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 three files named as follows:

- Homework6-YourIdentiKey.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- Homework6-YourIdentiKey.scala with your answers to the coding exercises
- Homework6-YourIdentiKey.smalla with your Smalla “stress” test.

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 Homework6.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.

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

2. **Dynamic Scoping.** We have discussed two kinds of scoping: static and dynamic. Use http://csci3155.cs.colorado.edu/pl-detective/hw/pldscoping.htm
to figure out if M\textsc{ystery} uses static or dynamic scoping (in this configuration). Your write up should describe the kind of scoping that M\textsc{ystery} uses and provide evidence for your conclusions. You may submit no more than three programs for this question. Programs that have parse errors (i.e., syntax errors) do not count. Every additional program will cost you 5% of the points for this question.

3. \textbf{Type Checker and Interpreter: IMP\textsc{Name}SMALLA}

In this question, we extend SMALLA with imperative features and different parameter passing modes.

In particular, we add imperative expressions for the following:

- Assignment $e_1 = e_2$ says write the value of $e_2$ to the memory location named by $e_1$.
- Dynamic memory allocation $\text{new}(e_1)$ says allocate a memory cell whose contents is initialized to the value of $e_1$.
- Dereference $*e_1$ says get the contents of the memory cell given by $e_1$.
- Address $a$ stands for some memory address. Addresses also become a new kind of value.

Note that addresses $a$ cannot be directly written in the source program but will arise during evaluation.

Like in Scala, we will have two kinds of variables: immutable (\texttt{val}) and mutable (\texttt{var}). Since we introduce local variables via function parameters, we annotate parameters to indicate whether they should be immutable or mutable. For example,

$$(\texttt{var } x : \texttt{Int}) \Rightarrow \ldots$$

says that $x$ is mutable. We will let no annotation to mean that the parameter is immutable (keeping the same syntax as before). That is,

$$(x : \texttt{Int}) \Rightarrow \ldots$$

says that $x$ is immutable. In both of these cases, the function is evaluated using call-by-value semantics.

We also add two other kinds parameter annotations to correspond to other parameter passing modes:

- \texttt{name} indicates that the function should be called using call-by-name. This choice means that the variable is immutable.
- \texttt{ref} indicates the function call should be called using call-by-reference. This choice means that the variable is mutable.

In summary, we have four kinds of parameter annotations $pann ::= \varepsilon | \texttt{name} | \texttt{var} | \texttt{ref}$ where the empty string $\varepsilon$ means the usual immutable call-by-value parameter.

We add two new types for pointers $\texttt{Ptr}[\tau_1]$, which is the type of memory addresses, and call-by-reference functions $\texttt{Ref}[\tau_1] \Rightarrow \tau_2$. There's no need to distinguish the other kinds of
functions in the type. For reference, the complete syntax of \texttt{ImpNamedMalla} is given in Figure 1.

We represent the abstract syntax of \texttt{ImpNamedMalla} using the AST types shown in Figure 2—the translation is fairly direct and is shown under each \texttt{case class} or \texttt{case object}. One important note is that addresses are represented by \texttt{Addr} objects, which is not a \texttt{case class}. A fresh memory address is represented by allocating a new \texttt{Addr} object. This representation is not particularly compact but very simple.

The typing judgment is defined in Figure 5 and Figure 6, which corresponds directly to a type synthesis algorithm. (i.e., given typing environment $\Gamma$ and an expression $e$, return a type $\tau$ such that $\Gamma \vdash e : \tau$). The first figure (Figure 5) lists the same rules for all of the constructs from before (plus call-by-name functions); the implementations of these rules are provided in the homework template.

The second figure (Figure 6) gives rules for the new imperative constructs. One extension to the typing environment $\Gamma$ that we need to make is to remember whether a variable is mutable or not. See the \texttt{TypEnv} type in the homework library, which maps variable names to types and a \texttt{Mutability} flag that is either \texttt{ReadOnly} or \texttt{Writable}. In Figure 6, we notate mutability of the variable on the mapping arrow: (1) $x \xrightarrow{\tau}$ means that $x$ is mutable and has type $\tau$, and (2) $x \rightarrow \tau$ (without any mark on the arrow) means that $x$ is immutable and has type $\tau$. We continue to write $\Gamma(x)$ to mean lookup the type of $x$. Viewing $\Gamma$ as a set of mapping, we write $x \xrightarrow{\tau} \tau \in \Gamma$ to mean that $\Gamma$ says that $x$ is mutable and has type $\tau$.

