Homework 10

The primary purpose of this assignment is to practice the use of monads. We reimplement our interpreter from Homework 9 using the state monad to thread the memory through the evaluation. We do not consider new language features in this version of our interpreter. In fact, we simplify the language by removing all parameter passing modes except call by value.

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 expressing computation currently unfamilar to you.

Problem 1 JakartaScript Interpreter with State (20 Points)

We start from our language in Homework 9, but we remove all parameter passing modes except for call-by-value parameters. The syntax of the new language is shown in Figure 1. In Figure 2, we show the updated and new AST nodes.

Type Checking. The inference rules defining the typing relation are given in Figures 3 and 4. The only change compared to Homework 9 is that we no longer have to handle the different parameter passing modes. The new type inference function

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

has already been provided for you.

Evaluation. Your task is to implement a monadic version of the `eval` function in Homework 9.

The new big-step operational semantics is given in Figures 5 and 6. The rules are identical to those given in Homework 9, except that we only support pass-by-value parameters in function expressions.

- The `eval` function now has the following signature

  ```
  def eval(e: Expr): State[Mem, Val]
  ```

  This function needs to be completed.

The `State[S, R]` type is defined for you and shown in Figure 7. The essence of `State[S, R]` is that it encapsulates a function of type `S => (S,R)`, which can be seen as a computation that returns a value of type `R` with an input-output state of type `S`. The class `State` is declared `abstract` because it is missing an implementation for the method `apply`. The `apply` method is the encapsulated function of type `S=>(S,R)`. Seeing `State[Mem, Val]` as an encapsulated `Mem => (Mem,Val)`, we see how the judgment form $\langle M, e\rangle \Downarrow \langle M', v\rangle$ corresponds to the signature of `eval`.

For the evaluation rules that do not involve imperative features, the memory M is simply threaded through. The main advantage of using the encapsulated computation `State[Mem, Val]` is that this common-case threading is essentially put into the `State` data structure. One can view `State[Mem, Val]` as a "collection" that holds a computation over

$$
\begin{array}{rcll}
n & \in & \mathit{Num} & \text{numbers (double)} \\
s & \in & \mathit{Str} & \text{strings} \\
a & \in & \mathit{Addr} & \text{addresses} \\
b \in & \mathit{Bool} & ::= \mathbf{true} \mid \mathbf{false} & \text{Booleans} \\
x & \in & \mathit{Var} & \text{variables} \\
\tau \in & \mathit{Typ} & ::= \mathbf{Bool} \mid \mathbf{Num} \mid \mathbf{string} \mid \mathbf{Undefined} \mid & \text{types} \\
 & & \quad (\overline{\cancel{\mathit{mode}}\ \tau}) \Rightarrow \tau_0 & \\
v \in & \mathit{Val} & ::= \mathbf{undefined} \mid n \mid b \mid s \mid a \mid & \text{values} \\
 & & \quad \mathbf{function}\ p(\overline{\cancel{\mathit{mode}}\ x{:}\tau})\, t\ e & \\
e \in & \mathit{Expr} & ::= x \mid v \mid \mathit{uop}\ e \mid e_1\, \mathit{bop}\, e_2 \mid e_1\ ?\ e_2\ :\ e_3 \mid & \text{expressions} \\
 & & \quad \mathbf{console.log}(e) \mid e_1(\overline{e}) \mid & \\
 & & \quad \mathit{mut}\ x = e_1 ; e_2 & \\
\mathit{uop} \in & \mathit{Uop} & ::= \texttt{-} \mid \texttt{!} \mid \texttt{*} & \text{unary operators} \\
\mathit{bop} \in & \mathit{Bop} & ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{>} \mid & \text{binary operators} \\
 & & \quad \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{,} \mid \texttt{=} & \\
 & p & ::= x \mid \epsilon & \text{function names} \\
 & t & ::= {:}\tau \mid \epsilon & \text{return types} \\
\mathit{mut} \in & \mathit{Mut} & ::= \mathbf{const} \mid \mathbf{let} & \text{mutability} \\
M \in & \mathit{Mem} & = \mathit{Addr} \rightharpoonup \mathit{Val} & \text{memories}
\end{array}
$$

Figure 1: Abstract syntax of JakartaScript

`Mem` that produces a `Val`, and thus, we can define a `map` method that creates an updated `State` "collection" holding the result of the callback `f` to the `map`. Applying `map` methods on different data structures is so frequent that Scala has an expression form

```
for (...) yield ...
```

that works for any data structure that defines a `map` method (cf., OSV).

