COM2001 — Advanced Programming Topics

Exercise Sheet 2: More Instance Declarations

Spring Semester

1 More instance declarations

When grading students' work, two of the main requirements of a grading system are that

- It should be possible to determine whether one grade is higher or lower than another; and
- It should be possible to determine whether a grade counts as a pass or a fail.

This can be modelled as a class:

```
class (Ord gs) ⇒ GradingSystem gs where
  isPass :: gs → Bool
  isFail :: gs → Bool

  -- default
  isFail = not ∘ isPass
  isPass = not ∘ isFail
```

Problem 1. What would you say (use your own judgment) is the "minimal complete definition" required for making a type into an instance of `GradingSystem`?

Solution. The user should define at least one of the functions: `isPass`, `isFail`.

There are many different grading systems in everyday use. For example:

- School exams are often assigned a grade ranging from A (highest) to E (lowest), with a special mark "U" (*unclassified*) for failing solutions;
- Our university modules are assigned a percentage ranging from 100% (highest) to 0% (lowest), together with a special mark "NC" (*Not Completed*) for students who were registered for the module but didn't do the assessment.
- Overall performance in a UK degree is often indicated by grades from the set { 1, 2.1, 2.2, 3, Pass, Fail }, and can be with or without "honours".

Problem 2. Define data types representing each of these grading systems, and show how to make them instances of `GradingSystem`.

Solution.

```
data School = A | B | C | D | E | U
  deriving (Eq, Ord)

-- This may vary according to exam type.
instance GradingSystem School where
  isPass grade
    | grade ≤ C  = True
    | otherwise  = False
```

```haskell
data DCSModule = DCSModule { percentage :: Int }
  deriving (Eq, Ord)

-- This is a simplified version of the situation.
instance GradingSystem DCSModule where
  isPass (DCSModule p)  = (p ≥ 40)
```

```haskell
data DegClass = Deg1 | Deg21 | Deg22 | Deg3 | DegPass | DegFail
  deriving (Eq, Ord)
data DegHons  = WithHons | WithoutHons
  deriving (Eq, Ord)
data Degree   = Degree DegClass DegHons
  deriving (Eq, Ord)

-- Use pattern matching
instance GradingSystem Degree where
  isPass (Degree DegFail _) = False
  isPass _                  = True
```

2 Correcting and editing instance declarations

Problem 3. Without running it, identify as many syntax and typing errors in the following Haskell code as you can:

```haskell
-- Comment this section out to run the code
--
  type UserInfo = UserInfo {                                    -- line 1
    Name        :: String;                                      -- line 2
    PhoneNumber :: Integer                                      -- line 3
    }                                                           -- line 4
                                                                -- line 5
  class Show UserInfo                                           -- line 6
    show :: UserInfo → String                                   -- line 7
    show UserInfo n p =    "(name: " + n + ", tele: " + p       -- line 8
--
--
```

Solution. There are lots of errors! Here are some of them, given by line number:

1. `type` should be `data`.
2. `Name` should start lower case. The semicolon should be a comma.
3. `PhoneNumber` should start lower case.
6. `class` should be `instance`. Line should end with `where`.
7. The type signature should not be repeated in an `instance` declaration.
8. The argument `UserInfo n p` should be in brackets. The bracket before `name` is unmatched. The plus signs should be `++`. The value `p` needs to be converted to a `String` before appending it.

Problem 4. Write a corrected version of this code and check that it correctly displays the details of a user called Bob whose phone number is 555-7890.

Solution.

```haskell
data UserInfo = UserInfo {
    name        :: String,
    phoneNumber :: Integer
    }

-- NB. This can be improved significantly
instance Show UserInfo where
    -- show :: UserInfo → String
    show (UserInfo n p) = "name: " ++ n ++ ", tele: " ++ show p

bob :: UserInfo
bob = UserInfo "Bob" 5557890
```

Problem 5. Edit your version of the code so that it can also display the details of a user called Mary whose phone number is unlisted.

Solution. Again, this needs improvement. For example, the international number `0044-114-222-1800` (this is the main DCS number) would be stored as the integer value `441142221800`. The `show` function needs to format the phone number and add back the two zeroes. How would you do that? Does it need more information to be stored about the user?

```haskell
data PhoneNumber = Phone Int | Unlisted

data UserInfo2 = UserInfo2 {
    name2        :: String,
    phoneNumber2 :: PhoneNumber
    }

-- NB. This can be improved significantly
instance Show UserInfo2 where
    -- show :: UserInfo2 → String
    show (UserInfo2 n p) = "name: " ++ n ++ ", tele: " ++ number
      where
        number = case p of
          Phone n   →  show n
          Unlisted  → "unlisted"

bob2 :: UserInfo2
bob2 = UserInfo2 "Bob" (Phone 5557890)

mary :: UserInfo2
mary = UserInfo2 "Mary" Unlisted
```