Homework 6

The purpose of this assignment is to grapple with how dynamic binding arises and how to formally specify semantics. Concretely, we will extend JAKARTASCRIPT with recursive functions and implement two interpreters. The first will be a big-step interpreter that is an extension of Homework 4 but it implements dynamic binding "by accident". The second will be a small-step interpreter that exposes evaluation order and implements static binding by substitution.

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 expressing computation currently unfamilar to you.

For submission instructions and the due date, please see the `README.md` file.

Problem 1 JAKARTASCRIPT Interpreter: Recursive Functions, and Dynamic Binding (12 Points)

We now have the formal tools to specify exactly how a JAKARTASCRIPT program should behave. Unless otherwise specified, we will continue to try to match JavaScript semantics as implemented by Node.js. Thus, it is still useful to write little test JavaScript programs and run them through Node.js to see how the tests should behave. Finding bugs in the JAKARTASCRIPT specification with respect to JavaScript is certainly deserving of extra credit.

In this homework, we extend JAKARTASCRIPT with recursive functions. This language is very similar to the LETREC language in Section 3.4 of Friedman and Wand.

For this problem, we try to implement functions as an extension of Homework 4 in the most straightforward way. What we will discover is that we have made a historical mistake and ended up with a form of dynamic binding.

The syntax of JAKARTASCRIPT for this homework is given in Figure 1. Note that the grammar specifies the abstract syntax using notation borrowed from the concrete syntax. The new constructs are highlighted. We have added function values **`function`** $p(x)e$ and function call expressions $e_1(e_2)$.

In a function value, the function name p can either be a variable name or empty. The variable name is used for recursion. For simplicity, all functions are one argument functions. Since functions are first-class values (i.e., they are treated like any other value), we can get multi-argument functions via currying.

We have also added a "marker" `typeerror` to the expression language. This marker is not part of the source language. It is used in our definition of the evaluation relations. We discuss this in more detail further below.

As in previous homework assignments, the concrete syntax accepted by the parser slightly differs from the abstract syntax in order to match the syntactic structure of JavaScript. The difference between the concrete and abstract syntax is highlighted in Fig. 2. As in Homework 4, the concrete syntax treats **`const`** declarations as statements rather than expressions. For function values, the body is surrounded by curly braces (i.e., `{ }`) and consists of an optional statement st_{opt} for **`const`** bindings followed by a **`return`** with an expression

$$
\begin{array}{rrll}
n \in & \mathit{Num} & & \text{numbers (double)} \\
s \in & \mathit{Str} & & \text{strings} \\
x \in & \mathit{Var} & & \text{variables} \\
b \in & \mathit{Bool} & ::= \mathbf{true} \mid \mathbf{false} & \text{Booleans} \\
v \in & \mathit{Val} & ::= \mathbf{undefined} \mid n \mid b \mid s \mid \mathbf{function}\ p(x)e \mid \mathsf{typeerror} & \text{values} \\
e \in & \mathit{Expr} & ::= x \mid v \mid \mathit{uop}\ e \mid e_1\,\mathit{bop}\,e_2 \mid e_1\ ?\ e_2\ :\ e_3 \mid & \text{expressions} \\
 & & \quad \mathbf{const}\ x = e_1; e_2 \mid \mathbf{console.log}(e) \mid e_1(e_2) & \\
\mathit{uop} \in & \mathit{Uop} & ::= - \mid \mathtt{!} & \text{unary operators} \\
\mathit{bop} \in & \mathit{Bop} & ::= + \mid - \mid * \mid / \mid \mathtt{===} \mid \mathtt{!==} \mid \mathtt{<} \mid \mathtt{>} \mid \mathtt{<=} \mid \mathtt{>=} \mid \mathtt{\&\&} \mid \mathtt{||} \mid \mathtt{,} & \text{binary operators} \\
 & p & ::= x \mid \epsilon & \text{function names}
\end{array}
$$

Figure 1: Abstract syntax

$$
\begin{array}{rll}
e & ::= \cdots \mid \xcancel{\mathbf{const}\ x = e_1; e_2} \mid \xcancel{\mathsf{typeerror}} \mid \xcancel{\mathbf{function}\ p(x)e} & \text{expressions} \\
 & \quad \mid (e) \mid \mathbf{function}\ p(x)\{\mathit{st}_{\mathit{opt}}\ \mathbf{return}\ e\} \mid x\ \mathtt{=>}\ e & \\
\mathit{st} & ::= \mathbf{const}\ x = e_1\mathtt{;} \mid e\mathtt{;} \mid\ \mathtt{;} \mid \{\mathit{st}\} \mid \mathit{st}_1\ \mathit{st}_2 & \text{statements} \\
\mathit{st}_{\mathit{opt}} & ::= \mathit{st} \mid \epsilon &
\end{array}
$$

Figure 2: Concrete syntax

e. The concrete syntax also supports anonymous functions x `=>` e, which correspond to **function** $(x)e$ in the abstract syntax.

