1. Consider the following Cool programs:

(a)

class A {
    x: A; -- line 2
    baz(): A {{x <- new A; x;}}; -- line 3
    bar(): A {new A}; -- line 4
    foo(): String {"COVID-19!");
}

class B inherits A {
    foo() : String {" ");
}

class C inherits A {
    foo() : String {" Bye,"};
}

class Main {
    main (): Object {
        let io : IO <- new IO, b : B <- new B, c : C <- new C
        in {{
            io.out_string(c.baz().foo());
            io.out_string(b.baz().foo());
            io.out_string(b.bar().baz().foo());
        }}
    }
};

What does this code currently print? By changing the types on lines 2-4 get this program to print "Bye, COVID-19!".
class Main {
    main (): Object {
        let io : IO <- new IO, x : Int <- 20 in {
            io.out_int (x);
            let x : Int < - 1 in {
                (* x <- YOUR CODE HERE ;*)
                io.out_int (x);
            });
            if x == 21 then
                io.out_string("x")
            else
                io.out_int(x)
            fi;
        }
    }
};

Replace (* x <- YOUR CODE HERE ;*) with at most one line of code containing an assignment to x that gets this code to print "2021x". If it is not possible, explain why.
2. Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[Int/y] ⊢ y + y : Int$:

\[
O[Int/y](y) = Int \\
\overline{O[Int/y], M, C ⊢ y : Int} \\
\overline{O[Int/y], M, C ⊢ y : Int} \\
\overline{O[Int/y], M, C ⊢ y + y : Int}
\]

Consider the following Cool program fragment:

```cool
1 class A {
2     i: Int;
3     b: Bool;
4     s: String;
5     o: SELF_TYPE;
6     foo(): SELF_TYPE { o };
7     bar(): Int { 2 * i + 1 };
8 };
9
class B inherits A {
10     a: A;
11     baz(a: Int, b: Int): Bool { a = b };
12     test(): Object { (* [Placeholder] *) };
13 };
```

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

\[
O = \emptyset[Int/i][Bool/b][String/s][SELF_TYPE_B/o][A/a][SELF_TYPE_B/self] \\
M = \emptyset[(SELF_TYPE)/(A, foo)][(Int)/(A, bar)] \\
\quad [(SELF_TYPE)/(B, foo)][(Int)/(B, bar)] \\
\quad [(Int, Int, Bool)/(B, baz)][(Object)/(B, test)]
\]

For each of the following expressions replacing (* [Placeholder] *), provide the type derivation and final type of the expression. You may omit subtyping relationships from the rules when the type is the same, e.g. Bool ≤ Bool

(a) b <- self.baz(i, self.bar())
(b) let b: B <- self.foo() in a <- b
(c) if b then a.foo() else self.foo() fi
3. Consider the following Cool program:

```cool
class Main {
    b: B;
    main (): Object {{
        b <- new B;
        b.foo();
    }};
}
```

Now consider the following implementations of the classes A and B. Analyze each version of the classes to determine if the resulting program will pass type checking and, if it does, whether it will execute without runtime errors. Please include a brief (1 - 2 sentences) explanation along with your answer. Note it is not sufficient to simply copy the output of the cool compiler, if it fails type checking be specific about which hypotheses cannot be satisfied for which rules.

(a)

```cool
class A {
    i: Int <- 1;
    a: SELF_TYPE;
    foo (): Int {i};
}

class B inherits A {
    j: Int <- 1;
    baz (): Int {{i <- 2 + i; i;}};
    foo (): Int {
        j <- a.baz () + a.foo();
        j;
    }
}
```

(b)

```cool
class A {
    i: Int <- 1;
    a: SELF_TYPE;
    foo (): Int {i};
}

class B inherits A {
    j: Int <- 1;
    baz (): Int {{i <- 2 + i; i;}};
    foo (): Int { let a: A <- new B in {
        j <- a@B.baz () + a.foo();
        j;
    }};
```
4. Consider the following extensions to Cool:

(a) Arrays.

\[
expr ::= \ldots
| \text{new } TYPE[expr]
| expr[expr]
| expr[expr] < - expr
\]

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 \texttt{Object} as its topmost supertype. An array object can be initialized with an expression similar to “\texttt{my} \_ \texttt{array}:T[] \leftarrow \texttt{new} T[n]”, where \( n \) is an \texttt{Int} indicating the size of the array. In the general case, any expression that evaluates to an \texttt{Int} can be used in place of \( n \). Thereafter, elements in the array can be accessed as “\texttt{my} \_ \texttt{array}[i]” and modified using an expression like “\texttt{my} \_ \texttt{array}[i] \leftarrow \texttt{value}”.

Provide new typing rules for Cool which handle the typing judgments for:

- \( O, M, C \vdash \text{new } T[ e_1 ] \)
- \( O, M, C \vdash e_1[ e_2 ] \)
- \( O, M, C \vdash e_1[ e_2 ] < - e_3 \)

Make sure your rules work with subtyping.
(b) Permissive method overriding.

