1. (8 pts) Consider the following program fragment in Cool (using standard Cool type rules, scoping rules and general semantics):

```cool
class A {
    f1(): Int {
        let a: Int in {
            a <- x;
            a;
        }
    }
    f2(): Int {
        let a: Int in {
            a <- 9;
            let x: Int <- 5 in {
                a <- f1();
            };
            a;
        };
        x: Int <- 0;
    }
}

class B inherits A {
    f3(): Int {
        let a: Int in {
            a <- x;
            let a: Int <- 6 in {
                a <- f2() + 3;
            };
            a;
        }
    }
    f4(): Int {
        let a: Int in {
            a <- x;
            let y: Int <- 7 in {
                let x: Int <- y in {
                    y <- 8;
                    a <- x;
                };
            };
            a;
        }
    }
    f5(): Object {
        let io: IO <- New IO, z: Int (* <- [Placeholder B] *) in {
            let w: Int <- 2 in {
                let z: Int <- w in {
                    (* [Placeholder A] *)
                    io.out_string("The secret will be: ");
                    io.out_int(z);
                }
            };
            io.out_string("The secret is: ");
        }
    }
}
```
Answer:

- **Statement 1** prints: 0. It prints the value of method `A.f2()` which evaluates to the value of `A.f1()`, which evaluates to `x`. `x` is not redefined within the local scope of `f1()`, so the next static scope is tried, which is the scope of class `A`. In this scope, `x` is defined and has value of 0 (note that we never override the value of `o.x` anywhere in the above fragment). The let statement of `f2` does not change the scope of `x` in `f1`, since Cool is statically scoped, rather than dynamically scoped.

- **Statement 2** prints: 0. It prints the value of method `B.f3()` which first sets the value of `a` to `x` (which resolves to `A.x` due to not being defined in the local scope and not being overridden by subclass `B`), then it enters a let block in which the value of `a` is obscured by a redefinition of `a` in a nested scope, then exits the let block removing the redefinition `a`, and finally evaluates to `a`. Since a new definition of variable `a` is present inside the scope of the let statement, changes to `a` within such statement can be safely ignored and won't affect the value of `a` in the outer scope.

- **Statement 3** prints: 7. It prints the value of method `B.f4()` which evaluates to the value of `a`. This variable is last set to `x` inside the two nested lets (none of them obscures value `a`, which remains scoped to the entire method). This value of `x` is set to `y` in the scope of the innermost let, which in turn is set to 7 by the outer let. Thus the value of `a` on exit from `B.f4()` is 7.
(b) \textbf{(4 pts)} Suppose you are told that you can only modify the code of the program fragment by replacing the comment including [Placeholder A] for any valid Cool expression you wish to write (your expression must match the Cool syntax for expr as given on page 16 of the Cool Manual). You are not allowed to edit the code anywhere else. Then, an adversary will take the code and replace only the comment including [Placeholder B] by a single valid Cool integer literal and run the program. Can you cause the line following [Placeholder A] to correctly predict the value printed in statement 4? If yes, provide the code fragment you should use to replace [Placeholder A]. If no, explain why. The resulting program after replacing both placeholders must be a valid Cool program.

\textbf{Answer:} This is not possible, as the only place the value of [Placeholder B] gets saved to is variable z in the scope of the let statement of lines 34-42. However, [Placeholder A] falls within the scope of the let statement of lines 36-39, which obscures that definition of z with a new definition setting z to the value of w (=2). Given this, the value of [Placeholder B] is not accessible at the point our code would be inserted and only becomes visible again right before the print of statement 4.

