CSCI 3155: Lab Assignment 4

Fall 2022
Checkpoint due Friday, October 14, 2022
Assignment due Friday, October 21, 2022

Learning Goals. The primary learning goals of this lab are to understand the following:

- programming with higher-order functions;
- static type checking and the interplay between type checking and evaluation;
- parameter passing modes: call-by-value versus call-by-name; and
- capture-avoiding substitution.

PL Ideas Static type checking and type safety. Records. Parameter passing modes.

FP Skills Higher-order functions. Collections and callbacks.

Concretely, we will extend JAVASCRIPTY with (immutable) objects and extend our small-step interpreter from Lab 3. Immutable objects are also called *records* in other languages. 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. In JavaScript, fields are called *properties* that can also be accessed via run-time strings. Since we will not support dynamic-property access, we will stick with the term "fields" used in more static languages. 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.

Parameters are always passed by value in JavaScript/TypeScript, so the parameter passing modes in JAVASCRIPTY is an extension beyond JavaScript/TypeScript. In particular, we consider parameter passing modes primarily to illustrate a language design decision and how the design decision manifests in the operational semantics.

Peer Teaching and Pair Programming. To be able to have someone to work closely with on the assignment, the instructor will randomly assign partners for the lab assignment; you will get a different partner for every lab assignment. You can pair program or work as closely as you like with your partner. However, note that **each student needs to submit** on Canvas, and you are individually responsible for your learning from the entirety of the assignment so that you can do well in your interview.

General Guidelines. You are welcome to talk about these questions beyond your teams. However, we ask that you code in pairs. See the collaboration policy for details, including the following:

> *Bottom line, feel free to use resources that are available to you as long as the use is **reasonable** and you **cite** them in your submission. However, copying answers directly or indirectly from solution manuals, web pages, or your peers is certainly unreasonable.*

Also, 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 preconditions 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 (e.g., with `sbt compile`). A program that does not compile will *not* be graded—no interview will be conducted.

Submission Instructions. We are using GitHub for assignment distribution and some autograding. You need to have a GitHub identity and must have your full name in your GitHub profile so that we can associate you with your submissions.

You will be editing/creating and submitting the the following files:

- `src/main/scala/jsy/student/Lab4.scala` with your solution to the coding exercises;
- `src/test/resources/lab4/mytest_`*somedescription*`.jsy` with a challenging test case for your JAVASCRIPTY interpreter.

There is no required write up in this lab. You are also likely to edit

- `src/test/scala/jsy/student/Lab4Spec.scala` with your additional tests;
- `src/main/scala/jsy/student/Lab4.worksheet.sc` for any scratch work; and

- `src/main/scala/jsy/student/Lab4.worksheet.js` for JavaScript experimentation.

You can also add JAVASCRIPTY tests in

- `src/test/resources/lab4/` as a pair of a `*.jsy` and a `*.ans` for a JAVASCRIPTY expression and the expected value of that expression, respectively.

Following good git practice, please make commits in small bits corresponding to completing small conceptual parts and push often so that your progress is evident. We expect that you have some familiarity with git from prior courses. If not, please discuss with your classmates and the course staff (e.g., via Piazza).

At any point, you may push your updated files to your GitHub repository for auto-grading. You need to push to your GitHub repository for the auto-testing part of your score, as well as to continue to the interview.

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 score. Please take advantage of your interview time to maximize the feedback that you are able to receive. Arrive at your interview ready with a working development environment to show your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Finally, upload the zip file of your repo to Canvas by clicking the `Code` button and then `Download ZIP`. The generated file should be named *your-lab-repo-name*`-main.zip`.

Getting Started. First, form a team of two and pick a team name. For our bookkeeping, please prefix your team name with the lab number and your CU IdentiKeys (i.e., your team should look something like `L4_`*your-identikey1*`_`*your-identikey2*`_anatomists`).

You must work in teams of two, and you will form teams in lab section. If you cannot connect with your partner, then please contact the course staff (via Piazza).

Then, log into Canvas and follow the GitHub Classroom link for setting up your Lab 4 repository with your team name. The first person will create the team, and the second person will select the team name from the existing team names. If you need to move teams after you have already created or joined a repository, you will need to contact the Course Manager (via Piazza) or work with a course staff member to move you manually.