An abstract syntax tree representation is provided for you in `ast.scala`. We also provide a parser and main driver for testing. The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 3.

A big-step operational semantics of JakartaScript is given in Figure 4. The rules may seem daunting at first, but reading them will become easier with practice. This homework is such an opportunity to practice.

A formal semantics enables us to describe the semantics of a programming language clearly and concisely. The initial barrier is getting used to the meta-language of judgment forms and inference rules. However, once you cross that barrier, you will see that we are telling you exactly how to implement the interpreter–it will almost feel like cheating!

In Figure 4, we define the relation $\mathit{env} \vdash e \Downarrow v$. This relation corresponds directly to the `eval` function that we are asking you to implement (not a coincidence). It similarly has three parts:

```
def eval(env: Env, e: Expr): Val
```

It takes as input a value environment `env` and an expression `e`, and returns a value.

In Homework 4, all expressions could be evaluated to something (because of type conver-

```
enum Expr extends Positional:
  ...
  // Function literals
  case Function(p: Option[String], x: String, e: Expr)
    // <~ function p(x) e
  // Function call expressions
  case Call(e1: Expr, e2: Expr) extends Expr // <~ e1(e2)

// Values
type Val = Num | Boolean | Str | Undefined.type | Function
```

Figure 3: The abstract syntax of JakartaScript as represented in Scala. After each `case`, we show the corresponding JavaScript expression represented by that case.

sion). With functions, we encounter one of the very few run-time errors in JavaScript: trying to call something that is not a function. In JavaScript and in JakartaScript, calling a non-function raises a run-time error. In the formal semantics, we model this with evaluating to the "marker" typeerror.

Such a run-time error is known as a *dynamic type error*. Languages are called dynamically typed when they allow all syntactically valid programs to run and check for type errors during execution.

In our Scala implementation, we will not clutter our `Expr` type with a typeerror marker. Instead, we will use a Scala exception `DynamicTypeError`:

```
case class DynamicTypeError(e: Expr)
```

to signal this case. In other words, when your interpreter discovers a dynamic type error, you should throw this exception using the following Scala code:

```
throw DynamicTypeError(e)
```

The argument `e` should be the input expression to `eval` where the type error was detected. One advantage of using a Scala exception for typeerror is that the marker does not need to be propagated explicitly as in the inference rules in Figure 5. In particular, your interpreter will implement the EvalTypeError rules explicitly, but the EvalPropagate rules are implemented implicitly with Scala's exception propagation mechanism.

Note in rule EvaleEqual, we disallow equality and disequality checks (i.e., `===` and `!==`) on function values. If either argument to a equality or disequality check is a function value, then we consider this a dynamic type error. This choice is a slight departure from JavaScript.

(a) First, write some JakartaScript programs and execute them as JavaScript programs. This step will inform how you will implement your interpreter and will serve as tests for your interpreter. You do not need to submit your programs as part of your solution. This step is purely for your own benefit.

Think of one test case that behaves differently under dynamic binding versus static binding (and does not crash with a dynamic type error). Once you have finished the implementation of both interpreters, run your interpreters on your test case using `sbt` to confirm that the two interpreters indeed behave differently.

(b) Then implement:

```
def eval(env: Env, e: Expr): Val
```

that evaluates a JakartaScript expression `e` in a value environment `env` to a value according to the evaluation judgment $env \vdash e \Downarrow v$. The following helper functions for converting values to numbers, Booleans, and strings are provided:

```
def toNumber(v: Val): Double
def toBoolean(v: Val): Boolean
def toStr(v: Val): String
```

We suggest the following step-by-step process:

1. Bring your Homework 4 implementation into Homework 6 and make sure your previous test cases work as expected.
2. Extend your implementation to support non-recursive functions. For function calls, you need to extend the environment for the formal parameter but not for the function itself. Do not worry yet about dynamic type errors.
3. Add support for checking for dynamic type errors. This requires changes in the cases for function calls and (dis)equality operators.
4. Check that your interpreter, unfortunately, implements dynamic binding instead of static binding.
5. Modify your implementation to support recursive functions.

