YOUR NAME – SUNet ID
CS143 Spring 2024 – 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 21, 2024 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 };
    three(): String { (* ??? BLANK 2 ??? *) };
};
class B inherits A {
    three(): String { (* ??? BLANK 3 ??? *) };
};
class C inherits A {
    two(): A { new A };
    three(): String { (* ??? BLANK 4 ??? *) };
};
class Main {
    main(): Object {
        let io: IO ← new IO,
            b : B  ← new B,
            c : C  ← new C
        in {
            io.out_string(c.two().three());
            io.out_string(" ");
            io.out_string(c.one().three());
            io.out_string(" ");
            io.out_string(b.one().three());
            io.out_string(" ");
            io.out_string(c.one().two().one().three());
        }
    };
};
```

Replace *only* blanks 1-4 on lines 3, 5, 8, and 12 respectively with **a single expression each (no blocks)** so that the code prints out `"Cool compilers are Cool"`.

Answer:

To print `"Cool compilers are Cool"`:

- Blank 1 should be `new SELF_TYPE`
- Blank 2 should be `"Cool"`
- Blank 3 should be `"are"`
- Blank 4 should be `"compilers"`

(b) To celebrate the recovery of the first-ever Cool program, the Cool development team plans to throw a celebration featuring their food, bananas. Here is an incomplete program written by one of the developers:

```
 1  class Main {
 2      main(): Object {
 3          let io: IO ← new IO, counter: Int ← 5 in {
 4              −− print 5 lines of bananas
 5              while 0 < counter loop {
 6                  io.out_string("ba");
 7                  let counter: Int ← 2 in {
 8                      −− print "nana" in a Cool way!
 9                      while 0 < counter loop {
10                          io.out_string("na");
11                          counter ← counter − 1;
12                      } pool;
13
14                      −− only print the "s" if we have more than one banana
15                      if (∗ INCOMPLETE ∗) then {
16                          io.out_string("s\n"); 1;
17                      } else 0;
18                  };
19
20                  −− decrement the print counter
21                  counter ← counter − 1;
22              } pool;
23          };
24      };
25  };
```

The developer needs your help with filling in the incomplete expression on line 15 so that the program prints the following output:

```
5 bananas
4 bananas
3 bananas
2 bananas
1 banana
```

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

Answer:
This is not possible. We need to be able to check the value of the `counter` variable created on line 3 (the "print `counter`"), but at line 15 the "print `counter`" variable is shadowed by the `counter` variable created on line 7 to print the "na"s. This means that the value of `counter` will be 0 each time line 15 is executed, regardless of the value of the print `counter`, so the code will print `"banana"` five times.

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

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

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:

```
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 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 $\leq$ 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:

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

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

Answer:

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

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

Answer:

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

3. Consider the following Cool program:

```
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
- 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)

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

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

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

(b)

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

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

Answer: This program will pass type checking and execute correctly. Here is the sequence of actions that will occur:

i. Upon initialization, the variable b initially contains $B(i = 1, a = \texttt{void}, j = 1)$.

ii. b.foo() initializes a to be another B.

iii. a.baz() sets a to be $B(i = 2, a = \texttt{void}, j = 1)$ and returns 2.

iv. a@A.foo() calls A.foo(), which returns $a.i = 2$.

v. b.foo() finally sets $b.j$ to 4 and returns 4.

vi. At the end, the main function returns 4. The final object b looks like

$$B(i = 1, a = B(i = 2, a = \texttt{void}, j = 1), j = 4)$$

4. Consider the following extensions to Cool:

   (a) Maps.

$$\begin{aligned} \textit{expr} ::= &\ \ldots \\ \mid &\ \textbf{new}\ \langle\ \text{TYPE}, \text{TYPE}\ \rangle \\ \mid &\ \textit{expr}\ [\ \textit{expr}\ ] \\ \mid &\ \textit{expr}\ [\ \textit{expr}\ ]\ \texttt{<-}\ \textit{expr} \end{aligned}$$

Note: In the above expression definitions, single brackets correspond to the actual `[` *and* `]` *characters, and not that the tokens between the brackets are optional like in Figure 1 of the Cool manual.*

A map is a key-value store, like `std::map` or `std::unordered_map` in C++ or dictionaries in Python. A key can be inserted into the map with a single associated value. A *map type* is defined as $\langle T_1, T_2\rangle$, where T_1 can be an Int or String and T_2 can be any type in Cool (including SELF_TYPE and other map types). T_1 is the type of the key, and T_2 is the type of the value. Note that the entire hierarchy of map types still has Object as its topmost supertype. Additionally, the subtype relation between map types is defined as follows:

$$\langle T_1, T_2\rangle \leq \langle T_1', T_2'\rangle \quad \text{if and only if } T_1 = T_1' \text{ and } T_2 \leq T_2'.$$

A map object can be initialized with an expression similar to

$$\textit{my\_map}\colon \langle \text{Int}, \text{Object}\rangle \leftarrow \textbf{new}\ \langle \text{Int}, \text{String}\rangle;$$

Thereafter, a key-value pair $\langle k, v\rangle$ can be inserted into the map with the syntax `my_map[k] <- v`. The value corresponding to a given key can be accessed with the syntax `my_map[k]`. Both of these expressions return the value, in this case `v`.

Provide new typing rules for Cool which handle the typing judgments for the three new forms of expressions: Map-New, Map-Access, and Map-Insert. As an example, your type rules should ensure the following given the earlier declaration:

$$O, M, C \vdash \textit{my\_map}[0]\ \texttt{<-}\ \text{“Hello”} : \text{Object} \qquad\qquad O, M, C \vdash \textit{my\_map}[1] : \text{Object}$$

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 in the Map-New rule.

Note: You do not need to consider the case when the map is accessed with a key that was never inserted into the map, as that would be handled at runtime and not by the type checker.

Answer:

$$\frac{\begin{array}{c} T_1 \in \{Int, String\} \\ T_2' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_2 = \texttt{SELF\_TYPE} \\ T_2 & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash \textbf{new}\ \langle T_1, T_2\rangle : \langle T_1, T_2'\rangle}\ \text{[Map-New]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \langle T_1, T_2 \rangle \\ O, M, C \vdash e_2 : S_1 \\ S_1 = T_1 \end{array}}{O, M, C \vdash e_1[e_2] : T_2} \text{ [Map-Access]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \langle T_1, T_2 \rangle \\ O, M, C \vdash e_2 : S_1 \\ O, M, C \vdash e_3 : S_2 \\ S_1 = T_1 \\ S_2 \leq T_2 \end{array}}{O, M, C \vdash e_1[e_2] \texttt{ <- } e_3 : T_2} \text{ [Map-Insert]}$$

(b) Multiple inheritance.

Cool's type system allows single inheritance, where one class inherits from at most one other class. However, many programming languages[3] 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`:

```
class Reader {
    read(): String {""};                    -- to be overridden by subclass
};
class Writer {
    write(s: String): SELF_TYPE {self}; -- to be overridden by subclass
};
class File inherits Reader, Writer {
    read(): String { ... }
    write(s: String): SELF_TYPE {{ ...; self; }}
};
class Stdin inherits Reader {
    read(): String { ... }
};
```

Now, most Cool code would continue to work if Cool is extended to support multiple inheritance. However, some Cool expressions would have undefined behavior without adjustments to its semantics. Identify one form of expression that would be undefined and explain why it would be undefined.

Answer:

There are three kinds of expressions that would be undefined:

i. **case** expressions. From section 7.9 of the Cool manual, given an expression with dynamic type C, a **case** expression always selects "the branch with the least type `<typek>` such that $C \leq$ `<typek>`." In other words, a **case** expression would try to find the most specific branch that still matches the input expression. However, if multiple inheritance is allowed, we could have a situation where two branches have types that are incomparable, or equally specific.
As a concrete example, consider the following expression:

```
case new File of
    r : Reader => ...;
    w : Writer => ...;
esac
```

It is undefined which branch should execute, since File $\leq$ Reader and File $\leq$ Writer, yet neither Reader nor Writer is "less than" (or more specific than) the other.

ii. **if** expressions. Suppose there exists a class File2 that also inherits from Reader and Writer:

```
class File2 inherits Reader, Writer {
    read(): String { ... };
    write(s: String): SELF_TYPE {{ ...; self; }};
};
```

and additionally there is code like

[3] Examples include C++, Go, and Python.

```
if ... then
    new File
else
    new File2
fi
```

The type of this expression is not well-defined, since both Reader and Writer are possible least upper bounds of File and File2.

iii. Dispatch expressions. Suppose there are two base classes that both define the same method in conflicting ways:

```
class IsTrue {
    test(): Bool { true };
};
class IsFalse {
    test(): Bool { false };
};
class Chimera inherits IsTrue, IsFalse {};
```

It is unclear which definition would be used for "(**new** Chimera).test()".

As an aside, it's interesting to see how real programming languages solve these problems.

i. For **case**, Go has a feature analogous to **case** called the type switch statement. Python 3.10 introduced the "pattern matching statement" with similar functionality. And a pattern matching syntax is proposed for C++ as well.
   In all three languages, the match statement chooses not the "least" (or "best") branch, but instead the first branch that matches. So in our example, all three languages would choose the Reader branch.
   For more details, refer to the specification of each of those languages:
   A. Go Programming Language Specification, `https://go.dev/ref/spec#Type_switches`;
   B. C++ P1371, §7.3 First Match rather than Best Match, `https://wg21.link/p1371r3#page=21`; and
   C. Python PEP-622, `https://peps.python.org/pep-0622/#match-semantics`.
ii. For **if**, Go does not have an equivalent expression type, while Python does not conduct static typing. C++ requires the two alternatives to be somewhat compatible in type, so our test case above would result in a type error. See `https://godbolt.org/z/vzqYce51e`.
iii. For dispatch, Python uses the order of inheritance to decide which parent class "wins". C++ and Go, on the other hand, forbid ambiguous calls to inherited methods. See the following "playground" links:
   A. C++, `https://godbolt.org/z/6WffGY864`
   B. Go, `https://go.dev/play/p/IyhGKmTOliO`

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=$\langle n+1, F_n\rangle$
```