The primary purpose of this lab to understand the interplay between type checking and evaluation. Concretely, we will extend JavaScript 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.

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 unfamiliar to you. Finally, make sure that your file compiles and runs (using Scala 2.9.2). 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. A link to the auto-testing system is available on the moodle.

Then, upload to the moodle exactly two 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

Replace YourIdentiKey with your IdentiKey. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

**Getting Started.** Download the code template Lab3.scala from the assignment page. Be sure to also write your name, your partner, and collaborators there.

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 JavaScript 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

```scala
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
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

```scala
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
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:

```scala
sealed abstract class Tree {
  def insert(n: Int): Tree = this match {
    case Empty => Node(Empty, n, Empty)
    case Node(l, d, r) =>
      if (n < d) Node(l insert n, d, r) else Node(l, d, r insert n)
  }

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

\[
f(\cdots(f(f(z, d_1), d_2)), \ldots, 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

```scala
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 ascending order (i.e., \( d_1 < d_2 < \cdots < d_n \)).

Example:

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

3. **JavaScript Type Checker**

As we have seen in the prior labs, dealing 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 JavaScript. We will not permit any type conversions and will guarantee the absence of dynamic type errors.
expressions

\[ e ::= x | n | b | \text{undefined} | uop e_1 | e_1 \ bop e_2 | e_1 ? e_2 : e_3 \]
\[ | \text{const } x = e_1 : e_2 | \text{jsy.print}(e_1) \]
\[ | \text{str } \text{function } p(\tilde{x}:\tilde{\tau})\text{tann } e_1 | e_1(e_2) \]
\[ | \{ f_1 : e_1, \ldots, f_n : e_n \} | e_1.f \]

values

\[ v ::= n | b | \text{undefined} | \text{str } \text{function } p(\tilde{x}:\tilde{\tau})\text{tann } e_1 \]
\[ | \{ f_1 : v_1, \ldots, f_n : v_n \} \]

unary operators

\[ \text{uop} ::= - | ! \]

binary operators

\[ \text{bop} ::= , | + | - | * | / | === | !== | < | <= | > | >= | && | || \]

values

\[ \tilde{\tau} ::= \text{number} | \text{bool} | \text{string} | \text{Undefined} | (\tilde{x}:\tilde{\tau}) \Rightarrow \tau' | \{ f_1 : \tau_1; \ldots; f_n : \tau_n \} \]

variables

\[ x \]

numbers (doubles)

\[ n \]

booleans

\[ b ::= \text{true} | \text{false} \]

strings

\[ \text{str} \]

function names

\[ p ::= x | \varepsilon \]

field names

\[ f \]

field read expressions

\[ e_1.f \]

field names

\[ f \]

field read expressions

\[ e_1.f \]

function names

\[ p ::= x | \varepsilon \]

field names

\[ f \]

field read expressions

\[ e_1.f \]

field names

\[ f \]

field read expressions

\[ e_1.f \]

Figure 1: Abstract Syntax of JAVASCRIPTY

In this lab, we extend JAVASCRIPTY with types, multi-parameter functions, and objects/records (see Figure 1). We have a language of types \( \tilde{\tau} \) 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 \( e_1, \ldots, e_n \) 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 dynamic dispatch. Our objects in this lab are also immutable. 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 \( \tilde{\tau} \) includes base types for numbers, booleans, strings, and undefined, as well as constructed types for functions \( (\tilde{x}:\tilde{\tau}) \Rightarrow \tau' \) and objects \( \{ f_1 : \tau_1; \ldots; f_n : \tau_n \} \).

As an aside, we have chosen 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.

In Figure 2, 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-
/* Functions */
case class Function(p: Option[String], params: List[(String,Typ)], tann: Option[Typ],
e1: Expr) extends Expr
    Function(p, (x,τ), tann, e1)

function p(x: τ)
tann e1

/* Objects */
case class Obj(fields: Map[String, Expr]) extends Expr
    Object(f: e)

case class GetField(e1: Expr, f: String) extends Expr
    GetField(e1, f)

case class Call(e1: Expr, args: List[Expr]) extends Expr
    Call(e1, args)

/* Types */
case class TFunction(params: List[(String,Typ)], tret: Typ) extends Typ
    TFunction(x: τ, r': Typ)

case class TObj(tfields: Map[String, Typ]) extends Typ
    TObj(f: τ)

Figure 2: Representing in Scala the abstract syntax of JavaScript. After each case class or case object, we show the correspondence between the representation and the concrete syntax.

