

# CSCI 3155: Lab Assignment 4

Fall 2015: Due Saturday, October 24, 2015

The primary purpose of this lab is to understand the interplay between type checking and evaluation. Concretely, we will extend JAVASCRIPTY with immutable objects and extend our small-step interpreter from Lab 3. Unlike all prior language constructs, object expressions do not have an *a priori* bound on the number of sub-expressions because an object can have any number of fields. To represent objects, we will use collections from the Scala library and thus will need to get used to working with the Scala collection API.

From your team of 8-10 persons in your lab section, find a new partner for this lab assignment (different from your Lab 1 partner). You will work on this assignment closely with your partner. However, note that **each student needs to submit** and are individually responsible for completing the assignment.

You are welcome to talk about these questions in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus.

*Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!*

*We will consider the following criteria in our grading:*

- How well does your submission answer the questions? *For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.*
- How clear is your submission? *If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any pre-conditions or invariants for your functions (either in comments, as assertions, or as require clauses as appropriate).*

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expression computation currently unfamiliar to you.

Finally, make sure that your file compiles and runs on COG. A program that does not compile will *not* be graded.

**Submission Instructions.** Upload to the moodle exactly four files named as follows:

- Lab4-*YourIdentiKey*.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- Lab4-*YourIdentiKey*.scala with your answers to the coding exercises
- Lab4Spec-*YourIdentiKey*.scala with any updates to your unit tests.
- Lab4-*YourIdentiKey*.jsy with a challenging test case for your JAVASCRIPTY interpreter.

Replace *YourIdentiKey* with your *IdentiKey* (e.g., I would submit Lab4-bec.pdf and so forth). Don't use your student identification number. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Submit your Lab4.scala file to COG for auto-testing. We ask that you submit both to COG and to moodle in case of any issues.

Sign-up for an interview slot for an evaluator. To fairly accommodate everyone, the interview times are strict and **will not be rescheduled**. Missing an interview slot means missing the interview evaluation component of your lab grade. Please take advantage of your interview time to maximize the feedback that you are able receive. Arrive at your interview ready to show your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

**Getting Started.** Clone the code from the Github repository with the following command:

```
git clone -b lab4 https://github.com/bechang/pppl-labs.git lab4
```

A suggested way to get familiar with Scala is to do some small lessons with Scala Koans (<http://www.scalakoans.org/>). Useful ones for Lab 4 are AboutHigherOrderFunctions, AboutLists, AboutPartialFunctions, and AboutInfixPrefixAndPostfixOperators.

1. **Feedback.** Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.
2. **Warm-Up: Collections.** To implement our interpreter for JAVASCRIPTY with objects, we will need to make use of collections from Scala's library. One of the most fundamental operations that one needs to perform with a collection is to iterate over the elements of the collection. Like many other languages with first-class functions (e.g., Python, ML), the Scala library provides various iteration operations via *higher-order functions*. Higher-order functions are functions that take functions as parameters. The function parameters are often called *callbacks*, and for collections, they typically specify what the library client wants to do for each element.

In this question, we practice both writing such higher-order functions in a library and using them as a client.

- (a) Implement a function

```
def compressRec[A](l: List[A]): List[A]
```

that eliminates consecutive duplicates of list elements. If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.

Example:

```
scala> compressRec(List(1, 2, 2, 3, 3, 3))
res0: List[Int] = List(1, 2, 3)
```

This test has been provided for you in the template.

For this exercise, implement the function by direct recursion (e.g., pattern match on `l` and call `compressRec` recursively). Do not call any `List` library methods.

This exercise is from Ninety-Nine Scala Problems:

<http://aperiodic.net/phil/scala/s-99/> .

Some sample solutions are given there, which you are welcome to view. However, it is strongly encouraged that you first attempt this exercise before looking there. The purpose of the exercise is to get some practice for the later part of this homework. Note that the solutions there do not satisfy the requirements here (as they use library functions). If at some point you feel like you need more practice with collections, the above page is a good resource.

- (b) Re-implement the `compress` function from the previous part as `compressFold` using the `foldRight` method from the `List` library. The call to `foldRight` has been provided for you. Do not call `compressFold` recursively or any other `List` library methods.
- (c) Implement a higher-order recursive function

```
def mapFirst[A](f: A => Option[A])(l: List[A]): List[A]
```

that finds the first element in `l` where `f` applied to it returns a `Some(a)` for some value `a`. It should replace that element with `a` and leave `l` the same everywhere else.