A technicality that we need to reason about type safety is that we need a mapping that tells us the types of memory cells. In particular, we need such a mapping to give a type to an address $a$. We write $\Lambda$ for a mapping from addresses to types, so our typing judgment is in actually extended with a fourth parameter $\Lambda$. However, since addresses cannot be directly given in the source program, this mapping is not actually needed to implement type checking on a source program (i.e., the last rule in Figure 6 is not implemented).

A small-step operational semantics that defines the meaning of all of the new and old language constructs of \texttt{ImpNamedMalla} is given in Figure 7 and Figure 8. The main change is that the judgment form that steps between a memory-expression configurations:

$$\langle M, e \rangle \rightarrow \langle M', e' \rangle.$$ 

A memory $M$ is a mapping from addresses $a$ to values $v$. Figure 7 defines the rules for all of the old constructs. They are updated to thread the memory but correspond to the rules before. The homework template implements all of these rules for you.

Rules that define the evaluation for the new constructs are given in Figure 8.

We observe that adding mutation requires a significant, global change to the way we formalize and implement an interpreter. Our formalization and our implementation experience suggests that while it is doable to reason about imperative programs, it is harder simply because there’s more “computation state” in the memory $M$ that is also both hidden and indirectly connected to executing program.

What we have after this exercise is quite cool! We have something that models the essence of imperative programming languages. As before, we are still missing recursive types. But
otherwise, we have a language that includes a type-safe C. The main thing that is not modelled is memory management (i.e., our memory state always grows and is never collected). With higher-order functions, we are also quite close to languages with objects (e.g., Java, C#, JavaScript, Python). A gap are notions of dynamic dispatch.

**Interpreter Harness.** We have provided a driver for your interpreter in

```
ImpNameSmallaMain.scala.
```

This “main” will read in a file with a test case, parse the file to create an AST, call your type checker, and then call your evaluator.

There a couple of notes about the concrete syntax expected by the parser:

- We have added a little syntactic sugar to introduce local `var` bindings (just like `val` bindings):

  ```
  var i = 2 + 2 in
  i + 1
  ```

  that is translated directly into

  ```
  ((var i: Int) => i + 1)(2 + 2)
  ```

- We have also added a little syntactic sugar to introduce type aliases:

  ```
  type MyInt = Int in
  (x: MyInt) => 3
  ```

  that is translated directly into

  ```
  (x: Int) => 3
  ```

In other words, we replace all type aliases with their definitions during parsing.

**Homework Template.** For this homework, the code template includes the pattern match for all of the rule cases. You can change them if you wish, but they are intended to provide a good start. You will extend only `inferType` and `step` in this homework assignment. The `substitute` function is provided to you in the library.

(a) Complete the function

```
def inferType(G: TypEnv, e: Expr): Typ
```

that implements a type synthesis algorithm according to the inference rules in Figure 6. Implementations for the non-imperative constructs are given in the homework template. If you encounter a type error, throw the `TypeError` exception using the type environment and expression on input:

```
throw new TypeError(G, e).
```
(b) Complete the function

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

that implements one step of evaluation according to the operational semantics. 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 one to evaluate any further.

As before, we assume as a pre-condition that the `e` argument to `step` is well-typed (according to `inferType`). You will notice that the rules will only make progress for well-typed Exprs. This design simplifies the `step` function quite a bit. We suggest that you implement `inferType` before `step` to better understand what it provides.

We also have that the `e` argument to `step` is a closed expression (i.e., does not have any free variables). Furthermore, our operational semantics maintains this invariant.

(c) Create a Smalla test input. We continue to encourage unit testing, but we do not ask for such tests for this homework. Instead, we ask for a Smalla test input that tests the whole system. Extra credit may be given for particularly tricky tests (e.g., breaks the reference solution or lots of your classmates solutions).

4. Optional/Not Graded: Parameter Passing Modes. In this question, you will discover the parameter passing mechanism for MYSTERY.

