1. For each of the following IC constructs, state whether it is well-typed in some typing context. If so, give the most general typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why.

   (a) `(new int[x.length])[x[2]]`

   (b) `if (x == v[x] && y == "true") x = y;`

   (c) `((a == b) == c) && (a == (b + "c"))`

   (d) `f(x)[x.length] = y[2]`

2. Suppose we extend IC with tuples of the following form. A tuple type is written as a sequence of types in parentheses. For example, the type `(int,bool,string)` represents a 3-tuple. The individual elements of the tuple can be accessed (i.e., read or written) in a manner similar to array elements. For example, if `x` has type `(int,bool,string)`, the expression `x[0]` has type `int`, `x[1]` has type `bool`, and `x[2]` has type `string`. Tuples are unlike arrays in that the index expression must be a constant. For simplicity, we assume that we require tuples don't contain class types; this ensures that different tuples cannot be subtypes of each other.

   (a) Explain why is it necessary to require that the index of a type expression must be a constant.

   (b) Write additional typing rules in the static semantics of IC for expressions and statements to support tuples.

   (c) Consider the types $\texttt{T}_1 =$`(int,(int,int)[])` and $\texttt{T}_2 =$`(int,(int,int))[]`. Consider a variable `x` having either type $\texttt{T}_1$ or type $\texttt{T}_2$. Write an expression which type-checks and has the same type in both cases; and an expression which type-checks if `x` has type $\texttt{T}_1$, but doesn't if `x` has type $\texttt{T}_2$. We require that `x`, `0`, and `1` are the only variables and constants in your expressions.

   (d) Syntactically, the tuple element access expression looks like an array element access expression. Will this create problems for type checking? Explain briefly.

3. Consider a C-like language that manipulates pointers. Statements have the following syntax:

$$S \ ::= \ \texttt{x} = n \ | \ \texttt{x} = \texttt{NULL} \ | \ \texttt{x} = \&\texttt{y} \ | \ \texttt{x} = \texttt{y} \ | \ \texttt{x} = *\texttt{y} \ | \ *\texttt{x} = \texttt{y}$$

   where n is an integer constant, and `x` and `y` are arbitrary variables. We consider that the only types for variables are integers and pointer types. If `T` is an arbitrary type, then `T`$*$ is the type for pointers to variables of type `T`. This allows to create multi-level pointers when `T` itself is a pointer type. The syntax for types is:

$$\texttt{T} ::= \texttt{int} \ | \ \texttt{T}*$$

   (a) Write typing rules for all of the assignment statements. Use `unit` to denote the type of statements.

   Now consider that we extend this syntax to model heap-allocated objects in C++. A declaration of the form `A* x` declares `x` to be a pointer to an object of class `A`. The assignment `x = new A` creates a new object of class `A` and stores a pointer to it in `x`. We add field assignment statements: `x->f = y` and `y = x->f`, where `x` is a pointer to an object, and `f` is a field of that object. However, we forbid declarations of the form `A x` (which essentially means that we don't allow stack-allocated objects). The types include integers, classes `C`, and pointers:

$$\texttt{T} = \texttt{int} \ | \ \texttt{C} \ | \ \texttt{T}*$$

   We assume that inheritance yields a subtype relation, and the typing rules for assignments use the subsumption rule for object values.

(b) We claim that covariant subtyping for pointer types is unsound. Show this subtyping rule and a counterexample program which would typecheck with that rule, but would produce a type error at run-time. You are allowed to use only the kinds of assignments presented in this problem. You can assume that the program contains two classes `A` and `B` such that `B` is a subclass of `A`, and contains a field `f`, which `A` doesn't.

(c) Contravariant subtyping for pointer types is also unsound. Write the contravariant subtyping rule and a program which would typecheck with that rule, but would produce a run-time type error.

(d) Assume that the language supports multiple inheritance. Show that field conflicts may occur even if the classes in the program all have different field names. Write a class hierarchy that shows such conflicts.