Example:

```
scala> mapFirst((i: Int) => if (i < 0) Some(-i) else None)(List(1,2,-3,4,-5))
res0: List[Int] = List(1, 2, 3, 4, -5)
```

- (d) Consider again the binary search tree data structure from Lab 1:

```
sealed abstract class Tree {
  def insert(n: Int): Tree

  def foldLeft[A](z: A)(f: (A, Int) => A): A = {
    def loop(acc: A, t: Tree): A = t match {
      case Empty => ???
      case Node(l, d, r) => ???
    }
    loop(z, this)
  }
}

case object Empty extends Tree
case class Node(l: Tree, d: Int, r: Tree) extends Tree
```

Here, we have implement the binary search tree `insert` as a method of `Tree`. For

def map[A,B](f: A ⇒ B)(l: List[A])  
: List[B]

map[Int, Bool](...)(List(1, 2, 3))

map(...)(List(1, 2, 3))  
: List[Int]

---

def f(a: Int, b: Int): Int

f: (Int, Int) ⇒ Int

def g(a: Int)(b: Int): Int

g: Int ⇒ (Int ⇒ Int)

def  $g(a: \text{Int}) = \{ (b: \text{Int}) \Rightarrow$

.....

}

this exercise, complete the higher-order method `foldLeft`. This method performs an in-order traversal of the input tree this calling the callback `f` to accumulate a result. Suppose the in-order traversal of the input tree yields the following sequence of data values:  $d_1, d_2, \dots, d_n$ . Then, `foldLeft` yields

$$f(\dots(f(f(z, d_1), d_2))\dots), d_n).$$

We have provided a test client `sum` that computes the sum of all of the data values in the tree using your `foldLeft` method.

(e) Implement a function

```
def strictlyOrdered(t: Tree): Boolean
```

as a client of your `foldLeft` method that checks that the data values of `t` as an in-order traversal are in strictly accending order (i.e.,  $d_1 < d_2 < \dots < d_n$ ).

Example:

```
scala> strictlyOrdered(treeFromList(List(1,1,2)))
res0: Boolean = false
```

### 3. JavaScripty Type Checker

As we have seen in the prior labs, dealing with conversions and checking for dynamic type errors complicate the interpreter implementation. Some languages restrict the possible programs that it will execute to ones that it can guarantee will not result in a dynamic type error. This restriction of programs is enforced with an analysis phase after parsing known as *type checking*. Such languages are called *strongly, statically-typed*. In this lab, we will implement a strongly, statically-typed version of JAVASCRIPTY. We will not permit any type conversions and will guarantee the absence of dynamic type errors.

In this lab, we extend JAVASCRIPTY with types, multi-parameter functions, and objects/records (see Figure 1). We have a language of types  $\tau$  and annotate function parameters with types. Functions can now take any number of parameters. We write a sequence of things using either an overbar or dots (e.g.,  $\bar{e}$  or  $e_1, \dots, e_n$  for a sequence of expressions). An object literal

$$\{f_1 : e_1, \dots, f_n : e_n\}$$

is a comma-separated sequence of field names with initialization expressions surrounded by braces. Objects in this lab are more like records or C-structs as opposed to JavaScript objects, as we do not have any form of mutation, dynamic extension, or dynamic dispatch. The field read expression  $e_1.f$  evaluates  $e_1$  to an object value and then looks up the field named  $f$ . An object value is a sequence of field names associated with values. The type language  $\tau$  includes base types for numbers, booleans, strings, and **undefined**, as well as constructed types for functions  $(\bar{x} : \tau) \Rightarrow \tau'$  and objects  $\{f_1 : \tau_1; \dots; f_n : \tau_n\}$ .

