CSCI 3155: Lab Assignment 3

Spring 2013: Due Saturday, March 2, 2013

The purpose of this lab is to grapple with how dynamic scoping arises and how to formally specifying semantics. Concretely, we will extend JavaScripty with recursive functions and implement two interpreters. The first will be a big-step interpreter that is an extension of Lab 2 but implements dynamic scoping "by accident." The second will be a small-step interpreter that exposes evaluation order and implements static scoping by substitution.

Like last time, find a partner. You will work on this assignment in pairs. However, note that **each student needs to submit a write-up** 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. 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 unfamilar to you. Finally, make sure that your file compiles and runs (using Scala 2.9.2 or Scala 2.10.0). A program that does not compile will *not* be graded.

Submission Instructions. First, submit your `.scala` file to the auto-testing system to get instant feedback on your code. You may submit as many times as you want, but please submit at least once before the deadline. We will only look at your last submission.

Then, upload to the moodle exactly two files named as follows:

- `Lab3.md` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- `Lab3.scala` with your answers to the coding exercises

Getting Started. Included in the Lab3 folder in the repository is the code template `Lab3.scala` for the assignment. Be sure to write your name, your partner, and collaborators in the header of the code template.

1. **Feedback**. Complete the survey in the survey.md file. Any non-empty answer will receive full credit.

expressions	$e ::= x \mid n \mid b \mid \textbf{undefined} \mid uop\ e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$
	$\mid \textbf{const}\ x = e_1; e_2 \mid \textbf{jsy.print}(e_1)$
	$\mid str \mid \textbf{function}\ p(x)\ e_1 \mid e_1(e_2) \mid \mathsf{typeerror}$
values	$v ::= n \mid b \mid \textbf{undefined} \mid str \mid \textbf{function}\ p(x)\ e_1$
unary operators	$uop ::= \texttt{-} \mid \texttt{!}$
binary operators	$bop ::= \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{\|\|}$
variables	x
numbers (doubles)	n
booleans	$b ::= \textbf{true} \mid \textbf{false}$
strings	str
function names	$p ::= x \mid \varepsilon$
value environments	$E ::= \cdot \mid E[x \mapsto v]$

Figure 1: Abstract Syntax of JavaScripty

2. JavaScripty Interpreter: Tag Testing, Recursive Functions, and Dynamic Scoping

We now have the formal tools to specify exactly how a JavaScripty program should behave. Unless otherwise specified, we will continue to try to match JavaScript semantics as implemented by Node.js/Google's V8 JavaScript Engine. Thus, it is still useful to write little test JavaScript programs and run it through Node.js to see how the test should behave. Finding bugs in the JavaScripty specification with respect to JavaScript is certainly deserving of extra credit.

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

For this question, we try to implement functions as an extension of Lab 2 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 scoping.

The syntax of JavaScripty for this lab 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 string literals *str*, function expressions **function** $p(x)\ e_1$, and function calls $e_1(e_2)$.

In a function expression, the function name p can either be an identifier or empty. The identifier is used for recursion. For simplicity, all functions are one argument functions. Since functions are first-class values, we can get multi-argument functions via currying.

We have also added a "marker" typeerror to the expression and value languages. These are not part of the source language but arise during evaluation. Its purpose will be explained further below.

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.

$$
\begin{array}{lll}
\text{statements} & s ::= \textbf{const}\ x = e \mid e \mid \{\ s_1\ \} \mid ; \mid s_1\ s_2 \\
\text{expressions} & e ::= \cdots \mid \textbf{const}\ x = e_1; e_2 \mid (e_1) \\
& \quad \mid \textbf{function}\ p(x)\ e_1 \mid \textbf{function}\ p(x)\ \{\ s_1\ \textbf{return}\ e_1\ \}
\end{array}
$$

Figure 2: Concrete Syntax of JavaScripty

```
case class Function(p: Option[String], x: String, e1: Expr) extends Expr
  Function(p, x, e1)   function p(x) e1
case class Call(e1: Expr, e2: Expr) extends Expr
  Call(e1, e2)   e1(e2)
```

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.

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 JavaScripty is given in Figure 4. This figure may be one of the first times that you are reading a formal semantics of a programming language. It may seem daunting at first, but it will be become easier with practice. This lab 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 judgment form $E \vdash e \Downarrow v$, which says informally, "In value environment E, expression e evaluates to value v." This relation has three parameters: E, e, and v. You can think of the other parts of the judgment as just punctuation. This judgment form corresponds directly to the `eval` function that we are asked to implement (not a coincidence). It similarly has three parts:

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

It takes as input a value environment `env` (E) and an expression `e` (e) returns a value v.

It is very informative to compare your Scala code for `env` with the inference rules that define $E \vdash e \Downarrow v$. One thing you should observe is that all of the rules are implemented, except EvalPlusString$_1$, EvalPlusString$_2$, and EvalCall. In essence, implementing those rules is your task for this question.

In Lab 2, all expressions could be evaluated to something (because of conversions). 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 JavaScripty, calling a non-function raises a run-time error. In the formal semantics, we model this with evaluating to the "marker" typeerror.

$\boxed{E \vdash e \Downarrow v}$

$$\textsc{EvalVar}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \textsc{EvalVal}\ \frac{}{E \vdash v \Downarrow v}$$

$$\textsc{EvalNeg}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad n' = -\,\text{toNumber}(v_1)}{E \vdash -\,e_1 \Downarrow n'} \qquad \textsc{EvalNot}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad b' = \neg\,\text{toBoolean}(v_1)}{E \vdash !\,e_1 \Downarrow b'}$$

$$\textsc{EvalSeq}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ ,\ e_2 \Downarrow v_2}$$

$$\textsc{EvalPlusNumber}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad n' = \text{toNumber}(v_1) + \text{toNumber}(v_2) \qquad v_1 \neq \mathit{str}_1 \qquad v_2 \neq \mathit{str}_2}{E \vdash e_1 + e_2 \Downarrow n'}$$

$$\textsc{EvalPlusString}_1\ \frac{E \vdash e_1 \Downarrow \mathit{str}_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad \mathit{str}' = \mathit{str}_1 + \text{toString}(v_2)}{E \vdash e_1 + e_2 \Downarrow \mathit{str}'} \qquad \textsc{EvalPlusString}_2\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow \mathit{str}_2 \qquad \mathit{str}' = \text{toString}(v_1) + \mathit{str}_2}{E \vdash e_1 + e_2 \Downarrow \mathit{str}'}$$

$$\textsc{EvalArith}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad n' = \text{toNumber}(v_1)\ \mathit{bop}\ \text{toNumber}(v_2) \qquad \mathit{bop} \in \{-, *, /\}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow n'}$$

$$\textsc{EvalInequalityNumber}\ \frac{\begin{array}{c}E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \\ b' = \text{toNumber}(v_1)\ \mathit{bop}\ \text{toNumber}(v_2) \qquad v_1 \neq \mathit{str}_1 \qquad v_2 \neq \mathit{str}_2 \qquad \mathit{bop} \in \{<, <=, >, >=\}\end{array}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow b'}$$

$$\textsc{EvalInequalityString}_1\ \frac{E \vdash e_1 \Downarrow \mathit{str}_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad b' = \mathit{str}_1\ \mathit{bop}\ \text{toString}(v_2) \qquad \mathit{bop} \in \{<, <=, >, >=\}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow b'}$$

$$\textsc{EvalInequalityString}_2\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow \mathit{str}_2 \qquad b' = \text{toString}(v_1)\ \mathit{bop}\ \mathit{str}_2 \qquad \mathit{bop} \in \{<, <=, >, >=\}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow b'}$$

$$\textsc{EvalEquality}\ \frac{\begin{array}{c}E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \\ v_1 \neq \textbf{function}\ p_1(x_1)\ e_1 \qquad v_2 \neq \textbf{function}\ p_1(x_2)\ e_2 \qquad b' = (v_1\ \mathit{bop}\ v_2) \qquad \mathit{bop} \in \{===, !==\}\end{array}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow b'}$$

$$\textsc{EvalAndTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{true} = \text{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ \&\&\ e_2 \Downarrow v_2} \qquad \textsc{EvalAndFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{false} = \text{toBoolean}(v_1)}{E \vdash e_1\ \&\&\ e_2 \Downarrow \textbf{false}}$$

$$\textsc{EvalOrTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{true} = \text{toBoolean}(v_1)}{E \vdash e_1\ ||\ e_2 \Downarrow \textbf{true}} \qquad \textsc{EvalOrFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{false} = \text{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ ||\ e_2 \Downarrow v_2}$$

$$\textsc{EvalPrint}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1\ \text{printed}}{E \vdash \textbf{jsy.print}(e_1) \Downarrow \textbf{undefined}} \qquad \textsc{EvalIfTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{true} = \text{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ ?\ e_2 : e_3 \Downarrow v_2}$$

$$\textsc{EvalIfFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \textbf{false} = \text{toBoolean}(v_1) \qquad E \vdash e_3 \Downarrow v_3}{E \vdash e_1\ ?\ e_2 : e_3 \Downarrow v_3} \qquad \textsc{EvalConst}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{E \vdash \textbf{const}\ x = e_1;\ e_2 \Downarrow v_2}$$

$$\textsc{EvalCall}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 = \textbf{function}\ (x)\ e' \qquad E \vdash e_2 \Downarrow v_2 \qquad E[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1(e_2) \Downarrow v'}$$

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

Figure 4: Big-step operational semantics of JavaScripty (with dynamic scoping).

$$
\begin{array}{rcll}
\text{toNumber}(n) & \stackrel{\text{def}}{=} & n & \\
\text{toNumber}(\textbf{true}) & \stackrel{\text{def}}{=} & 1 & \\
\text{toNumber}(\textbf{false}) & \stackrel{\text{def}}{=} & 0 & \\
\text{toNumber}(\textbf{undefined}) & \stackrel{\text{def}}{=} & \mathsf{NaN} & \\
\text{toNumber}(\mathit{str}) & \stackrel{\text{def}}{=} & \text{parse } \mathit{str} & \\
\text{toNumber}(\textbf{function } p(x)\ e_1) & \stackrel{\text{def}}{=} & \mathsf{NaN} & \\
 & & & \\
\text{toBoolean}(n) & \stackrel{\text{def}}{=} & \textbf{false} & \text{if } n = 0 \text{ or } n = \mathsf{NaN} \\
\text{toBoolean}(n) & \stackrel{\text{def}}{=} & \textbf{true} & \text{otherwise} \\
\text{toBoolean}(b) & \stackrel{\text{def}}{=} & b & \\
\text{toBoolean}(\textbf{undefined}) & \stackrel{\text{def}}{=} & \textbf{false} & \\
\text{toBoolean}(\mathit{str}) & \stackrel{\text{def}}{=} & \textbf{false} & \text{if } \mathit{str} = \text{""} \\
\text{toBoolean}(\mathit{str}) & \stackrel{\text{def}}{=} & \textbf{true} & \text{otherwise} \\
\text{toBoolean}(\textbf{function } p(x)\ e_1) & \stackrel{\text{def}}{=} & \textbf{true} & \\
 & & & \\
\text{toString}(n) & \stackrel{\text{def}}{=} & \text{string of } n & \\
\text{toString}(\textbf{true}) & \stackrel{\text{def}}{=} & \text{"}\texttt{true}\text{"} & \\
\text{toString}(\textbf{false}) & \stackrel{\text{def}}{=} & \text{"}\texttt{false}\text{"} & \\
\text{toString}(\textbf{undefined}) & \stackrel{\text{def}}{=} & \text{"}\texttt{undefined}\text{"} & \\
\text{toString}(\mathit{str}) & \stackrel{\text{def}}{=} & \mathit{str} & \\
\text{toString}(\textbf{function } p(x)\ e_1) & \stackrel{\text{def}}{=} & \text{"}\texttt{function}\text{"} &
\end{array}
$$

Figure 5: Conversion functions. We do not specify explicitly the parsing or string conversion of numbers. The conversion of a function to a string deviates slightly from JavaScript where the source code of the function is returned.

$$\boxed{E \vdash e \Downarrow v}$$

$$\textsc{EvalTypeErrorEquality}_1\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 = \textbf{function } p_1(x_1)\ e_1 \qquad \mathit{bop} \in \{===, !==\}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalTypeErrorEquality}_2\ \frac{E \vdash e_2 \Downarrow v_2 \qquad v_2 = \textbf{function } p_1(x_1)\ e_1 \qquad \mathit{bop} \in \{===, !==\}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalTypeErrorCall}\ \frac{v_1 \neq \textbf{function } p(x)\ e_1}{v_1(e_2) \longrightarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropagateUnary}\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash \mathit{uop}\ e_1 \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropagateBinary}_1\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropagateBinary}_2\ \frac{E \vdash e_2 \Downarrow \mathsf{typeerror}}{E \vdash e_1\ \mathit{bop}\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropagatePrint}\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash \textbf{print}(e_1) \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropagateIf}\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash e_1\ ?\ e_2 : e_3 \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropagateConst}\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash \textbf{const}\ x = e_1; e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropagateCall}_1\ \frac{E \vdash e_1 \Downarrow \mathsf{typeerror}}{E \vdash e_1(e_2) \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropagateCall}_2\ \frac{E \vdash e_2 \Downarrow \mathsf{typeerror}}{E \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}$$

