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. Make sure that your file compiles and runs. A program that does not compile will not be graded.

**Submission Instructions.** To submit, upload to the moodle exactly two files named as follows:

- **Homework4-YourIdentiKey.pdf** with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- **Homework4-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 **Homework4.scala** from the assignment page. Be sure to also write your name, your partner, and collaborators there. Do not alter the template except to rename with your IdentiKey, replace the code for the function implementation exercises, and give test cases. If you add other code to your file during development, please comment it out before submitting.

**Testing.** Write unit tests in the same way as in previous homework assignments. Templates for the tests are given in the homework skeleton. Recall that a test case is a Boolean expression that evaluates to true to indicate success.

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

2. **Evaluation Order.**
(a) **Mystery.** 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 Mystery, allow complicated expressions for $F$. For example, if $a$ is an array of procedures, one can do $a[2](5)$ in Mystery to call the procedure at $a[2]$ with argument 5. Mystery also allows a procedure to return a procedure (i.e., has higher-order functions like Scala). 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 Mystery evaluates the procedure being called before it evaluates its arguments. For example, with $f() (x)$ does Mystery call the function $f$ first or does it evaluate $x$ first (i.e., looks up the value of $x$). You may execute up to 4 PRINT statements.

ii. Discover whether or not Mystery evaluates arguments to a call from left-to-right. You may execute up to 4 PRINT statements.

Use the following link to submit programs:
http://csci3155.cs.colorado.edu/pl-detective/hw/pldeval.htm

In your write-up, explain how you came to a conclusion. You should provide evidence that supports your argument. You will lose 5% of the points for this question for each additional executed PRINT.

(b) **Smalla.** Consider the operational semantics for Smalla from class and shown in Figure 3.

i. What is the evaluation order for $e_1 + e_2$? Explain.

ii. Give a new set of inference rules for $e_1 + e_2$ that evaluates in the opposite order as the one specified in Figure 3?

3. **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) **Mystery.** Discover and indicate whether or not Mystery uses short-circuit evaluation of boolean operators (i.e., **AND**). Use this link to submit programs:
http://csci3155.cs.colorado.edu/pl-detective/hw/pldshortcircuit.htm

You should provide evidence that supports your argument. Your program should not execute PRINT more than three times. You will lose 5% of the points for this question for each additional executed PRINT.

(c) **Smalla.** Consider the operational semantics for Smalla from class and shown in Figure 3.
i. Does \( e_1 \&\& e_2 \) short circuit? Explain.

ii. Give a new set of inference rules for \( e_1 \&\& e_2 \) that does the opposite as the one specified in Figure 3 (i.e., short circuits if SMALLA does not or does not short circuit if SMALLA does).

4. **Continuations: Regular Expressions.** For this exercise, we will implement a simple, backtracking regular expression matcher. Our regular expressions will be represented using the following Scala case classes:

```scala
sealed abstract class RegExpr
  case object NoString extends RegExpr
  case object EmptyString extends RegExpr
  case class Single(c: Char) extends RegExpr
  case class Concat(re1: RegExpr, re2: RegExpr) extends RegExpr
  case class Union(re1: RegExpr, re2: RegExpr) extends RegExpr
  case class Star(sub: RegExpr) extends RegExpr
  case object AnyChar extends RegExpr
  case class Intersect(re1: RegExpr, re2: RegExpr) extends RegExpr
```

A regular expression defines a regular language (i.e., a set of strings). The first six constructors are the basic regular expression constants and operators. Let us write \( L(r) \) for the language specified by the regular expression \( r \):

- \( L(\text{NoString}) \overset{\text{def}}{=} \emptyset \), that is, the empty set.
- \( L(\text{EmptyString}) \overset{\text{def}}{=} \{""\}, \) that is, the set with the empty string.
- \( L(\text{Single}(c)) \overset{\text{def}}{=} \{"c"\}, \) that is, the set with the string matching the single character.
- **Concatenation.** A string \( s = s_1 s_2 \) is in \( L(\text{Concat}(r_1, r_2)) \) iff \( s_1 \) is in \( L(r_1) \) and \( s_2 \) is in \( L(r_2) \).
- **Union.** A string \( s \) is in \( L(\text{Union}(r_1, r_2)) \) iff \( s \) is in \( L(r_1) \) or \( s \) is in \( L(r_2) \).
- **Kleene Star.** A string \( s \) is in \( L(\text{Star}(r)) \) iff \( s \) is in zero-or-more concatenations of \( r \).

We add two extensions:

- **Any Character.** A string \( s \) is in \( L(\text{AnyChar}) \) iff \( s \) consists of any single character.
- **Intersection.** A string \( s \) is in \( L(\text{Intersect}(r_1, r_2)) \) iff \( s \) is in \( L(r_1) \) and \( s \) is in \( L(r_2) \).

We will write a regular expression matcher

```scala
def matchRE(re: RegExpr, s: String): Boolean
```

that given a regular expression \( re \) and a string \( s \) returns \textbf{true} if the string \( s \) belongs to the language described by the regular expression \( re \) and otherwise returns \textbf{false}. The implementation of \texttt{matchRE} is provided for you, which calls a helper function \texttt{matchREDo} that you provide.

We will implement our regular expression matcher using continuations. In particular, we will implement a helper function:
def matchREDo(re: RegExpr, chars: List[Char],
sc: List[Char] => Boolean): Boolean

This helper function will see if a prefix of chars matches the regular expression re. If there is a prefix match, then the success continuation is called with the remainder of chars that has yet to be matched. That is, the success continuation sc captures “what to do next if a prefix of chars successfully matches re.” If matchREDo discovers a failure to match, then it can “return false early.”