If you would like to look at the code before getting your own copy from GitHub Classroom, you may go to `https://github.com/csci3155/pppl-lab4`.

Correctness and Testing. While some test cases are provided to you, the correctness of your implementations is defined by the specification in this handout. In addition to the test cases provided to you directly in your GitHub repository, the **auto-grading may run additional tests** to which you do not have direct access. The auto-grading runs when you push changes to your GitHub repository, which offers you some quick, iterative, and repeatable feedback, while guiding you to think deeply about your code. While you do not have direct access to the actual input itself, each failing test is named, and you can use that information to get a sense of issues in your code to come up with your own test cases for testing those scenarios. The point here is that you won't always have every possible test case provided to you — both in a literal sense for this lab and in industry in general — so getting practice in developing test cases to

drive your coding is crucially important. A starting point is to look at the test cases provided to you directly and think about what isn't already being tested. Then, add some tests relating to these. It's impossible to test exhaustively, so try to find "edge" cases that test tricky aspects of your implementation. As the course progresses, you are gradually given fewer tests with the expectation that you are gradually better at creating your own test suites.

Checkpoint. The checkpoint is to encourage you to start the assignment early and it requires you to submit (i.e., push) your partial solution on GitHub a week before the assignment is due. You do not need to attempt everything a week early, but we want you to start working on it and make note in this handout any required questions in the checkpoint. This means that submitting the empty template that fails all tests is **not sufficient**. Failing to submit to the checkpoint will prevent you from proceeding to the interview.

1. **Feedback**. Complete the survey on the linked from the Canvas after completing this assignment. Any non-empty answer will receive full credit.

2. (Advised Completion: Week 1) **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](l: List[A])(f: A => Option[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(List(1,2,-3,4,-5)) { i => if (i < 0) Some(-i) else None }
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 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: $d_1, d_2, \ldots, d_n$. Then, `foldLeft` yields

$$\texttt{f}(\cdots(\texttt{f}(\texttt{f}(\texttt{z}, d_1), d_2))\cdots), d_n)\,.$$

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., $d_1 < d_2 < \cdots < d_n$).

Example:

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

Figure 1: Abstract Syntax of JAVASCRIPTY

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

3. **TypeScripty**. 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 τ, functions with a zero-or-more parameters (instead of exactly one) $p(\overline{x:\varsigma})\, t \Rightarrow e_1$ and parameter-passing modes $\varsigma ::= m\tau$, and objects $\{f_1 : v_1, \ldots, f_n : v_n\}$ (see Figure 1). We have a language of types τ and annotate function parameters with modes m and types τ. Functions can now take any number of parameters. We write a sequence of things using either an overbar or dots (e.g., $\overline{e}$ or $e_1, \ldots, e_n$ for a sequence of expressions). An object literal

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

   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

statements	$s ::= m\ x = e \mid e \mid \{\ s_1\ \} \mid ; \mid s_1\ s_2$
expressions	$e ::= \cdots \mid \cancel{m\ x = e_1; e_2} \mid (e_1)$
	$\mid \cancel{p(\overline{x:\varsigma})t \Rightarrow e_1} \mid \textbf{function}\ p(\overline{x:\varsigma})\,t\,\{\ s_1\ \textbf{return}\ e_1\ \} \mid (\overline{x:\varsigma}) \Rightarrow e$
moded types	$\varsigma ::= \cdots \mid \tau$

Figure 2: Concrete Syntax of JAVASCRIPTY

named f. An object value is a sequence of field names associated with values. The type language τ includes base types for numbers, booleans, strings, and **undefined**, as well as constructed types for functions $(\overline{x:\varsigma}) \Rightarrow \tau$ and objects $\{f_1 : \tau_1; \ldots; f_n : \tau_n\}$.

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.

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 with the **function** keyword, the body is surrounded by curly braces (i.e., { }) and consists of a statement s_1 for variable bindings followed by a **return** with an expression e_1. To be compatible with TypeScript syntax, we permit dropping the mode annotation on function parameter types in the concrete syntax, which is parsed as **const**.

