NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5100A—Industry Readiness: Typed Functional Programming
Academic Year 2024/2025, Semester 1

PRACTICAL EXAMINATION
Question Booklet

12 Nov 2024 **Time allowed:** 1h 30min

INSTRUCTIONS TO CANDIDATES (*please read carefully*):

1. This question booklet comprises **10 questions** and **10 printed pages** including the cover page. All questions must be answered correctly to obtain the full score.
2. This is an **open-book**, **open-internet** assessment. You are allowed to refer to materials and access any resources (online or otherwise), except:
   - You are **NOT allowed** to communicate with anyone. You must force-close your email client, messaging applications like Telegram, WeChat, etc before starting the assessment.
   - You are **NOT allowed** to post on any forums, such as Reddit, StackOverflow etc. as this constitutes communicating with others.
   - You are **NOT allowed** to use ChatGPT, ollama or similar tools using Large Language Models (LLMs) to generate answers.
3. You are **allowed** to use any other application of your choice, such as using GHC, GHCI, MS Excel, other IDEs etc.
4. You are **NOT allowed** to rely on any external dependencies. You can only import modules defined in Haskell's `base` library (which includes `Prelude`).
5. You are given "template files" `Section1.hs`, `Section2.hs` and `Section3.hs` and their respective support files. No testing files are provided to you. You should run your own tests.
6. Submit **only** `Section1.hs`, `Section2.hs` and `Section3.hs` to Canvas. No additional files will be accepted. Each section is graded **independently**. If your solution for one section depends on your solution for another section, you should remove the dependency by copy-pasting the depended code into the relevant file.
7. All programs will be graded on GHC 9.4.8.
8. Your solutions will be graded on correctness and programming style. In particular, use of `do` notation is strongly encouraged where idiomatic.
9. **Mobile phones and smartwatches should be turned off.**
10. The total attainable score is **40**. The assessment constitutes **40%** of your overall grade.
11. You **cannot** communicate with anyone other than the invigilators throughout the assessment.
12. **You must attempt the assessment on your own**. The University takes a zero-tolerance approach towards plagiarism and cheating.

SAFE DIALS [10 marks]

Most people store their valuables in a safe. Traditionally, safes had a mechanical dial as their lock. To unlock the safe, one has to turn the dial in a sequence of clockwise and counterclockwise rotations at the correct points.

We define the current state of the safe dial as the angle from the "north" or "upwards" direction, in degrees. For example, if the dial is currently facing north or upwards, then the dial is at 0 degrees. If it is 1 degree counterclockwise from the north, then the dial is 359 degrees. All dial states are modulo 360, i.e. saying that the dial is at 720 degrees is the same as saying that it is at 0 degrees.

Each safe has a password, represented as a list of dial states, $[s_1, s_2, \ldots, s_n]$. To unlock the safe with this password, one first turns the dial **clockwise** until the dial is at exactly state s_1, then **counterclockwise** until the dial is at exactly state s_2, then **clockwise** again until the dial is at state s_3, and so on, **alternating between clockwise and counterclockwise rotations**, until the dial is at s_n. All rotations must be of nonzero amounts, therefore if the current state of the dial is already at the next state of the password, the user must rotate once for 360 degrees in the appropriate direction. Once the dial is at s_n, the safe is unlocked.

The safe is unlocked only when a series of dial rotations **ends** with an unlocking sequence. For example, if the dial is at 0 degrees and the password is $[1, 0, 1]$, the series of rotations $[1, -1, 1]$ (where 1 is a clockwise rotation of 1 degree and -1 is a counterclockwise rotation of 1 degree) unlocks the safe, the series of rotations $[1, -1, 1, -1]$ does **not** unlock the safe (the dial rotations do not cause the dial states to end with the password sequence), and the series of rotations $[1, -1, 1, -1, 1]$ unlocks the safe because after performing the first two rotations, the last three rotations is an unlocking sequence.

Our goal is, given the current state of the dial, the password of the dial, and a list of rotations, to determine whether the safe is unlocked successfully after performing all the rotations.

Question 1 (Rotations ADT) [2 marks]. Define a `Rotation` ADT that represents either a clockwise rotation (`CW`) or a counterclockwise rotation (`CCW`). The rotation amounts are in degrees, which are positive `Int`s between 1 and 360 inclusive. Additionally, allow `Rotation` to derive the `Show` and `Eq` classes. Example runs follow.

```
ghci> CW 123
CW 123
ghci> CCW 1
CCW 1
```

Question 2 (Rotations List) [2 marks]. Define a function `toRotations` that receives a list of `Int`s, where each integer represents a rotation of the dial, and returns the equivalent as a list of `Rotation`s. Take note of the following:

1. Positive integers represent clockwise rotations, and negative integers represent counterclockwise rotations
2. All 0 values should be ignored
3. Consecutive rotations in the same direction should be represented as a single rotation, i.e. rotating clockwise by 1 degree, then rotating clockwise by 2 degrees, is the same as just rotating clockwise by 3 degrees
4. All rotation values should be between 1 and 360, inclusive.