Figure 6: Big-step operational semantics of JavaScripty: Dynamic type error rules.

Such a run-time error is known 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`:

```
class DynamicTypeError(e: Expr) extends Exception
```

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

```
throw new DynamicTypeError(e)
```

The argument should be the input expression to `eval` where the type error was detected. Another advantage of using a Scala exception for typeerror is that the marker does not need to be propagated explicitly (as in the inference rules).

Note in rule EvalEquality, 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 JavaScripty 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.

Give one test case that behaves differently under dynamic scoping versus static scoping (and does not crash). Explain the test case and how they behave differently in your write-up.

(b) Then, implement

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

that evaluates a JavaScripty expression `e` in a value environment `env` to a value according to the evaluation judgment $E \vdash e \Downarrow v$.

The following helper functions for converting values to numbers, booleans, and strings are provided:

```
def toNumber(v: Expr): Double
def toBoolean(v: Expr): Boolean
def toString(v: Expr): String
```

We suggest the following step-by-step process:

1. Extend your Lab 2 implementation with strings. The special string operations that we support is string concatenation via the $e_1 + e_2$ expression and the lexicographical ordering via the $<$, $<=$, $>$, and $>=$ operators. Thus, the changes you need to make is for the $e_1 + e_2$ expression case and the ordering operator cases.

2. Get your implementation working with non-recursive functions. On function calls, you need to extend the environment for the formal parameter but not for the function itself.
3. Modify your implementation to support recursive functions.

3. **JavaScripty Interpreter: Substitution and Evaluation Order**

In this question, 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, and we will end up with static, lexical scoping.

As an aside, substitution is not the only way to "fix" the scoping issue. Another way is to represent function values as *closures*, which is a pair of the function with the environment when it is defined. Substitution is a fairly simple way to get lexical scoping, but in practice, it is often not used because it is not the most efficient implementation.

The second thing that we do is move to implementing 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 substitute(e: Expr, v: Expr, x: String): Expr
```

that substitutes value `v` for all *free* occurrences of variable `x` in expression `e`. We advise defining `substitute` by recursion on `e`. The cases to be careful about are `ConstDecl` and `Function` because these are the variable binding constructs. In particular, `substitute` on expression

