1. (6 points) Consider the following class definitions.

```java
class X {
    z: Bool;
    c: Int;
    s: String;
    x: SELF_TYPE;
    foo(): SELF_TYPE { x };
    getC(): Int { c };
};
class Y inherits X {
    y: SELF_TYPE;
    h(b: Object): Object { (* EXPRESSION *) };
};
```

Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of (* EXPRESSION *) in the body of the method \( h \), show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

(a) (2 points) \( x \leftarrow y\).foo()
    Solution: SELF_TYPE
(b) (2 points) \( s.length() + z \)
(c) (2 points)
    ```java
    case x.foo() of
        z: X => z.getC();
        z: Y => z;
        l: Object => true;
    esac
    ```
    Solution: Object
2. A type derivation shows the inductive proof of a typing judgement as a tree. For example, the type derivation for \( O[\text{Int}/x] \vdash x + 1 : \text{Int} \) is given as follows:

\[
O[\text{Int}/x](x) = \text{Int} \\
O[\text{Int}/x] \vdash x : \text{Int} \\
O[\text{Int}/x] \vdash 1 : \text{Int} \\
O[\text{Int}/x] \vdash x + 1 : \text{Int}
\]

Show the type derivation for the following typing judgement:

\( O[\text{String}/\text{str}] \vdash \text{case str of a : Int = } 2; \ b : \text{Object} = \text{False}; \text{ esac} \)

Solution: (5 points)

\[
O[\text{String}/\text{str}](\text{str}) = \text{String} \\
O[\text{String}/\text{str}] \vdash \text{str : String} \\
2 \text{ is an integer constant} \\
O[\text{String}/\text{str}, \text{Int}/a] \vdash 2 : \text{Int} \\
O[\text{String}/\text{str}, \text{Object}/b] \vdash \text{False : Bool} \\
\text{Int} \sqcup \text{Bool} = \text{Object} \\
O[\text{String}/\text{str}] \vdash \text{case str of a : Int = } 2; \ b : \text{Object} = \text{False}; \text{ esac : Object}
\]

3. Suppose we want to add a special type to Cool that can either be an Int or a special value that represents “no result.” Our type MaybeInt will take two forms, Some(n), where n is an integer, and Nothing. Suppose we can create values of this form from the following compiler-implemented methods: (assume that they are defined Object so they are always accessible):

createSomething(n : Int) : MaybeInt
createNothing() : MaybeInt

For example, you could use MaybeInt to write a safe division function that handles division by zero:

\[
\text{safeDivision(numerator : Int, denominator : Int) : MaybeInt} \{ \\
\quad \text{if (denominator == 0) then} \\
\quad \quad \text{createNothing()} \\
\quad \text{else} \\
\quad \quad \text{createSomething(numerator ÷ denominator)} \\
\quad \}\]

Another example of a function that uses MaybeInt would be a integer parsing function. This function will return a MaybeInt with value Some(n) if the parse succeeds and value Nothing if it fails. The signature would then be:

\[
\text{parseInt(str: String) : MaybeInt}
\]

As an example, parseInt("123") would return Some(123) and parseInt("124a") would return Nothing. To be able to use the value inside of a MaybeInt, we will introduce pattern matching syntax to capture the value and bind it to a value in an expression. Similar to how a switch statement in other languages goes to a branch depending on the value of a variable, a match statement will go to a branch depending on the form of MaybeInt, while also possibly introducing a new value into the scope. The value of a match expression is then the value of the expression on the right side of the branch that was taken.

\[
\text{let maybe_int : MaybeInt <- parseInt("23") in} \\
\text{match(maybe_int) \{ \\
\quad \text{Some(n) => "The number is ".concat(n.to_string()),} \\
\quad \text{Nothing => "There was no number\"} \\
\}}
\]
This expression has the value “The number is 23”. The explicit grammar rule for match is:

$$
expr ::= \text{match } (expr) \{ \text{Some}(ID) => expr, \text{Nothing} => expr \}
$$

The identifier in the Some branch is introduced into the scope of the expression on the right side and initialized to the value inside the MaybeInt. Your task is to write a type checking rule for $\text{match } (e_1) \{ \text{Some}(id) => e_2, \text{Nothing} => e_3 \}$, as well as the operational semantics. Hint: For the operational semantics, model your solution off of the operational semantics for Let and If-True/False. Assume that when looking up a MaybeInt in the store that it will either return Some($v$) or Nothing, so you may need to write two separate rules.

Solution: (15 points)

Type Checking: (5 points)

\[
\begin{align*}
O, M, C & \vdash e_1 : \text{MaybeInt} \\
O[\text{Int/id}], M, C & \vdash e_2 : T_2 \\
O, M, C & \vdash e_3 : T_3 \\
O, M, C & \vdash \text{match } (e_1) \{ \text{Some}(id) => e_2, \text{Nothing} => e_3 \} : T_2 \sqcup T_3
\end{align*}
\]

MaybeInt-Some Operational Semantics: (5 points)

\[
\begin{align*}
& so, S_1, E \vdash e_1 : \text{Some}(v_1), S_2 \\
& l_1 = \text{newloc}(S_2) \\
& S_3 = S_2[v_1/l_1] \\
& E' = E[l_1/id] \\
& so, S_3, E' \vdash e_2 : v_2, S_4 \\
so, S_1, E & \vdash \text{match } (e_1) \{ \text{Some}(id) => e_2, \text{Nothing} => e_3 \} : v_2, S_4
\end{align*}
\]

MaybeInt-Nothing Operational Semantics: (5 points)

\[
\begin{align*}
& so, S_1, E \vdash e_1 : \text{Nothing}, S_2 \\
& so, S_2, E \vdash e_3 : v_3, S_3 \\
& so, S_1, E \vdash \text{match } (e_1) \{ \text{Some}(id) => e_2, \text{Nothing} => e_3 \} : v_3, S_3
\end{align*}
\]