CS143 Spring 2026 – Written Assignment 3

This assignment covers semantic analysis, including scoping, type systems, and code generation. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 19, 2026 at 11:59 PM PDT. Please review the course policies for more information: `https://web.stanford.edu/class/cs143/policies/`. A LaTeX template for writing your solutions is available on the course website.

1. The Un-Cool Society tore up a precious scroll containing the first-ever Cool program! Thankfully, most of the the code was able to be pieced together:

   (a)

```
class A {
    x: A;
    one(): A { x ← (* ??? BLANK 1 ??? *) };
    two(): A { x };
    string(): String { (* ??? BLANK 2 ??? *) };
};
class B inherits A {
    string(): String { (* ??? BLANK 3 ??? *) };
};
class C inherits A {
    init(): A { x ← new A };
    three(): A { new B };
    string(): String { (* ??? BLANK 4 ??? *) };
};
class Main {
    main(): Object {
        let io: IO ← new IO,
            c : C  ← new C
        in {
            c.init();
            io.out_string(c.three().one().string());
            io.out_string(" ");
            io.out_string(c.two().one().string());
            io.out_string(" ");
            io.out_string(c.one().string());
            io.out_string(" ");
            io.out_string(c.three().string());
        }
    };
};
```

Replace *only* blanks 1-4 on lines 3, 5, 8, and 13 respectively with **a single expression each (no blocks)** so that the code prints out `"Cool compilers are Cool"`.

Answer:

- **BLANK 1:** `self or new SELF_TYPE`
- **BLANK 2:** `"compilers"`
- **BLANK 3:** `"Cool"`
- **BLANK 4:** `"are"`

(b) Here is another incomplete COOL program that was recovered, that a COOL developer is trying to complete :

```
1
2  s Main {
3    main(): Object {
4      let io : IO ← new IO, n : Int ← 5, c : Int ← n in
5        while 0 < c loop {
6          io.out_int(c);
7          io.out_string(" ");
8          io.out_string("C-");
9          let n : Int ← (* YOUR CODE HERE *), c : Int ←0 in
10           while c < n loop {
11             io.out_string("OO-");
12             c ← c + 1;
13           } pool;
14         io.out_string("L\n");
15         c ← c − 1;
16       } pool
17   };
18 };
```

The developer does not understand scoping in COOL and needs your help with filling in the incomplete expression on line 15 so that the program prints the following output:

```
5 C-OO-L
4 C-OO-OO-L
3 C-OO-OO-OO-L
2 C-OO-OO-OO-OO-L
1 C-OO-OO-OO-OO-OO-L
```

Replace *(∗ YOUR CODE HERE ∗)* with a single expression such that the program prints the desired output, or explain why it is not possible.

Answer: $n - c + 1$

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 $O[\text{Int}/y], M, C \vdash y + y : \text{Int}$:

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

The [Var] and [Arith] labels refer to the corresponding inference rules in the Cool Reference Manual, section 12.2.[1]

Consider the following Cool program fragment:

```
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
10 class B inherits A {
11     a: A;
12     baz(x: Int, y: Int): Bool { x = y };
13     test(): Object { (* PLACEHOLDER *) };
14 };
```

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

$$O = \emptyset[\text{Int}/i][\text{Bool}/b][\text{String}/s][\text{SELF\_TYPE}_\text{B}/o][\text{A}/a][\text{SELF\_TYPE}_\text{B}/\textit{self}],$$

$$\begin{aligned} M = \emptyset&[(\text{SELF\_TYPE})/(\text{A}, \text{foo})][(\text{Int})/(\text{A}, \text{bar})] \\ &[(\text{SELF\_TYPE})/(\text{B}, \text{foo})][(\text{Int})/(\text{B}, \text{bar})] \\ &[(\text{Int}, \text{Int}, \text{Bool})/(\text{B}, \text{baz})][(\text{Object})/(\text{B}, \text{test})]. \end{aligned}$$

For each of the following expressions replacing *(∗ 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., Bool ≤ Bool). You also do not need to label each step with the inference rule name like we did above.

