# CS131 Compilers: Writing Assignment 3
# Due Saturday, May 15, 2021 at 23:59pm

## Name - Number

This assignment asks you to prepare written answers to questions on semantic analysis. Each of the questions has a short answer. 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. Written assignments are turned in at the start of lecture. You should use the Latex template provided at the course web site to write your solution.

I worked with: ()

Example for type rule in tex:

$$\frac{O[\text{Bo/x}][\text{Ob/x}](\text{x}) = \text{Ob}}{O[\text{Bo/x}][\text{Ob/x}] \vdash \text{x: Ob}}$$

1. (10 pts) Consider the following class definitions.

```
class A {
  i: Int
  o: Object
  b: B <- new B
  x: SELF_TYPE
  f(): SELF_TYPE {x}
}
class B inherits A {
  g(b: Bool): Object { (* EXPRESSION *) }
}
```

Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of (* EXPRESSION *) in the body of the method *g*, show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

(a) `x`
*Solution:* `SELF_TYPE_b`

(b) `self = x`
*Solution:*Bool

(c) `self = i`
*Solution:*error, the two sides of '=' is self and int, which is not equal.

(d) `let x: B <- x in x`
*Solution:*B

(e)
```
case o of
    o: Int => b;
    o: Bool => o;
    o: Object => true;
esac
```
*Solution:*Bool

2. (10 pts) Show the full type *derivation tree* for the following judgement: (You can use Bo as the type Bool and Ob as the type Object)

$$O[\text{Bool/x}] \vdash \texttt{x <- (let x:Object <- x in x = x): Bool}$$

*Solution:*

$$\dfrac{\dfrac{\dfrac{O[Bool/x] \vdash x : Ob \qquad O[Bool/x] \vdash x : Ob \quad \Rightarrow Ob = Ob}{0\ [\text{Bool/x}] \vdash \text{x=x : Bool}} \qquad O[Bool/x] \vdash x : Bo \quad Bo \leq Ob}{O[\text{Bool/x}] \vdash \text{let x:Object <- x in x = x : Bo}} \qquad Bo \leq Bo \quad O(x) = Bo}{O[\text{Bo/x}] \vdash \texttt{x <- (let x:Object <- x in x = x): Bo}}$$

3. (10 pts) Suppose we extend the grammar for Cool with a "**void**" keyword

$$\begin{array}{rcl} expr & ::= & \textbf{void} \\ & | & \ldots \end{array}$$

that is analogous to `null` in Java. (Currently objects are initialized to void if they have no other initializer specified, but there is no general-purpose **void** keyword.) We want to be able to use **void** whereever an object can be used, as in

```
let foo:Int <- if some_test
               then 5
               else void
               fi
in ...
```

Give a sound typing rule that we can add to the Cool specification to accomodate this new keyword.

*Solution:*

$$\dfrac{}{O, M, C \vdash \text{void: T}}$$

4. (10 pts) Suppose we extend Cool with exceptions by adding two new constructs to the Cool language.

$$\begin{array}{rcl} expr & ::= & \textbf{try}\ expr\ \textbf{catch}\ ID \Rightarrow expr \\ & | & \textbf{throw}\ expr \\ & | & \ldots \end{array}$$

Here **try**, **catch** and **throw** are three new terminals. "**throw** *expr*" returns *expr* to the closest dynamically enclosing catch block. Note that since **throw** expression returns control to a different location, we do not really care about the context in which throw is used. For example, (**throw** $false$) + 2 is a valid Cool expression (However, note that (**throw** $false$) + (2 + $true$) is not a valid Cool expression). Following is an example that uses the try-catch and throw constructs.

```
try
   if some_test1 then throw 34
   else if some_test2 then throw ‘‘undefined error’’
   else do_something fi fi
catch x =>
  case x of
      x:Int => do_something1
      x:String => do_something2
  esac
```

The above program fragment executes "do_something1" (with $x$ bound to the value 34) if "some_test1" evaluates to true. It executes "do_something2" (with $x$ bound to the value "undefined error") if "some_test1" evaluates to false but "some_test2" evaluates to true. It executes "do_something" if both "some_test1" and "some_test2" evaluate to false.

Give a set of new sound typing rules that we can add to the Cool specification to accommodate these two new constructs.

*Solution:*

$$\frac{O, M, C \vdash e : T_1}{O, M, C \vdash \text{throw e: } T}$$

$$\frac{O, M, C \vdash e_1 : \ T_1 \quad O[Object/x], M, C \vdash e_2 : \ T_2}{O, M, C \vdash \text{try } e_1 \text{ catch } x => e_2 : \ T_1 \cup T_2}$$

5. (10 pts) The Java programming language includes arrays. The Java language specification states that if $s$ is an array of elements of class $S$, and $t$ is an array of elements of class $T$, then the assignment $s = t$ is allowed as long as $T$ is a subclass of $S$. This typing rule for array assignments turns out to be unsound. (Java works around the fact that this rule is not statically sound by inserting runtime checks to generate an exception if arrays are used unsafely. For this question, assume there are no special runtime checks.)

