DRAFT: 2020-04-20 11:49:10-04:00

YOCTO-CFA

by
Leandro Facchinetti

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
August 2020

Contents

Chapter 1

Developing an Analyzer

1.1 The Analyzed Language: Yocto-JavaScript

Our first decision when developing an analyzer is which language it should analyze. In this dissertation we are interested in analysis techniques for higher-order functions, a feature which is supported by most languages, including JavaScript, Java, Python, Ruby, and so forth.

From all these options, we would like to choose JavaScript because it is the most popular language among programmers [30, 17], but JavaScript has many features besides higher-order functions that would complicate our analyzer, so we support only a *subset* of JavaScript features that are related to higher-order functions, resulting in a language that we call *Yocto-JavaScript* (JavaScript $\times 10^{-24}$). By design, every Yocto-JavaScript program is also a JavaScript program, but the converse does not hold.

Advanced

On the surface the choice of analyzed language is important because it determines how difficult the analyzer is to develop, but the analyzed language may also influence the analyzer's precision and running time. For example, there

is an analysis technique called k-CFA [33] that may be slower when applied to a language with higher-order functions than when applied to a language with objects, because the algorithmic complexity of the former is exponential and of the latter is polynomial [24].

Technical Terms

Yocto-JavaScript is a representation of something called the *λ-calculus* [34, § 6].

1.1.1 Values in Yocto-JavaScript

JavaScript has many kinds of values: strings (for example, `"Leandro"`), numbers (for example, `29`), arrays (for example, `["Leandro", 29]`), objects (for example, `{ name: "Leandro", age: 29 }`), and so forth. From all these kinds of values, Yocto-JavaScript supports only one: functions.

An Yocto-JavaScript function is written as `<parameter> => <body>`, for example, `x => x`, in which the `<parameter>` is called `x` and the `<body>` is a reference to the variable `x` (see § 1.1.2 for more on variable references). An Yocto-JavaScript function must have exactly one parameter. Because an Yocto-JavaScript function is a value, it may be passed as argument in a function call or returned as the result of a function call (see § 1.1.2 for more on function calls).

Technical Terms

The notation we use for writing functions is something called *arrow function expressions* [25]. The function given as example is called the *identity* function. The ability of acting as values is what characterizes these functions as *higher-order*.

1.1.2 Operations in Yocto-JavaScript

JavaScript has many operations: strings may have its characters accessed (for example, `"Leandro"[2]`, which results in `"a"`), numbers may be added together (for example, `29 + 1`, which results in `30`), and so forth. From all these operations, Yocto-JavaScript supports only two: functions may be called and variables may be referenced.

A function call is written as `<function>(<argument>)`, for example, `f(a)`, in which the `<function>` is a hypothetical function `f` and the `<argument>` is a hypothetical argument `a`. An Yocto-JavaScript function call must have exactly one argument (because an Yocto-JavaScript function must have exactly one parameter; see § 1.1.1). A variable reference is written as a bare identifier, for example, `x`.

The following is a complete Yocto-JavaScript program that exemplifies all the supported operations:

```
(x => x)(y => y)
```

This program is a function call in which the `<function>` is `x => x` and the `<argument>` is `y => y`. When called, an Yocto-JavaScript function returns the result of computing its `<body>` and the `<body>` of `x => x` is a reference to the variable `x`, so `x => x` is a function that returns its argument unchanged and the final result of the example above is `y => y`.

In general, all kinds of Yocto-JavaScript expressions (function definitions, function calls, and variable references) may appear in the `<body>` of a function definition, or as the `<function>` or `<argument>` of a call; for example, in the program `(f(a))(b)` the function call `f(a)` appears as the `<function>` of a call.

We use parentheses to resolve ambiguities on where function definitions start and end, and in which order operations are computed. For example, given hypothetical functions `f`, `g`, and `h`, in `(f(g))(h)` the call `f(g)` happens first and the

result is a function that is called with `h`, and in `f(g(h))` the call `g(h)` happens first and the result is passed as argument to `f`. If there are no parentheses, then nested function definitions are read right-to-left and a sequence of function calls are read left-to-right; for example, `x => y => x` is equivalent to `x => (y => x)` and `f(a)(b)` is equivalent to `(f(a))(b)`.

Technical Terms

The order in which operations are computed is something called their *precedence*, and operations that happen first are said to have *higher precedence*. Because of the order in which they are read, function definitions are said to be *right-associative* and function calls are said to be *left-associative*.

Advanced

1.1.3 The Computational Power of Yocto-JavaScript

Yocto-JavaScript has only a few features, which makes it the ideal language for discussing the analysis of higher-order functions, but is it *too* simple? In other words, in the process of pairing down JavaScript to define Yocto-JavaScript, have we removed features that make the language incapable of some computations? Perhaps surprisingly, the answer is negative: Yocto-JavaScript is equivalent to JavaScript (and Java, Python, Ruby, and so forth) in the sense that, with some effort, any program in any one of these languages may be translated into an equivalent program in any other of these languages [34, § 6].

As an example of how to carry out this translation, consider a JavaScript function of two parameters: `(x, y) => x`. This function is not supported by Yocto-JavaScript because it does not have exactly one parameter (see § 1.1.1), but we may encode it as a function that receives the first parameter and returns another function that receives the second parameter: `x => (y => x)`.

Similarly, we may encode a call with multiple arguments as a sequence of calls that passes one argument at a time; for example, `f(a, b)` may be encoded as `(f(a))(b)`.

Technical Terms

All the languages we are considering are said to be equivalent in terms of *computational power*: they are all *Turing complete* [34, § 7]. The translation technique for functions with multiple arguments is called *currying* [34, page 163].

For our goal of exploring analysis techniques, we are concerned only with computational power, but it is worth noting that programmers are more interested in other language properties: Does the language promote writing programs of higher quality? (It most probably does not [10].) Does the language improve productivity? Does the language work well for the domain of the problem? (For example, we would probably write an operating system in C and a web application in JavaScript, not the other way around.) Is the language more expressive than others? (Perhaps surprisingly, it is possible to make formal arguments about expressiveness without resorting to personal preference and anecdotal evidence [13].) Despite having the same computational power as other languages, Yocto-JavaScript fares badly in these other aspects: it is remarkably unproductive and inexpressive.

1.1.4 A Formal Grammar for Yocto-JavaScript

The description of Yocto-JavaScript given so far has been informal; the following is a grammar in *Backus–Naur Form* (BNF) [23] [9, § 4.2] that formalizes it:

```
e  ::=  (x => e) | e(e) | x                Expressions
x  ::=  <A JavaScript Identifier>          Variables
```

1.2 The Analyzer Language: TypeScript

After choosing our analyzed language (Yocto-JavaScript; see § 1.1), we must decide in which language to develop the analyzer itself. Despite our analyzed language being based on JavaScript, we may choose to develop the analyzer in any language (for example, JavaScript, Java, Python, Ruby, and so forth), because the analyzer treats the analyzed program as data. Still, from all these options, JavaScript does offer some advantages: it is the most popular [30, 17], and it includes convenient tools to manipulate JavaScript programs (and therefore Yocto-JavaScript programs as well; see § 1.3.15 and § 1.3.16). But JavaScript lacks a way to express the *types* of data structures, functions, and so forth, which we will need (for example, see § 1.3.2), so we choose to implement our analyzer in a JavaScript extension with support for types called *TypeScript* [7, 35, 11].

1.3 Step 0: Substitution-Based Interpreter

Having chosen the analyzed language (Yocto-JavaScript; see § 1.1) and the language in which to develop the analyzer itself (TypeScript; see § 1.2), we are ready to start the series of Steps in the development of the analyzer. The first Step is an interpreter that executes Yocto-JavaScript programs and produces the same outputs that would be produced by a regular JavaScript interpreter. This is a good starting point for two reasons: first, this interpreter is the basis upon which we will build the analyzer; and second, the outputs of this interpreter are the ground truth against which we will validate the outputs of the analyzer.