As an aside, we have chosen a syntax that is compatible with the TypeScript language that adds typing to JavaScript. TypeScript aims to be fully compatible with JavaScript, so it is not as strictly typed as JAVASCRIPTY in this lab.

expressions	$e ::= x \mid n \mid b \mid \mathbf{undefined} \mid uop\ e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$ $\mid \mathbf{const}\ x = e_1; e_2 \mid \mathbf{console.log}(e_1)$ $\mid str \mid \mathbf{function}\ p(\overline{x:\tau})\ tann\ e_1 \mid e_0(\overline{e})$ $\mid \{f_1 : e_1, \dots, f_n : e_n\} \mid e_1.f$
values	$v ::= n \mid b \mid \mathbf{undefined} \mid str \mid \mathbf{function}\ p(\overline{x:\tau})\ tann\ e_1$ $\mid \{f_1 : v_1, \dots, f_n : v_n\}$
unary operators	$uop ::= - \mid !$
binary operators	$bop ::= , \mid + \mid - \mid * \mid / \mid == \mid != \mid < \mid <= \mid > \mid >= \mid \&\& \mid   $
types	$\tau ::= \mathbf{number} \mid \mathbf{bool} \mid \mathbf{string} \mid \mathbf{Undefined} \mid (\overline{x:\tau}) \Rightarrow \tau' \mid \{f_1 : \tau_1; \dots; f_n : \tau_n\}$
variables	$x$
numbers (doubles)	$n$
booleans	$b ::= \mathbf{true} \mid \mathbf{false}$
strings	$str$
function names	$p ::= x \mid \varepsilon$
field names	$f$
type annotations	$tann ::= : \tau \mid \varepsilon$
type environments	$\Gamma ::= \cdot \mid \Gamma[x \mapsto \tau]$

Figure 1: Abstract Syntax of JAVASCRIPTY

statements	$s ::= \mathbf{const}\ x = e \mid e \mid \{s_1\} \mid ; \mid s_1\ s_2$
expressions	$e ::= \dots \mid \mathbf{const}\ x = e_1; e_2 \mid (e_1)$ $\mid \mathbf{function}\ p(\overline{x:\tau})\ tann\ e_1 \mid \mathbf{function}\ p(\overline{x:\tau})\ tann\ \{s_1\}\ \mathbf{return}\ e_1 \mid (\overline{x:\tau}) \Rightarrow e$

Figure 2: Concrete Syntax of JAVASCRIPTY

```

/* Functions */
case class Function(p: Option[String], params: List[(String, Typ)], tann: Option[Typ],
                    e1: Expr) extends Expr
    Function( $\overline{p}$ ,  $\overline{(x, \tau)}$ , tann, e1) function  $p(\overline{x: \tau})$  tann e1
case class Call(e1: Expr, args: List[Expr]) extends Expr
    Call(e1,  $\overline{e}$ ) e1( $\overline{e}$ )

/* Objects */
case class Obj(fields: Map[String, Expr]) extends Expr
    Object( $\overline{f: e}$ ) { $\overline{f: e}$ }
case class GetField(e1: Expr, f: String) extends Expr
    GetField(e1, f) e1.f

/* Types */
case class TFunction(params: List[(String, Typ)], tret: Typ) extends Typ
    TFunction( $\overline{(x: \tau)}$ ,  $\tau'$ )  $(\overline{x: \tau}) \Rightarrow \tau'$ 
case class TObj(tfields: Map[String, Typ]) extends Typ
    TObj( $\overline{f: \tau}$ ) { $\overline{f: \tau}$ }

```

Figure 3: Representing in Scala the abstract syntax of JAVASCRIPTY. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.

As before, the concrete syntax accepted by the parser is slightly less flexible than the abstract syntax in order to match the syntactic structure of JavaScript. For function expressions, the body is surrounded by curly braces (i.e., { }) and consists of a statement  $s_1$  for **const** bindings followed by a **return** with an expression  $e_1$ . We do, however, support the anonymous function form

$$(\overline{x: \tau}) \Rightarrow e_1$$

from TypeScript, which is syntactic sugar for

**function**  $\overline{(x: \tau)}$  { **return**  $e_1$  }.

In Figure 3, we show the updated and new AST nodes. We update Function and Call for multiple parameters/arguments. Object literals and field read expressions are represented by Object and GetField, respectively.

In this lab, we implement a type checker that is very similar to a big-step interpreter. Instead of computing the value of an expression by recursively computing the value of each sub-expression, we infer the type of an expression, by recursively inferring the type of each sub-expression. An expression is *well-typed* if we can infer a type for it.

