Module Code
COM00025I

BEng, BSc, MEng and MMath Degree Examinations 2020-21

Department Computer Science

Title Software 3

Issued: 09:00am on Saturday 21st August 2021.

Submission due: 09:00 on Sunday 22nd August 2021.

Feedback & Marks due: Wednesday 1st September 2021.

Time Allowed 24 Hours (NOTE: papers late by up to 30 minutes will be subject to a 5 mark penalty; papers later than 30 minutes will receive 0 marks).

Notification of errors in the paper may be made up to **one hour** after the start time. If you wish to raise a possible error it must be done through ⟨`cs-exams@york.ac.uk`⟩ with enough time for a response to be considered and made within the first hour.

Time Recommended THREE hours

Word Limit `Not Applicable.`

Allocation of Marks:

This paper has four questions. Question 1 is worth 35%, question 2 is worth 25%, question 3 is worth 20% and question 4 is worth 10%. The remaining 10% are allocated to style, clarity, and quality of code.

Instructions:

Candidates should answer **all** questions using Haskell. Failing to do so will result in a mark of 0%. For every question, a template has been provided in *SOF3Resit2021.hs*, and that is the **only** file to modify and submit. The **.hs** file has question descriptions, declarations – implementation as undefined and testing functions.

You may use any values defined in the standard 'Prelude', but you may not import any other module unless explicitly permitted. You may use Neil Mitchell's $Hoog\lambda e$ to discover useful functions in the *Prelude*.

Where tests accompany the English description of the problem, it is not necessarily enough for your implementation to pass the tests to be correct. Do not add any top-level definitions for which there is not already a skeleton provided in *SOF3Resit2021.hs*. This includes the value **main :: IO ()**. Should you do so, your file may not compile, and in that

case will receive **0 mark**. Where the solution uses helper values, they could be moved into a `where` clause.

Download the paper and the required source file from the VLE, in the "Assessment" section. You **must** save all your code in the *SOF3Resit2021.hs* file provided. **Do not** save your code anywhere else other than this file, failing to do so will result in a mark of **0%**.

Submit your *SOF3Resit2021.hs* file to the Department's Teaching Portal.

If a question is unclear, answer the question as best as you can, and note the assumptions you have made to allow you to proceed. Please inform ⟨`cs-exams@york.ac.uk`⟩ about any suspected errors on the paper immediately after you submit.

A Note on Academic Integrity

We are treating this online examination as a time-limited open assessment, and you are therefore permitted to refer to written and online materials to aid you in your answers.

However, you must ensure that the work you submit is entirely your own, and for the whole time the assessment is live you must not:

- communicate with departmental staff on the topic of the assessment
- communicate with other students on the topic of this assessment
- seek assistance with the assignment from the academic and/or disability support services, such as the Writing and Language Skills Centre, Maths Skills Centre and/or Disability Services. (The only exception to this will be for those students who have been recommended an exam support worker in a Student Support Plan. If this applies to you, you are advised to contact Disability Services as soon as possible to discuss the necessary arrangements.)
- seek advice or contribution from any third party, including proofreaders, online fora, friends, or family members.

We expect, and trust, that all our students will seek to maintain the integrity of the assessment, and of their award, through ensuring that these instructions are strictly followed. Failure to adhere to these requirements will be considered a breach of the Academic Misconduct regulations, where the offences of plagiarism, breach/cheating, collusion and commissioning are relevant: see AM1.2.1 *(Note this supercedes Section 7.3 of the Guide to Assessment)*.

1 (35 marks) Short questions

(i) [1 mark] Write a function `allSoft3` that takes a list of characters and returns `True` if every character is contained in "Software3" and `False` otherwise.

Your solution should satisfy:

```
allTest :: Bool
allTest =
    allSoft3 ""          == True  &&
    allSoft3 " "         == False &&
    allSoft3 "oft3w"     == True  &&
    allSoft3 "ofT3w"     == False &&
    allSoft3 "free"      == True  &&
    allSoft3 "Software3" == True

allSoft3 :: [Char] -> Bool

allSoft3 = undefined
```

(ii) [1 mark]

Write a function `three4th` that takes a fractional type and returns three-quarters of the input value.

Your solution should satisfy:

```
testThreeq :: Bool
testThreeq =
    three4th 0    == 0.0 &&
    three4th 1    == 0.75 &&
    three4th 8.0  == 6.0 &&
    three4th 4.8  == 3.5999999999999996 &&
    three4th (-8) == -6.0


three4th :: Fractional a => a -> a

three4th = undefined
```

(iii) [1 mark] Write a function `sqSum` that takes a list of numbers and returns the sum of the square of the numbers.

Your solution should satisfy:

```
testSum :: Bool
```

```
testSum =
     sqSum []                == 0 &&
     sqSum [1, 3, 4]         == 26 &&
     sqSum [1, 3, 4.0]       == 26.0 &&
     sqSum [1, 3, 4.2]       == 27.64 &&
     sqSum [1.0, 3.0, 4.0]   == 26.0 &&
     sqSum [1/2, 3/2, 4.0]   == 18.5 &&
     sqSum[-2, 3.0, 0.2, -1] == 14.04


sqSum :: Num ab => [ab] -> ab

sqSum = undefined
```

(iv) [2 marks] Write a function `same2other` that returns `True` when applied to a list with at least three elements of which the first two are the same as each other and at least one other element in the list apart from the second element, is also the same as the first element.

Your solution should satisfy:

```
testsame2other :: Bool
testsame2other =
    (same2other ""                        == False) &&
    (same2other "a"                       == False) &&
    (same2other [2]                       == False) &&
    (same2other "aaa"                     == True)  &&
    (same2other [8, 8, 8]                 == True)  &&
    (same2other [2, 2, 3, 4]              == False) &&
    (same2other [2, 2, 3, 4, 2]           == True)  &&
    (same2other "aatdaya"                 == True)  &&
    (same2other "aatdgyb"                 == False) &&
    (same2other [8, 8, 3, 8, 3, 6, 8, 12] == True)

same2other :: Eq a => [a] -> Bool
same2other = undefined
```

(v) [2 marks] Write a function `justVowels` that returns a list of all vowels in a string in the order and case they occur. Your solution should satisfy:

```
testVowels :: Bool
testVowels =
    justVowels "Hello World!"       == "eoo"  &&
    justVowels "Aaron562qe"         == "Aaoe" &&
    justVowels "sof3isGREATsoenjOY" == "oiEAoeO" &&
```

```
    justVowels "numberPLATE2021"    == "ueAE"

justVowels :: String -> String
justVowels = undefined
```

(vi) [3 marks] Write a function `revAllLower` that takes a string and returns the elements of the string in reverse order, converting every upper case character into lower case.

Your solution should satisfy:

```
testRev :: Bool
testRev =
    revAllLower ""                              == "" &&
    revAllLower "!reTupmoC"                     == "computer!" &&
    revAllLower "Software3"                     == "3erawtfos" &&
    revAllLower "Software3!"                    == "!3erawtfos" &&
    (revAllLower $  revAllLower "Software3!") == "software3!"


revAllLower :: String -> String
revAllLower = undefined
```

(vii) [5 marks] Write a function `findPlurals`, such that given a list over an `Ord` type, `a`, it returns a sorted list of all elements that occur at least twice.

Your solution should satisfy:

```
testfindPlurals :: Bool
testfindPlurals =
    (findPlurals ""                               == "") &&
    (findPlurals "THE1SOF1"                       == "1") &&
    (findPlurals "accommodation"                  == "acmo") &&
    (findPlurals "Accommodation"                  == "cmo") &&
    (findPlurals "THE2SOF2SYS1DAT1HCI1"           == "12HST") &&
    (findPlurals [1, 3, 4, 2, 3, 5, 7, 1, 9, 3] == [1,3]) &&
    (findPlurals [1, 3, 4, 2, 3, 5, 7, 1, 9, 5] == [1,3,5]) &&
    (findPlurals [1, 5, 4, 2, 3, 5, 7, 1, 9, 3] == [1,3,5])


findPlurals :: Ord a => [a] -> [a]

findPlurals = undefined
```

(viii) [5 marks] Given the following type `Course` to represent all university courses,

```
data Course = NICE | EASY | SOFT | HARD | HALF | FULL deriving (Show, Eq)
```

The rules on prerequisites are as follows:

To study EASY a student must have passed SOFT. To study HARD a student must have passed both EASY and NICE. To study FULL a student must have passed both SOFT and HALF. The rest of the courses do not have any prerequisites.

Given the type `Student` to represent any student's record, where `CMark` is the list of course and mark pair for the student.

```
data Student = Student SName Age College CMark


data College = Halifax | James | Langwith deriving (Show, Eq)

type SName   = String
type Mark    = Int
type Age     = Int
type CMark = [(Course, Double)]

benWalker, jBond, yWu, kSong, mGove :: Student
benWalker
 = Student "Ben Walker" 19 Halifax [(SOFT, 62), (EASY, 42), (FULL, 62)]
jBond = Student "James Bond" 21 Halifax [(SOFT, 42), (EASY, 42)]
mGove = Student "Mike Gove" 21 Halifax [(SOFT, 22), (EASY, 42)]
yWu = Student "Yang Wu" 18 Halifax [(SOFT, 22)]
kSong = Student "Kelly Song" 22 Halifax []
```

Given a student record `Student`, write a function `checkPrereqs` which returns `True` if the student has the prerequisites for all courses they have studied and `False` otherwise. The pass mark for a course is 40%.

Your solution should satisfy:

```
testPrereqs :: Bool
testPrereqs =
  (checkPrereqs benWalker == False) &&
  (checkPrereqs jBond     == True)  &&
  (checkPrereqs yWu       == True)  &&
  (checkPrereqs mGove     == False) &&
  (checkPrereqs kSong     == True)
```

```
checkPrereqs :: Student -> Bool

checkPrereqs = undefined
```

(ix) [5 marks] Write a function `numDiff` that computes the difference between the product of all even numbers and the sum of all odd numbers in a string. Assume every numeric character is an independent number.

Your solution should satisfy:

```
testND :: Bool
testND =
   numDiff "soft"               == 1 &&
   numDiff "soft2"              == 2 &&
   numDiff "soft3"              == -2 &&
   numDiff "char27481"          == 56 &&
   numDiff "3to15is117"         == -17 &&
   numDiff "some2743367numbers" == 28


numDiff :: String -> Int
numDiff = undefined
```

(x) [10 marks]

Consider the record of a Computer science student modelled by the type `CSStudent`

```
data CSStudent = CSStudent {sid :: SID, sname :: SName, stage :: Stage}
deriving (Ord, Eq, Show)

type SID     = Int

data Stage  = One | Two | Three | Four deriving (Ord, Eq, Show)
```

B-Trees are a generalisation of binary sort trees in which each node can have a larger, variable number of children.

Knuth's definition, a B-tree of order *m* is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every non-leaf node (except the root) has at least the ceil of *m/2* child nodes.
3. The root has at least two children unless it is a leaf node..
4. A non-leaf node with *k* children contains *k-1* keys.
5. All leaves appear in the same level.

Given the type `DBTree0` of a B-Tree:

```
data DBTree0 a = DBEmp Int | DBLeaf0 Int [a] | DBNode0 Int [a] [DBTree0 a]
        deriving (Eq, Show)
```

`DBEmp m` is an empty tree, `DBLeaf0 m xs` is a leaf node and `DBNode0 m xs ts` is a non-leaf node, all with order `m`.

```
csStud, students, csYork, dbThree, stdTest :: DBTree0 CSStudent
students = DBLeaf0 4
  [CSStudent {sid = 2, sname = "Mark Foster", stage = One},
  CSStudent {sid = 3, sname = "Juli Smith", stage = Two}]
stdTest  = DBLeaf0 4
  [CSStudent {sid = 1, sname = "Jane Hay", stage = One},
  CSStudent {sid = 3, sname = "Mat Bell", stage = Three}]
csStud   = DBEmp 4
csYork   = DBLeaf0 3
  [CSStudent {sid = 2, sname = "Mark Foster", stage = One},
  CSStudent {sid = 3, sname = "Juli Smith", stage = Two}]
dbThree  =
  DBNode0 3 [CSStudent {sid = 3, sname = "Kriss Wells", stage = One}]
  [DBLeaf0 3 [CSStudent {sid = 2, sname = "Sally Hodge", stage = Two}],
  DBLeaf0 3 [CSStudent {sid = 7, sname = "Greg Ovett", stage = Two},
  CSStudent {sid = 20, sname = "Ann Webb", stage = Four}]]
```