Problem 2 JakartaScript Interpreter: Substitution and Evaluation Order (28 Points)

In this problem, we will do two things. First, we will remove environments and instead use a language semantics based on substitution. This change will "fix" the scoping issue of dynamic binding, and we will end up with static binding.

As an aside, substitution is not the only way to static binding semantics. Another way is to represent a function value as a *closure*, which is an object that pairs the current environment at the point when the function is defined with a pointer to the function's definition. Substitution is a fairly simple way to get static binding, but in practice, it is rarely used because it is not the most efficient implementation.

Secondly, we will implement a small-step interpreter. A small-step interpreter makes explicit the evaluation order of expressions. These two changes are orthogonal, that is, one could implement a big-step interpreter using substitution or a small-step interpreter using environments.

(a) Implement

```
def subst(e: Expr, x: String, v: Val): Expr
```

that substitutes value `v` for all free occurrences of variable `x` in expression `e`. The function `subst` requires that `v` does not have free variables, so you do not need to worry about variable capturing. Still, be careful that you treat the variable binding constructs correctly (i.e., `ConstDecl` and `Function`) so that you only substitute free occurrences of `x` in `e` by `v`. In particular, calling `subst` on expression

```
x, (const x = 4; x)
```

with value `3` and variable `x` should return

```
3, (const x = 4; x)
```

not

```
3, (const x = 4; 3)
```

The function `subst` is a helper for the `step` function, but you might want to first implement all of the cases of `step` that do not depend on `subst`.

(b) Implement

```
def step(e: Expr): Expr
```

that performs one-step of evaluation by rewriting the input expression `e` into a "one-step reduced" expression. This one-step reduction should be implemented according to the small-step semantics $e \to e'$ defined in Figures 6, 7, and 8. We write $e[v/x]$ for substituting value v for all free occurrences of the variable x in expression e (i.e., this corresponds to `subst(`e`,`x`,`v`)` in the Scala implementation).

It is informative to compare the small-step semantics used in this problem and the big-step semantics from Problem 1. In particular, for all programs where dynamic binding is not an issue, your interpreters in this problem and the previous should behave the same. We have provided the functions `evaluate` and `iterateStep` that evaluate "top-level" expressions to a value using your interpreter implementations. Executing your interpreter on a JavaScript input file using the `run` command of `sbt` will evaluate the input program with both of these functions.

$$\frac{}{env \vdash v \Downarrow v}\ \textsc{EvalVal} \qquad \frac{env \vdash e_1 \Downarrow v_1 \quad \texttt{true} = toBool(v_1) \quad env \vdash e_2 \Downarrow v_2}{env \vdash e_1 \,\texttt{\&\&}\, e_2 \Downarrow v_2}\ \textsc{EvalAndTrue}$$

$$\frac{x \in \mathsf{dom}(env)}{env \vdash x \Downarrow env(x)}\ \textsc{EvalVar} \qquad \frac{env \vdash e_1 \Downarrow v_1 \quad \texttt{false} = toBool(v_1) \quad env \vdash e_2 \Downarrow v_2}{env \vdash e_1 \,\texttt{||}\, e_2 \Downarrow v_2}\ \textsc{EvalOrFalse}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2}{env \vdash e_1 \,\texttt{,}\, e_2 \Downarrow v_2}\ \textsc{EvalSeq} \qquad \frac{env \vdash e_1 \Downarrow v_1 \quad \texttt{false} = toBool(v_1)}{env \vdash e_1 \,\texttt{\&\&}\, e_2 \Downarrow v_1}\ \textsc{EvalAndFalse}$$

$$\frac{env \vdash e \Downarrow v \quad v \text{ printed}}{env \vdash \texttt{console.log}(e) \Downarrow \texttt{undefined}}\ \textsc{EvalPrint} \qquad \frac{env \vdash e_1 \Downarrow v_1 \quad \texttt{true} = toBool(v_1)}{env \vdash e_1 \,\texttt{||}\, e_2 \Downarrow v_1}\ \textsc{EvalOrTrue}$$

