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

# HASKELL BASICS, TYPES

**Due:** 22 Oct 2024 **Score**: [20 marks] (20%)

## PREAMBLE

In this assignment, we are going to re-create the bank accounts problem as seen in Question 9 of the exercises in Chapter 2 of the course notes[1], in Haskell. The goal of this assignment is to provide some exercises for you to create FP-style data types and functions in Haskell. You will get the most out of this assignment if you attempt Question 9 of Chapter 2's exercises in the course notes first. This is because once you have learnt how to do the exercise in FP-style code in Python, it will be much easier to do the same in Haskell. You may freely incorporate ideas from the provided solutions in your assignment submission.

The problem this assignment considers is a simulation of a simple banking system of bank accounts. There are several kinds of bank accounts which behave differently on certain operations, and our financial system receives operations that act on these accounts.

This assignment is worth **20 marks** which constitutes **20%** of your overall grade. It is due on **22 Oct 2024**. 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.

This assignment is scored solely on correctness. Ensure that your submission is compilable; uncompilable code will be penalized heavily. Additionally, 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 (`BankAccounts.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 `assignment1`. Compile `Main.hs` and run it:

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

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

[1] The exercise can be found in `https://yongqi.foo/it5100a-notes/types/sections/exercises.html`.

## BANK ACCOUNTS (HASKELL) [5 marks]

**Question 1 (Data Type)** [1 mark]. Create an Algebraic Data Type (ADT) called `BankAccount` that represents two kinds of bank accounts:

1. Normal bank accounts
2. Minimal bank accounts

Both kinds of accounts have an ID (`String`), an account balance (`Rational`) and an interest rate (`Rational`).

Example runs follow:

```
ghci> NormalAccount "abc" 1000 0.01
NormalAccount "abc" (1000 % 1) (1 % 100)
ghci> MinimalAccount "abc" 1000 0.01
MinimalAccount "abc" (1000 % 1) (1 % 100)
```

**Warn**: Ensure that you remove any `deriving` clauses for all ADTs you define in this assignment before testing and submission. In other words, **do not** write the following `deriving Show` statement:

```
data BankAccount =  ...
  deriving Show
```

The test suites will derive the right instances for you. If you do not know what `deriving` does, at least for this assignment, that is just fine.

**Question 2 (Accessor Functions)** [1 mark]. Create some accessor functions to access the fields of the bank accounts:

1. `accountId` retrieves a **`BankAccount`**'s account ID
2. `balance` retrieves a **`BankAccount`**'s account balance
3. `interest` retrieves a **`BankAccount`**'s interest rate

Example runs follow:

```
ghci> x = NormalAccount "abc" 1000 0.01
ghci> y = MinimalAccount "bcd" 2000 0.02
ghci> accountId x
"abc"
ghci> accountId y
"bcd"
ghci> balance x
1000 % 1
ghci> balance y
2000 % 1
ghci> interest x
1 % 100
ghci> interest y
1 % 50
```

**Question 3 (Basic Features)** [1 mark]. Now implement the following basic features:

1. `deposit` will deposit money into a bank account, giving the new state of the bank account with the money deposited
2. `deduct` deducts money from the bank account, returning a pair of
   (a) a **`Bool`** value describing whether the deduction succeeded
   (b) the new state of the bank account with the money (potentially) deducted

   Do not deduct money from the bank account if the amount to deduct exceeds the bank balance of the account.

Example runs follow:

```
ghci> x = NormalAccount "abc" 1000 0.01
ghci> y = MinimalAccount "bcd" 2000 0.02
ghci> deposit 1000 x
NormalAccount "abc" (2000 % 1) (1 % 100)
ghci> deduct 1000 x
(True,NormalAccount "abc" (0 % 1) (1 % 100))
ghci> deduct 2001 y
(False,MinimalAccount "bcd" (2000 % 1) (1 % 50))
```