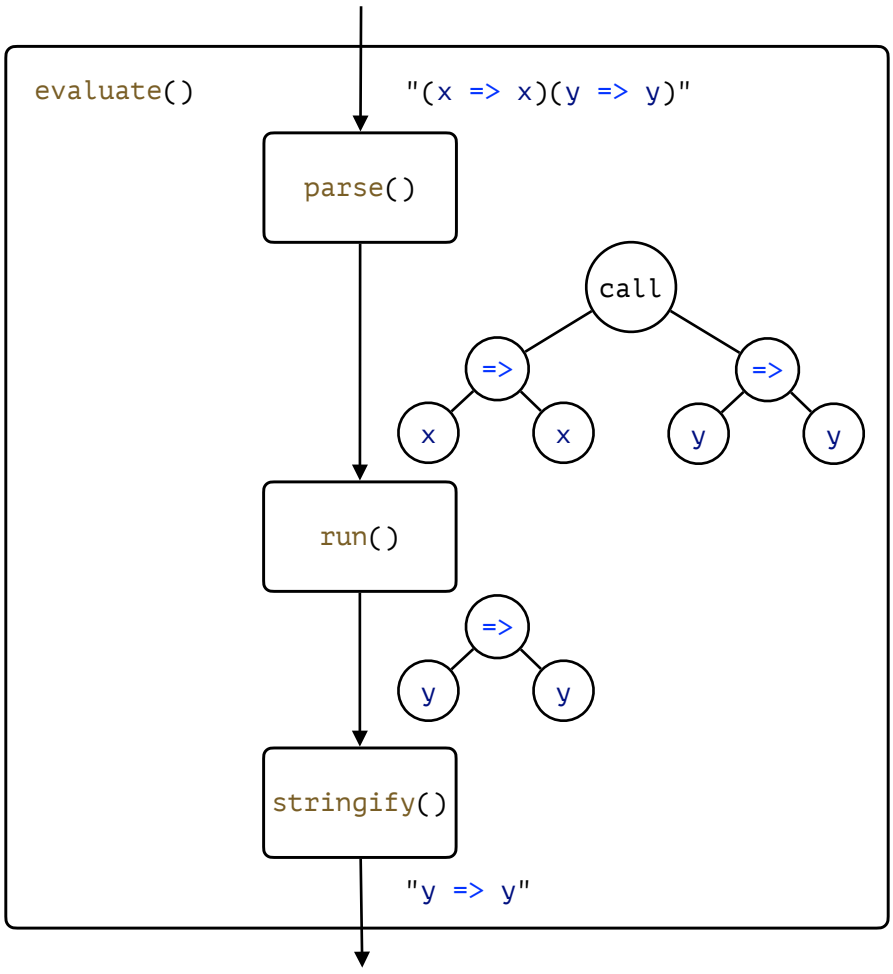
1.3.1 Architecture

Our interpreter is defined as a function called `evaluate()`, which receives as parameter an Yocto-JavaScript program represented as a string and returns the result of running it.

The following are two examples of how we will be able to use `evaluate()` by the end of Step 0 (the > represents the console):

```
> evaluate("x => x")
"x => x"
> evaluate("(x => x)(y => y)")
"y => y"
```

The implementation of `evaluate()` is separated into three parts called `parse()`, `run()`, and `stringify()`:



```
export function evaluate(input: string): string {
  return stringify(run(parse(input)));
}
```

The `parse()` function prepares the `input` for interpretation, converting it from a string into more convenient data structures (see § 1.3.2 for more on these data structures). The `run()` function is responsible for the interpretation itself. The `stringify()` function converts the outputs of `run()` into a human-readable format. In the following sections (§ 1.3.2–§ 1.3.14) we address the implementation of `run()`, deferring `parse()` to § 1.3.15 and `stringify()` to § 1.3.16.

In later Steps the implementations of `run()` and `stringify()` will change, but the architecture and therefore the implementations of `evaluate()` and `parse()` will remain the same.

Advanced

The `evaluate()` function is named after a native JavaScript function called `eval()` [28], which is similar to `evaluate()` but for JavaScript programs instead of Yocto-JavaScript. The `strinfigy()` function is named after a native JavaScript function called `JSON.stringify()` [29], which is used in the implementation (see § 1.4.11).

1.3.2 Data Structures to Represent Yocto-JavaScript Programs

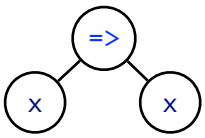
The `evaluate()` function receives an Yocto-JavaScript program represented as a string (see § 1.3.1), which is convenient for humans to write and read, but inconvenient for `run()` to manipulate directly, because `run()` is concerned with the *structure* of the program instead of the *text*: from `run()`'s perspective it does not matter, for example, whether a function is written as `x => x` or as `x=>x`. So before `run()` starts interpreting the program, `parse()` transforms it from a string into more convenient data structures (see § 1.3.15 for `parse()`'s implementation).

Technical Terms

The process of converting a program represented as a string into more convenient data structures is known as *parsing*, and the data structures are called the *Abstract Syntax Tree* (AST) of the program [9, § 4].

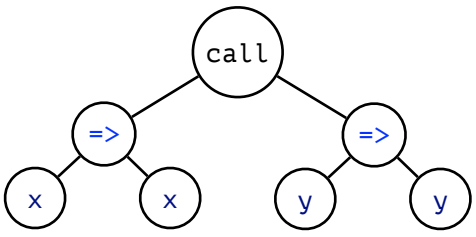
The following are two examples of Yocto-JavaScript programs followed by the data structures used to represent them, first in a high-level graphical representation and then in an equivalent low-level textual representation:

```
> parse("x => x")
```




```
{
  "type": "ArrowFunctionExpression",
  "params": [
    {
      "type": "Identifier",
      "name": "x"
    }
  ],
  "body": {
    "type": "Identifier",
    "name": "x"
  }
}
```

```
> parse("(x => x)(y => y)")
```




```
{
  "type": "CallExpression",
  "callee": {
```

```
    "type": "ArrowFunctionExpression",
    "params": [
      {
        "type": "Identifier",
        "name": "x"
      }
    ],
    "body": {
      "type": "Identifier",
      "name": "x"
    }
  },
  "arguments": [
    {
      "type": "ArrowFunctionExpression",
      "params": [
        {
          "type": "Identifier",
          "name": "y"
        }
      ],
      "body": {
        "type": "Identifier",
        "name": "y"
      }
    }
  ]
}
```

We choose to represent Yocto-JavaScript programs with the data structures above because they follow a specification called ESTree [6], and by adhering to this specification we may reuse tools from the JavaScript ecosystem (see § 1.3.15 and § 1.3.16).

In general, the data structures used to represent Yocto-JavaScript programs are of the following types (written as TypeScript types adapted from the ESTree types [8] to include only the features supported by Yocto-JavaScript):

```
type Expression = ArrowFunctionExpression | CallExpression | Identifier;

type ArrowFunctionExpression = {
  type: "ArrowFunctionExpression";
```

```
  params: [Identifier];
  body: Expression;
};

type CallExpression = {
  type: "CallExpression";
  callee: Expression;
  arguments: [Expression];
};

type Identifier = {
  type: "Identifier";
  name: string;
};
```

> **Advanced**
>
> The definitions above correspond to elements of the Yocto-JavaScript grammar (see § 1.1.4); for example, `Expression` corresponds to e.

In later Steps almost everything about the interpreter will change, but the data structures used to represent Yocto-JavaScript programs will remain the same.

1.3.3 An Expression That Already Is a Value

Example Program	`x => x`
Current Output	—
Expected Output	`x => x`

We start the definition of `run()` by considering the example above. As mentioned in § 1.3.2, the `run()` function receives as parameter an Yocto-JavaScript program represented as an `Expression`. The `run()` function is then responsible for interpreting the program and producing a value. In Yocto-JavaScript, the only kind of value is a function (see § 1.1.1), so we start the implementation of `run()` with the following (we use `throw` as a placeholder for code that has not be written yet to prevent the TypeScript compiler from signaling type errors):

```
type Value = ArrowFunctionExpression;

function run(expression: Expression): Value {
  throw new Error("NOT IMPLEMENTED YET");
}
```

The first thing that `run()` has to do is determine which type of `expression` it is given:

```
function run(expression: Expression): Value {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "CallExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "Identifier":
      throw new Error("NOT IMPLEMENTED YET");
  }
}
```

In our current example, the `expression` already is a `Value`, so it may be returned unchanged:

```
// run()
case "ArrowFunctionExpression":
  return expression;
```

1.3.4 A Call Involving Immediate Functions

Example Program	`(x => x)(y => y)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`y => y`

Interpreting function calls is the main responsibility of our interpreter. There are several techniques to do this and in Step 0 we use one of the simplest: when the interpreter encounters a function call, it substitutes the variable references in the body of the function that is called with the argument that is passed. This is similar to how we reason about functions in mathematics; for example, given the

function $f(x) = x+1$, we calculate $f(29)$ by substituting the references to x in f with the argument 29: $f(29) = 29+1$. The implementation of this substitution technique starts in this section and will only be complete in § 1.3.8.