$$\frac{env \vdash e \Downarrow v \quad v' = \texttt{-}\, toNum(v)}{env \vdash \texttt{-}\, e \Downarrow v'}\ \textsc{EvalUMinus} \qquad \frac{env \vdash e \Downarrow v \quad v' = \texttt{!}\ toBool(v)}{env \vdash \texttt{!}\ e \Downarrow v'}\ \textsc{EvalNot}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = toNum(v_1) + toNum(v_2) \quad v_1, v_2 \notin Str}{env \vdash e_1 \,\texttt{+}\, e_2 \Downarrow v}\ \textsc{EvalPlusNum}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = v_1 + toString(v_2) \quad v_1 \in Str}{env \vdash e_1 \,\texttt{+}\, e_2 \Downarrow v}\ \textsc{EvalPlusStr}_1$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = toString(v_1) + v_2 \quad v_2 \in Str}{env \vdash e_1 \,\texttt{+}\, e_2 \Downarrow v}\ \textsc{EvalPlusStr}_1$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = toNum(v_1)\, bop\, toNum(v_2) \quad bop \in \{\texttt{*}, \texttt{/}, \texttt{-}\}}{env \vdash e_1\, bop\, e_2 \Downarrow v}\ \textsc{EvalArith}$$

$$\frac{env \vdash e_d \Downarrow v_d \quad env' = env[x \mapsto v_d] \quad env' \vdash e_b \Downarrow v_b}{env \vdash \texttt{const}\ x \,\texttt{=}\, e_d \,\texttt{;}\, e_b \Downarrow v_b}\ \textsc{EvalConstDecl}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = toNum(v_1)\ bop\ toNum(v_2) \quad v_1 \notin Str \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{env \vdash e_1\ bop\ e_2 \Downarrow v}\ \textsc{EvalInequalNum}_1$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad v = toNum(v_1)\ bop\ toNum(v_2) \quad v_2 \notin Str \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{env \vdash e_1\ bop\ e_2 \Downarrow v}\ \textsc{EvalInequalNum}_2$$

$$\frac{env \vdash e_1 \Downarrow s_1 \quad env \vdash e_2 \Downarrow s_2 \quad v = s_1\ bop\ s_2 \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{env \vdash e_1\ bop\ e_2 \Downarrow v}\ \textsc{EvalInequalStr}$$

$$\frac{\begin{array}{c} env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2 \quad b = (v_1\ bop\ v_2) \\ v_1 \neq \texttt{function}\ p_1(x_1)e_3 \quad v_1 \neq \texttt{function}\ p_2(x_2)e_4 \quad bop \in \{\texttt{===}, \texttt{!==}\} \end{array}}{env \vdash e_1\ bop\ e_2 \Downarrow b}\ \textsc{EvalEqual}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad toBool(v_1) = \texttt{true} \quad env \vdash e_2 \Downarrow v_2}{env \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_2}\ \textsc{EvalIfThen}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad toBool(v_1) = \texttt{false} \quad env \vdash e_3 \Downarrow v_3}{env \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_3}\ \textsc{EvalIfElse}$$

$$\frac{env \vdash e_1 \Downarrow x \,\texttt{=>}\, e \quad env \vdash e_2 \Downarrow v_2 \quad env' = env[x \mapsto v_2] \quad env' \vdash e \Downarrow v}{env \vdash e_1(e_2) \Downarrow v}\ \textsc{EvalCall}$$

$$\frac{env \vdash e_1 \Downarrow v_1 v_1 = \texttt{function}\ x_1(x_2)e \quad env \vdash e_2 \Downarrow v_2 \quad env' = env[x_1 \mapsto v_1][x_2 \mapsto v_2] \quad env' \vdash e \Downarrow v}{env \vdash e_1(e_2) \Downarrow v}\ \textsc{EvalCallRec}$$