**Question 4 (Advanced Features)** [2 marks]. Now, implement the following 'advanced' features:

1. `compound` compounds the interest of a bank account. Given a bank account with balance $b$ and interest rate $i$, its new account balance after compounding will be $b(1 + i)$. However, minimal accounts will will have an administrative fee of $20 if its balance is (strictly) below $1000. This fee deduction happens **before** compounding. Bank balances will never go below $0, so e.g. if a minimal account has $19 in the bank, then after compounding, its balance will be $0, not -$1.
2. `transfer` performs a bank transfer. It receives
   (a) a transaction amount
   (b) a credit bank account (account where funds will come from)
   (c) a debit bank account (account where funds will be transferred to)
   The result of the bank transfer is a triplet (tuple of three elements) containing a boolean describing whether the transaction was successful, and the new states of the credit and debit accounts. Importantly, if the credit account does not have enough funds to make the transfer, the transaction should not happen.

Example runs follow:

```
ghci> x = NormalAccount "abc" 1000 0.01
ghci> y = MinimalAccount "bcd" 2000 0.02
ghci> z = MinimalAccount "def" 999 0.01
ghci> w = MinimalAccount "xyz" 19 0.01
ghci> compound x
NormalAccount "abc" (1010 % 1) (1 % 100)
ghci> compound (compound x)
NormalAccount "abc" (10201 % 10) (1 % 100)
ghci> compound y
MinimalAccount "bcd" (2040 % 1) (1 % 50)
ghci> compound z
MinimalAccount "def" (98879 % 100) (1 % 100)
ghci> compound w
MinimalAccount "xyz" (0 % 1) (1 % 100)
ghci> transfer 2000 x y
(False,NormalAccount "abc" (1000 % 1) (1 % 100),
  MinimalAccount "bcd" (2000 % 1) (1 % 50))
ghci> transfer 2000 y x
(True,MinimalAccount "bcd" (0 % 1) (1 % 50),
  NormalAccount "abc" (3000 % 1) (1 % 100))
```

## TABLES [7 marks]

Normally in Python if we were to represent a collection of bank accounts, we would use a dictionary. However, dictionaries are not built into Haskell's Prelude. We could use a list of key-value pairs as tuples, but that may present issues of its own.

Instead, what we shall do is to modify our Binary Search Tree (BST) implementation from Question 6 of the exercises in Chapter 2 of the course notes to act as a 'better' implementation for a table. This time, instead of our trees only containing values, we shall create a new kind of BST that stores both keys and values. The keys are unique and each correspond to a value; searching down the tree involves comparisons between the keys.

**Tip**: The implementation of our "dictionary" is very similar to our BST implementation in the exercises.

**Question 5 (BST Map Data Type)** [1 mark]. Define a new ADT `BSTMap` that defines a binary search tree map, containing key-value pairs such that keys are unique and ordered. Your BST should have two constructors:

1. `Empty`: this is the empty tree
2. `Node`: This node contains the left subtree, the key at the node, the value corresponding to this key, and the right subtree.

Example runs follow:

```
ghci> Empty
Empty
ghci> Node Empty 1 2 Empty -- {1: 2}
Node Empty 1 2 Empty
ghci> Node (Node Empty 0 2 Empty) 1 2 Empty -- {0: 2, 1: 2}
Node (Node Empty 0 2 Empty) 1 2 Empty
```

**Question 6 (Putting elements in BSTMap)** [2 marks]. Define a function `put` that puts a key-value pair in a tree. If the key is not already in the map, return the new state of the map with the key-value pair added. Otherwise, the new state of the map should have the previous value replaced by the new value.

Because adding a key-value pair of a binary search tree requires comparison of keys, for this to work, you will need the `Ord k ⇒` constraint added to your function's type signature. For example, if your key and value type parameters are `k` and `v` respectively, the `put` function's type should be `put :: Ord k ⇒ k -> v -> ....`

Example runs follow:

```
ghci> put 1 2 Empty
Node Empty 1 2 Empty
ghci> put 2 3 $ put 1 2 Empty
Node Empty 1 2 (Node Empty 2 3 Empty)
ghci> put 0 2 $ put 2 3 $ put 1 2 Empty
Node (Node Empty 0 2 Empty) 1 2 (Node Empty 2 3 Empty)
ghci> put 1 4 $ put 0 2 $ put 2 3 $ put 1 2 Empty
Node (Node Empty 0 2 Empty) 1 4 (Node Empty 2 3 Empty)
```