   Consider the following Java program, which type checks according to the preceeding rule:

```
class Mammal { String name; }

class Dog extends Mammal { void beginBarking() { ... } }

class Main {
    static public void main(String argv[]) {
        Dog x[] = new Dog[5];
        Mammal y[] = x;

        /* Insert code here */
    }
}
```

   Add code to the `main` method so that the resulting program is a valid Java program (i.e., it type checks statically and so it will compile), but the program could result in an error being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

   *Solution:*

```
class Mammal { String name; }

class Dog extends Mammal { void beginBarking() { ... } }

class Main {
    static public void main(String argv[]) {
        Dog x[] = new Dog[5];
        Mammal y[] = x;
```

```
            Mammal Horse = new Mammal();
            y[0] = Horse;
            x[0].beginBarking();
        }
    }
```

When doing Mammal y[]=x, it records the address of x[], which is new Dog[5]. When I give y[0] the Horse, which is the father class of Dog, it is considerd safe, since Dog ¡= Mammal. However, the actual is unsafe, Horse don't have beginBarking() but passed the type checking.

6. (10 pts) Now that you know why Java arrays are problematic, you decide to add an array construct to Cool with sound typing rules. An array containing objects of type A is declared as being of type Array(A) and one can create arrays in Cool using the new Array[A][e] construct, where e is an expression of type Int, specifying the size of the array. One can access elements in the array using the construct e1[e2] which yields the e2'th element in array e1, and one can insert elements into the array using the notation e1[e2] <- e3. Finally, as in Java, an assignment from one array a to an array b does not make copies of the elements contained in a, but addresses of elements.

   (a) (2 pts) Give a sound subtype relation for arrays in Cool, i.e., state the conditions under which the subtype relation Array(T)≤ T' is valid.
   *Solution:*
   $$\frac{T' = Array(T'')T \leq T'''}{\text{Array(T)} \leq T^{''}}$$

   (b) Give sound typing rules that are as permissive as possible for the following constructs:

      i. (2 pts) new Array[A][e]
      *Solution:*
      $$\frac{O, M, C \vdash e : Int}{\text{O,M,C} \vdash \text{new Array[A][e]: Array(A)}}$$

      ii. (2 pts) e1[e2]
      *Solution:*
      $$\frac{O, M, C \vdash e_1 : Array(A) \quad O, M, C \vdash e_2 : Int}{\text{O,M,C} \vdash \text{e1[e2]: Array(A)}}$$

      iii. (4 pts) e1[e2] <- e3. Assume the type of the whole expression is the type of e1.
      *Solution:*
      $$\frac{O, M, C \vdash e_1 : Array(A) \quad O, M, C \vdash e_2 : Int \quad O, M, C \vdash e_3 : T' \quad T' \leq A}{\text{O,M,C} \vdash \text{e1[e2] <- e3 : Array(A)}}$$

7. (10 pts) The C programming language includes variable-length arrays(VLA), a feature where we can allocate an auto array (on stack) of variable size, which was used widely in the Linux Kernel for resource allocation. But because of the tedious translation into the machine code and possible security issue, the recent kernel is VLA-free and C++ is partially abandon it. Therefore, the compiler shall do the semantic check for unacceptable following statements. Extend the typing semantics can be found at C manual 8.1.2 Function calls and C++ semantics page 7, so that the following C++ code will emit a semantic error.

```
template<class T>class array{
  int s;
  T* elements;
public:
  array(int n); // allocate "n" elements and let "elements" refer to them
  array(T* p, int n); // make this array refer to p[0..n-1]
  operator T*(){return elements;}
  int size()const{return s;}
```

```
  // the usual container operations, such as = and [], much like vector
};

void h(array<double>a); //C++
void g(int m,double vla[m]); //C99
void f(int m,double vla1[m],array<double>a1) {
    array<double> a2(vla1,m); // a2 refers to vla1
    double*p=a1; //p refers to a1's elements

    g(m,vla1);
    g(a1.size(),a1); // a bit verbose
    g(a1); //???
}
```

The calls marked with ? ? ? cannot be written in C++. Had they gotten past the type checking, the result would have executed correctly because of structural equivalence. If we somehow accept these calls, by a general mechanism or by a special rule for array and VLAs, arrays and VLAs would be completely interchangeable and a programmer could choose whichever style best suited taste and application.