Do not worry about making your implementation tail recursive.

Scala Note. In Scala, a case object gives a constructor with no parameters (such as, Nil). Here, case objects are used for NoString, EmptyString, and AnyChar:

case object NoString extends RegExpr
  case object EmptyString extends RegExpr
  case object AnyChar extends RegExpr

Hints. Star and Intersection are the most difficult cases. Consider completing the other cases first. Implementing the Concat case might help clarify how the success continuation is used.

Extra Credit. Prove that your implementation is correct. To get extra credit, you have to see Prof. Chang during office hours within one week after the submission deadline, explain the proof to him orally, and convince him that your implementation is correct via the proof.

5. Interpreter: DYNSMALLA. Let us consider a dynamically-typed variant of SMALLA, which we will call DYNSMALLA. For this exercise, we will implement a small-step interpreter based on the operational semantics of DYNSMALLA.

The syntax of DYNSMALLA given in Figure 1 is almost exactly the same as SMALLA. DYNSMALLA’s function types do not specify a return type. Instead, they have the form:

\[ \tau_1 \Rightarrow \text{Dyn} \]

Dyn is just part of the function type syntax for DYNSMALLA. It alludes to the fact that we dynamically type the body of functions. We also add a syntactic category for results that are either values or a distinguished symbol typeerror.

We represent the abstract syntax of DYNSMALLA using the AST types shown in Figure 2—the translation is direct. Results are not represented in Scala because in our DYNSMALLA interpreter, we will throw a Scala exception TypeError immediately when a typeerror is encountered.

The operational semantics of DYNSMALLA shown in Figure 3 are also almost exactly the same as for SMALLA, except that we have given explicit rules for when we check for a type error (rather than just getting stuck).

(a) Implement a function

   def substitute(v: Val, x: String, e: Expr): Expr

   that substitutes a Val v for Vars with name x in Expr e.

   Hints. You will want to recurse over the Expr AST.
(b) Complete the function

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

that implements one step of evaluation according to the operational semantics by adding cases to the pattern match on e. Do not change the default catch-all case

```python
case _ => throw new StuckError(e),
```

which you will want to have at the bottom below the cases that you add. This exception indicates that there is no one-step evaluation rule that allows to evaluate any further. If you encounter an ill-typed Expr according to the operational semantics, throw a TypeError exception. To throw this exception, use the expression:

```python
throw new TypeError(e)
```

where e is the input Expr to step when you have detected a type error. Make sure e is the input Expr to step at the call where you have detected the type error (not any subexpression).

We have provided in the template a helper function

```python
def typeOf(v: Val): Typ
```

that gives the type of a Val.

**Hints.**

- Do a direct translation of the evaluation rules in Figure 3 as much as you can. Your code should have a clear correspondence with the evaluation rules.
- To do so, you will want to make heavy use of pattern matching. Here some useful patterns that you may not have used previously:
  - **Typed Patterns** You can check that the value matched in a pattern has a particular type:
    ```python
case _ : Val => ...
```
    such as type Val in the above.
  - **Alternation Patterns** You can write a pattern that matches \( p_1 \) or \( p_2 \):
    ```python
    p_1 | p_2
    ```
  - **Patterns Guards** You can check a boolean condition along with a pattern match:
    ```python
    case p if (...) => ...
    ```
- You should maintain the invariant that the input Expr to step is not a Val.
- Your step function will be recursive. However, remember that step should not evaluate the Expr to a Val. We have provided a function

```python
def eval(e: Expr): Val
```

that tries to evaluate an Expr “fully” to a Val. It repeatedly calls your step and stops if it reaches a Val. You can use the eval function for your testing if you wish.
expressions  \[ e ::= x \mid r \mid uop e_1 \mid e_1 bop e_2 \mid \text{print}(e_1) \]
\[ \mid \text{if } (e_1) e_2 \text{ else } e_3 \mid e_1(e_2) \]
values \[ v ::= n \mid b \mid () \mid (x:\tau) => e_1 \]
types \[ \tau ::= \text{Int} \mid \text{Boolean} \mid \text{Unit} \mid \tau_1 \Rightarrow \text{Dyn} \]
booleans \[ b ::= \text{true} \mid \text{false} \]
unary operators \[ uop ::= - \mid ! \]
binary operators \[ bop ::= ; \mid + \mid - \mid * \mid < \mid == \mid && \mid || \]
variables \[ x \]
integers \[ n \]
results \[ r ::= v \mid \text{type error} \]

Figure 1: Syntax of DYNSMALLA.
sealed abstract class Expr
  case class Var(x: String) extends Expr
  case class Unary(uop: Uop, e1: Expr) extends Expr
  case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  case class Print(e1: Expr) extends Expr
  case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr
  case class Apply(e1: Expr, e2: Expr) extends Expr

sealed abstract class Val extends Expr
  case class N(n: Int) extends Val
  case class B(b: Boolean) extends Val
  case object U extends Val
  case class Fun(x: String, t: Typ, e1: Expr) extends Val

sealed abstract class Typ
  case object TInt extends Typ
  case object TBoolean extends Typ
  case object TUnit extends Typ
  case class TFunDyn(t: Typ) extends Typ

sealed abstract class Uop
  case object Neg extends Uop
  case object Not extends Uop

sealed abstract class Bop
  case object Semi extends Bop
  case object Plus extends Bop
  case object Minus extends Bop
  case object Times extends Bop
  case object Lt extends Bop
  case object Eq extends Bop
  case object And extends Bop
  case object Or extends Bop

Figure 2: Representing in Scala the abstract syntax of DYNSMALLA.
Figure 3: Small-step operational semantics of DYN\textsc{smallla}. 