**Question 7 (Key Membership)** [1 mark]. Define a function `in'` that receives a key and a `BSTMap`, and determines whether the key is present in the `BSTMap`. Example runs follow:

```
ghci> mp = put 1 4 $ put 0 2 $ put 2 3 (put 1 2 Empty)
ghci> 0 `in'` mp
True
ghci> 3 `in'` mp
False
```

**Question 8 (Obtaining Values)** [1 mark]. Define a function `get` that receives a key and a `BSTMap`, and returns the value associated with the key in the tree. Example runs folow:

**Tip**: If the key is not present in the tree, return `undefined`. In future lectures we will learn how to better deal with exceptions or "errors".

```
ghci> mp = put 0 2 $ put 2 3 $ put 1 2 Empty
ghci> get 0 mp
2
ghci> get 2 mp
3
ghci> get 1 mp
2
```

**Question 9 (Obtaining All Values)** [2 marks]. We shall write a function to obtain all the values in our `BSTMap`, just like `{0: 1, 2: 3}.values()` in Python. Define the function `values` which obtains all the values of a `BSTMap`. Since we are using a BST as our map implementation, `values` should obtain the pre-order of our BST map, giving us a list of values in *ascending order of their associated keys.*

```
ghci> mp = put 0 3 $ put 1 2 $ put 2 1 Empty
ghci> values mp
[3,2,1]
ghci> mp2 = put 1 10 $ put (-1) 100 $ put 0 1000 Empty
ghci> values mp2
[100,1000,10]
```

## OPERATING ON BANK ACCOUNTS [8 marks]

Now that we have relatively quick lookups of bank accounts in a tree, we can perform operations on the bank accounts in the tree. There are two kinds of operations we will consider for this problem: compounding all bank accounts, and performing transfers between two accounts in the map.

Some of these operations will fail, so when processing the operations on the accounts in a map, we will also return a list of boolean values describing whether each operation has succeeded.

**Question 10 (Operations Data Type)** [1 mark]. Create a new ADT called `Op` with two kinds of operations:

1. `Transfer` has a transfer amount, and credit bank account ID, and a debit bank account ID. This represents the operation where we are transferring the transfer amount from the credit account to the debit account.
2. `Compound`. This just tells the processor to compound all the bank accounts in the map. There should be no fields in this constructor.

**Question 11 (Processing One Operation)** [2 marks]. Write a function `processOne` that receives an operation and a `BSTMap` of bank accounts (keys are bank account IDs, and values are the corresponding bank accounts), and performs the operation on the bank accounts in the `BSTMap`. As a result, the function returns a pair containing:

1. A boolean value to describe whether the operation has succeeded
2. The resulting `BSTMap` containing the updated bank accounts after the operations have been processed.

There are several ways in which a `Transfer` operation may fail:

1. If either account IDs do not exist in the tree, the transfer will fail
2. If the credit account does not have sufficient funds, the transfer will fail
3. Otherwise, the transfer should proceed as per normal

Example runs follow:

```
ghci> alice = NormalAccount "alice" 1000 0.1
ghci> bob   = MinimalAccount "bob" 999 0.1
ghci> mp    = put "alice" alice (put "bob" bob Empty)
ghci> c     = Compound
ghci> t1    = Transfer 1000 "alice" "bob"
ghci> t2    = Transfer 1000 "bob" "alice"
ghci> processOne c mp
(True,Node
  (Node Empty "alice" (NormalAccount "alice" (1100 % 1) (1 % 10)) Empty)
  "bob" (MinimalAccount "bob" (10769 % 10) (1 % 10)) Empty)
ghci> processOne t1 mp
(True,Node
  (Node Empty "alice" (NormalAccount "alice" (0 % 1) (1 % 10)) Empty)
  "bob" (MinimalAccount "bob" (1999 % 1) (1 % 10)) Empty)
ghci> processOne t2 mp
(False,Node
  (Node Empty "alice" (NormalAccount "alice" (1000 % 1) (1 % 10)) Empty)
  "bob" (MinimalAccount "bob" (999 % 1) (1 % 10)) Empty)
```

**Question 12 (Processing All Operations)** [2 marks]. Define a function `processAll` that receives a list of operations and a `BSTMap` of bank accounts (the keys are bank account IDs, and the values are bank accounts). As a result, the function returns a pair containing:

1. A list of booleans where the $i^{\text{th}}$ boolean value describes whether the $i^{\text{th}}$ operation has succeeded
2. The resulting `BSTMap` containing the updated bank accounts after all the operations have been processed.