[1] See `https://web.stanford.edu/class/cs143/materials/cool-manual.pdf`, pp. 18–22.

[2] To draw proof trees in LaTeX, consider using the `ebproof` package. You can also use the tree in the template as an example.

(a) $b \leftarrow \textbf{self}.\text{baz}(i,\ 1)$;

Answer:

$$\dfrac{O(b) = \text{Bool} \qquad \dfrac{\dfrac{O(\textbf{self}) = \text{SELF\_TYPE}_B}{O, M, \text{B} \vdash \textbf{self} : \text{SELF\_TYPE}_B} \quad \dfrac{O(i) = \text{Int}}{O, M, \text{B} \vdash i : \text{Int}} \quad \dfrac{}{O, M, \text{B} \vdash 1 : \text{Int}} \quad M(\text{B}, \text{baz}) = (\text{Int}, \text{Int}, \text{Bool})}{O, M, \text{B} \vdash \textbf{self}.\text{baz}(i, 1) : \text{Bool}}}{O, M, \text{B} \vdash b \leftarrow \textbf{self}.\text{baz}(i, 1) : \text{Bool}}$$

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

Answer:

$$\dfrac{\dfrac{\dfrac{O(\textbf{self}) = \text{SELF\_TYPE}_B}{O, M, \text{B} \vdash \textbf{self} : \text{SELF\_TYPE}_B} \quad M(\text{B}, \text{foo}) = (\text{SELF\_TYPE})}{O, M, \text{B} \vdash \textbf{self}.\text{foo}() : \text{SELF\_TYPE}_B} \quad \text{SELF\_TYPE}_B \leq \text{A} \quad \dfrac{\dfrac{O[\text{A}/c](c) = \text{A}}{O[\text{A}/c], M, \text{B} \vdash c : \text{A}} \quad M(\text{A}, \text{foo}) = (\text{SELF\_TYPE})}{O[\text{A}/c], M, \text{B} \vdash c.\text{foo}() : \text{A}}}{O, M, \text{B} \vdash \textbf{let}\ c\colon \text{A} \leftarrow \textbf{self}.\text{foo}()\ \textbf{in}\ c.\text{foo}() : \text{A}}$$

(c) $\textbf{if}\ 1 \leq i\ \textbf{then self}.\text{foo}()\ \textbf{else}\ a.\text{foo}()\ \textbf{fi}$

Answer:

$$\dfrac{\dfrac{\dfrac{}{O, M, \text{B} \vdash 1 : \text{Int}} \quad \dfrac{O(i) = \text{Int}}{O, M, \text{B} \vdash i : \text{Int}}}{O, M, \text{B} \vdash 1 \leq i : \text{Bool}} \quad \dfrac{\dfrac{O(\textbf{self}) = \text{SELF\_TYPE}_B}{O, M, \text{B} \vdash \textbf{self} : \text{SELF\_TYPE}_B} \quad M(\text{B}, \text{foo}) = (\text{SELF\_TYPE})}{O, M, \text{B} \vdash \textbf{self}.\text{foo}() : \text{SELF\_TYPE}_B} \quad \dfrac{\dfrac{O(a) = \text{A}}{O, M, \text{B} \vdash a : \text{A}} \quad M(\text{A}, \text{foo}) = (\text{SELF\_TYPE})}{O, M, \text{B} \vdash a.\text{foo}() : \text{A}}}{O, M, \text{B} \vdash \textbf{if}\ 1 \leq i\ \textbf{then self}.\text{foo}()\ \textbf{else}\ a.\text{foo}()\ \textbf{fi} : \text{A}}$$

3. Consider the following Cool program:

```
1  class Main {
2      b: B;
3      main(): Object {{
4          b ← new B;
5          b.set().baz();
6      }};
7  };
```