Note that a well formed Cool expr cannot break out of the scope in which is declared, as it must only close scopes it itself opens (e.g. due to a let or case expression). This is in contrast to arbitrary text being accepted at [Placeholder A], in which case one can escape the scope as ”}; }; [escaped code] let w:...”.
2. (12 pts) Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[Int/y] \vdash y + y : Int$:

$$
\frac{O[Int/y](y) = Int \quad O[Int/y], M, C \vdash y : Int}{O[Int/y], M, C \vdash y + y : Int}
$$

Consider the following Cool program fragment:

```plaintext
class A {
    i: Int;
    j: Int;
    b: Bool;
    s: String;
    o: SELF_TYPE;
    foo(): SELF_TYPE { o };
    bar(): Int { 2 * i + j - i / j - 3 * j };
}
class B inherits A {
    p: SELF_TYPE;
    baz(a: Int, b: Int): Bool { a = b };
    test(c: Object): Object { (* [Placeholder C] *) };
}
```

Note that the environments $O$ and $M$ at the start of the method `test(...)` are as follows:

$$
O = \emptyset[Int/i][Int/j][Bool/b][String/s][SELF_TYPE_B/o][SELF_TYPE_B/p][Object/c][SELF_TYPE_B/self]
$$

$$
M = \emptyset[(SELF_TYPE)/(A, foo)][(Int)/(A, bar)][(Int, Int, Bool)/(B, baz)][(Object)/(B, test)]
$$

For each of the following expressions replacing [Placeholder C], provide the type derivation and final type of the expression, if it is well typed, otherwise explain why it isn’t. Assume Cool type rules (you may omit subtyping relationships from the rules when the type is the same, e.g. $\text{Bool} \leq \text{Bool}$).
(a) (3 pts)

\[ b \leftarrow p.\text{baz}(p.\text{bar}(), i) \]

Answer:

\[
\begin{array}{c}
\frac{O(p) = \text{SELF\_TYPE}_B}{O, M, B \vdash p : \text{SELF\_TYPE}_B} \\
\frac{O, M, B \vdash p : \text{SELF\_TYPE}_B \quad M(B, \text{baz}) = (\text{Int, Int, Bool})}{O, M, B \vdash \text{baz}(p.\text{bar}(), i) : \text{Bool}} \\
\frac{O(i) = \text{Int}}{O, M, B \vdash i : \text{Int}} \\
\end{array}
\]

And:

\[
\begin{array}{c}
\frac{O, M, B \vdash \text{baz}(p.\text{bar}(), i) : \text{Bool}}{O, M, B \vdash b \leftarrow \text{baz}(p.\text{bar}(), i) : \text{Bool}} \\
\frac{O(i) = \text{Int}}{O, M, B \vdash i : \text{Int}} \\
\end{array}
\]

Thus, type is \textbf{Bool}.

\textbf{Alternative:} Note that the last rule directly takes \(O(b) = \text{Bool}\) as a premise. This is consistent with the Cool Manual type rule for ASSIGN. However, the following alternative derivation, which uses a slightly different, but natural, version of the rule, is also acceptable:

\[
\begin{array}{c}
\frac{O, M, B \vdash \text{baz}(p.\text{bar}(), i) : \text{Bool}}{O, M, B \vdash b \leftarrow \text{baz}(p.\text{bar}(), i) : \text{Bool}} \\
\frac{O(i) = \text{Int}}{O, M, B \vdash i : \text{Int}} \\
\end{array}
\]
(b) (3 pts)

1          \[ p \leftarrow o.\text{foo}() \]

Answer:

\[
\begin{array}{c}
\frac{M(B, foo) = \text{SELF\_TYPE}_B \\ O(o) = \text{SELF\_TYPE}_B}{O, M, B \vdash o : \text{SELF\_TYPE}_B}
\end{array}
\]

\[
\begin{array}{c}
O(p) = \text{SELF\_TYPE}_B \\ O, M, B \vdash o.\text{foo}() : \text{SELF\_TYPE}_B
\end{array}
\]

\[
\begin{array}{c}
O, M, B \vdash p \leftarrow o.\text{foo}() : \text{SELF\_TYPE}_B
\end{array}
\]

Type is \text{SELF\_TYPE}_B.
(c) (3 pts)

1 \[ b \leftarrow \text{baz}(i+j, p.\text{bar}(i, o.\text{foo}())) \]

**Answer:** This expression will not type check, since \( p.\text{bar}(i, o.\text{foo}()) \) presents a type error. This is due to the fact that \( \text{bar} \) takes zero arguments and the expression passes two. The type mismatch is \((\text{Int}, \text{Int}, \text{Bool}) \neq \text{Int})\).
(d) (3 pts)

```plaintext
case c of
  s: Int => s;
  i: String => j;
  b: Object => i;
esac
```