In Figure 3, we show the updated and new AST nodes. The `Decl` node is a more general variable declaration node that takes a mode m:`Mode`; thus, the `ConstDecl`(x, e_1, e_2) node is replaced with `Decl`$(\texttt{Const}, x, e_1, e_2)$. We update `Function` and `Call` for multiple parameters and arguments using `List[(String,MTyp)]` and `List[Expr]`, respectively. Object literals is represented by an `Obj` node representing fields as a `Map[String,Expr]` where the field name f is represented by a `String`, and field read expressions are represented by a `GetField` node. The `TFunction` and `TObj` nodes represent function and object types (`Typ`), corresponding to the `Function` and `Obj` expression forms (`Expr`).

(a) (By the Checkpoint) **Testing New Language Features**. Submit at least one of your tests for a new language feature of this lab as

`src/test/resources/lab4/mytest_`*somedescription*`.jsy`

for the **checkpoint**. Focus on tricky edge cases to get better feedback.

(b) (Advised Completion: Week 1) **A Small-Step Interpreter for Statically-Typed JavaScripty with Multi-Parameter Functions and Objects**. In this part, we update the `substitute` and `step` implementations from Lab 3 for multi-parameter functions and objects. Because of static type checking, the `step` cases can be *greatly simplified*. We eliminate performing conversions. And what's cool is that we should no longer throw `DynamicTypeError`.

Instead, 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!

```
/* Declarations */
case class Decl(m: Mode, x: String, e1: Expr, e2: Expr) extends Expr
  Decl(m, x, e1, e2)   m x = e1; e2

/* Functions */
case class Function(p: Option[String], params: List[(String,MTyp)], tann: Option[Typ],
                    e1: Expr) extends Expr
  Function(p, (x,ς), t, e1)   p(x:ς)t => e1
case class Call(e1: Expr, args: List[Expr]) extends Expr
  Call(e1, e)   e1(e)

/* Parameter Modes */
case object Const extends Mode
  Const   const
case object Name extends Mode
  Name   name
case class MTyp(m: Mode, t: Typ)
  MTyp(m, τ)   mτ

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

/* Types */
case class TFunction(params: List[(String,MTyp)], tret: Typ) extends Typ
  TFunction(x:ς, τ)   (x:ς) ⇒ τ
case class TObj(tfields: Map[String, Typ]) extends Typ
  TObj(f:τ)   {f:τ}
```

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.

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 $\cdot \vdash e : \tau$ holds meaning `inferType(`e`)` does not throw `StaticTypeError`), then `step` should never throw a `StuckError`. This property of this variant of JAVASCRIPTY is known as *type safety*.

A small-step operational semantics is given in Figure 4. This semantics no longer has conversions compared to Lab 3. It is much simpler because of type checking—even with the addition of objects. It fits on one page!

As specified, SEARCHOBJECT is non-deterministic (ask yourself 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

```
(m: Map[K,V]).find(f: ((K,V)) => Boolean): Option[(K,V)].
```

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

1. **Core JAVASCRIPTY**. First, import `substitute`, `iterate`, `inequalityVal`, and 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 the exception `DynamicTypeError`. Follow the small-step operational semantics in Figure 4 to see how to simplify your Lab 3 code (e.g., start with DONEG).
2. **Objects**. 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? **Hints**: You might want to use your `mapFirst` function from the warm-up question. Helpful library methods here include

   ```
   (m: Map[K,V]).map(f: ((K,V)) => (J,U)): Map[J,U]
   (m: Map[K,V]).get(k: K): Option[V].
   ```

3. **Functions with Zero-Or-More Call-By-Value Parameters**. Then, work on the function cases assuming all parameters are call-by-value (i.e., the **const** mode). That is, ignore the possibility of call-by-name parameters (i.e., the **name** mode). **Hints**: You might want to use your `mapFirst` function from the warm-up question. Helpful library methods here include

   ```
   (l: List[A]).map(f: A => B): List[B]
   (l: List[A]).exists(f: A => Boolean): Boolean
   (la: List[A]).zip(lb: List[B]): List[(A,B)]
   (l: List[A]).forall(f: A => Boolean): Boolean
   (l: List[A]).foldRight(f: (A,B) => B): B.
   ```

   You may want to use `zip` in the `Call` case to match up formal parameters and actual arguments.

