1. Consider the following Cool programs:

(a) 
```
class A {
    x: A;
    baz(): A {x ← new SELF_TYPE};
    bar(): A {baz()};
    foo(): String {"am"};
}

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

class C inherits A {
    baz(): A {{ new A; }};
    foo(): String {"Therefore "};
}

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

What does this code currently print? Modify lines 2–4 so that this program prints “Therefore I am”.

Replace (\textit{\texttt{* YOUR CODE HERE *}}) with a single expression that gets this code to print "2023x". 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 \( O[\text{Int}/y], M, C \vdash y + y : \text{Int} \):

\[
\begin{align*}
O[\text{Int}/y](y) &= \text{Int} & [\text{Var}] & \quad O[\text{Int}/y](y) &= \text{Int} & [\text{Var}] \\
O[\text{Int}/y], M, C \vdash y : \text{Int} & \quad O[\text{Int}/y], M, C \vdash y : \text{Int} & [\text{Arith}]
\end{align*}
\]


Consider the following Cool program fragment:

```cool
class A {
  i: Int;
  b: Bool;
  s: String;
  o: SELF_TYPE;
  foo(): SELF_TYPE { o };
  bar(): Int { 2 * i + 1 };
};

class B inherits A {
  a: A;
  baz(x: Int, y: Int): Bool { x = y };
  test(): Object { (* PLACEHOLDER *) };
};
```

Note that the environments \( O \) and \( M \) at the start of the method \( \text{test}() \) are as follows:

\[
O = \emptyset[\text{Int}/i][\text{Bool}/b][\text{String}/s][\text{SELF_TYPE}_B/o][\text{SELF_TYPE}_B/a][\text{SELF_TYPE}_B/self],
\]

\[
M = \emptyset[(\text{SELF_TYPE})/(A, \text{foo})][(\text{Int})/(A, \text{bar})]
   \quad [(\text{SELF_TYPE})/(B, \text{foo})][(\text{Int})/(B, \text{bar})]
   \quad [(\text{Int}, \text{Int}, \text{Bool})/(B, \text{baz})][(\text{Object})/(B, \text{test})].
\]

For each of the following expressions replacing \( (*) \text{ PLACEHOLDER } * \), provide the inferred type of the expression, as well as its derivation as a proof tree.\(^2\) For brevity, you may omit subtyping relations where the same type is on both sides (e.g., \( \text{Bool} \leq \text{Bool} \)). You also do not need to label each step with the inference rule name like we did above.

---


\(^2\)To draw proof trees in \LaTeX, consider using the \texttt{ebproof} package. You can also use the tree in the template as an example.
(a) \{ s \leftarrow "world!"; b \leftarrow \textbf{self}.baz(i, 1); \}  

(b) \textbf{let} \ c: A \leftarrow \textbf{self}.foo() \ \textbf{in} \ c.\text{foo}()  

(c) \textbf{if} \ 1 \leq i \ \textbf{then} \ \textbf{self}.foo() \ \textbf{else} \ a.\text{foo}() \ \textbf{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 reference Cool
compiler: if it fails type checking, you must specify which hypotheses cannot be satisfied for
which rules.

(a)

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

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

(b)

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

