Exercise 1

Consider the typing rules in annex. Assuming an empty initial environment, type check the following expressions. Write down the derivation trees.

```
3 + 5
```

```
val x: Int = 4; val y: Int = x + x; x * y
```

Assuming that the initial environment is `{(x, Boolean), (power, (Int, Int) => Int)}`, type check the following expressions. Write down the derivation trees.

```
val x: Int = 7; if (x < 100) power(x, 10) else error("Too big!")
```

```
val x: Int = if (x) 1 else 0; x * 2
```

Exercise 2

In this exercise, we will extend our language with constructs to create and manipulate sequences. First of all, we add types to represent sequences. For every type `T`, we introduce the type `Seq[T]` which represents sequences of values of type `T`. We then add syntax to represent sequences. The syntax for literal sequences is given by the regular expression **`[`** `(`*`Expr`* **`,`** *`Expr`*`)*)?` **`]`**. For instance, the following are *syntactically* valid sequences:

```
[]
[1, 2, 3, 4]
[val x = 2; x, 1; 2; if (true) 42 else 17 + 3]
[1, true, x, ()]
```

We will use the binary operator ++ to represent concatenation of sequences (in addition to string concatenation). We also introduce two additional constructs: `atIndex` and `indexOf`.

1. `atIndex` takes as arguments a sequence and an index (an `Int`), and returns the value stored in the sequence at the specified index. It is unspecified what happens when the index is out of bounds (potentially a runtime crash).
2. `indexOf` takes two arguments and returns the index at which the second argument appears in the first. The first argument should be a sequence and the second a value of the appropriate type. The value returned will always be an `Int` (`-1` will be returned when the sequence doesn't contains the requested value).

Question 1

Write the typing rules for sequence literals, concatenation of sequences, `atIndex` and `indexOf`.

Question 2

Assuming an empty initial environment, type check the following expressions:

```
atIndex([1, 2, 3], 1) == 2

[1, 2, true]

val x: Boolean = true; [x, false, x]

atIndex([] ++ [], 0)
```

Is there something particular with the last example ?

Question 3

Being a huge fan of Scala, you decide to introduce for-comprehensions to your language. For-comprehensions are expressions of the form:

for { x_1 <- xs_1; x_2 <- xs_2; ...; x_n <- xs_n } yield e

Where x_1, x_2, …, x_n are variables, xs_1, xs_2, …, xs_n are expressions and `e` is an expression. The meaning of such expressions is the same as in Scala. Note that variables bound in earlier bindings can appear in subsequent bindings. For instance, the following should be valid for-comprehension expressions:

```
for {
  x <- [1, 2]
} yield x + 1

for {
  x <- [1, 2, 3];
  y <- [x > 1, false];
  z <- [x, if (y) 1 else 0]
} yield x + z
```

Your goal is to write down the typing rule(s) for for-comprehension expressions.

VARIABLE
$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T}$$

INT LITERAL
$$\frac{i \text{ is an integer literal}}{\Gamma \vdash i : \texttt{Int}}$$

STRING LITERAL
$$\frac{s \text{ is a string literal}}{\Gamma \vdash s : \texttt{String}}$$

UNIT
$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{Unit}}$$

BOOLEAN LITERAL
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : \texttt{Boolean}}$$

ARITH. BIN. OPERATORS
$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int} \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Int}}$$

ARITH. COMP. OPERATORS
$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int} \qquad op \in \{\texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

ARITH. NEGATION
$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{-}e : \texttt{Int}}$$

BOOLEAN BIN. OPERATORS
$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \qquad \Gamma \vdash e_2 : \texttt{Boolean} \qquad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

BOOLEAN NEGATION
$$\frac{\Gamma \vdash e : \texttt{Boolean}}{\Gamma \vdash \texttt{!}e : \texttt{Boolean}}$$

STRING CONCATENATION
$$\frac{\Gamma \vdash e_1 : \texttt{String} \qquad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1 \ \texttt{++} \ e_2 : \texttt{String}}$$

EQUALITY
$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \ \texttt{==} \ e_2 : \texttt{Boolean}}$$

SEQUENCE
$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \ \texttt{;} \ e_2 : T_2}$$

LOCAL VARIABLE DEFINITION
$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, n : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{val} \ n \ \texttt{:} \ T_1 \ \texttt{=} \ e_1 \ \texttt{;} \ e_2 : T_2}$$

FUNCTION/CLASS CONSTRUCTOR INVOCATION
$$\frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n \qquad \Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T}{\Gamma \vdash f(e_1, \ldots, e_n) : T}$$

IF-THEN-ELSE
$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if} \ \texttt{(}e_1\texttt{)} \ \texttt{\{}e_2\texttt{\}} \ \texttt{else} \ \texttt{\{}e_3\texttt{\}} : T}$$

ERROR
$$\frac{\Gamma \vdash e : \texttt{String}}{\Gamma \vdash \texttt{error(}e\texttt{)} : T}$$