Solutions to Written Assignment 3

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) (2 pt) `x`
Solution: $\texttt{SELF\_TYPE}_{\texttt{B}}$

(b) (2 pt) `self = x`
Solution: `bool`

(c) (2 pt) `self = i`
Solution: Error; `Int` objects can only be compared with other `Int` objects

(d) (2 pt) `let x: B <- x in x`
Solution: `B`

(e) (2 pt)

```
case o of
 o: Int => b;
 o: Bool => o;
 o: Object => true;
esac
```

Solution: `bool`

2. (10 pts)

(a) (8 pts) Give a very short Cool program (less than 10 lines) that does not type check under the typing rules given in the Cool manual, but would actually never exhibit a runtime error.
Solution:

```
class A { };
class B inherits A {
  f():Int {1}
}
class Main {
  main: Int {let x:A <- new B in x.f()}
}
```

(b) (2 pts) Explain why this program does not type check and why it does not have a runtime error.
Solution: This program does not type check because the static type of `x` is `A`, and `A` does not have a method called `f`. This program does not have a runtime error because `x` always has dynamic type `B`, and class `B` defines a method `f` with a suitable signature.

3. (10 pts) After writing many tedious `while` loops to test your Cool compiler, you are fed up and decide to add a `for` loop construct to Cool that looks as follows:

```
for Id:T <- e1 aslongas e2 do e3 rof
```

In this construct, `e1` is the initial value of `Id`, `e2` is a continuation predicate (i.e., the loop executes as long as `e2` holds), and `e3` is the body of the loop. Give a sound (and sensible) typing rule for the `for` loop construct. Solution:

$$\frac{\begin{array}{l} O \vdash T' = \begin{cases} \text{SELF\_TYPE}_C & \text{if T is SELF\_TYPE} \\ T & \text{otherwise} \end{cases} \\ O \vdash e_1 : T_1 \\ T_1 \leq T' \\ O[T'/Id] \vdash e_2 : \text{Bool} \\ O[T'/Id] \vdash e_3 : T_2 \end{array}}{O, M, C \vdash \text{for Id} : T \leftarrow e_1 \text{ aslongas } e_2 \text{ do } e_3 \text{ rof: Object}}$$

4. (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 preceding rule:

```
class Animal { String name; }

class Dog extends Animal { void bark() { ... } }

class Main {
    static public void main(String argv[]) {
        Dog x[] = new Dog[5];
        Animal 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 operation being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

Solution:

```
Dog x[] = new Dog[5];
Animal y[] = x;

Animal a_cat = new Animal();
y[0] = a_cat; (*)
x[0].bark();
```

The problem here is that arrays are not just lists of values – they represent memory locations into which we can store new data.

Normally we say that A $\leq$ B if objects of type A can safely be used anywhere that objects of type B can be used. That's not quite true with `Dog[]` $\leq$ `Animal []`, since an `Animal []` object can be safely used on the left-hand side of the assignment marked by (*), while a `Dog[]` object cannot. Java handles this by adding a runtime check to every array assignment that determines whether the right-hand side of the assignment matches the dynamic type of the array.

5. (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`.

   (a) (2 pts) Give a sound subtype relation for arrays in Cool, i.e., state the conditions under which the subtype relation `Array(`τ`)` $\leq \tau'$ is valid.
   Solution:

$$\frac{\tau' = Array(\tau'') \quad \tau'' = \tau}{Array(\tau) \leq \tau'}$$

   (b) Give sound typing rules that are as permissive as possible for the following constructs: (You do not need to worry about `SELF_TYPE` in this question.)

      i. (2 pts) `new Array[A][e]`
      Solution:

$$\frac{O, M, C \vdash e : \texttt{Int}}{O, M, C \vdash \texttt{new Array[A][e]} : Array(A)}$$

ii. (2 pts) `e1[e2]`
Solution:

$$\frac{O, M, C \vdash e1 : \mathit{Array}(\tau) \quad O, M, C \vdash e2 : \texttt{Int}}{O, M, C \vdash \texttt{e1[e2]} : \tau}$$

iii. (4 pts) `e1[e2] <- e3`. Assume the type of the whole expression is the type of `e1`.
Solution:

$$\frac{O, M, C \vdash e1 : \mathit{Array}(\tau) \quad O, M, C \vdash e2 : \texttt{Int} \quad O, M, C \vdash e_3 : \tau' \quad \tau' \leq \tau}{O, M, C \vdash \texttt{e1[e2]} \leftarrow \texttt{e3} : \mathit{Array}(\tau)}$$