Given its similarity to big-step evaluation, we can formalize a type inference algorithm in a similar way. In Figure 4, we define the judgment form  $\Gamma \vdash e : \tau$  which says informally, “In type environment  $\Gamma$ , expression  $e$  has type  $\tau$ .” We will implement a function

```
def typeInfer(env: Map[String, Typ], e: Expr): Typ
```

that corresponds directly to this judgment form. It takes as input a type environment env ( $\Gamma$ ) and an expression e ( $e$ ) returns a type Typ ( $\tau$ ). It is informative to compare these rules with the big-step operational semantics from Lab 3.



$\Gamma \vdash e : \tau$

$\frac{\text{TYPEVAR}}{\Gamma \vdash x : \Gamma(x)}$	$\frac{\text{TYPEREG} \quad \Gamma \vdash e_1 : \mathbf{number}}{\Gamma \vdash -e_1 : \mathbf{number}}$	$\frac{\text{TYPERNOT} \quad \Gamma \vdash e_1 : \mathbf{bool}}{\Gamma \vdash !e_1 : \mathbf{bool}}$	$\frac{\text{TYPESEQ} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_2}$
$\frac{\text{TYPEARITH} \quad \Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number} \quad bop \in \{+, -, *, /\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \mathbf{number}}$		$\frac{\text{TYPEPLUSSTRING} \quad \Gamma \vdash e_1 : \mathbf{string} \quad \Gamma \vdash e_2 : \mathbf{string}}{\Gamma \vdash e_1 + e_2 : \mathbf{string}}$	
$\frac{\text{TYPEINEQUALITYNUMBER} \quad \Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number} \quad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \mathbf{bool}}$			
$\frac{\text{TYPEINEQUALITYSTRING} \quad \Gamma \vdash e_1 : \mathbf{string} \quad \Gamma \vdash e_2 : \mathbf{string} \quad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \mathbf{bool}}$			
$\frac{\text{TYPEEQUALITY} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \text{ has no function types} \quad bop \in \{==, !=\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \mathbf{bool}}$			
$\frac{\text{TYPEANDOR} \quad \Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad bop \in \{\&\&,   \}}{\Gamma \vdash e_1 \text{ bop } e_2 : \mathbf{bool}}$		$\frac{\text{TYPEPRINT} \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathbf{console.log}(e_1) : \mathbf{Undefined}}$	
$\frac{\text{TYPEIF} \quad \Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau}$		$\frac{\text{TYPECONST} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{const } x = e_1; e_2 : \tau_2}$	
$\frac{\text{TYPECALL} \quad \Gamma \vdash e : (x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \dots, e_n) : \tau}$		$\frac{\text{TYPEOBJECT} \quad \Gamma \vdash e_i : \tau_i \quad (\text{for all } i)}{\Gamma \vdash \{ \dots, f_i : e_i, \dots \} : \{ \dots, f_i : \tau_i, \dots \}}$	
$\frac{\text{TYPEGETFIELD} \quad \Gamma \vdash e : \{ \dots, f : \tau, \dots \}}{\Gamma \vdash e.f : \tau}$	$\frac{\text{TYPERNUMBER} \quad \Gamma \vdash n : \mathbf{number}}{\Gamma \vdash n : \mathbf{number}}$	$\frac{\text{TYPEBOOL} \quad \Gamma \vdash b : \mathbf{bool}}{\Gamma \vdash b : \mathbf{bool}}$	$\frac{\text{TYPESTRING} \quad \Gamma \vdash str : \mathbf{string}}{\Gamma \vdash str : \mathbf{string}}$
$\frac{\text{TYPEUNDEFINED}}{\Gamma \vdash \mathbf{undefined} : \mathbf{Undefined}}$		$\frac{\text{TYPEFUNCTION} \quad \Gamma[x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \vdash e : \tau \quad \tau' = (x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \mathbf{function } (x_1 : \tau_1, \dots, x_n : \tau_n) e : \tau'}$	
$\frac{\text{TYPEFUNCTIONANN} \quad \Gamma[x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \vdash e : \tau \quad \tau' = (x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \mathbf{function } (x_1 : \tau_1, \dots, x_n : \tau_n) : \tau e : \tau'}$			
$\frac{\text{TYPERECFUNCTION} \quad \Gamma[x \mapsto \tau'] [x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \vdash e : \tau \quad \tau' = (x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \mathbf{function } x(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau e : \tau'}$			

Figure 4: Typing of JAVASCRIPTY.

The `TYPEEQUALITY` is slightly informal in stating

$\tau$  has no function types.

