Solutions to Written Assignment 3

1. Consider the following class definitions.

```
class A {
  i : Int;
  o : Object;
  a : A <- new B;
  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.

`1) i + i`

`Int`

`2) x`

$\texttt{SELF\_TYPE}_{\texttt{B}}$

`3) self = x`

`Bool`

`4) self = i`

Error: `Int` objects can only be compared with other `Int` objects

`5) let x :  B <- x in x`

`B`

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

`Bool`

`7) a.f().g(b)`

Error: The class `A` does not have a method named `g`

`8) f()`

$\texttt{SELF\_TYPE}_{\texttt{B}}$

2. Someone has proposed that Cool be extended to allow comparison, addition, and multiplication operations on `Bool` objects as well as on `Int` objects. The comparison, addition, and multiplication operations are now defined for any combination of `Int` and `Bool` operands. An addition or multiplication operation involving an operand of type `Bool` produces a result of type `Int` (the `Bool` object is converted to 1 if it has the value `true`, and to 0 if it has the value `false`).

   Write the additional type checking rules (as in the lecture and the Cool Reference Manual) for these operations on `Bool` objects.

   The original type checking rule for arithmetic operations remains the same for subtraction and division.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{-, /\} \end{array}}{O, M, C \vdash e_1\ op\ e_2 : Int} \quad \text{[Arith]}$$

   A new rule is added for addition and multiplication.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \\ op \in \{*, +\} \end{array}}{O, M, C \vdash e_1\ op\ e_2 : Int} \quad \text{[Add-Mul]}$$

   The rule for non-equality comparisons must be extended to allow for `Bool` operands.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \\ op \in \{<, \leq\} \end{array}}{O, M, C \vdash e_1\ op\ e_2 : Bool} \quad \text{[Compare]}$$

   The rule for equality comparisons must be changed to allow for comparisons between `Int` and `Bool` operands.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 = String \vee T_2 = String \Rightarrow T_1 = T_2 \\ T_1 \in \{Int, Bool\} \Rightarrow T_2 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \Rightarrow T_1 \in \{Int, Bool\} \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad \text{[Equal]}$$

3. (a) Your friend Damon feels constrained by the fact that every `while` expression in Cool evaluates to `void`. He would like to be able to write functions such as this:

```
f() : Bool {
  let x : Bool <- true in
    while x loop x <- false pool
};
```

Damon wants to change the semantics of the `while` expression so that the value of the expression is the value of the body on the last execution of the loop. He must define the value of a `while` expression in the case that the body of the loop is never evaluated, however.

Damon's first proposal is to define the value of the `while` expression to be `void` when the predicate of the loop is `false` initially and the body is never evaluated. He says that the type checker can now infer the static type of the `while` expression to be the static type of the body, because `void` is a member of every type. Give an example Cool program that shows how this change would cause a runtime type error (beyond the existing Cool runtime errors), and explain how the error occurs.

```
class Main {
  main() : Int {
    5 + while false loop 7 pool
  };
};
```

Under the proposed semantics, this program would be accepted by the type checker because the type checker infers the static type `Int` for the `while` expression. At runtime, however, the `while` expression evaluates to `void`, and so an integer is added to `void`. The result of this addition is undefined, as under the standard definition of Cool an expression whose static type is `Int` cannot take the value `void` at runtime. As such, a new runtime error is introduced into Cool.

(b) After seeing your example, Damon comes up with a new suggestion. He proposes to eliminate the requirement that the predicate expression in a loop must be of static type `Bool`. Now, the predicate can have any static type, and a new method `is_true() : Bool` is added to the `Object` class.

The predicate is evaluated before each iteration of the loop. If the value of the predicate is `void`, the loop terminates. Otherwise, the method `is_true()` of the value of the predicate is invoked, and the loop terminates if the value returned by the method is `false`. If the value returned by the method is `true`, then the body of the loop is evaluated, and the process repeats. The value of the `while` expression is determined as follows:

- If the body of the loop is never evaluated, then the value of the `while` expression is the value of the predicate (from the first evaluation of the predicate).
- Otherwise, the value of the `while` expression is the value of the body on the last execution of the loop.

Can these modified `while` semantics be type checked statically to accept Damon's sample function above, while ensuring type safety (i.e., that no runtime type error will occur)? If so, write the most flexible type rule (the rule that accepts the most correct programs) for the modified `while` expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

$$\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : T_1 \sqcup T_2} \quad \text{[Loop]}$$

Under this type rule, the basic types `Int`, `String`, and `Bool` will be inferred by the type checker for a `while` expression only if the type checker infers the same basic type for both the predicate and the body of the loop. The existing rules of Cool do not allow an expression with static type `Int`, `String`, or `Bool` to take the value `void` at runtime. As a result, a `while` expression for which the type checker infers one of the static types `Int`, `String`, or `Bool` will not evaluate to `void`. The only runtime errors caused by the `while` expression are the runtime errors due to `void` that already exist in Cool.

(c) Damon wants to extend Cool by allowing method assignments. He would like to add a new assignment expression of the following form.