Working on this question as well as studying and implementing the evaluation rules for function calls in Figure 8 are three very different ways to better understand parameter passing modes. They may mutually beneficial.

For the purpose of this question assume that the actual's type and the corresponding formal's type must be the same using structural type equality. Also assume that MYSTERY uses static scoping.

(a) For the following parameter passing implementations:

- pass-by-value
- pass-by-value-result
- pass-by-reference
- pass-by-name

give a realistic example for which the implementation is most suitable. You may want to consult Section 9.5.2 of Sebesta. Support your answer with clear arguments.

(b) Determine and describe the parameter passing mechanism in MYSTERY using the following link:

http://csci3155.cs.colorado.edu/pl-detective/hw/pldparam1.htm

Be sure to state how MYSTERY's implementation addresses the issues discussed in part (a). Try to keep your submissions to at most 10 PRINTs.
(c) Provide and discuss the evidence that supports your case.

(d) Repeat parts (b) and (c) from above using this link (which potentially uses a different parameter passing implementation):

   http://csci3155.cs.colorado.edu/pl-detective/hw/pldparam2.htm

5. **Optional/Not Graded: Recursive Types.** To round out IMPNAMESMALLA for fun, we can add recursive types to be able to talk about recursive data structures in Figure 9. The additions of **unfold** and **fold** expressions are primarily for type checking purposes.
expressions \[ e ::= x | n | b | () | (\text{pann}\text{x:}\tau) \Rightarrow e_1 \]
\[ \mid uop e_1 | e_1 \ bop \ e_2 | \text{print}(e_1) \]
\[ \mid \text{if} \ (e_1) \ e_2 \ \text{else} \ e_3 \mid e_1(e_2) \]
\[ \mid \{f_1 : e_1, \ldots, f_n : e_n\} \mid e_1.f \mid \text{inj}_{\{c_1 : \tau_1, \ldots, c_n : \tau_n\}}(c_i : e_i) \]
\[ \mid e \ \text{match} \ \text{case} \ c_1 : x_1 \Rightarrow e_1 \ \ldots \ \text{case} \ c_n : x_n \Rightarrow e_n \]
\[ \mid \text{def}g(\text{pann}\text{x:}\tau) = e_1 \ \text{in} \ e_2 \]
\[ \mid e_1 = e_2 \mid \text{new}(e_1) \mid *e_1 \mid a \]

values \[ v ::= n | b | () | (\text{pann}\text{x:}\tau) \Rightarrow e_1 \]
\[ \mid \{f_1 : v_1, \ldots, f_n : v_n\} \mid \text{inj}_{\{c_1 : \tau_1, \ldots, c_n : \tau_n\}}(c_i : v_i) \]
\[ \mid a \]

types \[ \tau ::= \text{Int} | \text{Boolean} | \text{Unit} | \tau_1 \Rightarrow \tau_2 \]
\[ \mid \{f_1 : \tau_1, \ldots, f_n : \tau_n\} | [c_1 : \tau_1, \ldots, c_n : \tau_n] \]
\[ \mid \text{Ptr}[\tau_1] | \text{Ref}[\tau_1] \Rightarrow \tau_2 \]

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

unary operators \[ uop ::= - | ! \]

binary operators \[ bop ::= ; | + | - | * | < | == | &\& | || \]

parameter annotations \[ pann ::= \varepsilon | \text{name} | \text{var} | \text{ref} \]

variables \[ x, g \]

field names \[ f \]

constructor names \[ c \]

integers \[ n \]

addresses \[ a \]

Figure 1: Syntax of IMPNAMESMALLA.
sealed abstract class Expr

case class Var(x: String) extends Expr
  Var(x)

case class Unary(uop: Uop, e1: Expr) extends Expr
  Unary(uop, e1)

case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  Binary(bop, e1, e2)

case class Print(e1: Expr) extends Expr
  Print(e1)

case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr
  If(e1, e2, e3)

case class Apply(e1: Expr, e2: Expr) extends Expr
  Apply(e1, e2)

case class Record(fields: Map[String, Expr]) extends Expr
  Record(fields)