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 3, 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

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

that corresponds directly to this judgment form. It takes as input a type environment \( \Gamma \) and an expression \( e \) \( (e) \) returns a type \( \text{Typ} \) \( (r) \). 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 \( \tau \) 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

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

where \( \text{tbad} \) is the type that is inferred sub-expression \( \text{esub} \) 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.
Figure 3: Typing of JavaScripty.
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 simplified greatly. We eliminate the need to perform conversions and should no longer throw DynamicTypeError.

We introduce another Scala exception type

```scala
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 ⊢ e : τ 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 4. 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. 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, complete the basic expression cases not given in the template (e.g., DoMinus, DoIfTrue).

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,

5. **Type Equality.** This question uses the PL-Detective. The most important part of this question is not the final answer (e.g., the type equality mechanism) but the reasoning that led
Figure 4: Small-step operational semantics of JavaScript.
to the answer. Thus, don’t leave the write up to the last minute! Also since the number of
attempts you can submit to the PL-Detective is limited, it is worth thinking carefully before
each submission. The limit is set high enough that you can waste about half of the attempts
and still get the full score. When you submit a program to the PL-Detective, it tells you how
many programs you have submitted without syntax errors so far. Note that type errors are
not syntax errors.

We study two type equality mechanisms: name type equality and structural type equality.
In this exercise, you will figure out the type equality mechanism used by MYSTERY. As with
previous exercises, you will accomplish this by submitting programs to the PL-Detective
and observing the results.

For the purpose of this assignment, you should assume that an assignment

\[
Expr_1 := Expr_2
\]

is legal only if the types of \(Expr_1\) and \(Expr_2\) are equal. Also, assume that the type of a con-
stant number (e.g., 5) is INTEGER.

(a) In this question, ignore TYPE declarations in MYSTERY. TYPE declarations are similar
to type in Scala or typedef in C and C++. For example,

\[
\text{TYPE } T = \text{INTEGER}
\]

in MYSTERY creates a new type name \(T\) for the type INTEGER. This declaration is sim-
ilar to

\[
\text{type } T = \text{Int}
\]

in Scala or

\[
\text{typedef int } T
\]

in C/C++.

i. Determine and describe how MYSTERY defines type equality for ARRAY, PROCEDURE,
and subrange types. Your description should be precise and complete enough that
a reader can take any two types (where both are array types, procedure types, or
subrange types) in MYSTERY and determine whether or not they are equal. You
may submit up to 8 programs to the PL-Detective. You will lose 5% of the points
for this question for each additional program that you use. Submitted programs
that fail due to syntax errors do not count in your total. Note that type errors are
not syntax errors. Use the following link for this exercise:

http://csci3155.cs.colorado.edu/pl-detective/hw/pldteq.htm

Hint. The issue of whether two types are name or structurally equal comes only
when they are not obviously different. For example, the subrange types \([0 \ T0\ 10]\)
and \([5 \ T0\ 20]\) are obviously different and thus you do not need to submit a pro-
gram to the PL-Detective to determine if they are equal or not.

Present an argument for your answer. Your argument should include the evidence
(e.g., programs you submitted to the PL-Detective and the output) along with a
carefully reasoned argument explaining how your answer is justified by your evi-
dence.
ii. Give one point in favor of and one point against MYSTERY's type equality mechanism (which you just discovered).

(b) In this question you will determine how MYSTERY's TYPE declarations affect type equality. In other words, after

\[ \text{TYPE } T = \langle \text{some type} \rangle \]

are \( T \) and \( \langle \text{some type} \rangle \) equal?

i. Determine and describe how MYSTERY's TYPE declarations affect type equality. Write this as if you are writing a language definition, that is, your answer to this question combined with your answer to question 2 should allow a reader to determine the equality of any two types in MYSTERY's including ones that use type names declared using TYPE. You may submit up to 2 programs to the PL-Detective in your investigation. Each additional program is charged 5% of the points for this question. Submitted programs that fail due to syntax errors do not count in your total. Note that type errors are not syntax errors. Use the following link for this exercise:

http://csci3155.cs.colorado.edu/pl-detective/hw/pldteq2.htm

Present an argument for your answer. Your argument should include the evidence (e.g., programs you submitted to the PL-Detective and the output) along with a carefully reasoned argument explaining how your answer is justified by your evidence.

ii. Give one point in favor of and one point against the semantics that you just discovered.