Introduction to Programming: Quiz 2 Solutions

Part A: Multiple choice

Questions 1–3 use these declarations:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving (Eq, Show, Ord, Enum)

data Shape = Circle Double | Square Double | Rectangle Double Double
  deriving (Eq, Ord, Show)
```

1. What does `[d | d ← [Sun ..], d > Thu]` evaluate to?

   Answer: (A) `[Fri,Sat]`

   The derived **`Enum`** instance lets `[Sun ..]` run from `Sun` to the last constructor, so it is the list `[Sun,Mon,Tue,Wed,Thu,Fri,Sat]`. The derived **`Ord`** instance orders the constructors in the order they are declared, so `d > Thu` holds exactly for `Fri` and `Sat`. Option (D) is wrong because range notation needs an **`Enum`** instance, not a numeric type.

2. Which of the following evaluates to **`True`**?

   Answer: (C) `Circle 9.0 < Square 1.0`

   The derived **`Ord`** instance compares constructors first, in declaration order, so `Circle` comes before `Square`, which comes before `Rectangle`. The fields are compared only when the constructors are equal.

   (A) `Square 1.0 < Circle 10.0` is **`False`**, since `Square` comes after `Circle`, whatever the fields are.

   (B) `Rectangle 1.0 5.0 < Rectangle 1.0 2.0` is **`False`**: the constructors agree, so the fields are compared from left to right, and the first fields are equal, but `5.0 < 2.0` is false.

   (C) `Circle 9.0 < Square 1.0` is **`True`**, since `Circle` comes before `Square`.

   (D) `Square 2.0 > Rectangle 1.0 1.0` is **`False`**, since `Square` comes before `Rectangle`.

3. What is the type of `map Rectangle [1.0, 2.0]`?

   Answer: (B) `[Double → Shape]`

The constructor has type `Rectangle :: Double → Double → Shape`. Mapping it over a list of `Double`s applies it to one argument at a time, and each partial application `Rectangle 1.0` has type `Double → Shape`. Partial application of a constructor is allowed, so (D) is wrong.

4. What is the most general type of `f`?

```
f x ys = length (filter (== x) ys) > 0
```

Answer: (D) `Eq a ⇒ a → [a] → Bool`

The section `(== x)` compares elements of `ys` with `x`, so both have the same type `a`, and this needs only `Eq a`. The comparison `> 0` is made on `length ( ... ) :: Int`, so it puts no `Ord` or `Num` constraint on `a`.

5. Consider the following code.

```
data Coin = H | T deriving (Eq, Show)

instance Show Coin where
  show H = "Heads"
  show T = "Tails"
```

What happens when we load this file and evaluate `show [H, T]`?

Answer: (D) Compile-time error: duplicate `Show Coin` instance

The clause `deriving (Eq, Show)` already generates an instance `Show Coin`, and the explicit `instance Show Coin` declares a second one. A type can have only one instance of a class, so the file does not load, and `show [H, T]` is never evaluated. Removing `Show` from the `deriving` clause would give `"[Heads,Tails]"`.

6. Which expression is rejected by the type checker?

Answer: (C) `[(+3)] == [(*5)]`

Equality on lists needs equality on the elements, and here the elements are functions. Functions have no `Eq` instance, so the type checker rejects the expression. The other three are accepted:

(A) tuples of a number and a `Char` have an `Eq` instance, and the result is `True`;
(B) strings have an `Ord` instance, and the result is `"abd"`;
(D) `read "42" + (1 :: Int)` is accepted because `+` forces `read "42"` to have type `Int`, and the result is `43`.

7. What happens when this definition is compiled?

```
average :: [Double] → Double
average xs = sum xs / length xs
```

Answer: (A) Compile-time error

The type of the division is `(/) :: Fractional a ⇒ a → a → a`, so both operands must have the same type. Here `sum xs :: Double` but `length xs :: Int`, so the definition does not type-check. The usual fix is `sum xs / fromIntegral (length xs)`. Option (D) is wrong because the definition never compiles; even with the fix, `average []` would be `0/0`, which is `NaN` for `Double`, not an exception.

8. Each expression is typed at the `ghci` prompt. Which one *never finishes* printing its result?

Answer: (D) `filter (< 5) [1..]`

GHCi prints `[1,2,3,4` and then hangs: `filter` goes on testing `5`, `6`, `7`, and so on, for ever, since it cannot know that no later element is less than `5`. The other three finish:

(A) `length` counts the elements without evaluating them, so the divisions by zero are never forced, and the result is `10`;
(B) `take 3` stops after the first three even numbers, giving `[2,4,6]`;
(C) `head` returns `2` without ever looking at `reverse [1..]`.

Part B: Program tracing

9. Consider the following code.

```
pick :: Int → [Int] → Int
pick _ []     = 0                   -- (1)
pick 0 (x:_)  = x                   -- (2)
pick n (_:xs) = pick (n-1) xs       -- (3)

nums :: [Int]
nums = map (*3) [1 ..]
```

(a) The equations of `pick` are tried from top to bottom, and within an equation the patterns are matched from left to right. Equation (1) has `_` as its first pattern, which forces nothing, and `[]` as its second, which forces the list. So the list is unfolded before `4-2` is evaluated. Equation (1) then fails, and equation (2) forces the first argument to compare it with `0`. That fails too, so equation (3) applies. Since `n` is shared, `n-1` uses the already computed value of `n`. Each unfolding of `map` and `[k ..]` is written as one step.

```
  pick (4-2) nums
= pick (4-2) (map (*3) [1 ..])            -- definition of nums
= pick (4-2) ((*3) 1 : map (*3) [2 ..])   -- (1) forces the list
= pick 2 ((*3) 1 : map (*3) [2 ..])       -- (2) forces n
= pick (2-1) (map (*3) [2 ..])            -- (3)
= pick (2-1) ((*3) 2 : map (*3) [3 ..])   -- (1) forces the list
= pick 1 ((*3) 2 : map (*3) [3 ..])       -- (2) forces n
= pick (1-1) (map (*3) [3 ..])            -- (3)
= pick (1-1) ((*3) 3 : map (*3) [4 ..])   -- (1) forces the list
= pick 0 ((*3) 3 : map (*3) [4 ..])       -- (2) forces n
= (*3) 3                                  -- (2)
= 9
```

The expressions `(*3) 1` and `(*3) 2` are discarded by the pattern `_` in equation (3), so they are never evaluated, and only `(*3) 3` is computed.

(b) (i) `pick 1 [5 `div` 0, 7, 8]`

Answer: `7`

Equation (1) fails because the list is not empty, and equation (2) fails because `1` is not `0`. So equation (3) rewrites the expression to `pick 0 [7, 8]`, and equation (2) then gives `7`. The head `5 `div` 0` is matched by the pattern `_` and discarded without being evaluated, so no error occurs.

(ii) `pick 1 [5, 7 `div` 0, 8]`

Answer: `*** Exception: divide by zero`

As in (i), the expression rewrites to `pick 0 [7 `div` 0, 8]`, and equation (2) returns the head `7 `div` 0` without evaluating it, since `x` is a plain variable. The result of `pick` is therefore the unevaluated expression `7 `div` 0`. Printing the result forces the division, and this raises the divide-by-zero exception.