(a) [2 marks] Consider a database table with records of computer science students organised as a B-Tree of order 4 and `SID` as the keys (for example, `students`). Write a function `csFind` which returns `True` if a student with the provided `SID` exists in the database, and `False` otherwise.

Your solution should satisfy:

```
testcsFind :: Bool
testcsFind =
   csFind students 1       == False &&
   csFind students 2       == True &&
   csFind students 3       == True &&
   csFind students 0       == False &&
   csFind students (-13)   == False
```

```
csFind :: DBTree0 CSStudent -> SID -> Bool

csFind = undefined
```


Continued.

(b) [8 marks] Consider a database table with records of computer science students organised as a B-Tree of order 4 and `SID` as keys (for example, `students`. Write a function `csInsert` which inserts the record of a computer science student into a database organised as a B-Tree. You may assume that the B-tree is either empty, or the root node is a non-full leaf-node. In all other cases the function should return the original database.

Your solution should satisfy:

```
testInsert :: Bool
testInsert =
   csInsert students CSStudent {sid =1, sname = "Yi Wu", stage = Four} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Yi Wu", stage = Four},
      CSStudent {sid = 2, sname = "Mark Foster", stage = One},
      CSStudent {sid = 3, sname = "Juli Smith", stage = Two}] &&
   csInsert csStud CSStudent {sid =1, sname = "Mike Brown", stage = One} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Mike Brown", stage = One}] &&
   csInsert stdTest CSStudent {sid = 1, sname = "Yi Wu", stage = Four} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Jane Hay", stage = One},
      CSStudent {sid = 3, sname = "Mat Bell", stage = Three}] &&
   csInsert
      (csInsert csStud CSStudent {sid = 1, sname = "Mike Brown", stage = One})
      CSStudent {sid=2, sname="Georgia Jones", stage=Two} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Mike Brown", stage = One},
      CSStudent {sid = 2, sname = "Georgia Jones", stage = Two}] &&
   csInsert
      (csInsert (csInsert
      csStud CSStudent {sid = 1, sname = "Mike Brown", stage = One})
      CSStudent {sid=2, sname="Georgia Jones", stage=Two})
      CSStudent {sid=3, sname="Tamara Berg", stage=Three} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Mike Brown", stage = One},
      CSStudent {sid = 2, sname = "Georgia Jones", stage = Two},
      CSStudent {sid = 3, sname = "Tamara Berg", stage = Three}] &&
   csInsert
      (csInsert (csInsert (csInsert
      csStud CSStudent {sid = 1, sname = "Mike Brown", stage = One})
      CSStudent {sid=2, sname="Georgia Jones", stage=Two})
      CSStudent {sid=3, sname="Tamara Berg", stage=Three})
      CSStudent {sid=7, sname="Eric Han", stage=Four} ==
      DBLeaf0 4 [CSStudent {sid = 1, sname = "Mike Brown", stage = One},
      CSStudent {sid = 2, sname = "Georgia Jones", stage = Two},
      CSStudent {sid = 3, sname = "Tamara Berg", stage = Three}]
```

```
csInsert :: DBTree0 CSStudent -> CSStudent -> DBTree0 CSStudent
csInsert = undefined
```

2 (25 marks) Type definitions

In this question you are asked to define types, as well as values. Each type, *Name*, is given a default implementation as an alias for the unit type: `type Name = ()`. You may change the keyword `type` to either `newtype` or `data` if appropriate, and you should replace the unit type, `()`, by a suitable type expression. You may derive any type classes, such as `Show`, that will be useful.

Marks will be mostly given for how well the type protects the programmer from mistakes, and also partly for the run-time efficiency of the type.