```
a; (const a = 4; a)
```

with value `3` for variable "`a`" should return

```
3; (const a = 4; a)
```

not

```
3; (const a = 4; 3)
```

This function is a helper for the `step` function, but you might want to implement all of the cases of `step` that do not require `substitute` first.

(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 judgment form $e \longrightarrow e'$ defined in Figures 7, 8, and 9. We write $e[v/x]$ for substituting value v for all free occurrences of the variable x in expression e (i.e., a call to `substitute`).

(c) Explain whether the evaluation order is deterministic.

$\boxed{e \longrightarrow e'}$

$$\textsc{DoNeg}\ \frac{n' = -\,\text{toNumber}(v)}{-\,v \longrightarrow n'} \qquad \textsc{DoNot}\ \frac{b' = \neg\,\text{toBoolean}(v)}{!\,v \longrightarrow b'} \qquad \textsc{DoSeq}\ \frac{}{v_1\ ,\ e_2 \longrightarrow e_2}$$

$$\textsc{DoPlusNumber}\ \frac{n' = \text{toNumber}(v_1) + \text{toNumber}(v_2) \qquad v_1 \neq str_1 \qquad v_2 \neq str_2}{v_1 + v_2 \longrightarrow n'} \qquad \textsc{DoPlusString}_1\ \frac{str' = str_1 + \text{toString}(v_2)}{str_1 + v_2 \longrightarrow str'}$$

$$\textsc{DoPlusString}_2\ \frac{str' = \text{toString}(v_1) + str_2}{v_1 + str_2 \longrightarrow str'} \qquad \textsc{DoArith}\ \frac{n' = \text{toNumber}(v_1)\ bop\ \text{toNumber}(v_2) \qquad bop \in \{-, *, /\}}{v_1\ bop\ v_2 \longrightarrow n'}$$

$$\textsc{DoInequalityNumber}\ \frac{b' = \text{toNumber}(v_1)\ bop\ \text{toNumber}(v_2) \qquad bop \in \{<, <=, >, >=\} \qquad v_1 \neq str_1 \qquad v_2 \neq str_2}{v_1\ bop\ v_2 \longrightarrow b'}$$

$$\textsc{DoInequalityString}_1\ \frac{b' = str_1\ bop\ \text{toString}(v_2) \qquad bop \in \{<, <=, >, >=\}}{str_1\ bop\ v_2 \longrightarrow b'}$$

$$\textsc{DoInequalityString}_2\ \frac{b' = \text{toString}(v_1)\ bop\ str_2 \qquad bop \in \{<, <=, >, >=\}}{v_1\ bop\ str_2 \longrightarrow b'}$$

$$\textsc{DoEquality}\ \frac{v_1 \neq \textbf{function}\ p_1(x_1)\ e_1 \qquad v_2 \neq \textbf{function}\ p_1(x_2)\ e_2 \qquad b' = (v_1\ bop\ v_2) \qquad bop \in \{===, !==\}}{v_1\ bop\ v_2 \longrightarrow b'}$$

$$\textsc{DoAndTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ \&\&\ e_2 \longrightarrow e_2} \qquad \textsc{DoAndFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ \&\&\ e_2 \longrightarrow \textbf{false}} \qquad \textsc{DoOrTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ ||\ e_2 \longrightarrow \textbf{true}}$$

$$\textsc{DoOrFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ ||\ e_2 \longrightarrow e_2} \qquad \textsc{DoPrint}\ \frac{v_1\ \text{printed}}{\textbf{jsy.print}(v_1) \longrightarrow \textbf{undefined}} \qquad \textsc{DoIfTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ ?\ e_2 : e_3 \longrightarrow e_2}$$

$$\textsc{DoIfFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ ?\ e_2 : e_3 \longrightarrow e_3} \qquad \textsc{DoConst}\ \frac{}{\textbf{const}\ x = v_1; e_2 \longrightarrow e_2[v_1/x]} \qquad \textsc{DoCall}\ \frac{v_1 = \textbf{function}\ (x)\ e_1}{v_1(v_2) \longrightarrow e_1[v_2/x]}$$

$$\textsc{DoCallRec}\ \frac{v_1 = \textbf{function}\ x_1(x_2)\ e_1}{v_1(v_2) \longrightarrow e_1[v_1/x_1][v_2/x_2]}$$

Figure 7: Small-step operational semantics of JavaScripty: Do rules.

$\boxed{e \longrightarrow e'}$

$$\textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{uop\ e_1 \longrightarrow uop\ e_1'} \qquad \textsc{SearchBinary}_1\ \frac{e_1 \longrightarrow e_1'}{e_1\ bop\ e_2 \longrightarrow e_1'\ bop\ e_2} \qquad \textsc{SearchBinaryArith}_2\ \frac{e_2 \longrightarrow e_2' \qquad bop \in \{+, -, *, /, <, <=, >, >=\}}{v_1\ bop\ e_2 \longrightarrow v_1\ bop\ e_2'}$$

$$\textsc{SearchEquality}_2\ \frac{e_2 \longrightarrow e_2' \qquad v_1 \neq \textbf{function}\ p(x)\ e_1 \qquad bop \in \{===, !==\}}{v_1\ bop\ e_2 \longrightarrow v_1\ bop\ e_2'} \qquad \textsc{SearchPrint}\ \frac{e_1 \longrightarrow e_1'}{\textbf{jsy.print}(e_1) \longrightarrow \textbf{jsy.print}(e_1')}$$

$$\textsc{SearchIf}\ \frac{e_1 \longrightarrow e_1'}{e_1\ ?\ e_2 : e_3 \longrightarrow e_1'\ ?\ e_2 : e_3} \qquad \textsc{SearchConst}\ \frac{e_1 \longrightarrow e_1'}{\textbf{const}\ x = e_1; e_2 \longrightarrow \textbf{const}\ x = e_1'; e_2} \qquad \textsc{SearchCall}_1\ \frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)}$$

