YOUR NAME – SUNet ID
CS143 Spring 2025 – Written Assignment 3

This assignment covers semantic analysis, including scoping, type systems. 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 20, 2025 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. **(16 pts)** Consider the following Cool programs:

   (a) **(8 pts)**

```
1    class A {
2        x: A; -- line 2
3        baz(): A {{x ←new A; x;}}; -- line 3
4        bar(): A {new A}; -- line 4
5        foo(): String {"Cool!"};
6    };
7    class B inherits A {
8        foo() : String {"are "};
9    };
10
11   class C inherits A {
12       foo() : String {"Compilers "};
13   };
14
15   class Main {
16       main (): Object {
17           let io : IO ←new IO, b : B ←new B, c : C ←new C in {{
18               io.out_string(c.baz().foo());
19               io.out_string(b.baz().foo());
20               io.out_string(b.bar().baz().foo());
21           }}
22       };
23   };
```

   What does this code currently print? By changing the types on lines 2-4 get this program to print "Compilers are Cool!".

   Answer:

(b) **(8 pts)** Here is an incomplete Cool program:

```
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 };
```

You need to fill 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:

2. **(24 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 $O[Int/y] \vdash y + y : Int$:

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

Consider the following Cool program fragment:

```
1    class A {
2        i: Int;
3        j: Int;
4        b: Bool;
5        s: String;
6        o: SELF_TYPE;
7        foo(): SELF_TYPE { o };
8        bar(): Int { 2 * i + j - i / j - 3 * j };
9    };
10   class B inherits A {
11      p: SELF_TYPE;
12      baz(a: Int, b:Int): Bool { a = b };
13      test(c: Object): Object { (* [Placeholder C] *) };
14   };
```

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

$O = \emptyset[\text{Int}/i][\text{Int}/j][\text{Bool}/b][\text{String}/s][\text{SELF\_TYPE}_B/o][\text{SELF\_TYPE}_B/p][\text{Object}/c][\text{SELF\_TYPE}_B/self]$

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

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. Bool $\leq$ Bool).

(a) **(6 pts)**

```
1          b ← p.baz(p.bar(),i)
```

Answer:

(b) **(6 pts)**

```
1          p ← o.foo()
```

Answer:

(c) **(6 pts)**

```
1          b ← baz(i+j,p.bar(i,o.foo()))
```

Answer:

(d) **(6 pts)**

```
1         case c of
2             s: Int => s;
3             i: String => j;
4             b: Object => i;
5         esac
```

Answer:

3. **(16 pts)** Consider the following Cool program:

```
1 class Main {
2     b: B;
3     main (): Object {{
4         b ← new B;
5         b.foo();
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 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) **(8 pts)**

```
1     class A {
2         i: Int ←1;
3         a: SELF_TYPE;
4         foo(): Int {i};
5     };
6
7     class B inherits A {
8         j: Int ←1;
9         baz(): Int {{i ←2 + i; i;}};
10        foo(): Int {{
11            j ← a.baz() + a.foo();
12            j;
13        }};
14    };
```

Answer:

(b) **(8 pts)**

```
1     class A {
2         i: Int ←1;
3         a: SELF_TYPE;
4         foo(): Int {i};
5     };
6
7     class B inherits A{
8         j: Int ←1;
9         baz(): Int {{ i ←2 + i; i; }};
10        foo(): Int { let a: A ←new B in {
11                j ← a@B.baz() + a.foo();
12                j;
13            }
14        };
15    };
```

Answer:

4. **(26 pts)** Consider the following extension to the Cool syntax as given on page 16 of the Cool Manual, which adds arrays to the language:

$$\begin{aligned} \text{expr} ::= & \text{ new TYPE[expr]} \\ | & \text{ expr[expr]} \\ | & \text{ expr[expr] } <- \text{ expr} \end{aligned} \tag{1}$$

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 "my_array:T[] ← 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 "my_array[i]" and modified using a expression like "my_array[i] ← value".

(a) **(18 pts)** Provide new typing rules for Cool which handle the typing judgments for: $O, M, C \vdash$ new T[e_1], $O, M, C \vdash e_1$[e_2] and $O, M, C \vdash e_1$[e_2] $<-$ e_3. Make sure your rules work with subtyping.

Answer:

(b) **(8 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:

5. **(18 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:

```
1  divide(numerator:Int, denominator:Int):MaybeInt {
2    if (denominator = 0) then
3        createNothing()
4    else
5        createSomething(numerator / denominator)
6    fi
7  }
```

   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:

```
1    let div_result: MaybeInt in {
2        div_result ←divide(i,j);
3        match(div_result) {
4            Some(n) => "The result was: ".concat(n.to_string()),
5            Nothing => "Can't divide by zero."
6        };
7    }
```

   The grammar rule for **match** is:

$$\text{expr} ::= \textbf{match}\ (\text{expr})\{\textbf{Some}(\textbf{n}) => \text{expr}, \textbf{Nothing} => \text{expr}\} \tag{2}$$

   (a) **(6 pts)** Write a type checking rule for the **match** expression, which preserves Cool's type safety guarantees.

   Answer:

(b) **(12 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: