Exam 2021-01

This was the online exam from January 2021. Students were permitted to use GHCi, notes etc., so the exam questions are a bit different to the usual hall-written exams.

Note that the format was not the usual one either, as it was 2 hours for part 1 + 2 hours for part 2.

NOTE: please do not leave the zoom room between part 1 and part 2. If you do not wish to take part 2 then you may leave but please notify the exam admin via the chat as instructed.

There are two parts to this exam. This is Part I, two hours long, from 8.30 to 10.25 (+ 10 mins extra to submit). There will be a 15 minute break between part I and part II.

You must pass Part I to get at pass on the course. Part I has a total of 24 points; 14 points guarantees a pass on the exam (3/G).

If you pass part I, then part II can contribute towards getting a grade 4 or 5 (VG).

Note the following

- *For each part Question, if your solution consists of more than a few lines of Haskell code, use your common sense to decide whether to include a short comment to explain your solution.*
- *You can use any of the standard Haskell functions in the following modules:*
  - *Prelude, Data.List, Data.Maybe, Data.Char, Test.QuickCheck*
- *Full points are given to solutions which are short, elegant, and correct. Fewer points may be given to solutions which are unnecessarily complicated or unstructured.*
- *You are encouraged to use the solution to an earlier part of a Question to help solve a later part — even if you did not succeed in solving the earlier part.*
- *Although you may use a Haskell compiler to write your code, the code will be graded in the usual way as for hand-written exams: we will not usually run your code, so small errors will be ignored. Your code does not have to compile. "Small errors" are usually involving syntax, indexing off-by-one or other minor points. "Small" does not refer to the number of characters needed to fix your solution (i.e. a large error may be fixable with a small number of symbols!)*
- *That said, apply the usual convention for formatting your code nicely. Please separate each question with a comment "line" -----.... Do not include unused code, and **do not submit multiple answers to the same question or part-question**, as we will only grade the first version that we read.*
- *By 10.25 please upload a single **.hs** file here in Canvas containing your solutions to Part I. You have until 10.35 to upload in case you have problems, but we strongly recommend that you upload by 10.25.*

1. (7 points) A web colour (colours used in displaying web pages) can be specified in several ways, including (i) an RGB format which is a triple of three numbers in the range 0 - 255, written e.g. "**rgb(255,0,0)**" in a web style sheet, (ii) an HSL colour space value, which is specified as an angle (an integer in the range 0-359), and two percentages (expressed as whole numbers in the range 0-100). An example of this might be the color red which can be written as "**hsl(0,100%,50%)**" in a web document.

   (a) Define a data type `WebColour` to represent a web colour so that both format (i) or format (ii) can be used.

   (b) Based on the description above, give the data type invariant for web colours `isValidWebColour :: WebColour -> Bool`.

   (c) Make your definition a member of the class Show so that your web colours are displayed as a web browser would expect to see them (i.e. as in the examples given above).

2. (2 points) Define a function

```
allProps :: [a -> Bool] -> a -> Bool
```

   which given a list of properties (boolean valued functions), and a value, determines whether all of the properties in the list are true for that value. Your definition of allProps should be recursive. Your definition should satisfy the following:

```
prop_allProps a = allProps [even,(>3),(<10)] 4
               && allProps [] a
```

3. (2 points) Define `allProps'`, which should be equivalent to `allProps`, but defined by using standard functions from the Prelude and/or list comprehensions.

4. (3 points) The following questions are about using a list of lists to represent a square grid of elements. For example a Sudoku puzzle a 9 x 9 square, can be represented as a list of nine lists, each with nine elements. We say that this is a square of *dimension* 9.

   Given a list of lists, define a function

```
dimension :: [[a]] -> Maybe Int
```

   such that `dimension xss` returns `Just n` if `xss` is a list of n lists, each of which has `n` elements, and returns `Nothing` otherwise. You may assume that the empty list has dimension `0`.

