }
                }
              }
            },
            {
              "from": 15,
              "to": 8,
              "type": "Normal",
              "label": "sequence.push(n)",
              "data": {
                "type": "CallExpression",
                "callee": {
                  "type": "MemberExpression",
                  "computed": false,
                  "object": {
                    "type": "Identifier",
                    "name": "sequence"
                  },
                  "property": {
                    "type": "Identifier",
                    "name": "push"
                  }
                },
                "arguments": [
                  {
                    "type": "Identifier",
                    "name": "n"
                  }
                ]
              }
            }
          ]
        }
      }
    ]
}
```

8.1.5 Abstract Syntax Tree

Shown below is the program's abstract syntax tree as generated by Esprima:


```
{
  "type": "Program",
  "body": [
    {
      "type": "FunctionDeclaration",
      "id": {
        "type": "Identifier",
        "name": "collatzSequence"
      },
      "params": [
        {
          "type": "Identifier",
          "name": "n"
        }
      ],
      "defaults": [],
      "body": {
        "type": "BlockStatement",
        "body": [
          {
            "type": "VariableDeclaration",
            "declarations": [
              {
                "type": "VariableDeclarator",
                "id": {
                  "type": "Identifier",
                  "name": "sequence"
                },
                "init": {
                  "type": "ArrayExpression",
                  "elements": [
                    {
                      "type": "Identifier",
                      "name": "n"
                    }
                  ]
                }
              }
            ],
            "kind": "var"
          },
          {
```

```
            "type": "IfStatement",
            "test": {
              "type": "BinaryExpression",
              "operator": "<=",
              "left": {
                "type": "Identifier",
                "name": "n"
              },
              "right": {
                "type": "Literal",
                "value": 0,
                "raw": "0"
              }
            },
            "consequent": {
              "type": "BlockStatement",
              "body": [
                {
                  "type": "ReturnStatement",
                  "argument": {
                    "type": "ArrayExpression",
                    "elements": []
                  }
                }
              ]
            },
            "alternate": null
          },
          {
            "type": "WhileStatement",
            "test": {
              "type": "BinaryExpression",
              "operator": "!==",
              "left": {
                "type": "Identifier",
                "name": "n"
              },
              "right": {
                "type": "Literal",
                "value": 1,
                "raw": "1"
              }
            },
            "body": {
              "type": "BlockStatement",
```

```
                "body": [
                  {
                    "type": "IfStatement",
                    "test": {
                      "type": "BinaryExpression",
                      "operator": "===",
                      "left": {
                        "type": "BinaryExpression",
                        "operator": "%",
                        "left": {
                          "type": "Identifier",
                          "name": "n"
                        },
                        "right": {
                          "type": "Literal",
                          "value": 2,
                          "raw": "2"
                        }
                      },
                      "right": {
                        "type": "Literal",
                        "value": 0,
                        "raw": "0"
                      }
                    },
                    "consequent": {
                      "type": "BlockStatement",
                      "body": [
                        {
                          "type": "ExpressionStatement",
                          "expression": {
                            "type": "AssignmentExpression",
                            "operator": "=",
                            "left": {
                              "type": "Identifier",
                              "name": "n"
                            },
                            "right": {
                              "type": "BinaryExpression",
                              "operator": "/",
                              "left": {
                                "type": "Identifier",
                                "name": "n"
                              },
                              "right": {
```

```
                        "type": "Literal",
                        "value": 2,
                        "raw": "2"
                      }
                    }
                  }
                }
              ]
            },
            "alternate": {
              "type": "BlockStatement",
              "body": [
                {
                  "type": "ExpressionStatement",
                  "expression": {
                    "type": "AssignmentExpression",
                    "operator": "=",
                    "left": {
                      "type": "Identifier",
                      "name": "n"
                    },
                    "right": {
                      "type": "BinaryExpression",
                      "operator": "+",
                      "left": {
                        "type": "BinaryExpression",
                        "operator": "*",
                        "left": {
                          "type": "Literal",
                          "value": 3,
                          "raw": "3"
                        },
                        "right": {
                          "type": "Identifier",
                          "name": "n"
                        }
                      },
                      "right": {
                        "type": "Literal",
                        "value": 1,
                        "raw": "1"
                      }
                    }
                  }
                }
```

```
                    ]
                  }
                },
                {
                  "type": "ExpressionStatement",
                  "expression": {
                    "type": "CallExpression",
                    "callee": {
                      "type": "MemberExpression",
                      "computed": false,
                      "object": {
                        "type": "Identifier",
                        "name": "sequence"
                      },
                      "property": {
                        "type": "Identifier",
                        "name": "push"
                      }
                    },
                    "arguments": [
                      {
                        "type": "Identifier",
                        "name": "n"
                      }
                    ]
                  }
                }
              ]
            }
          },
          {
            "type": "ReturnStatement",
            "argument": {
              "type": "Identifier",
              "name": "sequence"
            }
          }
        ]
      },
      "rest": null,
      "generator": false,
      "expression": false
    }
  ]
}
```