In Cool a subtype can only override a method with a method with exactly the same formal parameters and return type. Or as judgements (with some abuse of notion to quantify over the elements in environments):

\[
\begin{align*}
T_i &= S_i \quad 1 \leq i \leq n + 1 \\
(T_1, \ldots, T_n, T_{n+1}) &\leq (S_1, \ldots, S_n, S_{n+1})
\end{align*}
\]

Method Subtype

\[
\begin{align*}
T_c &= T_p \quad \lor \quad T_c \text{ inherits } T'_p \land T'_p \leq T_p \\
O, M, C \vdash \forall m \in M(T_p) : M(T_c, m) &\leq M(T_p, m)
\end{align*}
\]

Class Subtype

The Method Subtype rule says that if a class \(X\) has a method \(f\) and class \(Y\) has a method \(g\) to establish that \(f\) conforms to \(g\), i.e. \(M(X, f) \leq M(Y, g)\), we must show \(M(X, f) = (T_1, \ldots, T_n, T_{n+1}) = (S_1, \ldots, S_n, S_{n+1}) = M(Y, g)\).

The Class Subtype rules says that for a class \(T_c\) to be considered a subtype of a class \(T_p\) we must establish two things:

- \(T_c\) must either be equal to \(T_p\) or it must inherit from some class \(T'_p\) where \(T_p\) is a subtype of \(T'_p\)
- And for every method \(m\) on \(T_p\), \(T_c\) must also have a method \(m\) such that the types of the methods are conforming (as defined by the Method Subtype rule). I.e. \(M(T_c, m) = M(T_p, m)\).

Note in cool that we consider it an error for \(T_c\) to inherit from \(T_p\) but fail the second test.

The Method Subtype rule is more restrictive than necessary to ensure type safety. Rewrite it with new hypotheses so that \(T_i\) need not equal \(S_i\). Note your solution should still ensure type safety without changing the rules for dispatch. Specifically, given \(C \leq P\) with a method \(m\) if

\[
\begin{align*}
\text{results : } R_{n+1} &\leftarrow P.m(e_1: R_1, e_2: R_2, \ldots, en: R_n);
\end{align*}
\]

type checks then so should

\[
\begin{align*}
\text{results : } R_{n+1} &\leftarrow C.m(e_1: R_1, e_2: R_2, \ldots, en: R_n);
\end{align*}
\]
(c) Structural Typing.

Cool uses *nominal* subtyping. Meaning a class must explicitly inherit another class to be considered a subtype. In languages with *structural* subtyping a class $C$ is considered a subtype of a class $P$ if all of attributes and methods of $P$ have conforming attributes and methods on $C$. I.e. if for every method $m$ on $P$ with type $P_m = (T_1, ..., T_n, T_{n+1})$ there is a corresponding method $m$ on $C$ with type $C_m = (S_1, ..., S_n, S_{n+1})$ where $C_m \leq P_m$ then $C \leq P$. In other words, the Class Subtype rule defined above would only have the latter of the two hypotheses.

Now most things in Cool would continue to work if Cool used structural subtyping instead of nominal subtyping, however, there is one form of expression that would have undefined behavior. Identify which form of expression would be undefined and explain why it would be undefined.
5. Consider the following assembly language used to program a stack (r, r1, and r2 denote arbitrary registers):

1. `push r`: copies the value of r and pushes it onto the stack.
2. `top r`: copies the value at the top of the stack into r. This command does not modify the stack.
3. `pop`: discards the value at the top of the stack.
4. `swap`: swaps the value at top of the stack with the value right beneath it. E.g. if the stack was `<$ ... 5 2>` swap would change the stack to be `<$ ... 2 5>`
5. `r1 *= r2`: multiplies r1 and r2 and saves the result in r1. r1 may be the same as r2.
6. `r1 /= r2`: divides r1 with r2 and saves the result in r1. r1 may be the same as r2. remainders are discarded (e.g., 5 / 2 = 2).
7. `r1 += r2`: adds r1 and r2 and saves the result in r1. r1 may be the same as r2.
8. `r1 -= r2`: subtracts r2 from r1 and saves the result in r1. r1 may be the same as r2.
9. `jump r`: jumps to the line number in r and resumes execution.
10. `print r`: prints the value in r to the console.

The machine has two registers available to the program: `reg1`, and `reg2`. The stack is permitted to grow to a finite, but very large, size. If an invalid line number is invoked, pop is executed on an empty stack, swap is executed on stack with less than 2 elements, or the maximum stack size is exceeded, the machine crashes.

(a) Write code to print the Fibonacci sequence \(F_n = F_{n-1} + F_{n-2}\) where \(F_1 = F_2 = 1\) e.g. 1, 1, 2, 3, 5, 8, ...) starting at \(F_1\). Assume that the code will be placed at line 100, and will be invoked by pushing 1, 1 onto the stack (`<$ ... 1 1 >`) and storing 100 in `reg1`, and running `jump reg1`.

Your code should use the `print` opcode to display numbers in the sequence. You may not hardcode constants nor use any other instructions besides the ones given above. Your code should not terminate (or crash) after any amount of time. Assume that registers and the stack can hold arbitrarily large integers so computation will never overflow.

Hint it may help to comment each line with a symbolic machine state and think about what the state the code should be in at the end e.g.:

```
// initial: reg1=ptr reg2= stack=<F_{n-1}, F_{n-2}>
```

```
100 top reg2 // reg1=ptr reg2=F_{n-2} stack=<F_{n-1}, F_{n-2}>
101 pop // reg1=ptr reg2=F_{n-2} stack=<F_{n-1}>
```

```
// final: ???
```

Note: you are not required to do this but it will help us give you partial credit if you do.

(b) Write code to enumerate the factorials \(F_n = n \times F_{n-1}\) where \(F_1 = 1\) e.g. 1, 2, 6, 24, ...). With the same assumptions and restrictions as above.