We suggest that you extend the given template of the `eval` function case by case. For each case, first copy over your code for the corresponding case from the `eval` function of Homework 9. Then turn this code into a monadic version that hides the threading of the memory in the `State[Mem,Val]` data structure. Some of the cases are already provided for you.

```
/** Mutabilities */
enum Mut:
  case MConst, MLet // <~ const, let

/** Binary Operators */
enum Bop:
  ...
  case Assign // <~ =

/** Unary Operators */
enum Uop:
  ...
  case Deref // <~ *

/** Expressions */
type Params = List[(String, Typ)]
enum Expr:
  ...
  case Decl(mut: Mut, x: String, ed: Expr, eb: Expr) // <~ mut x = ed; eb

  case Function(p: Option[String], xs: Params, t: Option[Typ], e: Expr)
    // <~ function p(x1: τ1,...,xk: τk) t e

  /** Addresses */
  case Addr private[ast] (addr: Int) // <- a

/** Types */
enum Typ:
  ...
  case TFunction(ts: List[Typ], tret: Typ) // <~ (τ1,...,τk) => τret
```

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

$$\frac{}{\Gamma \vdash b : \texttt{Bool}}\ \textsc{TypeBool} \qquad \frac{}{\Gamma \vdash n : \texttt{Num}}\ \textsc{TypeNum} \qquad \frac{}{\Gamma \vdash s : \texttt{string}}\ \textsc{TypeStr}$$

$$\frac{}{\Gamma \vdash \texttt{undefined} : \texttt{Undefined}}\ \textsc{TypeUndefined}$$

$$\frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{-}\, e : \texttt{Num}}\ \textsc{TypeUMinus} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash \texttt{!}\ e : \texttt{Bool}}\ \textsc{TypeNot}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \texttt{Bool} \quad bop \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Bool}}\ \textsc{TypeAndOr}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \texttt{,}\ e_2 : \tau_2}\ \textsc{TypeSeq} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{console.log}(e) : \texttt{Undefined}}\ \textsc{TypePrint}$$

$$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{string}}\ \textsc{TypePlusStr}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \quad \Gamma \vdash e_2 : \texttt{Num} \quad bop \in \{\texttt{+}, \texttt{*}, \texttt{/}, \texttt{-}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Num}}\ \textsc{TypeArith}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{Num}, \texttt{string}\} \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{Bool}}\ \textsc{TypeInequal}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \text{ has no function types} \quad bop \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{Bool}}\ \textsc{TypeEqual}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}\ \textsc{TypeIf}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \Rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}\ \textsc{TypeCall}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (\texttt{const}, \tau_1)] \ldots [x_k \mapsto (\texttt{const}, \tau_k)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \ldots, \tau_k) \Rightarrow \tau\end{array}}{\Gamma \vdash (x_1 \texttt{:} \tau_1, \ldots, x_k \texttt{:} \tau_k)\ \texttt{=>}\ e : \tau'}\ \textsc{TypeFun}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x \mapsto \tau'][x_1 \mapsto (\texttt{const}, \tau_1)] \ldots [x_k \mapsto (\texttt{const}, \tau_k)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \ldots, \tau_k) \Rightarrow \tau\end{array}}{\Gamma \vdash \texttt{function}\ x(x_1 \texttt{:} \tau_1, \ldots, x_k \texttt{:} \tau_k) \texttt{:} \tau\ e\ : \tau'}\ \textsc{TypeFunRec}$$

Figure 3: Type checking rules for non-imperative primitives of JakartaScript (no changes compared to Homework 8)

$$\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (mut, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash mut\ x \,\texttt{=}\, e_d\,\texttt{;}\, e_b : \tau_b}\ \textsc{TypeDecl}$$

$$\frac{x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) = (mut, \tau)}{\Gamma \vdash x : \tau}\ \textsc{TypeVar}$$

$$\frac{\Gamma(x) = (\texttt{let}, \tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x \,\texttt{=}\, e : \tau}\ \textsc{TypeAssignVar}$$

Figure 4: Type checking rules for imperative primitives of JakartaScript

$$\frac{}{\langle M, v\rangle \Downarrow \langle M, v\rangle}\ \textsc{EvalVal} \qquad \frac{\langle M, e\rangle \Downarrow \langle M', n\rangle}{\langle M, \texttt{-}\, e\rangle \Downarrow \langle M', -n\rangle}\ \textsc{EvalUMinus} \qquad \frac{\langle M, e\rangle \Downarrow \langle M', b\rangle}{\langle M, \texttt{!}\ e\rangle \Downarrow \langle M', \texttt{!}\ b\rangle}\ \textsc{EvalNot}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{true}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalAndTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{false}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalOrFalse}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{false}\rangle}{\langle M, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle M_1, \textbf{false}\rangle}\ \textsc{EvalAndFalse} \qquad \frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{true}\rangle}{\langle M, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle M_1, \textbf{true}\rangle}\ \textsc{EvalOrTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{,}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalSeq}$$

$$\frac{\langle M, e\rangle \Downarrow \langle M', v\rangle \quad v \text{ printed}}{\langle M, \texttt{console.log}(e)\rangle \Downarrow \langle M', \textbf{undefined}\rangle}\ \textsc{EvalPrint}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad n = n_1 + n_2}{\langle M, e_1 \texttt{+} e_2\rangle \Downarrow \langle M_2, n\rangle}\ \textsc{EvalPlusNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, s_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, s_2\rangle \quad s = s_1 + s_2}{\langle M, e_1 \texttt{+} e_2\rangle \Downarrow \langle M_2, s\rangle}\ \textsc{EvalPlusStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad n = n_1\, bop\, n_2 \quad bop \in \{\texttt{*}, \texttt{/}, \texttt{-}\}}{\langle M, e_1\, bop\, e_2\rangle \Downarrow \langle M_2, n\rangle}\ \textsc{EvalArith}$$

$$\frac{\langle M, e_d\rangle \Downarrow \langle M_d, v_d\rangle \quad \langle M_d, e_b[v_d/x]\rangle \Downarrow \langle M', v_b\rangle}{\langle M, \textbf{const}\ x \texttt{=} e_d\texttt{;} e_b\rangle \Downarrow \langle M', v_b\rangle}\ \textsc{EvalConstDecl}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad b = n_1\ bop\ n_2 \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalInequalNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, s_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, s_2\rangle \quad b = s_1\ bop\ s_2 \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalInequalStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle \quad b = (v_1\ bop\ v_2)}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalEqual}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{true}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalIfThen}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \textbf{false}\rangle \quad \langle M_1, e_3\rangle \Downarrow \langle M_3, v_3\rangle}{\langle M, e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \Downarrow \langle M_3, v_3\rangle}\ \textsc{EvalIfElse}$$

Figure 5: Big-step operational semantics of non-imperative primitives of JakartaScript (no changes compared to Homework 9).

$$\frac{\langle M, e\rangle \Downarrow \langle M', v\rangle \quad a \in \mathsf{dom}(M')}{\langle M, \star a = e\rangle \Downarrow \langle M'[a \mapsto v], v\rangle}\ \textsc{EvalAssignVar}$$

$$\frac{a \in \mathsf{dom}(M)}{\langle M, \star a\rangle \Downarrow \langle M, M(a)\rangle}\ \textsc{EvalDerefVar}$$

$$\frac{\begin{array}{c}\langle M, e_d\rangle \Downarrow \langle M_d, v_d\rangle \quad a \notin \mathsf{dom}(M_d) \\ M' = M_d[a \mapsto v_d] \quad \langle M', e_b[\star a/x]\rangle \Downarrow \langle M'', v_b\rangle\end{array}}{\langle M, \texttt{let}\ x = v_d\,;e_b\rangle \Downarrow \langle M'', v_b\rangle}\ \textsc{EvalLetDecl}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \texttt{function}\ x_0(\overline{x_i : \tau_i})\texttt{:}\,\tau\ e \\ v_0' = ((\overline{x_i : \tau_i})\ \texttt{=>}\ e[v_0/x_0]) \quad \langle M_0, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(\overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallRec}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = x_1 : \tau_1, \overline{x_i : \tau_i}\ \texttt{=>} : \tau\ e \\ \langle M_0, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad v_0' = ((\overline{x_i : \tau_i})\ \texttt{=>}\ e[v_1/x_1]) \\ \langle M_1, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(e_1, \overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallConst}$$

$$\frac{\langle M, e_0\rangle \Downarrow \langle M_0,\ \texttt{=>}\ t\ e\rangle \quad \langle M_0, e\rangle \Downarrow \langle M', v\rangle}{\langle M, e_0()\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCall}$$

Figure 6: Big-step operational semantics of imperative primitives of JakartaScript.

```scala
abstract class State[S,+R]:
  def apply(s: S)

  def map[P](f: R => P): State[S,P] =
    new State((s: S) =>
      val (sp, r) = this(s) // same as this.apply(s)
      (sp, f(r))
    )

  def flatMap[P](f: R => State[S,P]): State[S,P] =
    new State((s: S) =>
      val (sp, r) = this(s)
      f(r)(sp) // same as f(r).apply(sp)
    )

object State:
  def apply[S,R](f: S => (S,R)): State[S,R] =
    new State(f)
  def init[S]: State[S,S] =
    State( s => (s, s) )
  def insert[S,R](r: R): State[S,R] =
    init map ( _ => r )
  def read[S,R](f: S => R) =
    init map ( s => (s, f(s)) )
  def write[S](f: S => S): State[S,Unit] =
    init flatMap ( s => State( _ => (f(s), ()) ) )
```

Figure 7: Implementation of the state monad