Consider the game of *Mastermind*, where one player constructs a secret, known as a "code", and their opponent has to guess the secret in a limited number of guesses. Each incorrect guess gives information to the guesser that they can use to refine their guess.

A secret is represented as four distinct colours, in a particular order, drawn from six possibilities:

```
data Colour = Red | Orange | Yellow | Green | Blue | Indigo
  deriving (Eq, Show, Read)
```

A guess is also four distinct colours, in a particular order.

(i) [4 marks] Implement a type, `Code`, to describe values that are secrets and guesses. `Code` must be an instance of the `Eq` type class.

```
type Code = () -- Replace with a suitable definition
```

(ii) [2 marks] Implement functions to turn a code into a list of length 4, and *vice versa*.

```
code2List :: Code -> [Colour]
list2Code :: [Colour] -> Code
code2List = undefined
list2Code = undefined

testCodeToFromList :: Bool
testCodeToFromList = let datum = [Red, Orange, Yellow, Green] in
                       code2List(list2Code datum) == datum
```

(iii) [4 marks] For each guess, the response is a pair of numbers:

1. The number of guess colours in the right position, and
2. The number of guess colours in the wrong position

Implement a type to represent responses to guesses. `Response` must be an instance of the `Eq` type class.

```
type Response = () -- Replace with a suitable definition.
```

(iv) [2 marks] Implement functions to construct and deconstruct a `Response`. Given `r :: Response`, `hits r` should evaluate to the reported number of right colours in the right location, and `near r` to the reported number of right colours in the wrong location.

```
mkResponse :: Int -> Int -> Response
hits, near :: Response -> Int
hits = undefined
near = undefined

test_mkResponse :: Bool
test_mkResponse = let r = mkResponse 2 1 in (hits r, near r) == (2, 1)
```

(v) [4 marks] Implement a function, `oneRound`, that takes a secret (a "code") and a guess and returns the appropriate response.

```
oneRound :: Code -> Code -> Response
oneRound = undefined

test_oneRound :: Bool
test_oneRound = oneRound (list2Code [Red, Orange, Yellow, Green])
     (list2Code [Green, Blue, Yellow, Orange]) == mkResponse 1 2
```

(vi) [5 marks] Given a type to represent the two players,

```
data Player = Encoder | Guesser deriving (Eq, Show)
```

define a type `Winner` whose values represent whether or not there is a winner in the current state of a game, and, if there is a winner, the winner's name.

```
type Winner = () -- Replace with a suitable definition.
```

(vii) [4 marks] Implement a function, `winner`, that given a response to the last guess and the remaining number of guesses reports the current winner, if any, and a function `ppWinner` that gives the name of the winner (`"Encoder"` or `"Guesser"`) or "No winner", as appropriate.

The guesser is the winner if the last guess exactly matches the pattern, and the encoder is the winner the guesser has run out of attempts.

[We have provided a definition, `noWinner :: String`, that you may use.]

```
winner :: Response -> Int -> Winner
ppWinner :: Winner -> String
noWinner :: String
noWinner = "No winner"
winner = undefined
ppWinner = undefined

test_winner :: Bool
```

```
test_winner = ppWinner (winner (mkResponse 2 1) 2) == noWinner
&& ppWinner (winner (mkResponse 2 1) 0) == "Encoder"
```

3 (20 marks) New supermarket

A new supermarket, *Superior Outlet For Fine Foods*, better known by its acronym, "SOFFF", or "SOF3", is being set up.

The software team producing the stocking database are trying to choose between two different representations for the current stock:

```
newtype StockF item = StockF {getStockF :: item -> Int}
newtype StockL item = StockL {getStockL :: [(item, Int)]} deriving (Eq, Show)
```

Here `item` is an arbitrary type, whose values are identifiers of the items stocked. `StockF` maintains the database as a function that, given an item identifier, returns the quantity in stock. `StockL` maintains the database as a list of pairs where the first element of each pair is an item identifier, and the second is the quantity in stock.

