Homework 8

The primary purpose of this assignment is to understand the interplay between type checking and evaluation. Concretely, you will extend your JakartaScript interpreter from Homework 7 with a static type checker.

Problem 1 Type Checking and Type Inference (16 Points)

The purpose of this exercise is to understand how the type inference rules work and what some of their limitations are. We suggest that you solve this exercise as a warm-up before you implement the type checker in Problem 2.

(a) Use the type inference rules from Figure 3 to determine for each of the following expressions whether the expression is well-typed. If an expression is not well-typed, explain why. If it is well-typed, give the inferred type. You do not need to show the individual inference steps. **(7 Points)**

(i)

```
const x = 2;
const y = x + 1;
x * y
```

(ii)

```
const x = true;
const y = x + 1;
x * y
```

(iii)

```
const f = function f(x: Num): Num => Num (
    (y: Num) => x === y ? 1 : y * f(x)(y + 1)
  );
f(3)
```

(iv)

```
const f = function f(x: Num): Num => Num (
    (y: Num) => x === y ? 1 : y * f(x)(y + 1)
  );
f(3) === f(3)
```

(v)

```
const f = function f(x: Num) (
    (y: Num) => x === y ? 1 : y * f(x)(y + 1)
  );
f(3)(1)
```

(b) For each of the following programs, find concrete types for the missing parameter type annotations τ_1, τ_2, and τ_3 such that the given program is well-typed according to the rules in Figure 3. If no such types exist, explain why. If you can find type annotations that make the program well-typed, what is the inferred type of `f` for your annotations? Are your chosen types the only annotations that work? If not, give at least one other choice of annotations that also makes the program well-typed. **(9 Points)**

(i)

```
1 const f = (x: τ1) => (y: τ2) => (z: τ3) => x(y(z));
2 const g = (x: Num) => x + 1;
3 const h = (x: Num) => x * 2;
4 f(h)(g)(3)
```

(ii)

```
1 const f = (x: τ1) => (y: τ2) => (z: τ3) => x(y(z));
2 const g = (x: τ3) => x;
3 f(g)(g)
```

(iii)

```
1 const f = (x: τ1) => (y: τ2) => x(y);
2 const g = (x: Bool) => x ? 1 : 0;
3 const h = (x: Num) => x + x;
4 f(g)(true) + f(h)(1)
```

Problem 2 Statically-Typed JakartaScript (24 Points)

In this exercise, we will implement a strongly, statically-typed version of JakartaScript. We will not permit any implicit type conversions and will guarantee the absence of dynamic type errors.

We extend JakartaScript with a language of types τ (see Figure 1). We also annotate function parameters with types and add optional return type annotations. Like last time, functions can 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). Note that we no longer need the special value typeerror to indicate a dynamic type error during evaluation. In Figure 2, we show the updated and new AST representation.

Your main task in this exercise is to 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 subexpression, we infer the type of an expression, by recursively inferring the type of each subexpression. 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 Γ, expression e has type τ." We will implement a function

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

$$
\begin{array}{rll}
n \in \mathit{Num} & & \text{numbers (double)} \\
s \in \mathit{Str} & & \text{strings} \\
x \in \mathit{Var} & & \text{variables} \\
b \in \mathit{Bool} & ::= \texttt{true} \mid \texttt{false} & \text{Booleans} \\
\tau \in \mathit{Typ} & ::= \texttt{Bool} \mid \texttt{Num} \mid \texttt{String} \mid \texttt{Undefined} \mid & \text{types} \\
 & \quad (\tau_1, \ldots, \tau_n) \Rightarrow \tau_0 & \\
v \in \mathit{Val} & ::= \texttt{undefined} \mid n \mid b \mid s \mid \texttt{function}\ p(\overline{x:\tau})t\ e \mid & \text{values} \\
 & \quad \cancel{\texttt{typeerror}} & \\
e \in \mathit{Expr} & ::= x \mid v \mid \mathit{uop}\ e \mid e_1\ \mathit{bop}\ e_2 \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \mid & \text{expressions} \\
 & \quad \texttt{const}\ x = e_1; e_2 \mid \texttt{console.log}(e) \mid e(e_1, \ldots, e_n) & \\
\mathit{uop} \in \mathit{Uop} & ::= - \mid \texttt{!} & \text{unary operators} \\
\mathit{bop} \in \mathit{Bop} & ::= + \mid - \mid * \mid / \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{,} & \text{binary operators} \\
p & ::= x \mid \epsilon & \text{function names} \\
t & ::= :\tau \mid \epsilon & \text{return types}
\end{array}
$$

Figure 1: Abstract syntax

that corresponds directly to this judgment form. It takes as input a type environment `env` (Γ) and an expression `e`, and returns a type (τ).