**Answer:**

\[
\begin{array}{cccc}
O(p) = \text{Object} & O(s/\text{Int})(s) = \text{Int} & O(i/\text{String})(j) = \text{Int} & O[b/\text{Object}](i) = \text{Int} \\
O, M, B + c : \text{Object} & O[s/\text{Int}], M, B + s : \text{Int} & O[i/\text{String}], M, B + j : \text{Int} & O[b/\text{Object}], M, B + i : \text{Int} \\
O, M, B + \text{case c of s: Int => s; i: String => j; b: Object => i; esac : Int} \\
\end{array}
\]

Type is \textbf{Int}.
3. (8 pts) Consider the following Cool program fragment:

```cool
class A {
    i: Int <- 14;
    f(): Int { i };
};
class B inherits A {
    g(): Int { f() + f() };
};
class C inherits B {
    h(): Int {
        let x: A <- new C in {
            x.g() + x.f();
        }
    };
};
class Main {
    main(): Int {
        let c: C <- new C in { c.h(); }
    };
};
```

(a) (4 pts) This code does not compile. Provide a complete but succinct explanation as to why that is the case. Please note that the error message of coolc does not count as an explanation, your answer must show that you understand the problem.

**Answer:** This code has a type error in line 11. The static type of x is A, which does not include a definition of method g, meaning that x.g() is a type error.
(b) (4 pts) Assume you are given a variant of Cool which is dynamically typed instead of statically typed. Would the behavior of the code above be safe, in the sense of not triggering a runtime type error, in such variant of Cool? Why or why not?

**Answer:** It would be safe. Dynamic Cool will only check the type of $x.g()$ at runtime, and at runtime the dynamic type of $x$ will always be $C$, since $x$ is only ever assigned “new $C$”. Since $C$ inherits from $B$, it has an implementation of $g()$, and thus $x.g()$ is dynamically type-safe in all possible executions of the program.
4. (16 pts) Consider the following extension to the Cool syntax as given on page 16 of the Cool Manual, which adds arrays to the language:

\[
\text{expr ::= new TYPE[ expr ]} \\
\quad | \text{expr[ expr ]} \\
\quad | \text{expr[ expr ] < − expr}
\]

This adds a new type \( T[] \) for every type \( T \) in Cool, including the basic classes. Note that the entire hierarchy of array types still has Object as its topmost supertype. An array object can be initialized with an expression similar to “\( \text{my\_array:T[]} \leftarrow \text{new T[n]} \)”, where \( n \) is an Int indicating the size of the array. In the general case, any expression that evaluates to an Int can be used in place of \( n \). Thereafter, elements in the array can be accessed as “\( \text{my\_array[i]} \)” and modified using an expression like “\( \text{my\_array[i]} \leftarrow \text{value} \)”.

(a) (4 pts) Provide new typing rules for Cool which handle the typing judgments for:

\[ O, M, C ⊢ e_1 : T_1, O, M, C ⊢ e_2 \] and \( O, M, C ⊢ e_1[e_2] < − e_3 \). Make sure your rules work with subtyping.

**Answer:**

**ArrayNew:**

\[
O, M, C ⊢ e_1 : T_1 \quad T_1 ≤ \text{Int} \quad T' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T = \text{SELF\_TYPE} \\ T & \text{otherwise} \end{cases} \\
O, M, C ⊢ \text{new T[e_1]} : T'[]
\]

Note that you may alternatively chose to disallow \( \text{SELF\_TYPE} \) as the type of a new array. Additionally, the Cool Manual states that it is an error to inherit from \text{Int}, so the only subtype of \text{Int} is \text{Int}. With this in mind, the above rule can also be given in a simplified form as:

\[
O, M, C ⊢ e_1 : \text{Int} \\
O, M, C ⊢ \text{new T[e_1]} : T[]
\]

**ArrayLoad:**

\[
O, M, C ⊢ e_1 : T[] \quad O, M, C ⊢ e_2 : T_2 \quad T_2 ≤ \text{Int} \\
O, M, C ⊢ e_1[e_2] : T
\]