Example runs follow.

```
ghci> toRotations [1, 2, 3]
[CW 6]
ghci> toRotations [-360, 360, -361, 0, -2, 725, 0]
[CCW 360, CW 360, CCW 3, CW 5]
```

Question 3 (Unlock Sequence) [3 marks]. Define a function `unlockSequence` that receives the current state of the dial as an `Int` between 0 and 359 (inclusive), the password of the safe as a list of `Int`s (each being between 0 and 359 as well), and returns a list of the fewest rotations needed to unlock the safe.

1. The length of the list you return must be the same as the length of the password.
2. Remember that the first rotation needed to unlock the safe is **always** a clockwise rotation.
3. If the password is empty, the safe is always unlocked, and no rotations are required to unlock the safe.

Example runs follow.

```
ghci> unlockSequence 0 [1, 0, 1, 0]
[CW 1, CCW 1, CW 1, CCW 1]
ghci> unlockSequence 0 [0, 0, 0]
[CW 360, CCW 360, CW 360]
ghci> unlockSequence 359 []
[]
```

Question 4 (Will It Open?) [3 marks]. It is finally time to solve the problem. Write a function `willOpen` that receives:

1. the current state of the dial as an `Int` between 0 and 359 inclusive,
2. a safe password represented as a list of `Int`s (each between 0 and 359 inclusive),
3. a list of rotations represented as a list of `Int`s where negative numbers represent counterclockwise rotations and positive numbers represent clockwise rotations, and 0 means "no rotation",

and returns `True` if the safe is unlocked after completing **all** the specified rotations, `False` otherwise. Example runs follow.

```
ghci> willOpen 0 [] [1] -- empty password
True
ghci> willOpen 359 [0] [1]
True
ghci> willOpen 1 [0] [-1] -- unlocking sequence always start with clockwise rotation
False
ghci> willOpen 2 [1, 0, 1] [-2, 1, -1, 1]
-- first rotation brings dial to 0, next three rotations unlock safe
True
ghci> willOpen 2 [1, 0, 1] [-2, 1, -1, 1, -360]
-- second-last rotation unlocks safe, but last counterclockwise rotation
-- re-locks safe
False
ghci> willOpen 2 [1, 0, 1] [-2, 1, -1, 2, 719]
-- second-last rotation and last rotation is the same as one clockwise rotation of
-- 1 degree
True
```

Tip: Use `toRotations` and `unlockSequence` to help you!

SIMPLE SHOP [15 marks]

We shall work with a simple shopping system consisting of users and products. Each user has a username, account balance and inventory. Each product has a product name, cost and current quantity owned by the shop. Our database consists very simply of a list of users and a list of products.

All the types have been given to you in `Section2Support.hs`. Do not amend them. Importantly, these are just type *aliases*, therefore, for example, a `Username` is *exactly the same* as a `String`, and you can (and should) use `Username`s as you would for regular `String`s.

```haskell
type Balance = Rational
type Username = String
type Inventory = [(ProductName, Quantity)]
type User = (Username, (Balance, Inventory))

type ProductName = String
type Quantity = Int
type Cost = Rational
type Product = (ProductName, (Cost, Quantity))

type Database = ([User], [Product])
```

An example database has also been provided to you. Do not amend it.

```haskell
exampleDb :: Database
exampleDb = (example_users, example_products) where
  example_users = [
        ("Alice", (100, [("Bow", 1), ("Arrow", 2)]))
      , ("Bob",   (150, [("Arrow", 1)]))
    ]
  example_products = [
        ("Bow", (50, 5))
      , ("Arrow", (10, 10))
    ]
```

Our goal in this section is to perform purchase transactions safely using the railway pattern. Assume throughout that all users and products have unique names.

In this section, use `do` notation and `guard` (from `Control.Monad`) where possible.

Question 5 (Polymorphic Updates) [3 marks]. Notice that `[User]` and `[Product]` are essentially lists of key-value pairs, where keys are usernames or product names respectively, and their corresponding values are information about that user or product respectively. This makes it easy for us to look up a user or product information using the `lookup` function in `Prelude`. However, updating information about users or products in `[User]`s and `[Product]`s is cumbersome.