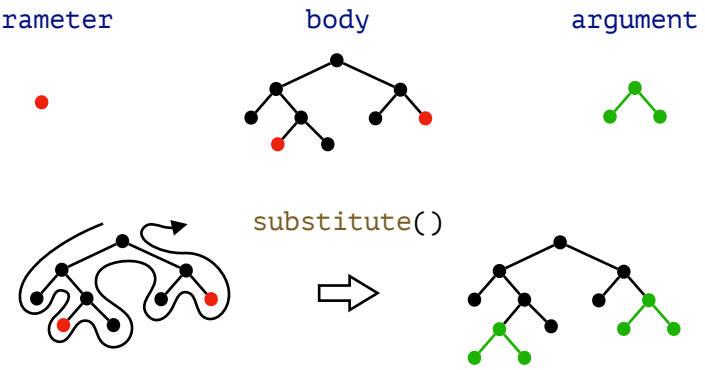
In the example we are considering both the function that is called (`x => x`) and the argument (`y => y`) are immediate functions, as opposed to being the result of other operations, so for now we may limit the interpreter to handle only this case:

```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  throw new Error("NOT IMPLEMENTED YET");
```

Next, we unpack the called function (using something called *destructuring assignment* [26]) and the argument:

```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = expression.arguments[0];
  throw new Error("NOT IMPLEMENTED YET");
```

Finally, we setup an auxiliary function called `substitute()` that implements the traversal of the `body` looking for references to `parameter` and substituting them:



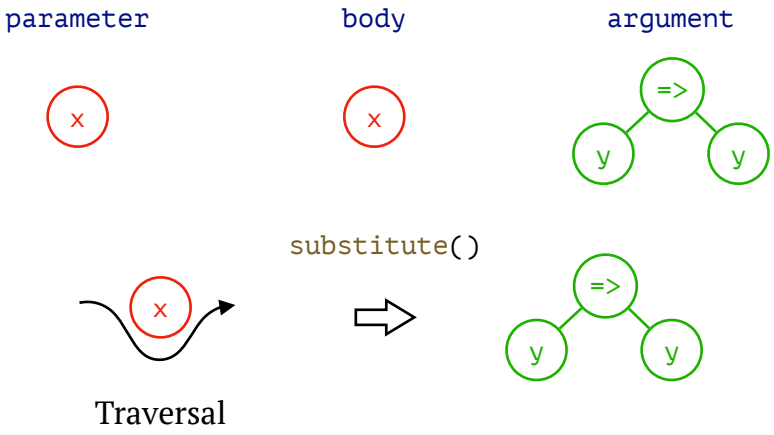
Traversal

```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = expression.arguments[0];
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    throw new Error("NOT IMPLEMENTED YET");
  }
```

Similar to `run()` itself, `substitute()` starts by determining which type of `expression` is passed to it:

```
function substitute(expression: Expression): Expression {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "CallExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "Identifier":
      throw new Error("NOT IMPLEMENTED YET");
  }
}
```

In our current example the `expression` is `x`, an `Identifier`, and it must be substituted with the `argument`:



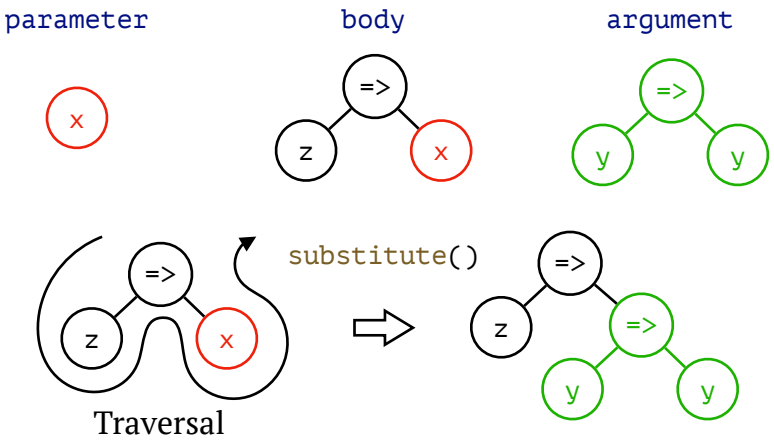
Traversal

```
// substitute()
case "Identifier":
  return argument;
```

1.3.5 Substitution in Function Definitions

Example Program	`(x => z => x)(y => y)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`z => y => y`

When `substitute()` (see § 1.3.4) starts traversing the `body` of the example above, the `expression` is an `ArrowFunctionExpression` (`z => x`), and we want substitution to proceed deeper to find and substitute `x`, so we call `substitute()` recursively (we use a feature called *spread syntax* [27] to build an `expression` based on the existing one with a new `body`):




```
// substitute()
case "ArrowFunctionExpression":
  return {
    ...expression,
    body: substitute(expression.body),
  };
```

1.3.6 Name Mismatch

Example Program `(x => z => z)(y => y)`

Current Output `z => y => y`

Expected Output `z => z`

The implementation of `substitute()` in the case of `Identifier` introduced in § 1.3.4 *always* substitutes variable references, regardless of whether they refer to the `parameter` of the called function. For example, in the program above `substitute()` is substituting the `z` even though the `parameter` is `x`. To fix this, we check whether the variable reference matches the `parameter`, and if it does not then we prevent the substitution by retuning the variable reference unchanged:

```
// substitute()
case "Identifier":
  if (expression.name !== parameter.name) return expression;
  return argument;
```

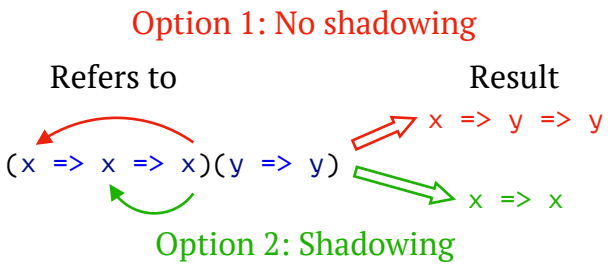
1.3.7 Name Reuse

Example Program `(x => x => x)(y => y)`

Current Output `x => y => y`

Expected Output `x => x`

In the program above, there are two options for the variable reference `x` on the right of the second `=>`: it may refer to the first (outer) `x` on the left of the first `=>`, in which case the output of the program would be `x => y => y`; or it may refer to the second (inner) `x` on the left of the second `=>`, in which case the output of the program would be `x => x`:



Currently `substitute()` is implementing Option 1, but this leads to an issue: we are not able to reason about the inner function `x => x` independently; we must know where it appears and whether a variable called `x` is already defined there.

> **Technical Terms**
>
> We say that the problem with Option 1 is that it defeats something called *local reasoning*. We say that Option 2 exhibits a behavior called *shadowing*, and that the outer `x` is *shadowed* by the inner `x`, because there is no way to refer to the outer `x` from the body of the inner function.

We avoid this issue by modifying `substitute()` to implement Option 2, which is also the choice of JavaScript and every other popular programming language. We change `substitute()`'s behavior when encountering a function definition so that if the parameter of the function definition matches the param-

eter that `subsitute()` is looking for, then `subsitute()` returns the function unchanged, preventing further substitution (there is no recursive call to `substitute()` in this case):

```
// substitute()
case "ArrowFunctionExpression":
  if (expression.params[0].name === parameter.name) return expression;
  return {
    ...expression,
    body: substitute(expression.body),
  };
```

1.3.8 Substitution in Function Calls

Example Program	`(x => z => x(x))(y => y)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`z => (y => y)(y => y)`

This case is similar to § 1.3.5: all `substitute()` has to do is continue traversing the function call recursively:

```
// substitute()
case "CallExpression":
  return {
    ...expression,
    callee: substitute(expression.callee),
    arguments: [substitute(expression.arguments[0])],
  };
```

1.3.9 An Argument That Is Not Immediate

Example Program	`(x => z => x)((a => a)(y => y))`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`z => y => y`

In all example programs we considered so far the argument to a function call is an immediate function definition, but in general arguments may be the result of

function calls themselves. We fix this by calling `run()` recursively on the argument (we also remove the check that the argument is an immediate function definition; if it is, then the recursive call to `run()` returns the immediate function unchanged; see § 1.3.3):

```
// run()
case "CallExpression":
  if (expression.callee.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = run(expression.arguments[0]);
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    // ...
  }
```

Technical Terms

This technique of calling `run()` recursively to produce an immediate function for the argument characterizes the interpreter as *big-step*.

Advanced

The notion that the argument is interpreted to produce a value as soon as the function call is encountered characterizes Yocto-JavaScript as a *call-by-value* language [31]. JavaScript itself and most other popular programming languages are call-by-value as well, but there is a notable exception, Haskell, which is a *call-by-need* language. In a call-by-need language the argument is interpreted only if it is *needed*, for example, if it is used in the function position of another call (see § 1.3.10), or if it is the result of the program (see § 1.3.11). In a call-by-need language the result of the program above would

be `z => ((a => a)(y => y))`. And there is yet another policy for when to interpret arguments called *call-by-name*: the difference between call-by-name and call-by-need is that in a call-by-name language the an argument may be computed multiple times if it is used multiple times, but in a call-by-need language an argument is guaranteed to be computed at most once.