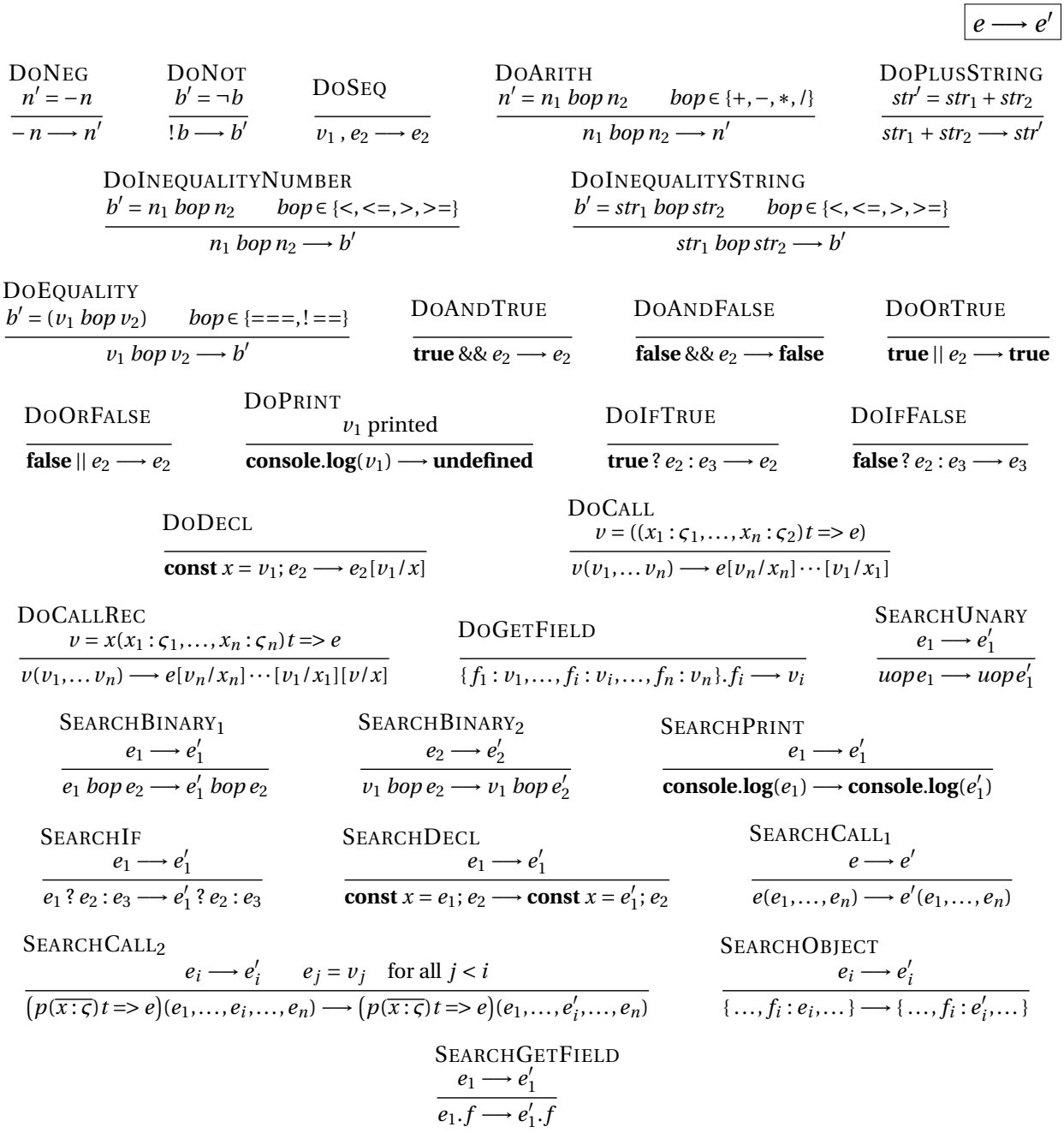

Figure 4: Small-step operational semantics of JavaScripty

The cases for objects and functions use collections, so be sure to complete the functional programming warm-up question before attempting them. You can also work on the next part `typeof` before finishing `step`. In fact, it is often easier to "incrementally grow the language" by going language-feature by-language-feature for all functions rather than function-by-function.