Example runs follow:

```
ghci> alice = NormalAccount "alice" 1000 0.1
ghci> bob   = MinimalAccount "bob" 999 0.1
ghci> mp    = put "alice" alice (put "bob" bob Empty)
ghci> c     = Compound
ghci> t1    = Transfer 1000 "alice" "bob"
ghci> t2    = Transfer 1000 "bob" "alice"
ghci> processAll [t2,c,t2,t1] mp
([False,True,True,True],
  Node (Node Empty "alice" (NormalAccount "alice" (1100 % 1) (1 % 10)) Empty)
  "bob" (MinimalAccount "bob" (10769 % 10) (1 % 10)) Empty)
```

**Question 13 (Polymorphic Processing)** [3 marks]. If you were careful with your implementation of `processAll`, you might notice that if we `processOne` were a parameter of the `processAll` function, then nothing about the implementation of `processAll` depends on the types like `Op`, `BSTMap` or `Bool`. Thus we should make this function polymorphic!

Our goal is to write a polymorphic function `process` that can process any list (like a list of operations) over a state (like a `BSTMap`) and produces a result list and an updated state after performing stateful processing over the list. It should be defined such that `process processOne` should be the exact same function as `processAll` as you have defined earlier:

```
ghci> alice = NormalAccount "alice" 1000 0.1
ghci> bob   = MinimalAccount "bob" 999 0.1
ghci> mp    = put "alice" alice (put "bob" bob Empty)
ghci> c     = Compound
ghci> t1    = Transfer 1000 "alice" "bob"
ghci> t2    = Transfer 1000 "bob" "alice"
ghci> process processOne [t2,c,t2,t1] mp
([False,True,True,True],
  Node (Node Empty "alice" (NormalAccount "alice" (1100 % 1) (1 % 10)) Empty)
  "bob" (MinimalAccount "bob" (10769 % 10) (1 % 10)) Empty)
```

Furthermore, the best part of this polymorphic function is that it can be used in any situation where we need this stateful accumulation over a list. For example, we can define a function that tests if a number $n$ is co-prime to a list of other numbers, and if it is indeed co-prime to all of the input numbers, add $n$ to the state list:

```haskell
divides a b = b `mod` a == 0
gatherPrime :: Int -> [Int] -> (Bool, [Int])
gatherPrime n [] = (True, [n])
gatherPrime n ls
  | any (`divides` n) ls = (False, ls)
  | otherwise     = (True, ls ++ [n])
```

Example uses of this follow:

```
ghci> gatherPrime 2 []
(True, [2])
ghci> gatherPrime 3 [2]
(True, [2,3])
ghci> gatherPrime 4 [2,3]
(False,[2,3])
```

This way, we can use `process` to generate prime numbers and do primality testing!

```
ghci> primes n = process gatherPrime [2..n] []
ghci> primes 10
([True,True,False,True,False,True,False,False,False],[2,3,5,7])
ghci> primes 30
([True,True,False,True,False,True,False,False,False, -- 2 to 10
  True,False,True,False,False,False,True,False,True,False, -- 11 to 20
  False,False,True,False,False,False,False,False,True,False], -- 21 to 30
  [2,3,5,7,11,13,17,19,23,29])
```

Proceed to define the `process` function. Example runs are as above.

**Tip**: Recall that we want to define `process` such that `processAll` can be defined as `process processOne`. As such, ensure that the type of `process processOne` has the same type as `processAll`. To make `process` polymorphic, think of changing `Op`, `BSTMap String BankAccount` and `Bool` into type variables.

**– End of Assignment 1 –**