Again, we can alternatively list \( T_2 \) as \text{Int} directly, since \text{Int} cannot be subclassed. Note that whether or not we allow the expression “\( \text{new SELF\_TYPE[]} \)”, we don’t need to check for \text{SELF\_TYPE} here, since our rule for \text{New} already handles that case and gives the array a \( \text{SELF\_TYPE}_C \) for the specific class \( C \). There are some subtleties that arise if we want to allow a method to return a \( \text{SELF\_TYPE[]} \) value or take arguments of that type as arguments. However, as we will see soon, array subtyping is more restrictive than normal class subtyping, making \( \text{SELF\_TYPE[]} \) a lot less useful than \( \text{SELF\_TYPE} \).

**ArrayStore:**

\[
O, M, C ⊢ e_1 : T[] \quad O, M, C ⊢ e_2 : T_2 \quad T_2 ≤ \text{Int} \quad O, M, C ⊢ e_3 : T_3 \quad T_3 ≤ T \\
O, M, C ⊢ e_1[e_2] < − e_3 : T_3
\]

Note that we assign the whole expression the type of \( e_3 \). This is not specified by the description of our array extension in itself, and is not the only valid answer for this exercise, but it most closely resembles the rule for \text{ASSIGN}. 

11
(b) **(4 pts)** Consider the following subtyping rule for arrays:

\[
\frac{T_1 \leq T_2}{T_1[\ ] \leq T_2[\ ]}
\]

This rule means that \( T_1[\ ] \leq T_2[\ ] \) whenever it is the case that \( T_1 \leq T_2 \), for any pair of types \( T_1 \) and \( T_2 \).

While plausible on first sight, the rule above is incorrect, in the sense that it doesn’t preserve Cool’s type safety guarantees. Provide an example of a Cool program (with arrays added) which would type check when adding the above rule to Cool’s existing type rules, yet lead to a type error at runtime.

**Answer:**

```java
1 class A { };
2 class B inherits A {
3     g(): Int { 1 };
4 }
5 class Main {
6     va: A[];
7     vb: B[] <- New B[1];
8     main(): Int {{
9         va <- vb;
10         va[0] <- New A;
11         vb[0].g(); -- error
12     }};
13 }
```

This will type check at compile time, since each of the statements in `Main.main()` type checks correctly: the first uses the subtyping rule given above and the standard `ASSIGN` rule to assign a `B[]` array to `va`, which has type `A[]`; the second simply initializes `va[0]` with a new `A` object; and the last one retrieves the first object (of static type `B`) in the array `vb` and calls `g()` on it, which is a valid method for `B`. The runtime error arises from the fact that `vb[0] = va[0]` has an actual type of `A`, not `B`. Because we can assign a mutable array of class `B` to a mutable array of the superclass `A`, we can end up with an array containing objects that don’t match the static type of the array, which will violate our assumptions the moment we extract those objects from the array and try to use them.
(c) **(4 pts)** In the format of the subtyping rule given above, provide the least restrictive rule for the relationship between array types (i.e. under which conditions is it true that $T_1[] \leq T'$ for a certain $T'$ or $T'' \leq T_1[]$ for a certain $T''$?) which preserves the soundness of the type system. The rule you introduce must not allow assignments between non-array types that violate the existing subtyping relations of Cool.

**Answer:**

The least restrictive rule for the relationship between mutable array types is as follows:

$$
\frac{T_1 = T_2}{T_1[] \leq T_2[]}
$$

An array type is only a subtype of another array type if their allowable contents are of the same type (which implies that they are the exact same array type: $T_1[] = T_2[]$). Additionally, every array is a subtype of Object and has no subtyping relationship with any other non-array type.

This typing rule for mutable arrays is generally refereed to as invariant typing. The rule of the last exercise is called covariant typing and is incorrect for arrays. Mutable arrays are neither covariant, nor contravariant (a related rule in which a dependent type $X[[T]]$ is only a subtype of another $X[[G]]$ if $T$ is a supertype of $G$).
(d) **(4 pts)** Add another extension to the language for immutable arrays (denoted by the type $T()$). Analogous to questions 4a and 4c, for this extension, provide: the additional syntax constructs to be added to the listing of page 16 of the Cool manual, the typing rules for these constructs and the least restrictive subtyping relationship involving these tuple types. It is not necessary that this extension interact correctly with mutable arrays as defined above, but feel free to consider that situation.

