NATIONAL UNIVERSITY OF SINGAPORE

Department of Computer Science, School of Computing

IT5100A—Industry Readiness: Typed Functional Programming

Academic Year 2025/2026, Semester 1

ASSIGNMENT 2

TYPECLASSES AND THE RAILWAY PATTERN

Due: 23 Sep 2025 **Score**: [30 marks] (30%)

PREAMBLE

JavaScript Object Notation (JSON)[1] is a frequently used data-interchange format. A JSON value is one of the following things:

1. An object in the form of `{ key1: value1, key2: value2, ... }` where keys are strings and values are JSON values
2. A list of the form `[v1, v2, ...]` where each `v` is a JSON value
3. A string
4. A number (we shall restrict ourselves to integers)
5. A boolean
6. `null`

Here is an example JSON value:

```
{ "name":    "Bob"
, "id":      1234
, "friends": ["Alice", "Charlie", null]
, "courses": {
      "IT5100A": {
        "completed": true,
        "score":     47
      }
    , "IT5100B": {
        "completed": false,
        "score":     null
      }
  }
, "email":   null }
```

We can represent `JSON` values using a **`JSON`** algebraic data type, defined as such:

[1] See `https://www.json.org/json-en.html`.

```haskell
data JSON = O [(String, JSON)]
          | L [JSON]
          | N Int
          | S String
          | B Bool
          | Null
  deriving Eq
```

Then, our JSON object from above can be represented in Haskell like so:

```haskell
myJson :: JSON
myJson = O [("name", S "Bob")
          , ("id", N 1234)
          , ("friends", L [S "Alice", S "Charlie", Null])
          , ("courses", O [("IT5100A", O [("completed", B True)
                                        , ("score", N 47)])
                          ,("IT5100B", O [("completed", B False)
                                        , ("score", Null)])])
          , ("email", Null)]
```

However, working with JSON values can be unwieldly. Firstly, JSON values are recursive data structures, so when a transformation on, say, all integer values in a JSON object is to be performed, we must recursively traverse into the structure of the JSON values and update every the integer values. As such, any transformation of JSON values will involve a large amount of uninteresting "boilerplate" that merely traverses into the JSON structure. In addition, JSON values are untyped (aside from being able to discriminate the type of the value), as such, we are not guaranteed that an object, say, has a key called `"courses"`, leaving us with the possibilities of having runtime errors by querying data from a JSON value that does not exist.

In this assignment, we are going to work on some simple JSON operations using type classes and the railway pattern.

This assignment is worth **30 marks** which constitutes **30%** of your overall grade. It is due on **23 Sep 2025**. Late submissions reduce the attainable marks by **4 marks** for each late day or part thereof. For example, if you submit your solution one day and one hour late, you will not be able to receive more than 12 marks.

Some of these questions have some requirements. Thus, even though your solution to a question operates correctly, your solution must be implemented under their respective requirements to obtain the full score. Ensure that your submission is compilable; uncompilable code will be penalized heavily. You are not allowed to use any external dependencies. Only modules from Haskell's `base` library (which include `Prelude`) can be used.

You are given a template file (`JSONOps.hs`) and a public test suite. You may use the test files to run your tests. Test your code *section by section*. That is, after completing solutions for a section, open `Main.hs` and uncomment the `import` statement for that section, and uncomment the corresponding list item in `assignment2`. Compile `Main.hs` and run it:

```
ghc Main.hs
./Main
```

You are to submit *only* `JSONOps.hs` to Canvas. We will use a separate private test suite to test your code.

TRANSFORMATIONS [15 marks]

Deeply nested/recursive data structures are a pain to deal with. For example, adding 1 to every component integer in a deeply-nested tuple, say `(1, (2, ((3, (4, 5)), 6)))`, is incredibly frustrating.

Actually, type classes can be of great help here. To show this, we have defined a new class called **`TransformOn`**:

```haskell
class TransformOn a b where
  transformOn :: (a -> a) -> b -> b
```

The idea of this type class is that an instance of **`TransformOn`** `a b` is able to apply a transformation of `a` (having type `a -> a`) *everywhere* in a term of type `b` wherever a term of type `a` occurs.

To give you some concrete examples, let us try to define a way to apply an **`Int`** transformation function on an **`Int`**. This should be straightforward, since we can just apply the function directly!

```haskell
instance TransformOn Int Int where
  transformOn :: (Int -> Int) -> Int -> Int
  transformOn f x = f x
  -- point-free style: transformOn = id
```

Then notice that as long as the elements of a tuple can be transformed using an **`Int`** transformer, then the tuple *itself* can also be transformed with that **`Int`** transformer!

```haskell
instance (TransformOn Int a, TransformOn Int b) ⇒ TransformOn Int (a, b) where
  transformOn :: (Int -> Int) -> (a, b) -> (a, b)
  transformOn f (x, y) = (transformOn f x, transformOn f y)
```

And just like that, we are able to transform all the integers in a deeply nested tuple! All we need to do is to apply the `transformOn` method onto whatever **`Int`** function you want, and this gives us a function that transforms *all* **`Int`**s occurring in *any* transformable type!

```haskell
addDeeply :: TransformOn Int a ⇒ Int -> a -> a
addDeeply k = transformOn (+k)
```

Let's try this out!

```
ghci> :{
ghci| myDeepInts :: (Int, (Int, ((Int, (Int, Int)), Int)))
ghci| myDeepInts = (1, (2, ((3, (4, 5)), 6)))
ghci| :}
ghci> addDeeply 1 myDeepInts
(2, (3, ((4, (5, 6)), 7)))
ghci> addDeeply 3 myDeepInts
(4, (5, ((6, (7, 8)), 9)))
```

Voila! Just from these type class instances, we are able to apply any `Int` transformation on any deeply nested tuple of integers using `transformOn`!