1.3.10 A Function That Is Not Immediate

Example Program `((z => z)(x => x))(y => y)`

Current Output `NOT IMPLEMENTED YET`

Expected Output `y => y`

This is the dual of § 1.3.9 for the called function, and the solution is the same: to call `run()` recursively (we also remove the check of whether the function is immediate):

```
// run()
case "CallExpression":
  const {
    params: [parameter],
    body,
  } = run(expression.callee);
  const argument = run(expression.arguments[0]);
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    // ...
  }
```

1.3.11 Continuing to Run After a Function Call

Example Program `(x => (z => z)(x))(y => y)`

Current Output `NOT IMPLEMENTED YET`

Expected Output `y => y`

This is similar to § 1.3.9 and § 1.3.10: the result of substitution may be not an immediate function but another call, and more work may be necessary to interpret it. We solve this with yet another recursive call to `evaluate()` (we also remove yet another check and inline the `substitutedBody` variable):

```
// run()
case "CallExpression":
  const {
    params: [parameter],
    body,
  } = run(expression.callee);
  const argument = run(expression.arguments[0]);
  return run(substitute(body));
  function substitute(expression: Expression): Expression {
    // ...
  }
```

1.3.12 A Reference to an Undefined Variable

Example Program `(x => y)(y => y)`

Current Output `NOT IMPLEMENTED YET`

Expected Output `Reference to undefined variable: y`

The only case in which `run()` may encounter a variable reference directly is if the referenced variable is undefined, otherwise `substitute()` would have already substituted it (see § 1.3.4–§ 1.3.11). In this case, we throw an exception:

```
// run()
case "Identifier":
  throw new Error(`Reference to undefined variable:
↪  ${expression.name}`);
```

Example Program `x => y`

Current Output `x => y`

Expected Output `x => y`

If the reference to an undefined variable occurs in the body of a function that is not called, then we do not reach the case addressed in this section and an exception is not thrown. This is consistent with JavaScript's behavior.

1.3.13 The Entire Runner

The implementation of the `run()` function is complete:

```
type Value = ArrowFunctionExpression;

function run(expression: Expression): Value {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      return expression;
    case "CallExpression":
      const {
        params: [parameter],
        body,
      } = run(expression.callee);
      const argument = run(expression.arguments[0]);
      return run(substitute(body));
      function substitute(expression: Expression): Expression {
        switch (expression.type) {
          case "ArrowFunctionExpression":
            if (expression.params[0].name === parameter.name) return
 ↪  expression;
            return {
              ...expression,
              body: substitute(expression.body),
            };
          case "CallExpression":
            return {
              ...expression,
              callee: substitute(expression.callee),
              arguments: [substitute(expression.arguments[0])],
            };
          case "Identifier":
            if (expression.name !== parameter.name) return expression;
            return argument;
        }
      }
    case "Identifier":
      throw new Error(`Reference to undefined variable:
 ↪  ${expression.name}`);
  }
}
```

Advanced

1.3.14 An Operational Semantics for the Interpreter

What we accomplished so far in this section is more than defining an interpreter for Yocto-JavaScript; we also defined formally the *meaning* of Yocto-JavaScript programs: an Yocto-JavaScript program means what the interpreter produces for it. The definition of the meaning of programs in a language is something called the *semantics* of the language, and there are several techniques to specify semantics; the one we are using so far is known as a *definitional interpreter* [32].

A definitional interpreter has some advantages over other techniques for specifying semantics: it is easier to understand for most programmers, and it is executable. But a definitional interpreter also has one disadvantage: to understand the meaning of an Yocto-JavaScript program we have to understand an interpreter written in TypeScript. To address this, there are other techniques for defining semantics that do not depend on other programming languages, and in this section we introduce one of them: *operational semantics* [18, 14, 16].

First, we extend the grammar from § 1.1.4 with the notion of values that is equivalent to the type `Value` (see § 1.3.13, line 1):

$$v \quad ::= \quad x \texttt{ => } e \quad \text{Values}$$

Next, we define a *relation* $e \Rightarrow v$ using *inference rules* that are equivalent to the behavior of `run()` (see § 1.3.13, lines 3–36):

$$\frac{}{v \Rightarrow v} \qquad \frac{e_f \Rightarrow x_p \texttt{ => } e_b \qquad e_a \Rightarrow v_a \qquad e_b[x_p \backslash v_a] \Rightarrow v}{e_f(e_a) \Rightarrow v}$$

Finally, we define a *metafunction* $e[x\backslash v] = e$ that is equivalent to the behavior of `substitute()` (see § 1.3.13, lines 14–32):

$$\begin{aligned}
(x \texttt{ => } e)[x_p \backslash v_a] &= x \texttt{ => } (e[x_p \backslash v_a]) && \text{if } x \neq x_p \\
(x_p \texttt{ => } e)[x_p \backslash v_a] &= x_p \texttt{ => } e \\
(e_f(e_a))[x_p \backslash v_a] &= (e_f[x_p \backslash v_a])((e_a[x_p \backslash v_a])) \\
x[x_p \backslash v_a] &= x && \text{if } x \neq x_p \\
x_p[x_p \backslash v_a] &= v_a
\end{aligned}$$

1.3.15 Parser

The parser is responsible for converting an Yocto-JavaScript program written as a string into data structures that are more convenient for the runner to manipulate (see § 1.3.1 for a high-level view of the architecture and § 1.3.2 for the definition of the data structures). We choose to represent Yocto-JavaScript programs with data structures that are compatible with a specification for representing JavaScript programs called ESTree [6, 8] because it allows us to reuse tools from the JavaScript ecosystem, including a parser called Esprima [3], and the Esprima Interactive Online Demonstration [4], which shows the data structures used to represent a given program.

Our strategy to implement the Yocto-JavaScript parser is to delegate most of the work to Esprima and check that the program is using only features supported by Yocto-JavaScript. The following is the full implementation of the parser:

```
1 function parse(input: string): Expression {
```

```
2   const program = esprima.parseScript(input, { range: true },
    ↪  checkFeatures);
3   const expression = (program as any).body[0].expression as Expression;
4   return expression;
5   function checkFeatures(node: estree.Node): void {
6     switch (node.type) {
7       case "Program":
8         if (node.body.length !== 1)
9           throw new Error(
10            "Unsupported Yocto-JavaScript feature: Program with multiple
    ↪  statements"
11          );
12        break;
13      case "ExpressionStatement":
14        break;
15      case "ArrowFunctionExpression":
16        break;
17      case "CallExpression":
18        if (node.arguments.length !== 1)
19          throw new Error(
20            "Unsupported Yocto-JavaScript feature: CallExpression with
    ↪  multiple arguments"
21          );
22        break;
23      case "Identifier":
24        break;
25      default:
26        throw new Error(`Unsupported Yocto-JavaScript feature:
    ↪  ${node.type}`);
27    }
28  }
29 }
```

Line 1: The parser is defined as a function called `parse()`, which receives the program `input` represented as a `string` and returns an `Expression` (see § 1.3.2).

Line 2: Call `esprima.parseScript()`, which parses the `input` as a JavaScript program and produces a data structure following the ESTree specification. The `esprima.parseScript()` function also detects syntax errors, for example, in the program `x =>`, which is missing the function body.

The `{ range: true }` argument causes Esprima to include in the generated data structures some information about the part of the `input` from where they came. We do not use this information (it is not even part of the definition of the data structures; see § 1.3.2), but in programs with expressions that repeat, for example, `x => x => x`, this information distinguishes the `x`s.

We pass as argument to `esprima.parseScript()` a function called `checkFeatures()` which is called with every fragment of data structure that represents a part of the program. The purpose of `checkFeatures()` is to check that the program uses only the features that are supported by Yocto-JavaScript.

Line 3: Extract the single `Expression` from within the `Program` returned by `esprima.parseScript()`. The `as <something>` forms sidestep the TypeScript type checker and assert that the `expression` is of the correct type. This is safe to do because of `checkFeatures()`.

Line 5: The `checkFeatures()` function, which is passed to `esprima.parseScript()` is called with every fragment of data structure used to represent the program. These fragments are called *nodes*, because the data structure as a whole forms a *tree*, also known as the *Abstract Syntax Tree* (AST) of the program (see § 1.3.2). The `checkFeatures()` does not return anything (`void`); its purpose is only to throw an exception in case the program uses a feature that is not supported by Yocto-JavaScript.

Lines 6, 7, 13, 15, 17, 23, 25: Similar to `run()` and `substitute()` (see § 1.3.13), `checkFeatures()` starts by determining which type of `estree.Node` it is given.

Lines 8–11: Check that the `Program` contains a single statement. This prevents programs such as `x => x; y => y`.

Lines 13, 15: `ExpressionStatement`s and `ArrowFunctionExpression`s are supported in Yocto-JavaScript unconditionally. We could check that the `ArrowFunctionExpression` includes only one parameter and that this parameter is a variable (as opposed to being a pattern such as `[x, y]`, for example), but this would be redundant because Esprima already calls `checkFeatures()` with other unsupported `node`s that subsume these cases. For example, given the program `(x, y) => x`, which is a function of multiple parameters, Esprima calls `checkFeatures()` with a `node` of type `SequenceExpression`. Similarly, given the program `([x, y]) => x`, which is a function in which the parameter is a pattern, Esprima calls `checkFeatures()` with a `node` of type `ArrayExpression`.