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
- 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)

```
1  class A {
2      a_str: String ← "Cool";
3      a: SELF_TYPE;
4      set(): SELF_TYPE {{ a ←new A; }};
5      foo(): String {a_str};
6  };
7
8  class B inherits A {
9      b_str: String ← "Language";
10     foo(): String { a_str.concat(b_str)};
11     baz(): String {
12         foo().concat(a@A.foo())
13     };
14 };
```

Answer: This program does not pass type checking. The type of attribute `a` on line 3 is $\text{SELF\_TYPE}_{\texttt{A}}$, and the inferred type of the expression it is being set to is `A`. However, $\texttt{A} \not\leq \text{SELF\_TYPE}_{\texttt{A}}$, so this breaks the [Attr-Init] type-checking rule.

(b)

```
1  class A {
2      a_str: String ← "Cool";
3      a: A;
4      set(): A {{ a ← new A; }};
5      foo(): String {a_str};
6  };
7
8  class B inherits A {
9      b_str: String ← "Language";
10     foo(): String { a_str.concat(b_str)};
```

```
11      baz(): String {
12          foo().concat(a@A.foo())
13      };
14  };
```

Answer: This program does not pass type checking. The return type of the `set` method (defined in class `A`) is `A`. The method `baz` is then dispatched on this result, but class `A` does not define `baz` (only class `B` does). Hence this is not a valid `[Dispatch]` expression.

(c)

```
1   class A {
2       a_str: String ← "Cool";
3       a: A;
4       set(): SELF_TYPE {{ a ←new SELF_TYPE; }};
5       foo(): String {a_str};
6   };
7
8   class B inherits A {
9       b_str: String ← "Language";
10      foo(): String { a_str.concat(b_str)};
11      baz(): String {
12          foo().concat(a@A.foo())
13      };
14  };
```

Answer: This program passes type checking, however it causes a runtime execution error.

The return type of `set` is `SELF_TYPE`, so `b.set()` has type `B`, and `baz` is defined in class `B`, so the `[Dispatch]` rule is satisfied for this case unline part (b).

However, this program causes a runtime error. The execution proceeds as follows:

i. `b.set()` is called on `b` whose dynamic type is `B`. Since `set()` returns `SELF_TYPE`, at runtime `SELF_TYPE` resolves to `B`.

ii. The body of `set` executes `a <- new SELF_TYPE`, which creates a new `B` object. So `a` is now a fresh `B` instance.

iii. `baz()` is then dispatched on this new `B` object. Inside `baz`:

```
foo().concat(a@A.foo())
```

iv. `a@A.foo()` performs a static dispatch of `foo` on `a`. However, the `a` attribute of the newly created `B` object is uninitialized, so it is **void**.

v. Dispatching on **void** causes a **runtime error**.

(d)

```
1   class A {
2       a_str: String ← "Cool";
3       a: A;
4       set(): SELF_TYPE {{ a ←new SELF_TYPE; self;}};
5       foo(): String {a_str};
6   };
7
```

```
8  class B inherits A {
9      b_str: String ← "Language";
10     foo(): String { a_str.concat(b_str)};
11     baz(): String {
12         foo().concat(a@A.foo())
13     };
14 };
```

Answer: This program passes type checking and executes successfully. The execution proceeds as follows:

i. `b <- new B` (line 4): `b` is initialized as a new `B` object. `b` has its own `a_str = "Cool"`, `b_str = "Language"`, and `a = void`.

ii. `b.set()` (line 5): `set()` is called on `b` whose dynamic type is `B`. The body executes:

   A. `a <- new SELF_TYPE`: since the dynamic type of `self` is `B`, this creates a new `B` object and assigns it to `a`. This new `B` has `a_str = "Cool"` and `b_str = "Language"`.