We intend this statement to say that the structure of  $\tau$  has no function types. The helper function `hasFunctionTyp` is intended to return **true** if a function type appears in the input and **false** if it does not, so this statement can be checked by taking the negation of a call to `hasFunctionTyp`.

To signal a type error, we will use a Scala exception

```
case class StaticTypeError(tbad: Typ, esub: Expr, e: Expr) extends Exception
```

where `tbad` is the type that is inferred sub-expression `esub` of input expression `e`. These arguments are used to construct a useful error message. We also provide a helper function `err` to simplify throwing this exception.

We suggest the following step-by-step order to complete `typeInfer`.

1. First, complete the cases for the basic expressions excluding `Function`, `Call`, `Obj`, and `GetField`.
2. Then, work on these remaining cases. These cases use collections, so be sure to complete the previous question before attempting these cases. You can also work on step before finishing `typeInfer`.

**Hints:** Helpful library methods here include `map`, `foldLeft`, `zipped`, `foreach`, and `mapValues`. You may want to use `zipped` in the `Call` case to match up formal parameters and actual arguments.

#### 4. JavaScripty Small-Step Interpreter

In this question, we update `substitute` and `step` from Lab 3 for multi-parameter functions and objects. Because of type checking, the step cases can be greatly simplified. We eliminate the need to perform conversions and should no longer throw `DynamicTypeError`.

We introduce another Scala exception type

```
case class StuckError(e: Expr) extends Exception
```

that should be thrown when there is no possible next step. This exception looks a lot like `DynamicTypeError` except that the intent is that it should never be raised. It is intended to signal a coding error in our interpreter rather than an error in the `JAVASCRIPTY` test input.

In particular, if the `JAVASCRIPTY` expression  $e$  passed into `step` is closed and well-typed (i.e., judgment  $\vdash e : \tau$  holds meaning `inferType(e)` does not throw `StaticTypeError`), then `step` should never throw a `StuckError`. This property of `JAVASCRIPTY` is known as *type safety*.

A small-step operational semantics is given in Figure 5. This semantics no longer has conversions compared to Lab 3. It is much simpler because of type checking (e.g., even with the addition of objects, it fits on one page).

As specified, `SEARCHOBJECT` is non-deterministic (why?). As we view objects as an unordered set of fields, it says an object expression takes can take a step by stepping on any of its component fields. To match the reference implementation, you should make the step go on the first non-value as given by the left-to-right iteration of the collection using `Map.find`.

We suggest the following step-by-step order to complete `substitute` and `step`.

1. First, import the cases from your Lab 3 `step` function (perhaps excluding `Call`) and simplify them to remove calls to conversions (e.g., `toNumber`) and cases that throw `DynamicTypeError`. Follow the small-step operational semantics in Figure 5 to see how to simplify your Lab 3 code (e.g., start with `DONEG`).
2. Then, work on the object cases. These are actually simpler than the function cases. **Hint:** Note that field names are different than variable names. Object expressions are not variable binding constructs—what does that mean about `substitute` for them?
3. Then, work on the function cases. **Hints:** You might want to use your `mapFirst` function from the warm-up question. Helpful library methods here include `map`, `foldRight`, `zipped`, and `forall`.