Our goal in this section is to be able to exploit this programming pattern on our `JSON` type.

Question 1 (JSON-on-JSON Transformers) [8 marks]. Define a type class instance of `TransformOn JSON JSON` that applies a JSON transformation everywhere in a JSON value! Although a JSON transformation can directly transform a JSON value, we must also recursively apply the transformation on all JSON values that are contained within the outer JSON value. Example runs will be shown after the next question.

Question 2 (Handy JSON Transformers) [7 marks]. Define the following functions:

1. `addAll k x` adds `k` to every integer occurring in `x`
2. `negateAll x` negates all occurring boolean values in `x`—i.e. true becomes false, and false becomes true
3. `filterNull x` transforms all JSON values where:
   - key-value pairs in JSON objects where values are `null` are removed
   - `null` values in JSON lists are removed

Tip: All of your functions should be written as `transformOn f` for some `f`. In addition, since the `transformOn` function should already handle the recursion for you, your function `f` only needs to handle the top layer of the transformation.
We'll give you an example for free. A function that increases every JSON number by one everywhere can be written like this:

```
incAll = transformOn aux where
  aux :: JSON -> JSON
  aux (N x) = N (x + 1)
  aux x = x
```

Example runs follow.

```
ghci> :{
ghci| myJson :: JSON
ghci| myJson = O [("name", S "Bob"), ("id", N 1234)
ghci|           , ("friends", L [S "Alice", S "Charlie", Null])
ghci|           , ("courses", O [("IT5100A", O [("completed", B True)
ghci|                                         , ("score", N 47)])
ghci|                           ,("IT5100B", O [("completed", B False)
ghci|                                         , ("score", Null)])])
ghci|           , ("email", Null)]
ghci| :}
ghci> myJson
{ "name": "Bob", "id": 1234, "friends": ["Alice","Charlie", null]
, "courses": {
      "IT5100A": { "completed": true, "score": 47 }
    , "IT5100B": { "completed": false, "score": null } }
, "email": null }
ghci> addAll 10 myJson
{ "name": "Bob", "id": 1244, "friends": ["Alice","Charlie", null]
, "courses": { "IT5100A": { "completed": true, "score": 57 }
             , "IT5100B": { "completed": false, "score": null } }
, "email": null }
ghci> negateAll myJson
{ "name": "Bob", "id": 1234, "friends": ["Alice","Charlie", null]
, "courses": { "IT5100A": { "completed": false, "score": 47 }
             , "IT5100B": { "completed": true, "score": null } }
, "email": null }
ghci> filterNull myJson
{ "name": "Bob", "id": 1234, "friends": ["Alice","Charlie"]
, "courses": { "IT5100A": { "completed": true, "score": 47 },
               "IT5100B": { "completed": false } } }
```

JSON QUERIES [15 marks]

Querying data from JSON values is a potentially empty action. For example, trying to get the underlying integer from a JSON string gives nothing; querying a key in a JSON object where the key doesn't exist in the object also gives nothing. As such, we want to define functions that safely obtain information from JSON values.

Question 3 (Getting Numbers) [5 marks]. This is the most straightforward. Write a function `getN x` that retrieves the underlying integer from a JSON number. If `x` is not a JSON number, return nothing. You do not need to recursively search for numbers in a JSON value. Example runs follow.

```
ghci> getN (N 123)
Just 123
ghci> getN (S "123")
Nothing
ghci> getN (O [("number", N 123)])
Nothing
```

Question 4 (Querying Keys) [5 marks]. JSON objects are kind of like dictionaries in Python. Recall that in Python, dictionaries have a `get` method that retrieves the value corresponding to a key. However, if the key does not exist, `get` returns `None`.

Write a function `queryKey k x` and obtains the value associated with the key `k` in the JSON value `x`. If `k` is not a key in `x`, or if `x` is not a JSON object, then return nothing. You do not need to recursively search for keys. Assume that all keys in every JSON object are unique. Example runs follow.

```
ghci> -- re-use myJson
ghci> queryKey "name" myJson
Just "Bob"
ghci> queryKey "age" myJson
Nothing
ghci> queryKey "x" (L [O [("x", N 1)]])
Nothing
```

Tip: The `lookup` function is particularly useful here.

Tip: Although it looks like `queryKey "name" myJson` returned `Just` of some string, actually the string is a pretty-printed JSON string. Thus, `queryKey` should return a JSON value.

Question 5 (Getting Scores) [5 marks]. Now we want to obtain the score of a particular course from a student that is represented as a JSON value. In particular, a student is of the following JSON structure, where `AB1234X` and `CD5678Y` are course codes:

```
{ "courses" : {
    "AB1234X": {
      "score": ...,
      ...
    },
    "CD5678Y": {
      "score": ...,
      ...
    }, ...
}, ...}
```

For example, `myJson` from all the examples earlier is considered a valid student.

Then, write a function `getScore course_code student` that retrieves the score of the `course_code` course from `student`. The returned score is to be an `Int`; therefore, if the score of the course in the student JSON value is anything other than a number, or the student does not fit the correct JSON structure, then nothing should be returned. Of course, if `student` does not adhere to the structure of a student, then nothing should be returned as well. Use `do` notation. Example runs follow.

```
ghci> getScore "IT5100A" myJson
Just 47
ghci> getScore "IT5100B" myJson
Nothing
ghci> getScore "AB123" (N 45)
Nothing
```

– End of Assignment 2 –