(c) (Advised Completion: Week 2) **Static Type Checking**. In this part, we then implement a static type checker that up front rules out programs that would get stuck in taking steps. This type checker 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 5, we define the judgment form $\Gamma \vdash e : \tau$ which says informally, "In type environment Γ, expression e has type τ." We will implement a function

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

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

The TYPEEQUALITY is slightly informal in stating

$$\tau \text{ has no function types}.$$

We intend this statement to say that the structure of τ 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 `typeof`.

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 functional programming warm-up question before attempting them.

Hints: In addition to the previously mentioned library methods, helpful ones here include

```
(l: List[A]).foldLeft(f: (B,A) => B): B
(l: List[A]).foreach(f: A => Unit): Unit
(l: List[A]).length: Int
(m1: Map[K,V]).++(m2: Map[K,V]): Map[K,V]
```

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{TypeVar}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{TypeNeg}\ \frac{\Gamma \vdash e_1 : \textbf{number}}{\Gamma \vdash -e_1 : \textbf{number}} \qquad \textsc{TypeNot}\ \frac{\Gamma \vdash e_1 : \textbf{bool}}{\Gamma \vdash !e_1 : \textbf{bool}} \qquad \textsc{TypeSeq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 , e_2 : \tau_2}$$

$$\textsc{TypeArith}\ \frac{\Gamma \vdash e_1 : \textbf{number} \qquad \Gamma \vdash e_2 : \textbf{number} \qquad bop \in \{+, -, *, /\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{number}} \qquad \textsc{TypePlusString}\ \frac{\Gamma \vdash e_1 : \textbf{string} \qquad \Gamma \vdash e_2 : \textbf{string}}{\Gamma \vdash e_1 + e_2 : \textbf{string}}$$

$$\textsc{TypeInequalityNumber}\ \frac{\Gamma \vdash e_1 : \textbf{number} \qquad \Gamma \vdash e_2 : \textbf{number} \qquad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypeInequalityString}\ \frac{\Gamma \vdash e_1 : \textbf{string} \qquad \Gamma \vdash e_2 : \textbf{string} \qquad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypeEquality}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \text{ has no function types} \qquad bop \in \{===, !==\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypeAndOr}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool} \qquad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}} \qquad \textsc{TypePrint}\ \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \textbf{console.log}(e_1) : \textbf{Undefined}}$$

$$\textsc{TypeIf}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : \tau} \qquad \textsc{TypeDecl}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash m\ x = e_1; e_2 : \tau_2}$$

$$\textsc{TypeCall}\ \frac{\Gamma \vdash e : (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) \Rightarrow \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau} \qquad \textsc{TypeObject}\ \frac{\Gamma \vdash e_i : \tau_i \qquad \text{(for all } i\text{)}}{\Gamma \vdash \{\ldots, f_i : e_i, \ldots\} : \{\ldots, f_i : \tau_i, \ldots\}}$$

$$\textsc{TypeGetField}\ \frac{\Gamma \vdash e : \{\ldots, f : \tau, \ldots\}}{\Gamma \vdash e.f : \tau} \qquad \textsc{TypeNumber}\ \frac{}{\Gamma \vdash n : \textbf{number}} \qquad \textsc{TypeBool}\ \frac{}{\Gamma \vdash b : \textbf{bool}} \qquad \textsc{TypeString}\ \frac{}{\Gamma \vdash str : \textbf{string}} \qquad \textsc{TypeUndefined}\ \frac{}{\Gamma \vdash \textbf{undefined} : \textbf{Undefined}}$$

$$\textsc{TypeFunction}\ \frac{\Gamma[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) \Rightarrow \tau}{\Gamma \vdash \quad (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) => e \quad : \tau'}$$

$$\textsc{TypeFunctionAnn}\ \frac{\Gamma[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) \Rightarrow \tau}{\Gamma \vdash \quad (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) : \tau => e \quad : \tau'}$$

$$\textsc{TypeRecFunction}\ \frac{\Gamma[x \mapsto \tau'][x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) \Rightarrow \tau}{\Gamma \vdash \quad x(x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) : \tau => e \quad : \tau'}$$

Figure 5: Typing of JAVASCRIPTY.

(d) (Advised Completion: Week 2, as time permits) **Call-By-Name Parameters**. In this part, we consider extend JAVASCRIPTY functions with call-by-name parameters. We consider the formalization and concept of call-by-name parameters as part of this lab but the implementation as supplemental in support of their understanding. There are no test cases in the auto-grader that check for call-by-name.