Figure 4: Big-step operational semantics of JakartaScript (with dynamic binding)

$$\frac{env \vdash e_1 \Downarrow \texttt{function}\, p(x)e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorEqual}_1$$

$$\frac{env \vdash e_2 \Downarrow \texttt{function}\, p(x)e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorEqual}_2$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad v_1 \neq \texttt{function}\, p(x)e}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorCall}$$

$$\frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropIf} \qquad \frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1\, bop\, e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropBop}_1$$

$$\frac{env \vdash e \Downarrow \mathsf{typeerror}}{env \vdash uop\, e \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropUop} \qquad \frac{env \vdash e_2 \Downarrow \mathsf{typeerror} \quad bop \notin \{\texttt{\&\&}, \texttt{||}\}}{env \vdash e_1\, bop\, e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropBop}_2$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad v_1 \neq \mathsf{typeerror} \quad \texttt{true} = toBool(v_1) \quad env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1\, \texttt{\&\&}\, e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropAnd}$$

$$\frac{env \vdash e_1 \Downarrow v_1 \quad v_1 \neq \mathsf{typeerror} \quad \texttt{false} = toBool(v_1) \quad env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1\, \texttt{||}\, e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropOr}$$

$$\frac{env \vdash e \Downarrow \mathsf{typeerror}}{env \vdash \texttt{console.log}(e) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropPrint} \qquad \frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropCall}_1$$

$$\frac{env \vdash e_d \Downarrow \mathsf{typeerror}}{env \vdash \texttt{const}\ x = e_d ; e_b \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropConst} \qquad \frac{env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropCall}_2$$

Figure 5: Big-step operational semantics of JakartaScript: dynamic type error rules

$$\frac{e_1 \to e_1'}{e_1\, bop\, e_2 \to e_1'\, bop\, e_2}\ \textsc{SearchBop}_1 \qquad \frac{bop \notin \{\texttt{,}, \texttt{\&\&}, \texttt{||}, \texttt{===}, \texttt{!==}\} \quad e_2 \to e_2'}{v_1\, bop\, e_2 \to v_1\, bop\, e_2'}\ \textsc{SearchBop}_2$$

$$\frac{e_2 \to e_2' \quad v_1 \neq \texttt{function}\, p(x)e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{v_1\, bop\, e_2 \to v_1\, bop\, e_2'}\ \textsc{SearchEqual}$$

$$\frac{e \to e'}{uop\, e \to uop\, e'}\ \textsc{SearchUop} \qquad \frac{e \to e'}{\texttt{console.log}(e) \to \texttt{console.log}(e')}\ \textsc{SearchPrint}$$

$$\frac{e_1 \to e_1'}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_1'\ \texttt{?}\ e_2\ \texttt{:}\ e_3}\ \textsc{SearchIf} \qquad \frac{e_d \to e_d'}{\texttt{const}\ x = e_d ; e_b \to \texttt{const}\ x = e_d' ; e_b}\ \textsc{SearchConst}$$

$$\frac{e_1 \to e_1'}{e_1(e_2) \to e_1'(e_2)}\ \textsc{SearchCall}_1 \qquad \frac{e_2 \to e_2'}{v_1(e_2) \to v_1(e_2')}\ \textsc{SearchCall}_2$$

Figure 6: Small-step operational semantics of JakartaScript: search rules

$$\frac{v' = -\mathit{toNum}(v)}{\texttt{-}\, v \to v'}\ \textsc{DoUMinus} \qquad \frac{v' = !\mathit{toBool}(v)}{\texttt{!}\ v \to v'}\ \textsc{DoNot} \qquad \frac{}{v_1\ \texttt{,}\ e_2 \to e_2}\ \textsc{DoSeq}$$

$$\frac{v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2) \quad v_1, v_2 \notin \mathit{Str}}{v_1 \texttt{+} v_2 \to v}\ \textsc{DoPlusNum}$$

$$\frac{v = s_1 + \mathit{toStr}(v_2)}{s_1 \texttt{+} v_2 \to v}\ \textsc{DoPlusStr}_1 \qquad \frac{v = \mathit{toStr}(v_1) + s_2}{v_1 \texttt{+} s_2 \to v}\ \textsc{DoPlusStr}_2$$

$$\frac{v = \mathit{toNum}(v_1)\, \mathit{bop}\, \mathit{toNum}(v_2) \quad \mathit{bop} \in \{\texttt{*}, \texttt{-}, \texttt{/}\}}{v_1\, \mathit{bop}\, v_2 \to v}\ \textsc{DoArith}$$

$$\frac{b = \mathit{toNum}(v_1)\ \mathit{bop}\ \mathit{toNum}(v_2) \quad \mathit{bop} \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\} \quad v_1 \notin \mathit{Str}}{v_1\ \mathit{bop}\ v_2 \to b}\ \textsc{DoInequalNum}_1$$