5. (3 points) Given the following type intended to represent squares of Integers

```
data IntSquare = Square [[Int]]
  deriving (Eq,Show)
```

give a suitable `Arbitrary` instance for `IntSquare` so that the following test can be run (and passed) by quickCheck:

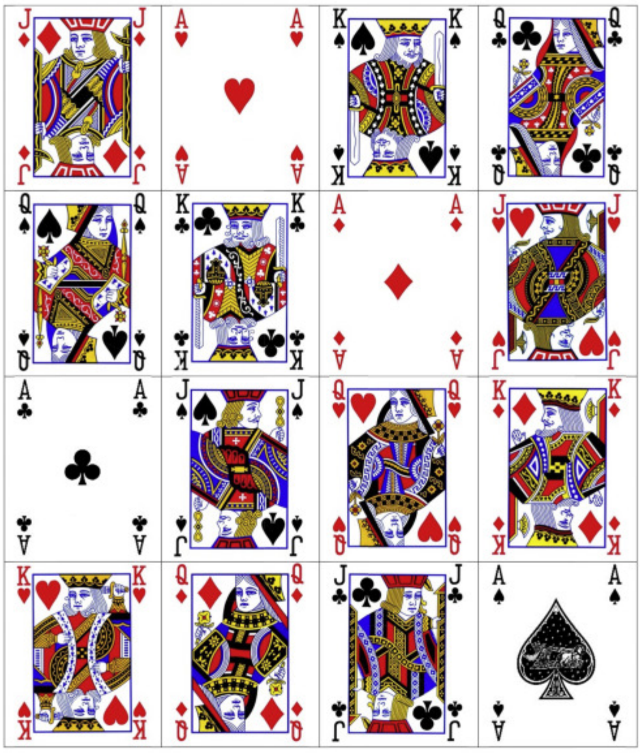
```
prop_Square (Square s) = isJust (dimension s)
```

6. (2 points) A square of dimension n has two main diagonals, each with n elements. There are many ways to compute the list of elements on one of the main diagonals. Dave's favourite is `\xss -> zipWith (!!) xss [0..]`. In this question you are to complete the following incomplete definition to compute a diagonal by providing a recursive definition of the helper function `diagHelper`. Which of the two diagonals you choose, and in which order the elements appear is up to you. You can assume that the input is a square.

```
diag xss = diagHelper (length xss)   -- do not modify this line
  where diagHelper ...
```

Hints: Remember that you can use `xss` in the definition of `diagHelper` as it is a local definition.

7. (5 points) A royal square is a square of cards of dimension 4, in which each row, column and major diagonal contains one card of each suit, and one card of each rank from Jack to Ace. The following picture shows an example of a Royal square:



Swapping any two cards in this square will result in something which is *not* a royal square. Given the following definitions:

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
  deriving (Eq, Ord, Show)

data Suit = Hearts | Spades | Diamonds | Clubs
        deriving (Eq, Enum, Bounded, Show)
        -- permits [Hearts..Clubs] etc
```