   B. `self`: the method returns `self`, which is `b` itself (not the new object). Since `set()` returns `SELF_TYPE`, the return type resolves to `B`.

iii. `b.set().baz()` (line 5): since `set()` returns `self` (i.e. `b`), `baz()` is now called on `b`. Inside `baz`:

   A. `foo()`: dispatches `foo` on `self` (which is `b`, a `B` object), calling `B`'s `foo()`, which returns `a_str.concat(b_str) = "Cool".concat("Language") = "CoolLanguage"`.

   B. `a@A.foo()`: performs a static dispatch of `foo` on `a`. Since `a` was assigned `new B` in step 2, `a` is not void. The static dispatch forces `A`'s `foo()` to be called, which returns `a_str = "Cool"`.

   C. `foo().concat(a@A.foo())` returns `"CoolLanguage".concat("Cool") = "CoolLanguageCool"`.

4. Consider the following extensions to Cool:

   (a) Tuples.

$$
\begin{aligned}
\mathit{expr} ::= \ & \ldots \\
& \mid \textbf{new} \ \langle \ \text{TYPE} \ [\![, \text{TYPE}]\!]^* \ \rangle \ [\ \mathit{expr} \ [\![, \mathit{expr}]\!]^* \ ] \\
& \mid \mathit{expr} \ [\ \text{INT} \ ]
\end{aligned}
$$

   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:

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

   A tuple object can be initialized with an expression similar to

$$\mathit{my\_tuple} : \langle \text{Int}, \text{Object} \rangle \leftarrow \textbf{new} \ \langle \text{Int}, \text{String} \rangle [42, \text{“}\texttt{answer}\text{”}];$$

   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:

$$O, M, C \vdash \mathit{my\_tuple}[0] : \text{Int} \qquad\qquad O, M, C \vdash \mathit{my\_tuple}[1] : \text{Object}$$

   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.

Answer:

$$\frac{\begin{array}{c} m=n \\ O,M,C \vdash e_1 : S_1 \\ \vdots \\ O,M,C \vdash e_m : S_m \end{array} \qquad T'_i = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T_i = \text{SELF\_TYPE} \\ T_i & \text{otherwise} \end{cases} \quad \forall i \in \{1,\ldots,n\} \qquad S_i \leq T'_i \quad \forall i \in \{1,\ldots,n\}}{O, M, C \vdash \textbf{new} \ \langle T_1, T_2, \ldots, T_n \rangle \ [e_1, e_2, \ldots, e_m] : \langle T'_1, T'_2, \ldots, T'_n \rangle} \ \textsc{Tuple-New}$$

$$\frac{O, M, C \vdash e : \langle T_1, T_2, \ldots, T_n \rangle \qquad i \text{ is an integer constant} \qquad 0 \leq i \leq n-1}{O, M, C \vdash e[i] : T_{i+1}} \ \textsc{Tuple-Index}$$

(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. The rules can also be written formally as the following typing judgements (using slightly informal notation to quantify over the elements in environments):

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

$$\frac{\begin{array}{c} T_c = T_p \quad \vee \quad (T_c \textbf{ inherits } T'_p \wedge T'_p \leq T_p) \\ M \vdash \forall m \in M(T_p) \colon M(T_c, m) \leq M(T_p, m) \end{array}}{M \vdash T_c \leq T_p} \text{ Class Subtype}$$

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 $M(X, f) = (T_1, ..., T_n, T_{n+1}) = (S_1, ..., S_n, S_{n+1}) = M(Y, g)$.

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., $M(T_c, m) \leq M(T_p, m)$.

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

$$out \leftarrow (p\colon P).m(e_1, e_2, \ldots, e_n);$$

type checks then so should

$$out \leftarrow (c\colon C).m(e_1, e_2, \ldots, e_n);$$

for the same arguments and output variable.

Answer:

$$\frac{\begin{array}{c} S_i \leq T_i \quad 1 \leq i \leq n \\ T_{n+1} \leq S_{n+1} \end{array}}{(T_1, ..., T_n, T_{n+1}) \leq (S_1, ..., S_n, S_{n+1})} \text{ Method Subtype}$$

This corresponds to allowing super types in arguments and sub type in the return.

