1. [10 pts.] **Errors**

Consider the following Scala code fragment.

``` scala
def foo (i: Int, a: Array[Int]) : Int = {
    var s : String
    val j : Int = i - i - 1
    if (a(j) > 2) then
        s = s + "bar"
    else
        0
}
```

Identify at least three language violations in this code. For each one, indicate whether it is a static error, checked runtime error, or unchecked runtime error.
2. [10 pts.] **Type Inference**

Consider the following expressions written in a variant of our usual toy language syntax, including integers, booleans, and single-argument functions (constructed with `fun` and applied with `@`). The grammar of types is

\[
\tau ::= \alpha_i \mid \text{int} \mid \text{bool} \mid (\tau_1 \rightarrow \tau_2)
\]

where type variables \(\alpha_1, \alpha_2, \alpha_3, \ldots\) denote arbitrary unknown types (i.e. universally quantified types).

Give the most polymorphic possible type for each entire expression. For example, the most polymorphic type of \((\text{fun } x \ 42)\) is \((\alpha_1 \rightarrow \text{int})\). Hint: To infer the types you can extract and solve constraints, but this is not necessary; it is probably easier to figure them out informally.

(a) [2 pts.] \((@ \ (\text{fun } x \ x) \ 42)\)

(b) [2 pts.] \((\text{fun } x \ (\text{fun } y \ x))\)

(c) [2 pts.] \((\text{fun } x \ (@ x \ \text{true}))\)

(d) [2 pts.] \((\text{fun } x \ (\text{fun } y \ (\text{not} \ (@ x \ y))))\)

(e) [2 pts.] \((\text{fun } x \ (\text{fun } y \ (\text{fun } z \ (@ (@ x \ z) (@ y z))))))\)
3. [10 pts.] **Typing of Exception Handling**

In lecture, we examined formal typing rules of the form $TE \vdash e : t$ for a simple expression language. Suppose we extend this language with new expression forms, `throw` and `catch`, for raising and handling exceptions. To keep things simple, assume that there is only one kind of exception and that it carries no value information.

The `throw` expression throws an exception, which causes control to pass immediately to the nearest dynamically enclosing handler. Handlers are introduced by expressions `(catch $e_1$ $e_2$)`. This means to evaluate $e_1$ with the handler $e_2$ active. If no exception is thrown during the evaluation of $e_1$, then the value of the whole expression is just that of $e_1$. If an exception is thrown during evaluation of $e_1$, and is not caught within some nested handler in $e_1$, it will be caught by this handler: $e_2$ will be evaluated, and its value will be used as the value of the whole `catch` expression.

Example 1: the program `(catch (+ 2 throw) 3)` evaluates to 3.

Example 2: the program `(catch (+ 2 2) 3)` evaluates to 4.

Give typing rules for `catch` and `throw`, in the style of the rules described in lecture. Assume that we have just two types, `Int` and `Bool`. Your rules should constrain programs as little as possible, while still ensuring type safety.
4. [10 pts.] **Dynamic Function Binding**

```scala
class ScalaProgrammer {
  def who() = "Scala programmer"
  def code() = "code"
  def hacks() = "hacks " + code()
}

class OOProgrammer extends ScalaProgrammer {
  override def who() = "OO programmer"
  override def code() = "classes"
}

class FunctionalProgrammer extends ScalaProgrammer {
  override def who() = "Functional programmer"
  override def hacks() = "uses pattern matching"
}

val main = {
  val team = List(new OOProgrammer(), new FunctionalProgrammer())
  for (p <- team)
    println (p.who() + " " + p.hacks())
}
```

(a) [6 pts.] Assuming normal Scala semantics, where all member functions are treated as dynamically dispatched, what is the output of this program?

(b) [4 pts.] Suppose we translated this example into C++, which uses type-based static binding by default, so that none of the member functions in this example were treated as dynamically dispatched. Assuming `team` were declared to be a suitable container object with elements of type `ScalaProgrammer`, what would the output of this program be then?
5. [10 pts.] **Variant Records**

Consider the following Haskell program fragment:

```haskell
data T = P Int T T
        | Q Bool

f :: T -> Int
f x =
    case x of
        P i u v -> i * f u + f v
        Q b -> if b then 1 else 0

h :: T -> T -> Int
h x1 x2 = f x1 + f x2
```

Note that functions and constructors in Haskell are all Curried, and they can be applied without putting parentheses around their arguments. As an example, just to check your understanding:

\[ h \left( Q \left( \text{True} \right) \right) \left( P \left( 3 \right) \left( Q \left( \text{True} \right) \right) \left( Q \left( \text{False} \right) \right) \right) \]

evaluates to 4.