Lines 18–21: Check that the `CallExpression` contains a single argument. This prevents programs such as `f(a, b)`.

Line 23: `Identifier`s are supported in Yocto-JavaScript unconditionally. An `Identifier` may be an expression or the parameter of an `ArrowFunctionExpression`.

Line 26: All other types of `estree.Node` are not supported by Yocto-JavaScript. This includes programs such as `29` (`estree.Literal`) and `const f = x => x` (`estree.VariableDeclarator`).

In later Steps almost everything about the interpreter will change, but the parser will remain the same.

1.3.16 Stringifier

The stringifier transforms a `Value` produced by `run()` into a human-readable format (see § 1.3.1 for a high-level view of the architecture). Similar to what happened in the parser (see § 1.3.15), we may implement the stringifier by reusing existing tools from the JavaScript ecosystem, because we are representing Yocto-JavaScript programs and values with data structures that follow the ESTree specification. In particular, we use a library called Escodegen [2] to generate a string representation of an ESTree data structure, and a library called Prettier [5] to format that string. The following is the full implementation of the stringifier:

```
1 function stringify(value: Value): string {
2   return prettier
3     .format(escodegen.generate(value), {
4       parser: "babel",
5       semi: false,
6       arrowParens: "avoid",
7     })
8     .trim();
9 }
```

Line 4: Prettier needs to parse and regenerate the string representing the value, in an architecture similar to that of the interpreter (see § 1.3.1), and it may use different parsers. We choose Babel [1], which is the default parser for JavaScript (Prettier also supports formatting other languages, for example, TypeScript and Markdown).

Line 5: Instruct Prettier not to produce semicolons at the end of the line, for example, `x => y` instead of `x => y;`.

Line 6: Instruct Prettier not to wrap parameters in parentheses, for example, `x => y` instead of `(x) => y`.

Line 8: Remove the newline at the end of the output produced by Prettier.

1.3.17 Programs That Do Not Terminate

Example Program `(f => f(f))(f => f(f))`

Current Output `DOES NOT TERMINATE`

Expected Output `DOES NOT TERMINATE`

Technical Terms

This example program is also known as the Ω-*combinator*. The function `f => f(f)` that is part of this program is also known as the *U-combinator* ($\Omega = (U)(U)$).

Yocto-JavaScript may express any program that a computer may run (see § 1.1.3), including some programs that do not terminate. For example, consider the program above, which is the shortest non-terminating program in Yocto-JavaScript. The following is a trace of the first call to `run()`:

Line (see § 1.3.13)

```
 3        expression  =  (f => f(f))(f => f(f))
 9         parameter  =  f
10              body  =  f(f)
12          argument  =  f => f(f)
13  substitute(body)  =  (f => f(f))(f => f(f))
```

The result of substitution is the same as the initial expression, so when it is passed as argument to the recursive call to `run()` in line 13, it causes `run()` to go into an infinite loop.

Example Program `(f => (f(f))(f(f)))(f => (f(f))(f(f)))`

Current Output `DOES NOT TERMINATE`

Expected Output `DOES NOT TERMINATE`

There are also programs for which interpretation does not terminate that never produce the same expression twice. For example, consider the program above, which is a variation of the first program in which every variable reference to `f` has been replaced with `f(f)`. The following is a trace of the first call to `run()`:

```
Line  (see § 1.3.13)              where F = f => (f(f))(f(f))
   3          expression  =  F(F)
   9           parameter  =  f
  10                body  =  (f(f))(f(f))
  12            argument  =  F
  13  substitute(body)    =  (F(F))(F(F))
```

The result of substitution (`(F(F))(F(F))`) is an expression that contains the initial expression (the first `F(F)`) in addition to some extra work (the second `F(F)`), so when it is passed as argument to the recursive call to `run()` in line 13, it causes `run()` to go into an infinite loop. Unlike what happened in the first example, when interpreting this program the expressions that are passed to `run()` never repeat themselves:

```
(F(F))
(F(F))(F(F))
(F(F))(F(F))(F(F))
(F(F))(F(F))(F(F))(F(F))
                ⋮
```

Non-termination is what we expect from an interpreter, but not from an analyzer, and as the second example demonstrates, preventing non-termination is not as simple as checking whether `run()` is being called with the same expression multiple times. As we move forward from an interpreter to an analyzer in the next Steps one of the main issues we address is termination: even if it takes a long time, an analyzer must eventually finish its work regardless of the program it is given.

Advanced

Detecting non-termination in an interpreter without losing any information about the original program is a problem that cannot be solved, regardless of the sophistication of the detector and the computational power available to it. The problem, which is known as the *halting problem*, is said to be *uncomputable* [34, § 8], and is a direct consequence of the Turing completeness of Yocto-JavaScript (see § 1.1.3). In our analyzer we will guarantee termination by allowing some information to be lost.

1.4 Step 1: Environment-Based Interpreter

The interpreter in Step 0 may not terminate for some programs, and preventing non-termination is one of the main issues we must address when developing an analyzer (see § 1.3.17). The source of non-termination in Step 0 is substitution, which may produce infinitely many new expressions and cause the interpreter to loop forever. In Step 1, we modify the interpreter so that it does not perform substitution, and as a consequence it considers only the finitely many expressions found in the input program. The interpreter in Step 1 may still not terminate, but that is due to other sources of non-termination that will be addressed in subsequent Steps.

1.4.1 Avoiding Substitution by Introducing Environments and Closures

When the interpreter from Step 0 encounters a function call, it produces a new expression by traversing the body of the called function and substituting the references to the parameter with the argument, for example:

Example Program (see § 1.3.5)	`(x => z => x)(y => y)`
Step 0 Output	`z => (y => y)`

The issue with this strategy is that the expression `z => (y => y)` does not exist in the original program, and as mentioned in § 1.3.17, there is a possibility that the interpreter tries to produce infinitely many of these new expressions and loops forever. In Step 1 we want to avoid producing new expressions, so that the interpreter has to consider only the finitely many expressions found in the original program. We accomplish this by interpreting function calls with a different strategy: instead of performing substitution, we maintain a map from variables to the values with which they would have been substituted, for example:

Example Program	`y => y`
Step 1 Output	⟨`(y => y)`, []⟩
Example Program	`(x => z => x)(y => y)`
Step 1 Output	⟨`(z => x)`, [`x` ↦ ⟨`(y => y)`, []⟩]⟩

Technical Terms

A map from variables to the values with which they would have been replaced (for example, [`x` ↦ ⟨`(y => y)`, []⟩]) is something called an *environment*. A function along with an environment (for example, ⟨`(z => x)`, [`x` ↦ ⟨`(y => y)`, []⟩]⟩) is something called a *closure* [19].

In Step 1 we have a different notion of *value*: while in Step 0 the interpreter produced a *function*, now it produces a *closure*. It is possible to use the closure produced in Step 1 to recreate the function that would have been produced in Step 0 by substituting the variable references in the function body with the corresponding values from the environment found in the closure. We may do this to check that the outputs of the interpreters are equivalent, but in Step 1 we do not perform

substitution in the regular course of interpretation; we add more mappings to the environment and look up variable references in the environment.

Alternative Argument

Another way to reason about an environment-based interpreter is that it is a substitution-based interpreter in which substitutions are delayed until they are needed.

1.4.2 New Data Structures

The following are the data structures used to represent environments and closures:

```
type Value = Closure;

type Closure = {
  function: ArrowFunctionExpression;
  environment: Environment;
};

type Environment = MapDeepEqual<Identifier["name"], Value>;
```

Implementation Details

The `MapDeepEqual` data structure is provided by a JavaScript package developed by the author called Collections Deep Equal [12]. A `MapDeepEqual` is similar to a `Map`, except that the keys are compared by value, not by reference, for example:

```
> new Map([[{ age: 29 }, "Leandro"]]).get({ age: 29 });
undefined
> new MapDeepEqual([[{ age: 29 }, "Leandro"]]).get({ age: 29 });
"Leandro"
```

The occurrences of `{ age: 29 }` are objects with the same key and value, but they are not the same object.

1.4.3 Adding an Environment to the Runner

The runner needs to maintain an environment, so we modify the implementation of `run()` from § 1.3.13 to introduce an auxiliary function called `step()` that receives an `environment` as an extra parameter:

```
function run(expression: Expression): Value {
  return step(expression, new MapDeepEqual());
  function step(expression: Expression, environment: Environment): Value
↪   {
    switch (expression.type) {
      case "ArrowFunctionExpression":
        return expression;
      case "CallExpression":
        const {
          params: [parameter],
          body,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        return step(substitute(body), environment);
        function substitute(expression: Expression): Expression {
          switch (expression.type) {
            case "ArrowFunctionExpression":
              if (expression.params[0].name === parameter.name)
                return expression;
              return {
                ...expression,
                body: substitute(expression.body),
              };
            case "CallExpression":
              return {
                ...expression,
                callee: substitute(expression.callee),
                arguments: [substitute(expression.arguments[0])],
              };
            case "Identifier":
              if (expression.name !== parameter.name) return expression;
              return argument;
          }
        }
      case "Identifier":
        throw new Error(`Reference to undefined variable:
↪ ${expression.name}`);
    }
```

```
37    }
38  }
```

Line 3: We define the `step()` auxiliary function that receives an `environment` as an extra parameter.