```
<exprB>.g <- <exprA>.f
```

Suppose that `<exprA>` evaluates to an object `a` of class `A`, and `<exprB>` evaluates to an object `b` of class `B`. Furthermore, `A` has a method named `f` and `B` has a method named `g`, and the two methods `f` and `g` have the same signature (the signature consists of the number of arguments, the types of the formal parameters, and the return type). The effect of the assignment would be to set the body of the method `g` of the object `b` to the body of the method `f` of the object `a`, so that subsequent invocations of the method `g` belonging to `b` would execute the body of the method `f` belonging to `a`. The value of the assignment expression would be `void`.

Damon says that, if `B` is a subclass of `A` (a descendant of `A` in the inheritance graph), then the inheritance rules of Cool guarantee that this operation is type safe.

Can Damon's method assignment expression be type checked statically to guarantee type safety? If so, write the most flexible type rule for the method assignment expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

This method assignment expression cannot be type checked statically while ensuring type safety because the dynamic types of the objects involved in the assignment may be different from the static types. The type checker cannot guarantee at compile time that the dynamic type of the object to which the method is being assigned is a subclass of the dynamic type of the object that contains the method being assigned.

```
class A {
  x : Int;
  f(z : Int) : Int {
    z + x
  };
};
class B inherits A {
  y : Int;
  f(z : Int) : Int {
    z + x + y
  };
};
```

```
class Main {
  main() : Object {
    let a : A <- new A,
        b : A <- new B in {
      a.f <- b.f;
      a.f(1);
    }
  };
};
```

Any static type rule for the method assignment expression would have to allow the method assignment `a.f <- b.f` in this program, because the static types of `a` and `b` are the same. However, executing this program would lead to a runtime error because the object `a` does not have an attribute `y`.

4. Suppose `f` is a function with a call to `g` somewhere in the body of `f`.

```
f(...) {
 ... g(...) ...
}
```

We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following two functions for computing positive powers of 2.

```
f(x : Int, acc : Int) : Int { if (0 < x) then f(x - 1, acc * 2) else acc fi };
g(x : Int) : Int { if (0 < x) then (2 * g(x - 1)) else 1 fi };
```

Here `f(x, 1)` = `g(x)` $= 2^x$ for $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not. A function in which all recursive calls are tail calls is called *tail recursive*.

(a) Here is a non-tail recursive function for computing factorials.

```
fact(n : Int) : Int { if (0 < n) then (n * fact(n - 1)) else 1 fi };
```

Write a tail recursive function `fact2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