The TypeEquality rule is slightly informal in stating "τ 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`** iff a function type appears in the input, so this rule can be implemented 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, e: Expr) extends JsException
```

where `tbad` is the type that is inferred for expression `e`. These arguments are used to construct a useful error message. We also provide a helper function `err` to simplify throwing this exception.

It is informative to compare the typing rules with the new rules for our big-step operational semantics in Figure 4. Note that the only modification compared to the evaluation rules of Homework 7 is that we no longer use implicit type conversion functions. Moreover, we now require that the number of arguments provided in a call exactly matches the number of parameters of the called function. We rely on the type checker to detect all the cases where the evaluation may get stuck. The implementation of the interpreter will indicate such cases by throwing a `StuckError`. If you implement your type checker correctly, you will never observe a `StuckError` during the evaluation of a well-typed program.

Complete the functions `subst`, `eval`, and `typeInfer` provided in the code package. We suggest the following step-by-step order:

(a) First, bring over the missing cases for the implementations of `subst` and `eval` from

```
/** Types */
enum Typ:
  case TBool // <~ Bool
  case TNum // <~ Num
  case TStr // <~ String
  case TUndefined // <~ Undefined
  case TFunction(ts: List[Typ], tret: Typ) // <~ (τ1,...,τk) => τret

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

  case Call(e: Expr, es: List[Expr]) // <~ e(e1,...,ek)
```

Figure 2: The abstract syntax of JakartaScript as represented in Scala. After each **case**, we show the corresponding JavaScript expression represented by that case.

Homework 7. You will only need to make minor modifications to account for the updated representation of function expressions, which now include type annotations.

(b) Then complete the function `typeInfer`. Start with the cases for the basic expressions. Then do the more complex cases for `Function` and `Call`.

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

$$\frac{}{\Gamma \vdash \texttt{undefined} : \texttt{Undefined}}\ \textsc{TypeUndefined} \qquad \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \textsc{TypeVar}$$

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

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

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

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

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

$$\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto \tau_d] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \texttt{const}\ x \texttt{=} e_d \texttt{;} e_b : \tau_b}\ \textsc{TypeConst}$$

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

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

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

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

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

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

$$\frac{\Gamma' = \Gamma[x \mapsto \tau'][x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_n] \quad \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \ldots, \tau_n) \Rightarrow \tau}{\Gamma \vdash \texttt{function}\ x(x_1 \texttt{:} \tau_1, \ldots, x_n \texttt{:} \tau_n) \texttt{:} \tau\ e\ : \tau'}\ \textsc{TypeFunctionRec}$$

Figure 3: Type checking rules for JakartaScript

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

$$\frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow v_2}{e_1 \,\texttt{\&\&}\, e_2 \Downarrow v_2}\ \textsc{EvalAndTrue} \qquad \frac{e_1 \Downarrow \texttt{false} \quad e_2 \Downarrow v_2}{e_1 \,\texttt{||}\, e_2 \Downarrow v_2}\ \textsc{EvalOrFalse}$$

$$\frac{e_1 \Downarrow \texttt{false}}{e_1 \,\texttt{\&\&}\, e_2 \Downarrow \texttt{false}}\ \textsc{EvalAndFalse} \qquad \frac{e_1 \Downarrow \texttt{true}}{e_1 \,\texttt{||}\, e_2 \Downarrow \texttt{true}}\ \textsc{EvalOrTrue}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \,\texttt{,}\, e_2 \Downarrow v_2}\ \textsc{EvalSeq} \qquad \frac{e \Downarrow v \quad v \text{ printed}}{\texttt{console.log(}e\texttt{)} \Downarrow \texttt{undefined}}\ \textsc{EvalPrint}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 \,\texttt{+}\, e_2 \Downarrow n}\ \textsc{EvalPlusNum} \qquad \frac{e_1 \Downarrow s_1 \quad e_2 \Downarrow s_2 \quad s = s_1 + s_2}{e_1 \,\texttt{+}\, e_2 \Downarrow s}\ \textsc{EvalPlusStr}$$

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

$$\frac{e_d \Downarrow v_d \quad e_b[v_d/x] \Downarrow v_b}{\texttt{const}\ x \texttt{=} e_d \texttt{;} e_b \Downarrow v_b}\ \textsc{EvalConstDecl}$$

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

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

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad b = (v_1\ bop\ v_2)}{e_1\ bop\ e_2 \Downarrow b}\ \textsc{EvalEqual}$$

$$\frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow v_2}{e_1 \,\texttt{?}\, e_2 \,\texttt{:}\, e_3 \Downarrow v_2}\ \textsc{EvalIfThen} \qquad \frac{e_1 \Downarrow \texttt{false} \quad e_3 \Downarrow v_3}{e_1 \,\texttt{?}\, e_2 \,\texttt{:}\, e_3 \Downarrow v_3}\ \textsc{EvalIfElse}$$

$$\frac{\begin{array}{c} e_0 \Downarrow x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n \texttt{=>}\ t\ e \\ e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \quad e' = e[v_1/x_1] \ldots [v_n/x_n] \quad e' \Downarrow v \end{array}}{e_0\texttt{(}e_1, \ldots, e_n\texttt{)} \Downarrow v}\ \textsc{EvalCall}$$

$$\frac{\begin{array}{c} e_0 \Downarrow v_0 \quad v_0 = \texttt{function}\ x_0\texttt{(}x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n\texttt{)}t\ e \\ e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \quad e' = e[v_0/x_0] \ldots [v_n/x_n] \quad e' \Downarrow v \end{array}}{e_0\texttt{(}e_1, \ldots, e_n\texttt{)} \Downarrow v}\ \textsc{EvalCallRec}$$

Figure 4: Big-step operational semantics of JakartaScript