Line 2: The `environment` starts empty.

Lines 11–13: The recursive calls to `run()` are changed to recursive calls to `step()` and the `environment` is propagated.

The listing above does not compile yet because we are not producing closures. In the following sections we fix this by considering how to manage the `environment` for each type of `expression`.

1.4.4 A Function Definition

Example Program	`x => x`
Current Output	—
Expected Output	⟨`(x => x)`, []⟩

When the interpreter encounters a function definition, it captures the current `environment` in a closure:

```
// step()
case "ArrowFunctionExpression":
  return { function: expression, environment };
```

1.4.5 A Function Call

Example Program	`(x => z => x)(y => y)`
Current Output	—
Expected Output	⟨`(z => x)`, [`x` ↦ ⟨`(y => y)`, []⟩]⟩

First, we remove `substitute()`, which is the goal of Step 1:

```
// step()
case "CallExpression":
  const {
    params: [parameter],
    body,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(body, environment);
```

Next, we fix the pattern that matches the result of the interpretation of the called function to take in account the closure:

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
    environment: functionEnvironment,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(body, environment);
```

Finally, we modify the recursive call to `step()` that interprets the function body so that it receives a new augmented `environment` including a mapping from the `parameter` (for example, `x`) to the `argument` (for example, ⟨`(y => y)`, []⟩):

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
    environment: functionEnvironment,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(
    body,
    new MapDeepEqual(environment).set(parameter.name, argument)
  );
```

1.4.6 Name Reuse

Example Program `(x => x => z => x)(a => a)(y => y)`

Current Output ⟨`(z => x)`, [`x` ↦ ⟨`(y => y)`, []⟩]⟩

Expected Output ⟨`(z => x)`, [`x` ↦ ⟨`(y => y)`, []⟩]⟩

If a name is reused (for example, `x` in the example program above), then the second time it is encountered by `step()` it is overwritten in the `environment` (see the call to `set()` in line 13 of § 1.4.5, which overwrites an existing map key). This causes the variable reference to `x` to refer to the second (inner) `x`, which is the expected behavior (it is what we called Option 2 in § 1.3.7).

1.4.7 A Variable Reference

Example Program `(x => x)(y => y)`

Current Output —

Expected Output ⟨`(y => y)`, []⟩

Example Program `(x => y)(y => y)`

Current Output —

Expected Output `Reference to undefined variable: y`

Example Program `x => y`

Current Output —

Expected Output ⟨`(x => y)`, []⟩

When we encounter a variable reference, we look it up in the current environment:

```
// step()
case "Identifier":
  const value = environment.get(expression.name);
  if (value === undefined)
```

```
    throw new Error(
      `Reference to undefined variable: ${expression.name}`
    );
  return value;
```

1.4.8 A Function Body Is Evaluated with the Environment in Its Closure

Example Program

```
(f => (x => f(x))(a => a))((x => z => x)(y => y))
```

Current Output ⟨(a => a),[f ↦ ⟨(z => x),[x ↦ ⟨(y => y),[]⟩]⟩]⟩

Expected Output ⟨(y => y),[]⟩

This example program shows the difference between the current environment with which an expression is evaluated and the environment that comes from a closure. The following is a trace of the first call to `step()`, when a closure is created:

Trace 1: First Call to `step()` · Closure Creation

Line (see § 1.4.5)

```
            expression.callee = f => (x => f(x))(a => a)
expression.arguments[0] = (x => z => x)(y => y)
10                 argument = ⟨(z => x),[x ↦ ⟨(y => y),[]⟩]⟩
```

And the following is a trace of the recursive call to `step()` in which that closure is called:

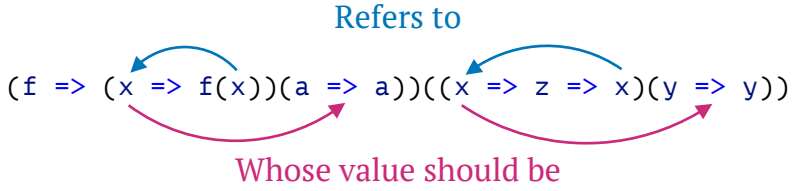
Trace 2: Recursive Call to `step()` · Closure Call

Line	(see § 1.4.5)		
	`expression`	=	`f(x)`
	`expression.callee`	=	`f`
	`expression.arguments[0]`	=	`x`
	`environment`	=	[x ↦ ⟨(a => a), [⋯]⟩, ⋯]
9	`step( ⋯ )`	=	⟨(z => x), [x ↦ ⟨(y => y), []⟩]⟩
5	`parameter`	=	`z`
6	`body`	=	`x`
8	`functionEnvironment`	=	[x ↦ ⟨(y => y), []⟩]

Paused before line 10

At this point, there are two expressions left to evaluate: the argument (`expression.arguments[0]`; line 10), and the body of the called function (`body`; lines 11–14). Both of these expressions have the same code (`x`), and our current implementation looks up this variable both times on the current `environment`, which produces the same value: ⟨`(a => a)`, [⋯]⟩.

But this leads to an issue: we may not reason about `z => x` by looking only at where it is defined; we must also examine all the places in which it may be called. This is the same issue we had to solve when considering name reuse in Step 0 (see § 1.3.7). We would like, instead, for each `x` to refer to the value that existed in the environment where the closure is *created*, not where it is *called*:



To implement this, we change the recursive call to `step()` that evaluates the function body so that it uses the environment coming from the closure

(`functionEnvironment`) instead of the current environment (`environment`):

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
    environment: functionEnvironment,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(
    body,
    new MapDeepEqual(functionEnvironment).set(parameter.name, argument)
  );
```

Technical Terms

The principle of being able to reason about a function only by looking at its definition is something called *local reasoning* (see § 1.3.7). The treatment given to the environment before this section is something called *dynamic scoping*, because the *scope* of a variable (where a variable is defined) is *dynamic*, depending on where the function is called. The treatment given to the environment in this section is something called *static scoping* or *lexical scoping*, because the scope of a variable is determined before we start interpreting the program.

Advanced

There are languages that implement dynamic scoping. In some cases dynamic scoping is the only option, for example, in the original implementation of LISP [21], though that was later considered a mistake [22]. In some cases dynamic scoping is the default, but there is an option to use static scoping, for example, in Emacs Lisp [20, § 12.10]. In some cases dynamic scoping is an ex-

tra feature to be used sparingly, for example, in Racket's `parameterize` [15, § 4.13].

1.4.9 The Entire Runner

We completed the changes necessary to transform the `run()` function from the substitution-based interpreter in Step 0 into an environment-based interpreter:

```
type Value = Closure;

type Closure = {
  function: ArrowFunctionExpression;
  environment: Environment;
};

type Environment = MapDeepEqual<Identifier["name"], Value>;

function run(expression: Expression): Value {
  return step(expression, new MapDeepEqual());
  function step(expression: Expression, environment: Environment): Value
  ↪ {
    switch (expression.type) {
      case "ArrowFunctionExpression":
        return { function: expression, environment };
      case "CallExpression":
        const {
          function: {
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        return step(
          body,
          new MapDeepEqual(functionEnvironment).set(parameter.name,
  ↪ argument)
        );
      case "Identifier":
        const value = environment.get(expression.name);
        if (value === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
```

```
            );
          return value;
      }
    }
}
```

Advanced

1.4.10 Operational Semantics

We adapt the operational semantics from § 1.3.14 to the interpreter defined in Step 1. First, we change the notion of values:

$$\begin{array}{rcll} v & = & \langle (x \texttt{ => } e), \rho \rangle & \text{Values / Closures} \\ \rho & = & \{x \mapsto v, \cdots\} & \text{Environments} \end{array}$$

We then define the relation $\rho \vdash e \Rightarrow v$ to be equivalent to the new implementation of `run()`:

$$\frac{}{\rho \vdash (x \texttt{ => } e) \Rightarrow \langle (x \texttt{ => } e), \rho \rangle}$$

$$\frac{\rho \vdash e_f \Rightarrow \langle (x \texttt{ => } e_b), \rho_f \rangle \qquad \rho \vdash e_a \Rightarrow v_a \qquad \rho_f \cup \{x \mapsto v_a\} \vdash e_b \Rightarrow v}{\rho \vdash e_f(e_a) \Rightarrow v}$$

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

1.4.11 Stringifier

We modify the stringifier from § 1.3.16 to support closures. For example, the following is the representation of the closure from § 1.4.5:

$$\langle (\texttt{z => x}), [\texttt{x} \mapsto \langle (\texttt{y => y}), [] \rangle] \rangle$$