Suppose we wish to write a more-or-less equivalent object-oriented Scala program fragment, but using *dynamic dispatch* in place of pattern matching over constructors. Type T will be represented by

```scala
abstract class T {
    def f() : Int
}
```

with two subclasses P and Q. Function h will be implemented as

```scala
def h(x1:T, x2:T) : Int = x1.f() + x2.f()
```

and we want \( h \left( Q \left( \text{true} \right) , P \left( 3, Q \left( \text{true} \right) , Q \left( \text{false} \right) \right) \right) \) to evaluate to 4.

Give definitions for the subclasses P and Q.

Note: Of course, it would be possible to reproduce the Haskell program’s pattern matching using Scala pattern matching, but that is not what this question asks you to do; you must *not* use pattern matching!
6. [10 pts.] **Nested Procedures and Objects**

There is a relationship between nested functions in a procedural language and class member functions in an object-oriented language, under which the fields of the class correspond to the free variables of the functions.

Consider the following Scala code:

```scala
case class M(x:Boolean, y:Int, z:Int) {
  def f(w:Int) = if (x)
    z + w
  else
    w - 42
  def g(w:Int) = w + y
}
def h() = {
  val m = M(true,10,20)
  m.f(30) + m.g(40)
}
```

Calling `h()` returns the answer 100.

(a) [2 pts.] Identify the free variables of function `f` and the free variables of function `g`.

(b) [8 pts.] Let `R` be the record type defined by

```scala
case class R(f: Int => Int, g: Int => Int)
```

Write down a Scala function

```scala
M : (x:Boolean,y:Int,z:Int) => R
```

such that, if we replace the class definition `M` above by your function definition `M`, function `h` still behaves the same way (e.g. calling it still returns 100).
7. [10 pts.] **Efficiency of built-in types**

Consider a language with built-in support for an *immutable* character string type, including literals and a string concatenation operator (denoted `++`) with the same semantics as `+` on strings in Java or Scala.

Suppose compilers for the language are allowed to choose one of two methods for representing strings. Under both methods, strings are allocated boxed in the heap, so string values are heap pointers. Under method A, every string is represented as a contiguous array of characters, of exactly the required length, terminated with a null character. (For example, this is how C represents strings.) Under method B, every string is represented as a singly-linked list of single characters (i.e. one character per linked record), linked in string order, terminated by a null pointer. (For example, this is how Haskell represents strings.)

Describe a timing experiment that can be used to determine which method is being employed by your compiler. The experiment should consist of two short programs (in pseudo-code) that are to be compiled and run. If method A is being used, the run times of the programs should be about equal; if method B is being used, program #1 should run much faster than program #2.

Hint: How much work does it take to concatenate a single character to the beginning or the end of an existing string?
8. [10 pts.] Abstract Data Types

A bag is a data type that is just like a set, except that it can contain multiple copies of a given element.

Here is a Scala description of a suitable signature for a simple abstract data type $B$ of immutable bags of elements of type $A$:

```scala
abstract class BagSig[A] {
  type B
  val empty : B
  def insert(b:B,y:A) : B
  def delete(b:B,y:A) : B
  def union(b:B,b2:B) : B
  def count(b:B,x:A) : Int
}
```

The semantics of these operators should be obvious. In particular, $\text{count}(b,x)$ returns the number of elements in the bag $b$ that are equal to $x$. (Obviously, $\text{count}$ should always return a non-negative number.)

There are many possible ways to make a concrete implementation matching this signature. Here is one, in which each bag, i.e. value of type $B$, is represented by the history of the constructors used to build the bag. This makes the implementation of the constructor functions trivial, and puts all the work into the $\text{count}$ function.

```scala
class Bag[A] extends BagSig[A] {
  sealed abstract class B
  private case object EmptyB extends B
  private case class InsertB(b:B,y:A) extends B
  private case class DeleteB(b:B,y:A) extends B
  private case class UnionB(b1:B,b2:B) extends B

  val empty : B = EmptyB
  def insert(b:B,y:A) : B = InsertB(b,y)
  def delete(b:B,y:A) : B = DeleteB(b,y)
  def union(b1:B,b2:B) : B = UnionB(b1,b2)
  def count(b:B,x:A) : Int = b match {
    case EmptyB => 0
    fill in the remaining cases here:
  }
}
```

Your task is to complete the definition of the $\text{count}$ function above.