```
fact2(n : Int, acc : Int) : Int {
  if (n = 0)
  then acc
  else fact2(n - 1, acc * n)
  fi
};
```

(b) Recall from lecture that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` as they compute 4!, showing the tree of activation records (each node of the tree shows the invocation of a function, and the arguments). How can a compiler make the execution of the tail recursive function `fact2` more efficient than that of `fact`? (Hint: Compare the stack space required for `fact(99)` with the stack space required for `fact2(99)`. Can `fact2` use fewer activation records?)

The function invocations, with arguments, are as follows.

$$\texttt{fact}(4) \rightarrow \texttt{fact}(3) \rightarrow \texttt{fact}(2) \rightarrow \texttt{fact}(1) \rightarrow \texttt{fact}(0)$$
$$\texttt{fact2}(4, 1) \rightarrow \texttt{fact2}(3, 4) \rightarrow \texttt{fact2}(2, 12) \rightarrow \texttt{fact2}(1, 24) \rightarrow \texttt{fact2}(0, 24)$$

A compiler can compile a tail recursive function such as `fact2` so that a new activation record is not needed for every recursive invocation of `fact2` in a computation. Suppose that the method `fact2` is invoked with the arguments 4 and 1 from a function `main`. The return address in the activation record for this invocation is an address R that specifies an instruction in the body of `main`.

Since `fact2` is tail recursive, after the recursive invocation of `fact2` with the arguments 3 and 4 has completed, no further computation must be done in the body of `fact2` to complete the invocation `fact2(4, 1)`. The return value of `fact2(3, 4)` is the return value of `fact2(4, 1)`. As a result, the compiler can replace the activation record for `fact2(4, 1)` with the activation record for `fact2(3, 4)`. The return address for `fact2(3, 4)` is the same as the return address for `fact2(4, 1)` (R), and the return value of `fact2(4, 1)` is now set during the execution of `fact2(3, 4)`. Figure 1 illustrates the stack space saved by this compiler optimization, showing the activation records during the invocation `fact2(3, 4)`.

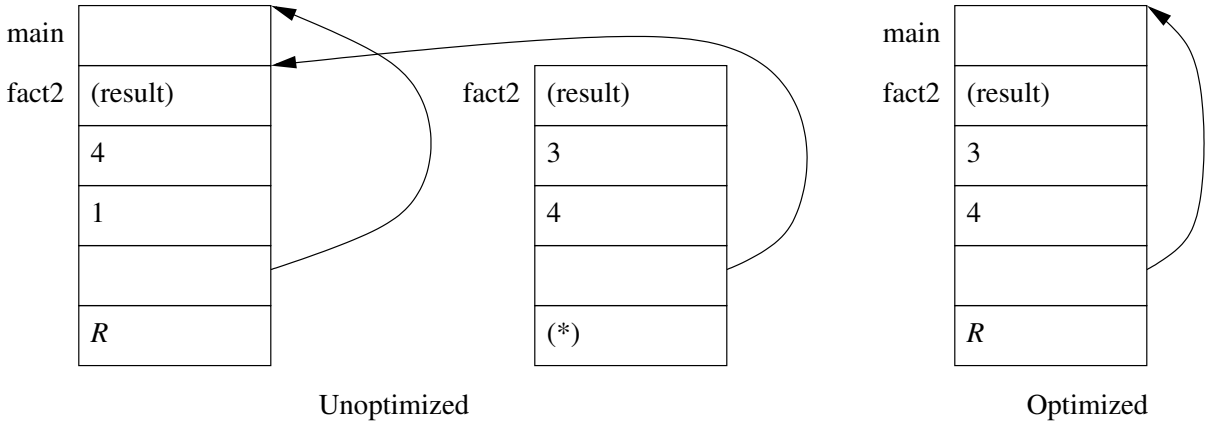

Figure 1: Activation records during the function invocation `fact2(3, 4)`.

5. In some languages, a class can have multiple methods with the same name, as long as these methods differ in the number and/or types of formal parameters. This is referred to as method *overloading*.

   Suppose we would like to add method overloading to Cool. Now, when generating code for a dispatch expression $e_0.f(e_1, \ldots, e_n)$, the compiler may need to choose the method to dispatch to (i.e., which slot in the dispatch table to jump to) amongst several valid possibilities. Let T_i be the static type of e_i for $i = 0, 1, \ldots, n$. Suppose that the compiler chooses a method f for the dispatch such that:

   - T_0 has a method f with n formal parameters of types $P_1, \ldots, P_n$; and
   - $T_i \leq P_i$ for $i = 1, \ldots, n$; and
   - If T_0 has more than one method named f, then, for any other method named f with n formal parameters $Q_1, \ldots, Q_n$ satisfying $T_i \leq Q_i$ for $i = 1, \ldots, n$, it must be the case that $P_i \leq Q_i$

for $i = 1, \ldots, n$. In other words, $P_1, \ldots, P_n$ are the most specific parameter types for a method named f that could be invoked.

If a *unique* method exists under these rules, then the dispatch is accepted by the type checker. If more than one method satisfies these conditions, then the type checker signals a type error at compile time.

Method overriding occurs as described in the original Cool Reference Manual. Specifically, a method defined in a child class overrides any method with the identical signature in the parent class.

Consider the following Cool program:

```
class A inherits IO {
  f(a : Object, b : Object) : Object { out_string("1") };   (* offset 0 *)
  f(a : Object, b : Int) : Object { out_string("2") };      (* offset 1 *)
};
class B inherits A {
  f(a : Object, b : Object) : Object { out_string("3") };   (* offset 0 *)
  f(a : Int, b : Object) : Object { out_string("4") };      (* offset 2 *)
};
class Main {
  main() : Object {
    let a : A <- new B,
        b : B <- new B,
        x : Object <- new Object,
        y : Object <- 1,
        z : Int <- 2 in
      (* DISPATCH *)
  };
};
```

For each of the following dispatch expressions, give the output of the program when `(* DISPATCH *)` is replaced by the dispatch expression, or specify that a type error would occur.

The compiler chooses an offset in the dispatch table for a method invocation at compile time, based on the static types of the arguments. When the invocation occurs at runtime, the method that executes is the method at that offset in the dispatch table. The following table shows the method offset chosen by the compiler (assuming the offsets are fixed according to the numbers in the comments in the program) and the output for each of the dispatch expressions.

Dispatch	Offset	Output
`a.f(x, x)`	0	`3`
`a.f(x, y)`	0	`3`
`a.f(x, z)`	1	`2`
`a.f(y, x)`	0	`3`
`a.f(y, y)`	0	`3`
`a.f(y, z)`	1	`2`
`a.f(z, x)`	0	`3`
`a.f(z, y)`	0	`3`
`a.f(z, z)`	1	`2`
`b.f(x, x)`	0	`3`
`b.f(x, y)`	0	`3`
`b.f(x, z)`	1	`2`
`b.f(y, x)`	0	`3`
`b.f(y, y)`	0	`3`
`b.f(y, z)`	1	`2`
`b.f(z, x)`	2	`4`
`b.f(z, y)`	2	`4`
`b.f(z, z)`	1 or 2	TYPE ERROR