$$\textsc{SearchCall}_2\ \frac{e_2 \longrightarrow e_2'}{\big(\textbf{function}\ p(x)\ e_1\big)(e_2) \longrightarrow \big(\textbf{function}\ p(x)\ e_1\big)(e_2')}$$

Figure 8: Small-step operational semantics of JavaScripty: Search rules.

$\boxed{e \longrightarrow e'}$

$$\textsc{TypeErrorEquality}_1\ \frac{bop \in \{===, !==\}}{(\textbf{function}\ p(x)\ e_1)\ bop\ e_2 \longrightarrow \mathsf{typeerror}} \qquad \textsc{TypeErrorEquality}_1\ \frac{bop \in \{===, !==\}}{v_1\ bop\ (\textbf{function}\ p(x)\ e_2) \longrightarrow \mathsf{typeerror}}$$

$$\textsc{TypeErrorCall}\ \frac{v_1 \neq \textbf{function}\ p(x)\ e_1}{v_1(e_2) \longrightarrow \mathsf{typeerror}}$$

$$\textsc{PropagateUnary}\ \frac{}{uop\ \mathsf{typeerror} \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateBinary}\ \frac{}{\mathsf{typeerror}\ bop\ e_2 \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateBinary}\ \frac{}{v_1\ bop\ \mathsf{typeerror} \longrightarrow \mathsf{typeerror}}$$

$$\textsc{PropagatePrint}\ \frac{}{\textbf{jsy.print}(\mathsf{typeerror}) \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateIf}\ \frac{}{\mathsf{typeerror}\ ?\ e_2 : e_3 \longrightarrow \mathsf{typeerror}}$$

$$\textsc{PropagateConst}\ \frac{}{\textbf{const}\ x = \mathsf{typeerror}; e_2 \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateCall}_1\ \frac{}{\mathsf{typeerror}(e_2) \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateCall}_2\ \frac{}{v_1(\mathsf{typeerror}) \longrightarrow \mathsf{typeerror}}$$

Figure 9: Small-step operational semantics of JavaScripty: Dynamic type error rules.

It is informative to compare the small-step semantics used in this question and the big-step semantics from the previous one. In particular, for all programs where dynamic scoping is not an issue, your interpreters in this question 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.

4. **Evaluation Order**.

   (a) **Scala**. Calls are amongst the most complicated expression constructs in programming languages. There are many, many different ways of supporting calls and different languages often differ in their support for calls. In this question, we will consider only the operand evaluation order issues with calls; later in the semester we will consider other issues.

   Let's consider a call schematically $F(A_1, \ldots, A_n)$. F is the procedure being called. While we are used to calling a procedures directly (e.g., `print()`), most languages, including Scala, allow complicated expressions for F. For example, if `a` is an array of procedures, one can do `a._2(5)` in Scala to call the procedure at `a._2` with argument `5`. Scala also allows a procedure to return a procedure (i.e., has higher-order functions). Thus, if a procedure `f` returns a procedure, then one could write `f()(x)`. This calls the procedure returned by `f` with the argument `x`. Needless to say, most languages allow one to pass arbitrarily complicated arguments (i.e., A_i).

      i. Discover whether Scala evaluates the procedure being called before it evaluates its arguments. For example, with `f()(x)` does Scala call the function `f` first or does it evaluate `x` first (i.e., looks up the value of `x`).

      ii. Discover whether or not Scala evaluates arguments to a call from left-to-right.

   In your write-up, explain how you came to a conclusion. Provide the code and output you used to determine your conclusion. You should provide evidence that supports your argument.

   (b) **JavaScripty**. Consider the small-step operational semantics for JavaScripty shown in Figures 7, 8, and 9. What is the evaluation order for $e_1 + e_2$? Explain.

5. **Short-Circuit Evaluation**. In this question, we will discuss some issues with short-circuit evaluation.

   (a) **Concept**. Give an example that illustrates the usefulness of short-circuit evaluation. Explain your example.

   (b) **Scala**. Discover and indicate whether or not Scala uses short-circuit evaluation of boolean operators (i.e., `&&`). You should provide evidence that supports your argument, including code listing and program output.

   (c) **JavaScripty**. Consider the small-step operational semantics for JavaScripty shown in Figures 7, 8, and 9. Does e_1 && e_2 short circuit? Explain.