```
data Card = Card Rank Suit
  deriving (Eq,Show)
```

Define a function

```
isRoyal :: [[Card]] -> Bool
```

That checks whether the argument is a royal square. You may not make any assumptions about the argument.

Part 2

1. **(2 points)** In part 1 you wrote a function `allProps :: [a -> Bool] -> a -> Bool`. Give a definition of `allProps` using `foldr`, `(&&)` and no other Prelude functions (you may use local helper functions).

You can give the same definition if you already used `foldr` for part 1.

2. **(8 points)** Consider the following language for arithmetic expressions with variables and a square root operator.

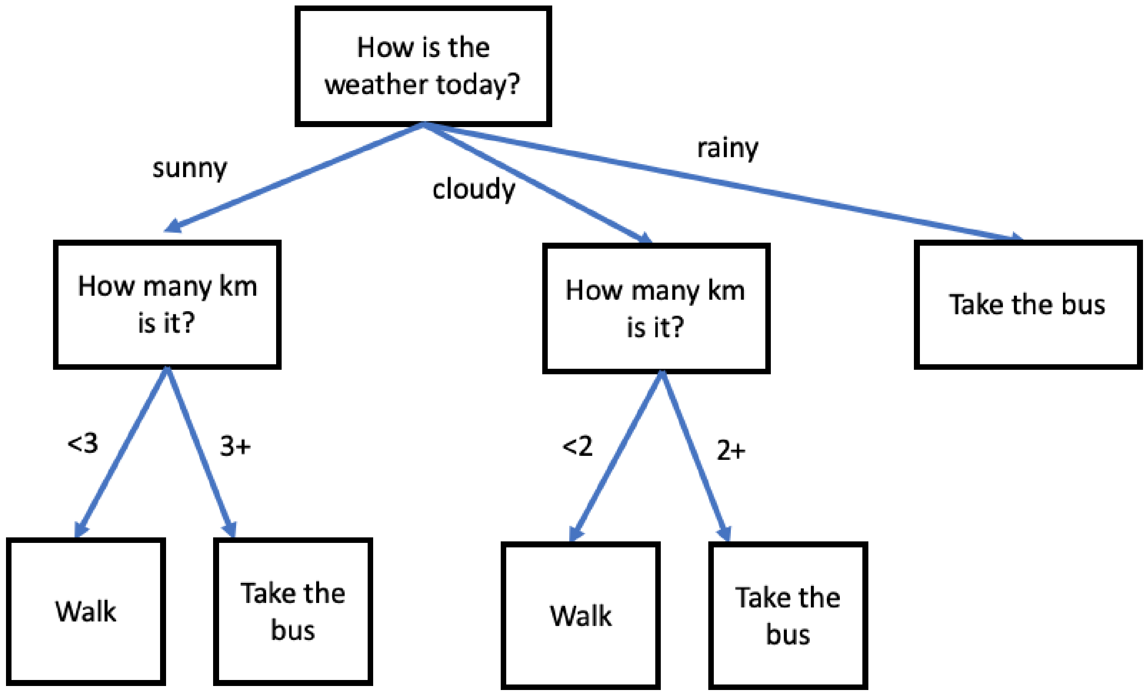
```
type VarName = String
data Exp = Var VarName | Add Exp Exp | SquareRoot Exp | Num Float
   deriving Show
```

Define an evaluator

`eval :: [(VarName,Float)] -> Exp -> Maybe Float`

which takes a table of the values for variables, and an expression, and evaluates the expression. It should give `Nothing` whenever there is an undefined variable in the expression, or if there is a square root of a negative number. For full points make use of the fact that `Maybe` is in class `Monad` to simplify your code a little (max 6 points otherwise).

3. **(8 points)** The following picture represents a decision tree. In this example the tree gives advice on how you should get to work (when there is no pandemic).



Define a parameterised data type `DTree q a` to represent a tree with questions (the things in the boxes) of type q and responses (the labels on the branches) of type a. The number of branches from any node in the tree should ***not*** be limited to a maximum of 3 (as in this example) but can be any non negative number.

Define a program `runDTree :: DTree String String -> IO()` which interactively "runs" the decision tree by asking the user questions, as in the following example using `transportDT`, a representation of the DTree for the diagram above:

```
*Main> :t transportDT
transportDT :: DTree String String
*Main> runDTree transportDT
How is the weather? ["sunny","cloudy","rainy"]
sunny
How many km is it? ["<3","3+"]
3
Answer not recognised
How many km is it? ["<3","3+"]
3+
Take the bus
*Main>
```

(Here the text in bold italics was typed by the user)

4. (6 points) A *Bag* or *Multiset* is like a set, but an element can occur multiple times. Just like a set, there is no notion of the order of elements in a Bag.

In this question you are to define an abstract data type for bags, in the form of a module (if you wish to compile your solution you need to either put it in a separate file temporarily or comment out the module header)

Your solution should use the following internal representation of a Bag:

`data Bag a = Bag (a -> Integer)`

where the function is used to tell you how many copies of the element there are in the Bag. So for example if `Bag f` is a bag of strings which contains three copies of `"spam"`, then `f "spam"` would give the value `3`. Using functions to represent Bags is not very efficient, but it makes some things very easy (e.g. working with infinite bags). You are to implement the following three Bag operations:

```
-- return the number of times a given element occurs in a Bag
countBag :: Bag a -> a -> Integer

-- compute the union of two Bags.
unionBag :: Bag a -> Bag a -> Bag a

-- make a Bag from all elements satisfying a given property
makeBag :: (a -> Bool) -> Bag a
```

countBag, unionBag, and makeBag should be related by the following properties:

```
prop_union b1 b2 a = countBag b1 a + countBag b2 a == countBag (unionBag b1 b2) a
prop_makeBag p a = (countBag (makeBag p) a == 1)
             `xor` (countBag (makeBag (not . p)) a == 1)
                    where xor = (/=) -- exclusive or
```

But note that these cannot be run by quickCheck for technical reasons (additional quickCheck magic would be needed to generate random functions, but you do not have to worry about that here).