```cool
class B inherits A {
    j: Int ← 1;
    baz(): Int {i ← i + j};
    foo(): Int {
        a ← new SELF_TYPE;
        j ← a.baz() + a@A.foo();
    };
```
4. Consider the following extensions to Cool:

(a) Tuples.

\[
\text{expr ::= \ldots}
\]

| new \{ TYPE [], TYPE\}^* [ expr [], expr\}^* ]
| expr [ INT ]

A tuple is a fixed-size list of values of potentially different types. Empty tuples are not allowed. We define a new family of types called tuple types \( \langle T_1, T_2, \ldots, T_n \rangle \), where \( T_1, T_2, \ldots, T_n \) could be any type in Cool (including SELF_TYPE and other tuple types). Note that the entire hierarchy of tuple types still has Object as its topmost supertype. Additionally, the subtype relation between tuple types is defined as follows:

\[ \langle T_1, T_2, \ldots, T_n \rangle \leq \langle T'_1, T'_2, \ldots, T'_n \rangle \text{ if and only if } T_i \leq T'_i \text{ for all } i. \]

A tuple object can be initialized with an expression similar to

\[
\text{my_tuple: } \langle \text{Int, Object} \rangle \leftarrow \text{new } \langle \text{Int, String} \rangle [42, "\text{answer"}];
\]

Thereafter, the \( i^{th} \) element in the tuple can be accessed as “my_tuple[\( i \)]”. Tuple elements are 0-indexed. The tuple index is an integer literal that is always known at compile time. Provide new typing rules for Cool which handle the typing judgments for the two new forms of expressions. As an example, your type rules should ensure the following given the earlier declaration:

\[
O, M, C \vdash \text{my_tuple}[0] : \text{Int} \quad O, M, C \vdash \text{my_tuple}[1] : \text{Object}
\]

Hint: See [New] in the Cool manual for an example that deals with SELF_TYPE in a way similar to how you will have to.

\[
O, M, C \vdash \text{new } \langle T_1, T_2, \ldots, T_n \rangle [e_1, e_2, \ldots, e_m] : \quad \text{[Tuple-New]}
\]

\[
\quad O, M, C \vdash e[i] : \quad \text{[Tuple-Index]}
\]
(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:

\[ T_i = S_i \quad \forall i \in \{1, \ldots, n + 1\} \]  \[ (T_1, \ldots, T_n, T_{n+1}) \leq (S_1, \ldots, S_n, S_{n+1}) \]  \[ \text{[Method-Subtype]} \]

\[ T_c = T_p \]  \[ M \vdash T_c \leq T_p \]  \[ \text{[Class-Subtype1]} \]

\[ T_c \text{ inherits } T'_p \]  \[ M \vdash T'_p \leq T_p \]  \[ M \vdash \forall m \in M(T_p): M(T_c, m) \leq M(T_p, m) \]  \[ M \vdash T_c \leq T_p \]  \[ \text{[Class-Subtype2]} \]

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

1. \( T_c \) must either be equal to \( T_p \); or
2. (a) \( T_c \) must inherit from some class \( T'_p \) where \( T'_p \) is a subtype of \( T_p \), and
   (b) 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., \( M(T_c, m) \leq M(T_p, m) \).

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

\[ \text{out \leftarrow (p: P).m(e_1, e_2, \ldots, e_n);} \]

type checks then so should

\[ \text{out \leftarrow (c: C).m(e_1, e_2, \ldots, e_n);} \]

for the same arguments and output variable.
(c) Multiple inheritance.

Cool’s type system allows single inheritance, where one class inherits from at most one other class. However, many programming languages[4] allow a class to inherit from multiple superclasses. This is especially useful for “interface”-like classes: a hypothetical File class can inherit from both Reader and Writer, while standard input only inherits from Reader:

```plaintext
1 class Reader {
2   read(): String {""};
3 };
4 class Writer {
5   write(s: String): SELF_TYPE {self};
6 };
7 class File inherits Reader, Writer {
8   read(): String { ... }
9   write(s: String): SELF_TYPE {{ ...; self; }}
10 };
11 class Stdin inherits Reader {
12   read(): String { ... }
13 };
```

Now, most Cool code would continue to work if Cool is extended to support multiple inheritance. However, one kind of Cool expressions would have undefined behavior without adjustments to its semantics. 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 \((r, r_1, \text{ and } r_2 \text{ denote arbitrary registers})\):

- **push** \(r\): copies the value of \(r\) and pushes it onto the stack.
- **top** \(r\): copies the value at the top of the stack into \(r\). This command does not modify the stack.
- **pop**: discards the value at the top of the stack.
- **swap**: swaps the value at top of the stack with the value right beneath it. E.g., if the stack was \(⟨\$, \ldots, 5, 2⟩\) swap would change the stack to be \(⟨\$, \ldots, 2, 5⟩\)
- \(r_1 *= r_2\): multiplies \(r_1\) and \(r_2\) and saves the result in \(r_1\). \(r_1\) may be the same as \(r_2\).
- \(r_1 /= r_2\): divides \(r_1\) with \(r_2\) and saves the result in \(r_1\). \(r_1\) may be the same as \(r_2\). Remainders are discarded (e.g., \(5/2 = 2\)).
- \(r_1 += r_2\): adds \(r_1\) and \(r_2\) and saves the result in \(r_1\). \(r_1\) may be the same as \(r_2\).
- \(r_1 -= r_2\): subtracts \(r_2\) from \(r_1\) and saves the result in \(r_1\). \(r_1\) may be the same as \(r_2\).
- **jump** \(r\): jumps to the line number in \(r\) and resumes execution.
- **print** \(r\): prints the value in \(r\) to the console.

The machine has two registers available to the program: \(\text{reg1}\), and \(\text{reg2}\). The stack is permitted to grow to a finite, but very large, size. If an invalid line number is invoked, a number is divided by zero, **top** or **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.

Write code to enumerate and print the factorials \(F_n = n \times F_{n-1}\) where \(F_1 = 1\); e.g., \(1, 2, 6, 24, \ldots\) 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 \(⟨\$, \ldots, 1, 1⟩\), storing 100 in \(\text{reg1}\), and running **jump** \(\text{reg1}\).

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. There is no need to keep the number in memory after it has been printed out. 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. (You are not required to do this but it will help us give you partial credit if you do.) E.g.:

```assembly
// initial :  reg1=100  reg2=     stack=⟨n, F_{n-1}⟩
100  top  reg2  // reg1=100  reg2=F_{n-1}  stack=⟨n, F_{n-1}⟩
101  pop       // reg1=100  reg2=F_{n-1}  stack=⟨n⟩
// final :  ???
```