**Answer:**

*Syntax:* Note that for immutable arrays, we must somehow combine the rule for New with a rule that sets the values of all elements in the array. The syntax for array loading can be identical to that of the mutable case.

\[
\text{expr ::= new TYPE[ ] < \{(expr, expr)\}}
\]

\[
\text{| expr[ expr ] (2)}
\]

*Typing rules:*

\[
\forall i \in [1, n] \quad O, M, C \vdash e_i : T_i \quad T_i \leq T
\]

\[
O, M, C \vdash \text{new } T[ ] < \{(e_1, \ldots, e_n) : T()\}
\]

\[
O, M, C \vdash e_1 : T() \quad O, M, C \vdash e_2 : T_2 \quad T_2 \leq \text{Int}
\]

\[
O, M, C \vdash e_1[ e_2 ] : T
\]

*Subtyping rules:* Because immutable arrays don’t allow modification using a reference of a subtype, the issue shown in question 4b does not arise. Thus, immutable arrays are covariant:

\[
\frac{T_1 \leq T_2}{T_1( ) \leq T_2( )}
\]
5. (12 pts) Consider another extension to the Cool language. In this case, we wish to add a special type to Cool that can either be an Int or a special value that represents “no result”. This MaybeInt type will take two forms: Some(n) where n is an integer, and Nothing. The compiler will provide two methods: createSomething(n: Int): MaybeInt and createNothing(): MaybeInt which are defined in the Object class and produce each of the values of the MaybeInt type.

An example of where this type would be useful is a function like:

```cool
divide(numerator: Int, denominator: Int): MaybeInt {
    if (denominator = 0) then
        createNothing()
    else
        createSomething(numerator / denominator)
    fi
}
```

Which provides an implementation of an integer division that will not need to throw an exception when faced with a denominator of 0, but will return Nothing instead.

To be able to use the value inside of a MaybeInt, we add a pattern matching statement match to our language. Similar to how a switch statement works in other languages, match will go to a branch depending on the form of MaybeInt passed to it at runtime, while also possibly introducing a new value into the scope. The value of a match expression is the value of the expression on the right side of the branch that was taken. Example:

```cool
let div_result: MaybeInt in {
    div_result <- divide(i, j);
    match (div_result) {
        Some(n) => "The result was: ".concat(n.to_string()),
        Nothing => "Can’t divide by zero."
    }
}
```

The grammar rule for match is:

\[
expr ::= \text{match} (expr)\{\text{Some}(n) => expr, \text{Nothing} => expr\}
\] (3)

(a) (4 pts) Write a type checking rule for the match expression, which preserves Cool’s type safety guarantees.

**Answer:**

\[
\frac{O, M, C \vdash e_0 : \text{MaybeInt} \quad O[\text{Int}/n], M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \text{match}(e_0)\{\text{Some}(n) => e_1, \text{Nothing} => e_2\} : T_1 \sqcup T_2}
\]
(b) (8 pts) Give the operational semantics of the **match** expression. Consider referring to the operational semantics for Let and If-True/False from the Cool manual. Assume that when looking up **MaybeInt** in the store, it will either return **Some(n)** or **Nothing**, so you may need to write two separate rules.

**Answer:**

**Operational semantics for Nothing:**

\[
\frac{\text{so, } S, E \vdash e_0 : \text{Nothing}, S_1 \quad \text{so, } S_1, E \vdash e_2 : v, S_2}{\text{so, } S, E \vdash \text{match}(e_0)\{\text{Some}(n) => e_1, \text{Nothing} => e_2\} : v, S_2}
\]

**Operational semantics for Some(v):**

\[
\frac{\text{so, } S, E \vdash e_0 : \text{Some}(v), S_1 \quad l_1 = \text{newloc}(S_1) \quad \text{so, } S_1[v/l_1], E[l_1/n] \vdash e_1 : v_1, S_2}{\text{so, } S, E \vdash \text{match}(e_0)\{\text{Some}(n) => e_1, \text{Nothing} => e_2\} : v_1, S_2}
\]