$e \longrightarrow e'$

$\frac{\text{DoNEG} \quad n' = -n}{-n \longrightarrow n'}$	$\frac{\text{DoNOT} \quad b' = \neg b}{!b \longrightarrow b'}$	$\frac{\text{DoSEQ}}{v_1, e_2 \longrightarrow e_2}$	$\frac{\text{DoARITH} \quad n' = n_1 \text{ bop } n_2 \quad \text{bop} \in \{+, -, *, /\}}{n_1 \text{ bop } n_2 \longrightarrow n'}$	$\frac{\text{DoPLUSSTRING} \quad str' = str_1 + str_2}{str_1 + str_2 \longrightarrow str'}$
$\frac{\text{DoINEQUALITYNUMBER} \quad b' = n_1 \text{ bop } n_2 \quad \text{bop} \in \{<, <=, >, >=\}}{n_1 \text{ bop } n_2 \longrightarrow b'}$		$\frac{\text{DoINEQUALITYSTRING} \quad b' = str_1 \text{ bop } str_2 \quad \text{bop} \in \{<, <=, >, >=\}}{str_1 \text{ bop } str_2 \longrightarrow b'}$		
$\frac{\text{DoEQUALITY} \quad b' = (v_1 \text{ bop } v_2) \quad \text{bop} \in \{==, !=\}}{v_1 \text{ bop } v_2 \longrightarrow b'}$	$\frac{\text{DoANDTRUE}}{\mathbf{true} \ \&\& \ e_2 \longrightarrow e_2}$	$\frac{\text{DoANDFALSE}}{\mathbf{false} \ \&\& \ e_2 \longrightarrow \mathbf{false}}$		
$\frac{\text{DoORTRUE}}{\mathbf{true} \    \ e_2 \longrightarrow \mathbf{true}}$	$\frac{\text{DoORFALSE}}{\mathbf{false} \    \ e_2 \longrightarrow e_2}$	$\frac{\text{DoPRINT} \quad v_1 \text{ printed}}{\mathbf{console.log}(v_1) \longrightarrow \mathbf{undefined}}$	$\frac{\text{DoIFTRUE}}{\mathbf{true} ? e_2 : e_3 \longrightarrow e_2}$	
$\frac{\text{DoIFFALSE}}{\mathbf{false} ? e_2 : e_3 \longrightarrow e_3}$	$\frac{\text{DoCONST}}{\mathbf{const} \ x = v_1; e_2 \longrightarrow e_2[v_1/x]}$	$\frac{\text{DoCALL} \quad v = \mathbf{function} \ (x_1 : \tau_1, \dots, x_n : \tau_n) \text{ tann } e}{v(v_1, \dots, v_n) \longrightarrow e[v_n/x_n] \cdots [v_1/x_1]}$		
$\frac{\text{DoCALLREC} \quad v = \mathbf{function} \ x(x_1 : \tau_1, \dots, x_n : \tau_n) \text{ tann } e}{v(v_1, \dots, v_n) \longrightarrow e[v_n/x_n] \cdots [v_1/x_1][v/x]}$		$\frac{\text{DoGETFIELD}}{\{f_1 : v_1, \dots, f_i : v_i, \dots, f_n : v_n\}.f_i \longrightarrow v_i}$	$\frac{\text{SEARCHUNARY}}{e_1 \longrightarrow e'_1}$	
$\frac{\text{SEARCHBINARY}_1 \quad e_1 \longrightarrow e'_1}{e_1 \text{ bop } e_2 \longrightarrow e'_1 \text{ bop } e_2}$	$\frac{\text{SEARCHBINARY}_2 \quad e_2 \longrightarrow e'_2}{v_1 \text{ bop } e_2 \longrightarrow v_1 \text{ bop } e'_2}$	$\frac{\text{SEARCHPRINT} \quad e_1 \longrightarrow e'_1}{\mathbf{console.log}(e_1) \longrightarrow \mathbf{console.log}(e'_1)}$		
$\frac{\text{SEARCHIF} \quad e_1 \longrightarrow e'_1}{e_1 ? e_2 : e_3 \longrightarrow e'_1 ? e_2 : e_3}$	$\frac{\text{SEARCHCONST} \quad e_1 \longrightarrow e'_1}{\mathbf{const} \ x = e_1; e_2 \longrightarrow \mathbf{const} \ x = e'_1; e_2}$	$\frac{\text{SEARCHCALL}_1 \quad e \longrightarrow e'}{e(e_1, \dots, e_n) \longrightarrow e'(e_1, \dots, e_n)}$		
$\frac{\text{SEARCHCALL}_2 \quad e_i \longrightarrow e'_i}{(\mathbf{function} \ p(\overline{x} : \tau) \text{ tann } e)(v_1, \dots, v_{i-1}, e_i, \dots, e_n) \longrightarrow (\mathbf{function} \ p(\overline{x} : \tau) \text{ tann } e)(v_1, \dots, v_{i-1}, e'_i, \dots, e_n)}$				
$\frac{\text{SEARCHOBJECT} \quad e_i \longrightarrow e'_i}{\{ \dots, f_i : e_i, \dots \} \longrightarrow \{ \dots, f_i : e'_i, \dots \}}$		$\frac{\text{SEARCHGETFIELD} \quad e_1 \longrightarrow e'_1}{e_1.f \longrightarrow e'_1.f}$		

Figure 5: Small-step operational semantics of JAVASCRIPTY