A good way to understand this is considering functions of 1 argument. suppose we have sets (types) A, B, X, Y where $A \subseteq B$ and $X \subseteq Y$, a function $f : B \to X$ is also a function $A \to Y$ as every element in A is mapped to an element Y by f. In other words functions $B \to X$ are a subtype of functions $A \to Y$.

It is tempting to use the rule $T_i \leq S_i \quad 1 \leq i \leq n+1$, however, this would lead to the same problems as allowing SELF_TYPE as parameter

5. Consider the following assembly language used to program a stack (r, r_1, and r_2 denote arbitrary registers):

   - **push** *offset* r: copies the value of r and pushes it onto the stack with a provided offset. An offset of 0 pushes a value to the top of the stack, an offset of 1 pushes a value to the second-highest position in the stack, etc.
   - **read** *offset* r: copies the value at the provided offset from the top of the stack into r. This command does not modify the stack. An offset of 0 reads the value at the top of the stack, an offset of 1 reads the value at the second-from-top of the stack, etc.
   - **pop** *offset*: discards the value at the provided offset from the top of the stack. An offset of 0 pops the value at the top of the stack, an offset of 1 pops the value at the second-highest position in the stack, etc.
   - $r_1 *= r_2$: 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: **reg1**, and **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, **push**, **read**, or **pop** is executed with an invalid offset, 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 $\langle \$, \ldots, 1, 1 \rangle$, 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. 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.:

   // initial : *reg1*=100 *reg2*= *stack*=$\langle n, F_{n-1} \rangle$

   100 **read** 0 reg2 *// reg1*=100 *reg2*=F_{n-1} *stack*=$\langle n, F_{n-1} \rangle$
   101 **pop** 0 *// reg1*=100 *reg2*=F_{n-1} *stack*=$\langle n \rangle$
   102 ...

   // final : *???*

Answer:

// initial : *reg1=100* *reg2=* *stack=*$\langle n, F_{n-1}\rangle$

100 **read** 0 reg2 *// reg1=*100 *reg2=*F_{n-1} *stack=*$\langle n, F_{n-1}\rangle$
101 **pop** 0 *// reg1=*100 *reg2=*F_{n-1} *stack=*$\langle n\rangle$
102 **push** 1 reg1 *// reg1=*100 *reg2=*F_{n-1} *stack=*$\langle 100, n\rangle$
103 **read** 0 reg1 *// reg1=*n *reg2=*F_{n-1} *stack=*$\langle 100, n\rangle$
104 **pop** 0 *// reg1=*n *reg2=*F_{n-1} *stack=*$\langle 100\rangle$
105 reg2 $*=$ reg1 *// reg1=*n *reg2=*F_n *stack=*$\langle 100\rangle$
106 **print** reg2 *// reg1=*n *reg2=*F_n *stack=*$\langle 100\rangle$
107 **push** 1 reg2 *// reg1=*n *reg2=*F_n *stack=*$\langle F_n, 100\rangle$
108 reg2 $/=$ reg2 *// reg1=*n *reg2=*1 *stack=*$\langle F_n, 100\rangle$
109 reg2 $+=$ reg1 *// reg1=*n *reg2=*$n+1$ *stack=*$\langle F_n, 100\rangle$
110 **push** 2 reg2 *// reg1=*n *reg2=*$n+1$ *stack=*$\langle n+1, F_n, 100\rangle$
111 **read** 0 reg1 *// reg1=*100 *reg2=*$n+1$ *stack=*$\langle n+1, F_n, 100\rangle$
112 **pop** 0 *// reg1=*100 *reg2=*$n+1$ *stack=*$\langle n+1, F_n\rangle$
113 **jump** reg1 *// reg1=*100 *reg2=*$n+1$ *stack=*$\langle n+1, F_n\rangle$

// final : *reg1=*100 *reg2=*$n+1$ *stack=*$\langle n+1, F_n\rangle$