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

case class Inj(t: TUnion, c: String, e: Expr) extends Expr
  Inj(t, c, e)

case class Match(t: Typ, e: Expr, cases: Map[String, (String, Expr)]) extends Expr
  Match(t, e, cases)

case class DefFun(fun: String, pann: PAnn, x: String, t1: Typ, t2: Typ, e1: Expr, e2: Expr) extends Expr
  DefFun(fun, pann, x, t1, t2, e1, e2)

case class Assign(e1: Expr, e2: Expr) extends Expr
  Assign(e1, e2)

case class New(e1: Expr) extends Expr
  New(e1)

case class Deref(e1: Expr) extends Expr
  Deref(e1)

case class N(n: Int) extends Expr
  N(n)

case class B(b: Boolean) extends Expr
  B(b)

case object U extends Expr
  U

class Addr extends Expr
  LibHomework6$Addr
  new Addr

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

case object TInt extends Typ
   TInt   Int

case object TBoolean extends Typ
   TBoolean  Boolean

case object TUnit extends Typ
   TUnit   Unit

case class TFun(t1: Typ, t2: Typ) extends Typ
   TFun(t1, t2)   t1 ⇒ t2

case class TRecord(fields: Map[String, Typ]) extends Typ
   TRecord(Map(f1 -> t1, ...))   {f1 : t1, ...}

case class TUnion(constrs: Map[String, Typ]) extends Typ
   TUnion(Map(c1 -> t1, ...))   [c1 : t1, ...]

case class TPtr(t1: Typ) extends Typ
   TPtr(t1)   Ptr[t1]

sealed abstract class PAnn

case object AnnVal extends PAnn
   AnnVal   ε

case object AnnName extends PAnn
   AnnName   name

case object AnnVar extends PAnn
   AnnVar   var

case object AnnRef extends PAnn
   AnnRef   ref

Figure 3: Representing in Scala the abstract syntax of IMPNAMESMALLA.
sealed abstract class Uop
  case object Neg extends Uop
    Neg -
  case object Not extends Uop
    Not !

sealed abstract class Bop
  case object Semi extends Bop
    Semi ;
  case object Plus extends Bop
    Plus +
  case object Minus extends Bop
    Minus -
  case object Times extends Bop
    Times *
  case object Lt extends Bop
    Lt <
  case object Eq extends Bop
    Eq =
  case object And extends Bop
    And &&
  case object Or extends Bop
    Or ||

Figure 4: Representing in Scala the abstract syntax of RECORDUNIONSMALLA.
\[\Gamma \vdash e : \tau\]

\[
\begin{array}{c}
\Gamma \vdash x : \Gamma(x) \\
\Gamma \vdash -e : \text{Int} \\
\Gamma \vdash !e_1 : \text{Boolean} \\
\Gamma \vdash e_1 : \text{Int} \\
\Gamma \vdash e_2 : \text{Int} \\
\Gamma \vdash e_1 \text{ bop } e_2 : \text{Int} \\
\Gamma \vdash e_1 : \tau \\
\Gamma \vdash e_2 : \tau \\
\tau \neq \tau_1 \Rightarrow \tau_2 \\
\Gamma \vdash e_1 == e_2 : \text{Boolean} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{Boolean} \\
\Gamma \vdash e_2 : \text{Boolean} \\
\text{bop} \in \{+, -, *, \&\&, ||\} \\
\Gamma \vdash e_1 \text{ bop } e_2 : \text{Boolean} \\
\Gamma \vdash e_1 : \tau \\
\Gamma \vdash e_2 : \tau \\
\tau \neq \tau_1 \Rightarrow \tau_2 \\
\Gamma \vdash e_1 == e_2 : \text{Boolean} \\
\Gamma \vdash \text{print}(e_1) : \text{Unit} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau \\
\Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_3 : \tau \\
\Gamma \vdash \text{if}(e_1) e_2 \text{ else } e_3 : \tau \\
\Gamma \vdash e : \tau \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e : [\ldots, c \vdash \tau_1, \ldots] \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : [\ldots, c_i : \tau_i, \ldots] \\
\Gamma \vdash e_i : \tau_i \quad \text{(for all } i\text{)} \\
\Gamma \vdash e \text{ match } c_i : x_i \Rightarrow e_j \cdots : \tau \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash n : \text{Int} \\
\Gamma \vdash b : \text{Boolean} \\
\Gamma \vdash () : \text{Unit} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash (x : \tau) => e : \tau' \\
\Gamma \vdash \text{name } x : \tau \Rightarrow e : \tau \Rightarrow \tau' \\
\end{array}
\]