Like short-circuiting, call-by-name is another form of lazy evaluation where some subexpression is conditionally evaluated. In particular, call-by-name does not evaluate the function argument to a value before starting to evaluate this function body. Instead, it takes the unevaluated argument expression and substitutes it for the formal parameter. Consider a function call with a single call-by-name parameter, then the derived DOCALL-like rule is as follows:

$$\frac{v = ((x_1 : \textbf{name}\,\tau_1)\,t \Rightarrow e)}{v(e_1) \longrightarrow e[e_1/x_1]}$$

If it is possible that the function body e does not use the formal parameter x_1 in some executions, then the argument expression e_1 is not evaluated in those cases. While call-by-name parameters do not exist in JavaScript, they do in some other languages (e.g., Scala). They can be used to implement functions that look like control structures (i.e., **while**, **for**, etc.).

To implement call-by-name, the final wrinkle is that it requires substituting an arbitrary expression into another expression. Thus, we must be careful to avoid free variable capture (cf., Notes 3.2). We did not have to consider this case before because we were only ever substituting values that did not have free variables.

In this lab, you will need to modify your `substitute` function to avoid free variable capture. To do this, first implement a function

def `rename(e: Expr)(fresh: String => String): Expr`

that renames an expression `e` using the `fresh` parameter to decide how to rename *bound* variables in `e`. The inner helper function

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

recurses over expression `e` with a renaming environment `env` that maps original names to new names for the free variables of `e` (as determined by code `fresh` when the variable was bound).

Then, modify `substitute`

def `substitute(e: Expr, esub: Expr, x: String): Expr`

with a call to `rename` to rename expression `e` to avoid capturing the free variables of `esub`. You can call the function

def `freeVars(e: Expr): Set[String]`

provided in `jsy.lab4.ast` to compute the set of free variables of an expression.

Finally, implement the `isRedex` relation following the judgment form $m \vdash e$ redex in Figure 6 and update `step` to use it for variable declarations and function calls according to the small-step judgment form $e \longrightarrow e'$ in Figure 4.

$\boxed{m \vdash e \text{ redex}}$

$$\textsc{ConstMode}\ \frac{e \neq v}{\textbf{const} \vdash e \text{ redex}} \qquad \text{no rule for } \textbf{name}$$

$\boxed{e \longrightarrow e'}$

$$\textsc{DoDecl}\ \frac{m \not\vdash e_1 \text{ redex}}{m\,x = e_1; e_2 \longrightarrow e_2[e_1/x]} \qquad \textsc{DoCall}\ \frac{v = (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n)\,t \Rightarrow e \qquad m_i \not\vdash e_i \text{ redex} \qquad \text{for all } i}{v(e_1, \ldots e_n) \longrightarrow e[e_n/x_n] \cdots [e_1/x_1]}$$

$$\textsc{DoCallRec}\ \frac{v = x(x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n)\,t \Rightarrow e \qquad m_i \not\vdash e_i \text{ redex} \qquad \text{for all } i}{v(e_1, \ldots e_n) \longrightarrow e[e_n/x_n] \cdots [e_1/x_1][v/x]} \qquad \textsc{SearchDecl}\ \frac{e_1 \longrightarrow e_1' \qquad m \vdash e_1 \text{ redex}}{m\,x = e_1; e_2 \longrightarrow m\,x = e_1'; e_2}$$

$$\textsc{SearchCall}_2\ \frac{e_i \longrightarrow e_i' \qquad m_i \vdash e_i \text{ redex} \qquad m_j \not\vdash e_j \text{ redex} \qquad \text{for all } j < i}{\big(p(\overline{x : \varsigma})\,t \Rightarrow e\big)(e_1, \ldots, e_i, \ldots, e_n) \longrightarrow \big(p(\overline{x : \varsigma})\,t \Rightarrow e\big)(e_1, \ldots, e_i', \ldots, e_n)}$$

Figure 6: Define expressions that are reducible under a mode $m \vdash e$ redex that is then used to generalize the small-step operational semantics from Figure 4 with call-by-name parameters. The inference rules for the $e \longrightarrow e'$ judgment form defined here replace the ones by the same name in Figure 4; all other rules are unchanged.