```
{
  "function": "z => x",
  "environment": [
    [
      "x",
      {
        "function": "y => y",
        "environment": []
      }
    ]
  ]
}
```

The following is the modified implementation of `stringify()`:

```
1  function stringify(value: any): string {
2    return JSON.stringify(
3      value,
4      (key, value) => {
5        if (value.type !== undefined)
6          return prettier
7            .format(escodegen.generate(value), {
8              parser: "babel",
9              semi: false,
10             arrowParens: "avoid",
11           })
12           .trim();
13       return value;
14     },
15     2
16   );
17 }
```

Line 1: Change the input type from `Value` to `any`, because this implementation of `strinfigy()` supports any data structure, including the data structures we will define in later Steps.

Line 2: Call `JSON.stringify()` [29], which traverses any data structure and converts it into a string.

Line 4: Provide a `replacer` that is responsible for converting data structures that represent Yocto-JavaScript programs into strings.

Line 5: Check whether a data structure represents an Yocto-JavaScript program by checking the existence of a field called `type` (see § 1.3.2), in which case we use the previous implementation of `stringify()` (see § 1.3.16) to produce a string.

Line 15: Format the output with indentation of two spaces.

This implementation of `stringify()` supports not only closures but any data structure (because of `JSON.stringify()`), so it will remain the same in the following Steps.

1.4.12 Programs That Do Not Terminate

The programs that do not terminate in Step 0 (see § 1.3.17) do not terminate in Step 1 either, because the interpreters are equivalent except for the technique used to interpret function calls, but the sources of non-termination are different. In Step 0 substitution may produce infinitely many expressions, including expressions that do not occur in the original program. In Step 1 the interpreter considers only the finitely many expressions that occur in the original program, but it may produce infinitely many environments.

This difference is significant because there are programs that produce infinitely many different expressions in Step 0, but produce the same expression and environment repeatedly in Step 1, and in these cases we could detect non-termination by checking whether the runner is in a loop with the same arguments, for example:

`(F(F))`, where `F = f => (f(f))(f(f))`

Step 0	Step 1
`(F(F))`	⟨`(F(F))`, [`f` ↦ ⟨`F`, []⟩]⟩
`(F(F))(F(F))`	⟨`(F(F))`, [`f` ↦ ⟨`F`, []⟩]⟩
`(F(F))(F(F))(F(F))`	⟨`(F(F))`, [`f` ↦ ⟨`F`, []⟩]⟩
`(F(F))(F(F))(F(F))(F(F))`	⟨`(F(F))`, [`f` ↦ ⟨`F`, []⟩]⟩
⋮	⋮

But this strategy is insufficient to guarantee termination, because there are programs that do not terminate which produce infinitely many different environments, for example:

`(f => c => f(f)(x => c))(f => c => f(f)(x => c))(y => y)`

or `F(F)(y => y)`, where `F = f => c => f(f)(C)` and `C = x => c`

⟨`f(f)(C)`, [`c` ↦ ⟨`(y => y)`, []⟩, ⋯]⟩

⟨`f(f)(C)`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`(y => y)`, []⟩, ⋯]⟩, ⋯]⟩

⟨`f(f)(C)`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`(y => y)`, []⟩, ⋯]⟩, ⋯]⟩, ⋯]⟩

⟨`f(f)(C)`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`C`, [`c` ↦ ⟨`(y => y)`, []⟩, ⋯]⟩, ⋯]⟩, ⋯]⟩, ⋯]⟩

⋮

The program above is a variation on the shortest non-terminating program, `(f => f(f))(f => f(f))`, in which each `f => f(f)` receives an additional parameter `c`, and each `f(f)` receives an additional argument `x => c`.

The interpreter in Step 1 may produce infinitely many different environments because environments may be nested. That is the issue that we address in Step 2.

Technical Terms

The nesting of environments in Step 1 characterizes them as something called *recursive data structures*: data structures that may contain themselves. The data structures used to represented Yocto-JavaScript programs are recursive

as well (see § 1.3.2).

1.5 Step 2: Store-Based Interpreter

The source of non-termination in Step 1 is the nesting of the environments (see § 1.4.12). In Step 2, we address this issue by introducing a layer of indirection between a name in the environment and its corresponding value. The interpreter in Step 2 may still not terminate, but that is due to other sources of non-termination that will be addressed in subsequent Steps.

1.5.1 Avoiding Nested Environments by Introducing a Store

In Step 1 a closure contains an environment mapping names to other closures, which in turn contain their own environments mapping to yet more closures. In Step 2, we remove this circularity by introducing a layer of indirection: an environment maps names to *addresses*, which may be used to lookup values in a *store*, for example:

(See § 1.4.7)	**Step 1**	**Step 2**
Variable Reference	`x`	`x`
Environment	[`x` ↦ ⟨`(y => y)`,[]⟩]	[`x` ↦ `0`]
Store	—	[`0` ↦ ⟨`(y => y)`,[]⟩]
Value	⟨`(y => y)`,[]⟩	⟨`(y => y)`,[]⟩

Each closure continues to include its own environment, because it needs to look up variable references from where the closure was created (see § 1.4.8), but there is only one store for the entire interpreter, and we avoid ambiguities by allocating different addresses, for example:

(See § 1.4.8)	**Step 1**	**Step 2**
`environment`	$[\texttt{x} \mapsto \langle \texttt{(a => a)}, [\cdots] \rangle, \cdots]$	$[\texttt{x} \mapsto \texttt{0}, \cdots]$
`funcEnv.`	$[\texttt{x} \mapsto \langle \texttt{(y => y)}, [] \rangle]$	$[\texttt{x} \mapsto \texttt{1}]$
Store	—	$[\texttt{0} \mapsto \langle \texttt{(a => a)}, [\cdots] \rangle,$ $\texttt{1} \mapsto \langle \texttt{(y => y)}, [] \rangle]$

The runner must return the store along with the value, for the variable references to be looked up, for example:

Example Program (see § 1.4.5)	`(x => z => x)(y => y)`
Step 1 Output	$\langle \texttt{(z => x)}, [\texttt{x} \mapsto \langle \texttt{(y => y)}, [] \rangle] \rangle$
Step 2 Output	`value` = $\langle \texttt{(z => x)}, [\texttt{x} \mapsto \texttt{0}] \rangle$
	`store` = $[\texttt{0} \mapsto \langle \texttt{(y => y)}, [] \rangle]$

Advanced

The technique used in Step 2 is related to the run-time environments that are the target of traditional compilers for languages such as C [9, § 7]: an environment corresponds to some of the data stored in an activation frame on the call stack, and the store corresponds to the heap.

1.5.2 New Data Structures

The following are the data structures used to represent environments, stores, and addresses:

```
type Environment = MapDeepEqual<Identifier["name"], Address>;

type Store = MapDeepEqual<Address, Value>;

type Address = number;
```

1.5.3 Adding a Store to the Runner

We modify the implementation of `run()` from § 1.4.9 to introduce a `store`:

```
1 function run(expression: Expression): { value: Value; store: Store } {
2   const store: Store = new MapDeepEqual();
3   return { value: step(expression, new MapDeepEqual()), store };
4   function step(expression: Expression, environment: Environment): Value
    ↪  {
5     // ...
6   }
7 }
```

Lines 1 and 3: The `store` is returned because it is necessary to look up variable references in the `value`.

Lines 2 and 4: The `store` is unique for the whole interpreter, unlike `environment`s which are different for each closure, so we create only one store that is always available to `step()` instead of adding it as an extra parameter.

1.5.4 Adding a Value to the Store

Example Program	`(x => z => x)(y => y)`
Current Output	—
Expected Output	`value` = ⟨`(z => x)`, [`x` ↦ `0`]⟩
	`store` = [`0` ↦ ⟨`(y => y)`, []⟩]

In Step 1, when we encounter a function call we extend the `functionEnvironment` with a mapping from the `parameter.name` to the `argument` (see § 1.4.5, 1.4.8). In Step 2, we introduce the `store` as a layer of indirection:

```
1 // step()
2 case "CallExpression": {
3   const {
```

```
4       function: {
5         params: [parameter],
6         body,
7       },
8       environment: functionEnvironment,
9     } = step(expression.callee, environment);
10    const argument = step(expression.arguments[0], environment);
11    const address = store.size;
12    store.set(address, argument);
13    return step(
14      body,
15      new MapDeepEqual(functionEnvironment).set(parameter.name, address)
16    );
17  }
```

Line 11: Allocate an `address`. We use the `store.size` as the `address` because as the `store` grows this number changes, so it is guaranteed to be unique throughout the interpretation of a program.

Line 15: Extend the `functionEnvironment` with a mapping from the `parameter.name` to the `address`.

Line 12: Extend the `store` with a mapping from the `address` to the `argument`. This is mutating the unique `store` that is available to the entire `step()` function, not creating a new `store` in the same way that extending an `environment` creates a new `environment` (see line 15).

1.5.5 Retrieving a Value from the Store

Example Program	`(x => x)(y => y)`
Current Output	—
Expected Output	`value` = ⟨`(y => y)`, []⟩
	`store` = [`0` ↦ ⟨`(y => y)`, []⟩]

In Step 1 we retrieved values directly from the `environment` (see § 1.4.7), but in Step 2 we have to go through the `store`:

```
1  // step()
2  case "Identifier": {
3    const address = environment.get(expression.name);
4    if (address === undefined)
5      throw new Error(
6        `Reference to undefined variable: ${expression.name}`
7      );
8    return store.get(address)!;
9  }
```

Line 3: Retrieve the `address` from the `environment`.

Line 8: Retrieve the `value` from the `store` at the given `address` found in the `environment`. The `address` is guaranteed to be in the `store` because we extend the `store` and the `environment` together (see § 1.5.4), so we use `!` to indicate that `get()` may not return `undefined`.

1.5.6 The Entire Runner

We completed the changes necessary to remove the circularity between closures and environments:

```
1  type Environment = MapDeepEqual<Identifier["name"], Address>;
2
3  type Store = MapDeepEqual<Address, Value>;
4
5  type Address = number;
6
7  function run(expression: Expression): { value: Value; store: Store } {
8    const store: Store = new MapDeepEqual();
9    return { value: step(expression, new MapDeepEqual()), store };
10   function step(expression: Expression, environment: Environment): Value
     ↪ {
11     switch (expression.type) {
12       case "ArrowFunctionExpression": {
13         return { function: expression, environment };
14       }
15       case "CallExpression": {
16         const {
17           function: {
```

```
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        const address = store.size;
        store.set(address, argument);
        return step(
          body,
          new MapDeepEqual(functionEnvironment).set(parameter.name,
↪   address)
        );
      }
      case "Identifier": {
        const address = environment.get(expression.name);
        if (address === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return store.get(address)!;
      }
    }
  }
}
```

Advanced

1.5.7 Operational Semantics

We adapt the operational semantics from § 1.5.7 to the interpreter defined in Step 2. First, we change the notion of environments:

$$\begin{array}{lcll} \rho & = & \{x \mapsto A, \cdots\} & \text{Environments} \\ \sigma & = & \{A \mapsto v, \cdots\} & \text{Stores} \\ A & = & \mathbb{N} & \text{Addresses} \end{array}$$

We then define the relation $\rho, \sigma \vdash e \Rightarrow \langle v, \sigma \rangle$ to be equivalent to the new implementation of `run()`:

$$\frac{}{\rho, \sigma \vdash (x \texttt{ => } e) \Rightarrow \langle\langle (x \texttt{ => } e), \rho\rangle, \sigma\rangle}$$

$$\frac{\rho, \sigma \vdash e_f \Rightarrow \langle\langle (x \texttt{ => } e_b), \rho_f\rangle, \sigma_f\rangle \qquad \rho, \sigma_f \vdash e_a \Rightarrow \langle v_a, \sigma_a\rangle \qquad A = |\sigma_a| \qquad \rho_f \cup \{x \mapsto A\}, \sigma_a \cup \{A \mapsto v_a\} \vdash e_b \Rightarrow \langle v, \sigma_v\rangle}{\rho, \sigma \vdash e_f(e_a) \Rightarrow \langle v, \sigma_v\rangle}$$

$$\frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x))}$$

1.5.8 Programs That Do Not Terminate

The programs that do not terminate in Step 1 (see § 1.4.12) do not terminate in Step 2 either, because the interpreters are equivalent except for the treatment of environments, but the sources of non-termination are different. In both cases the issue is that the interpreter may create infinitely many environments, but in Step 1 the environments are nested, and in Step 2 they contain different addresses, for example:

`(f => c => f(f)(x => c))(f => c => f(f)(x => c))(y => y)`

or `F(F)(y => y)`, where `F = f => c => f(f)(C)` and `C = x => c`

Step 1

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \langle \texttt{(y => y)}, [] \rangle, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{(y => y)}, [] \rangle, \cdots] \rangle, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{(y => y)}, [] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \langle \texttt{(y => y)}, [] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle$

$\vdots$

Step 2

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \texttt{0}, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \texttt{1}, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \texttt{2}, \cdots] \rangle$

$\langle \texttt{f(f)(C)}, [\texttt{c} \mapsto \texttt{3}, \cdots] \rangle$

$\vdots$

$[\texttt{0} \mapsto \langle \texttt{(y => y)}, [] \rangle, \texttt{1} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \texttt{0}, \cdots] \rangle, \texttt{2} \mapsto \langle \texttt{C}, [\texttt{c} \mapsto \texttt{1}, \cdots] \rangle, \cdots]$

We address this issue in Step 3.

1.6 Step 3: Finitely Many Addresses

1.6.1 The Entire Runner

We completed the changes necessary to produce only finitely many addresses:

```
type Value = SetDeepEqual<Closure>;

type Address = Identifier;

function run(expression: Expression): { value: Value; store: Store } {
  const store: Store = new MapDeepEqual();
  return { value: step(expression, new MapDeepEqual()), store };
  function step(expression: Expression, environment: Environment): Value
  ↪ {
```

```
    switch (expression.type) {
      case "ArrowFunctionExpression": {
        return new SetDeepEqual([{ function: expression, environment }]);
      }
      case "CallExpression": {
        const value: Value = new SetDeepEqual();
        for (const {
          function: {
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } of step(expression.callee, environment)) {
          const argument = step(expression.arguments[0], environment);
          const address = parameter;
          store.merge(new MapDeepEqual([[address, argument]]));
          value.merge(
            step(
              body,
              new MapDeepEqual(functionEnvironment).set(parameter.name, address)
            )
          );
        }
        return value;
      }
      case "Identifier": {
        const address = environment.get(expression.name);
        if (address === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return store.get(address)!;
      }
    }
  }
}
```

Bibliography

[1] Babel. https://babeljs.io. Accessed 2020-04-06.

[2] Escodegen. https://github.com/estools/escodegen. Accessed 2020-02-18.

[3] Esprima. https://esprima.org. Accessed 2020-01-21.

[4] Esprima Interactive Online Demonstration. https://esprima.org/demo/parse.html. Accessed 2020-01-21.

[5] Prettier. https://prettier.io. Accessed 2020-02-18.

[6] The ESTree Spec. https://github.com/estree/estree. Accessed 2020-01-21.

[7] TypeScript Documentation. https://www.typescriptlang.org/docs. Accessed 2020-01-17.

[8] TypeScript Types for the ESTree Spec. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/1911a1fbbe30af03f0f38a915c4bf0620d251fc6/types/estree/index.d.ts. Accessed 2020-01-27.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 2006.

[10] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.*, 41(4), October 2019.

[11] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] Leandro Facchinetti. Collections Deep Equal. `https://github.com/leafac/collections-deep-equal`. Accessed 2020-04-01.

[13] Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17(1):35 – 75, 1991.

[14] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[15] Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. `https://docs.racket-lang.org/guide/`. Accessed 2020-04-13.

[16] Mike Grant, Zachary Palmer, and Scott Smith. *Principles of Programming Languages*. 2020.

[17] JetBrains. The State of Developer Ecosystem 2019. `https://www.jetbrains.com/lp/devecosystem-2019/`. Accessed 2020-01-14.

[18] Gilles Kahn. Natural Semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

[19] Peter J Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[20] Bil Lewis, Dan LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual*. AAA, 2015.

[21] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, April 1960.

[22] John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978.

[23] Matthew Might. The Language of Languages. `http://matt.might.net/articles/grammars-bnf-ebnf/`. Accessed 2020-01-17.

[24] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and Exploiting the *k*-CFA Paradox: Illuminating Functional vs Object-Oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 305–315, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Mozilla. Arrow Function Expressions. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions`. Accessed 2020-01-16.

[26] Mozilla. Destructuring Assignment. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment`. Accessed 2020-01-27.

[27] Mozilla. Spread Syntax. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax`. Accessed 2020-02-03.

[28] Mozilla. `eval()`. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`. Accessed 2020-02-13.

[29] Mozilla. `JSON.stringify()`. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`. Accessed 2020-04-13.

[30] Stack Overflow. Developer Survey Results 2019. `https://insights.stackoverflow.com/survey/2019`. Accessed 2020-01-14.

[31] G.D. Plotkin. Call-By-Name, Call-By-Value and the λ-Calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[32] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery.

[33] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[34] Tom Stuart. *Understanding Computation: From Simple Machines to Impossible Programs*. O'Reilly Media, 1st edition, 2013.

[35] Basarat Ali Syed. TypeScript Deep Dive. `https://basarat.gitbook.io/typescript/`. Accessed 2020-01-17.