Name: **PID:**

Quiz 4

Please indicate for each of the following statements whether or not it is correct. (Correct answer: 1 point; incorrect answer: -0.25 points) **[5 points]**

1. **With static type checking, types are associated <u>with values</u>; with dynamic type checking, types are associated <u>with expressions/variables</u>. True or false:_____**
   False; this is reversed.
2. **Interpreters usually employ dynamic type checking. True or false:_____**
   True; dynamic type checking == runtime checks, which interpreters do anyway.
3. **Static type checking cannot be applied to recursive types such as lists because the set of possible values is infinite. True or false:_____**
   False; static type checking examines the syntax tree (expressions), not the sets of values.
4. **Type errors are more likely to remain undetected in languages that allow type coercion. True or false:_____**
   True; coercion automatically converts non-equivalent types and can thus mask errors.
5. **Composite types such as Haskell's tuples fundamentally requires runtime type checking.**
   True or false:_____
   False; disjoint union types require runtime checks (in strongly-typed languages).

In the context of type equivalence, briefly explain structural equivalence and the problem that name equivalence addresses. **[2 points]**

Under structural equivalence, two composite type are considered equivalent if they have the same components (usually in the same order).

This can cause similar, but semantically non-equivalent, type to be considered equivalent by the type checker. For example, the following types are semantically different but have the same structure.

```
data Dog = Dog {name :: String, age :: Int}
data Cat = Cat { name :: String, age :: Int}
```

Name equivalence avoids this problem by considering types with different names to not be equivalent.

Explain the difference between a primitive type and a composite type. [1 point]

A primitive type is "atomic"; its values are considered to be a unit and cannot be split into smaller parts. Primitive types are usually built into the language.

A composite type is defined in terms of simpler types and usually has named components (field names).

Explain the difference between an anonymous function and a closure. [2 points]

An anonymous function (or function literal) is a function that is defined without being bound to a name. In Haskell and Python, such functions are called lambda expressions.

A closure is a nested function with free variables, where the free variables are bound to values from the enclosing scope(s). (A closure "captures" the bindings of the free variables.)

An anonymous function can be a closure, and a closure can be anonymous, but not all anonymous functions are closures (= have free variables), nor are all closures anonymous.