Figure 5: Typing of the non-imperative constructs of IMPNAMESMALLA.
\( \text{Var } x : \text{Int} = 0 \)  
\( x = \text{true} \)

\[
\begin{align*}
\frac{x \to \tau \in \Gamma}{\Gamma \vdash x = e_2 : \text{Unit}} & \quad \frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash e : \tau'} & \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{var } x : \tau = > e : \tau \to \tau'}
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{new}(e) : \text{Ptr}[\tau]} & \quad \frac{\Gamma \vdash e : \text{Ptr}[\tau]}{\Gamma \vdash *e : \tau} & \quad \frac{\Gamma \vdash e_1 : \text{Ptr}[\tau]}{\Gamma \vdash e_2 : \tau} & \quad \frac{\Gamma \vdash e_1 = e_2 : \text{Unit}}{\Gamma \vdash *e_1 = e_2 : \text{Unit}}
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma \vdash (\text{ref } x : \tau) = > e : \text{Ref}[\tau] \Rightarrow \tau'}{\Gamma \vdash e : \tau'} & \quad \frac{\Gamma \vdash e_1 : \text{Ref}[\tau]}{\Gamma \vdash e_2 : \text{Ptr}[\tau]} & \quad \frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(x_2) : \tau'} & \quad \frac{\Gamma \vdash e_1 : \text{Ref}[\tau]}{\Gamma \vdash e_2 : \text{Ptr}[\tau]} & \quad \frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(\ast e_2) : \tau'} & \quad \frac{\Gamma \vdash \Lambda a : \text{Ptr}[\Lambda(a)]}{\Gamma \vdash \Lambda a : \text{Ptr}[\Lambda(a)]}
\end{align*}
\]

Figure 6: Typing of the imperative constructs of IMPNAMESMALLA.
Figure 7: Small-step operational semantics of the non-imperative constructs of call-by-value IMPNAMESMALLA.
Figure 8: Small-step operational semantics of the call-by-name and imperative constructs of IMPNAMESMALLA.
expressions \( e ::= \cdots \mid \text{unfold}(e_1) \mid \text{fold}_\tau(e_1) \)

types \( \tau ::= \cdots \mid T \mid \text{rec } T \cdot \tau \)

type variables \( T \)

case class Unfold(e1: Expr) extends Expr
  Unfold(e1) \text{ unfold}(e1)

case class Fold(t: Typ, e1: Expr) extends Expr
  Fold(t, e1) \text{ fold}_\tau(e1)

case class TVar(tvar: String) extends Typ
  TVar(T)

case class TRec(tvar: String, t1: Typ) extends Typ
  TRec(T, \tau) \text{ rec } T \cdot \tau

\[ \Gamma \vdash e : \tau \]

\( \Gamma \vdash e_1 : \text{rec } T \cdot \tau \)

\( \Gamma \vdash \text{unfold}(e_1) : ([\text{rec } T \cdot \tau]) / T \tau \)

\( \Gamma \vdash \text{fold}_{\text{rec } T \cdot \tau}(e_1) : \text{rec } T \cdot \tau \)

\( \langle M, e \rangle \rightarrow \langle M', e' \rangle \)

\textbf{DOUNFOLDFOLD}

\( \langle M, \text{unfold}(\text{fold}_\tau(v)) \rangle \rightarrow \langle M, v \rangle \)

\textbf{STEPUNFOLD}

\( \langle M, e_1 \rangle \rightarrow \langle M', e'_1 \rangle \)

\( \langle M, \text{unfold}(e_1) \rangle \rightarrow \langle M', \text{unfold}(e'_1) \rangle \)

\textbf{STEPFOLD}

\( \langle M, e_1 \rangle \rightarrow \langle M', e'_1 \rangle \)

\( \langle M, \text{fold}_\tau(e_1) \rangle \rightarrow \langle M', \text{fold}_\tau(e'_1) \rangle \)

Figure 9: Extending IMPNAMESMALLA with recursive types.