$$\frac{b = \mathit{toNum}(v_1)\ \mathit{bop}\ \mathit{toNum}(v_2) \quad \mathit{bop} \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\} \quad v_2 \notin \mathit{Str}}{v_1\ \mathit{bop}\ v_2 \to b}\ \textsc{DoInequalNum}_2$$

$$\frac{b = s_1\ \mathit{bop}\ s_2 \quad \mathit{bop} \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{s_1\ \mathit{bop}\ s_2 \to b}\ \textsc{DoInequalStr}$$

$$\frac{v_1 \neq \textbf{function}\ p_1(x_1)e_1 \quad v_2 \neq \textbf{function}\ p_2(x_2)e_2 \quad b = v_1\ \mathit{bop}\ v_2 \quad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{v_1\ \mathit{bop}\ v_2 \to b}\ \textsc{DoEqual}$$

$$\frac{\mathit{toBool}(v_1) = \textbf{true}}{v_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_2}\ \textsc{DoIfThen} \qquad \frac{\mathit{toBool}(v_1) = \textbf{false}}{v_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_3}\ \textsc{DoIfElse}$$

$$\frac{\textbf{false} = \mathit{toBool}(v_1)}{v_1\, \texttt{\&\&}\, e_2 \to v_1}\ \textsc{DoAndFalse} \qquad \frac{\textbf{true} = \mathit{toBool}(v_1)}{v_1\, \texttt{\&\&}\, e_2 \to e_2}\ \textsc{DoAndTrue}$$

$$\frac{\textbf{true} = \mathit{toBool}(v_1)}{v_1\ \texttt{||}\ e_2 \to v_1}\ \textsc{DoOrTrue} \qquad \frac{\textbf{false} = \mathit{toBool}(v_1)}{v_1\ \texttt{||}\ e_2 \to e_2}\ \textsc{DoOrFalse}$$

$$\frac{}{\textbf{const}\ x \texttt{=} v_d \texttt{;} e_b \to e_b[v_d/x]}\ \textsc{DoConst} \qquad \frac{v_1 = x\ \texttt{=>}\ e_1}{v_1(v_2) \to e_1[v_2/x]}\ \textsc{DoCall}$$

$$\frac{v \text{ printed}}{\textbf{console.log}(v) \to \textbf{undefined}}\ \textsc{DoPrint} \qquad \frac{v_1 = \textbf{function}\ x_1(x_2)e_1}{v_1(v_2) \to e_1[v_1/x_1][v_2/x_2]}\ \textsc{DoCallRec}$$

Figure 7: Small-step operational semantics of JakartaScript: do rules

$$\frac{v_1 = \textbf{function}\ p(x)e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{v_1\ bop\ e_2 \rightarrow \textsf{typeerror}}\ \textsc{TypeErrorEqual}_1$$

$$\frac{v_2 = \textbf{function}\ p(x)e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{v_1\ bop\ v_2 \rightarrow \textsf{typeerror}}\ \textsc{TypeErrorEqual}_2$$

$$\frac{v_1 \neq \textbf{function}\ p(x)e}{v_1(e_2) \rightarrow \textsf{typeerror}}\ \textsc{TypeErrorCall}$$

$$\frac{}{\textsf{typeerror}\ ?\ e_2\ :\ e_3 \rightarrow \textsf{typeerror}}\ \textsc{PropIf} \qquad \frac{}{\textsf{typeerror}\ bop\ e_2 \rightarrow \textsf{typeerror}}\ \textsc{PropBop}_1$$

$$\frac{}{uop\ \textsf{typeerror} \rightarrow \textsf{typeerror}}\ \textsc{PropUop} \qquad \frac{}{e_1\ bop\ \textsf{typeerror} \rightarrow \textsf{typeerror}}\ \textsc{PropBop}_2$$

$$\frac{}{\textbf{console.log}(\textsf{typeerror}) \rightarrow \textsf{typeerror}}\ \textsc{PropPrint} \qquad \frac{}{\textsf{typeerror}(e_2) \rightarrow \textsf{typeerror}}\ \textsc{PropCall}_1$$

$$\frac{}{\textbf{const}\ x = \textsf{typeerror}; e_b \rightarrow \textsf{typeerror}}\ \textsc{PropConst}_1 \qquad \frac{}{v_1(\textsf{typeerror}) \rightarrow \textsf{typeerror}}\ \textsc{PropCall}_2$$

Figure 8: Small-step operational semantics of JakartaScript: dynamic type error rules