Write a function `update` that receives a key and a value and a list of key-value pairs. If the key does not exist in the list, `update` prepends the key-value pair onto the list. If the key already exists in the list, then its corresponding value is updated with the new value. In effect, `update k v ls` in Haskell now behaves similarly to `ls[k] = v` in Python, where `ls` is a dictionary. Example runs follow.

```
ghci> update "a" 1 []
[("a", 1)]
ghci> update "b" 2 (update "a" 1 [])
[("b", 2), ("a", 1)]
ghci> update "a" 3 (update "b" 2 (update "a" 1 []))
[("b", 2), ("a", 3)]
ghci> update "Alice" (0, []) (fst exampleDb)
[("Alice", (0 % 1, [])), ("Bob", (150 % 1, [("Arrow", 1)]))]
```

Question 6 (Deducting User Balances) [3 marks]. Write a function `deductUserBalance` that receives a `Username` `username`, a `Balance` `balance`, and a `Database`, and returns a new database where `username`'s account balance has been deducted by `balance`. The transaction should fail if any of the following are true:

1. `balance` is not strictly positive
2. `username` is not a user in the database
3. the user has insufficient account balance

Because the transaction could fail, we shall work with the `Maybe` monad. Example runs follow.

```
ghci> deductUserBalance "Alice" 10 exampleDb
Just ([("Alice",(90 % 1,[("Bow",1),("Arrow",2)])),("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Bow",(50 % 1,5)),("Arrow",(10 % 1,10))])
ghci> deductUserBalance "Alice" 101 exampleDb
Nothing -- insufficient account balance
ghci> deductUserBalance "Alice" 0 exampleDb
Nothing -- can't deduct balance by 0
ghci> deductUserBalance "Charlie" 1 exampleDb
Nothing -- Charlie does not exist
```

Question 7 (Deducting Product Quantities) [3 marks]. Write a function `deductProductQuantity` that receives a **`ProductName`** `n`, **`Quantity`** `q` and the **`Database`**, and returns the new database where the quantity of `n` in the database has been reduced by `q`. The transaction should fail if any of the following are true:

1. `q` is not strictly positive
2. `n` is not a product in the database
3. the quantity of `n` in the database is strictly less than `q`

Once a product's quantity in the database reaches 0, the entry is removed from the database. Example runs follow:

```
ghci> deductProductQuantity "Bow" 4 exampleDb
Just ([("Alice",(100 % 1,[("Bow",1),("Arrow",2)])),("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Bow",(50 % 1,1)),("Arrow",(10 % 1,10))])
ghci> deductProductQuantity "Bow" 5 exampleDb
Just ([("Alice",(100 % 1,[("Bow",1),("Arrow",2)])),("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Arrow",(10 % 1,10))]) -- Bows no longer in shop
ghci> deductProductQuantity "Bow" 6 exampleDb
Nothing -- not enough Bows in shop
ghci> deductProductQuantity "Bow" 0 exampleDb
Nothing -- can't deduct 0
ghci> deductProductQuantity "Cake" 1 exampleDb
Nothing -- Cakes do not exist in shop
```

Question 8 (Adding Products to User Inventories) [3 marks]. Write a function `addProductToUser` that receives a **`ProductName`** `p`, a **`Quantity`** `q`, a **`Username`** `u`, and a **`Database`**, and returns the new state of the database where `u`'s inventory has been added with `q` more of product `p`. The transaction should fail if either `q` is not strictly positive or if `u` is not a user in the database. Example runs follow.

```
ghci> addProductToUser "Bow" 10 "Alice" exampleDb
Just ([("Alice",(100 % 1,[("Bow",11),("Arrow",2)])),("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Bow",(50 % 1,5)),("Arrow",(10 % 1,10))])
ghci> addProductToUser "Cake" 10 "Alice" exampleDb
Just ([("Alice",(100 % 1,[("Cake",10),("Bow",1),("Arrow",2)]))
      ,("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Bow",(50 % 1,5)),("Arrow",(10 % 1,10))])
ghci> addProductToUser "Bow" 0 "Alice" exampleDb
Nothing -- Can't add 0 quantity
ghci> addProductToUser "Bow" 1 "Charlie" exampleDb
Nothing -- Charlie does not exist as a user
```

Question 9 (Purchasing Items) [3 marks]. Now it is finally time to solve our problem. Write a function `userPurchaseProduct` that receives a **`Username`** `u`, a **`ProductName`** `p`, a **`Quantity`** `q` and a **`Database`**, and returns the new state of the database where the user `u` purchases `q` amounts of product `p`. In other words, several things should happen in this transaction:

1. `u`'s account balance should be deducted by the total cost of the purchase
2. The quantity of the product in the database should be deducted by `q`
3. `u` should have an additional `q` amount of `p`

If any of the three things above fail, the entire transaction should fail as well. Example runs follow.

```
ghci> userPurchaseProduct "Alice" "Bow" 2 exampleDb
Just ([("Alice",(0 % 1,[("Bow",3),("Arrow",2)])),
       ("Bob",(150 % 1,[("Arrow",1)]))]
     ,[("Bow",(50 % 1,3)),("Arrow",(10 % 1,10))])
ghci> userPurchaseProduct "Bob" "Arrow" 10 exampleDb
Just ([("Alice",(100 % 1,[("Bow",1),("Arrow",2)])),
       ("Bob",(50 % 1,[("Arrow",11)]))]
     ,[("Bow",(50 % 1,5))])
ghci> userPurchaseProduct "Bob" "Arrow" 11 exampleDb
Nothing -- not enough arrows in products
ghci> userPurchaseProduct "Alice" "Bow" 3 exampleDb
Nothing -- Alice does not have enough money
ghci> userPurchaseProduct "Alice" "Bow" 0 exampleDb
Nothing -- cannot purchase 0 quantity
ghci> userPurchaseProduct "Charlie" "Bow" 1 exampleDb
Nothing -- Charlie does not exist
ghci> userPurchaseProduct "Alice" "Cake" 1 exampleDb
Nothing -- Cake does not exist
```

Tip: Use the three functions from earlier to make your life easier. In addition, you can use the `fromIntegral` function which converts an integer to a rational number.

GOING SOMEWHERE [15 marks]

In this section we are going to write a simple program that receives, as console input, a map file, a source city, and a destination city, and prints to the console the total number of ways to go from the source to the destination.

A map file is a nonempty file with several lines; each line consists of several words delimited by spaces. Each word represents the name of a city. A line in the map file $w_1 \; w_2 \; \dots \; w_n$ denotes that there is a road going from w_1 to w_2, a road going from w_1 to w_3, ..., and a road going from w_1 to w_n. All roads are one-way. In essence, the map is an *adjacency list*. Assume that:

1. the map never has duplicate entries
2. the map will never have cycles

For example, given the following file `map0.txt`:

```
A B C
B D
```

`map0.txt` says that there are roads going from `A` to `B`, `A` to `C`, and `B` to `D`. We can also go from `A` to `D`, by taking the route `A` to `B` to `D`.

Question 10 (Going Somewhere) [15 marks]. You are on your own in this section. Complete `Section3.hs` so that it defines a program that reads three lines of input from the user:

1. The file name of the map file
2. The name of the source city as a single word
3. The name of the destination city as a single word

and prints the number of ways to get from the source node to the destination node. Assume that the map file always exists, and that the user will only enter city names as single words without spaces.

Your implementation should be fast enough to handle a map with 100 cities within one second. Use memoization with the provided **`State`** monad to help you.

Tip:
- The functions `readFile`, `words` and `lines` from Haskell's **`Prelude`** will be useful to you.
- The **`State`** monad given to you in `Section3Support.hs` has three functions, `put`, `get` and `modify`. All of these are defined exactly as in the course notes.
- Some example map files have been provided to you. Use them for your own testing.

Let us show some example program executions with the following map files. First, we have `map0.txt` from before:

```
A B C
B D
```

Next, we have `map1.txt`:

```
A B C D
B C E
C D E
```

Example executions of the program are as follows. Lines shown with a prefix `>` are inputs to the terminal/console, and all user inputs are in **bold**.

```
> ghc Section3.hs
[1 of 2] Compiling Main (Section3.hs, Section3.o)
[2 of 2] Linking Section3 [Objects changed]
> ./Section3
map0.txt
A
D
1
> ./Section3
map0.txt
B
C
0
> ./Section3
map0.txt
PasirRis
Tampines
0
> ./Section3
map1.txt
A
E
3
```

Explanations for these test cases as follows:

1. To go from `A` to `D`, we travel from `A` to `B` to `D`. Thus, there is only one path.
2. There are no paths that allow us to travel from `B` to `C`. Recall that roads are one-way.
3. Both `PasirRis` and `Tampines` do not exist in the map file.
4. There are three paths to take:
   (a) `A` to `B` to `C` to `E`
   (b) `A` to `B` to `E`
   (c) `A` to `C` to `E`

– End of Practical Examination –