A small concrete example of item is `Item`, where

```
data Item = Loaf_White_Small | Loaf_Brown_Large | Single_Apple | Raisins_1kg
        deriving (Eq, Show)
```

but any type instantiating type class `Eq` can be used.

To help the software team decide between representations they have commissioned you to produce pairs of functions for each task of interest. For each task below give **two** functions that solve the problem, one function for each representation.

(i) [4 marks] Report how many instances of a given item are in stock.

```
countF :: StockF item -> item -> Int
countL :: Eq item => StockL item -> item -> Int
countF = undefined
countL = undefined

test_count :: Bool
test_count = let f Single_Apple = 30
                 f Raisins_1kg = 6
                 f _           = 0
        in countF (StockF f) Single_Apple == 30
        &&
        countL (StockL [(Loaf_White_Small, 0), (Single_Apple, 30)])
          Single_Apple == 30
```

(ii) [8 marks] The supermarket can be restocked. Implement the two operations that produce the updated database. Note that the incoming delivery of stock can also be represented as a value of type `StockF` or `StockL` as appropriate. The first parameter to each function is the stock in the shop, the second is the stock on the lorry bringing the new items.

```
restockF :: StockF item -> StockF item -> StockF item
restockL :: Eq item => StockL item -> StockL item -> StockL item
restockF = undefined
restockL = undefined

test_restock :: Bool
test_restock =
  let f Single_Apple = 30
      f Raisins_1kg  =  6
      f _            =  0
      g Raisins_1kg      = 3
      g Loaf_White_Small = 7
      g _                = 0
      r = restockF (StockF f) (StockF g)
  in countF r Single_Apple == 30 && countF r Raisins_1kg == 9
  &&
  let r = restockL (StockL [(Single_Apple, 30),(Raisins_1kg, 6)])
          (StockL [(Raisins_1kg, 3), (Loaf_White_Small, 7)])
  in countL r Single_Apple == 30 && countL r Raisins_1kg == 9
```

(iii) [8 marks] The supermarket receives orders, which it fills as far as it can: for example, if four apples are requested, but the supermarket only has two, then only two will be removed from the stock. Implement the two operations that produce the updated database.

Note that the order can also be represented as a value of type `StockF` or `StockL` as appropriate. The first parameter to each function is the stock in the shop, the second is the order.

This operation does **not** report the unfulfilled part of the order.

```
fillOrderF :: StockF item -> StockF item -> StockF item
fillOrderL :: Eq item => StockL item -> StockL item -> StockL item
fillOrderF = undefined
fillOrderL = undefined

test_fillOrder :: Bool
test_fillOrder =
        let f Single_Apple = 30
              f Raisins_1kg  =  6
              f _            =  0
              g Raisins_1kg      = 9
              g Loaf_White_Small = 7
              g _                = 0
              r = fillOrderF (StockF f) (StockF g)
        in countF r Single_Apple == 30 && countF r Raisins_1kg == 0
```

```
&&
let r = fillOrderL (StockL [(Single_Apple, 30), (Raisins_1kg, 6)])
          (StockL [(Raisins_1kg, 9), (Loaf_White_Small, 7)])
in countL r Single_Apple == 30 && countL r Raisins_1kg == 0
```

4 (10 marks) Proof

Recall the definition of the `ProofLayout` type constructor:

```
infixr 0 :=: -- the fixity and priority of the operator
data ProofLayout a = QED | a :=: ProofLayout a deriving Show
```

Consider the following definitions from the standard prelude:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) = f (g x)            -- (.).0

maybe :: a -> (b -> a) -> Maybe b -> a
maybe k f Nothing  = k       -- maybe.0
maybe k f (Just x) = f x     -- maybe.1
```

By exhibiting a suitable value of type `ProofLayout`, prove that

```
FOR-ALL g :: a -> b, k :: a, f :: c -> a {g . maybe k f == maybe (g k) (g . f)}
```

Marks will be given for correct annotation, as well as correct proof steps.

Hint

Use a case analysis over the structure of values drawn from a `Maybe` type.

```
distribMaybe :: (a -> b) -> a -> (c -> a) -> (Maybe c) -> ProofLayout b
distribMaybe = undefined
```

End of examination paper