CSC3403 S1 2022 Assignment 2

Functional Programming — Due Date: 03 May 2022

Objective

This assignment aligns with Course Learning Outcome 3. It assesses your ability to:

- Analyse and compare different language paradigms, with a particular emphasis on functional programming languages.

Marks

There are **two (2)** tasks each comprising several parts in Assignment 2. There are **20 marks** and Assignment 2 comprises **20%** of your final mark.

Submission

- Submit a single plain text file (extension `.txt`) containing all definitions. Please use your user id as the name of the file; e.g. `u12345678.txt`.
- The text file must only contain valid Haskell code and comments.
- **Use the specified function name and type signatures as your code will be tested by a Haskell function expecting that function name**.
  - You may however write any 'helper' functions that you need to be able to solve any of the following problems.
- Final submission is via the submission link for Assignment 2 on the CSC3403 Study Desk.
- Unless stated otherwise **do not use library functions that are not in the Haskell standard prelude**[1]. This constraint is so that you gain practice in simple Haskell recursive programming.
- You must use techniques **which have been presented in the course**. Using sophisticated code and methods which have not been presented will attract penalties.
- You can use Hugs98 or GHCi on your own systems or use the provided resources on the USQ Turbo system used in the course Workshops for your code.

[1] The Haskell 2010 standard prelude definition is available at https://www.haskell.org/onlinereport/haskell2010/haskellch9.html

Late Submission of Assignments[2]

Students can apply for an extension of time to submit an assignment at any time **up to** the deadline. Students are advised to make a request for an extension as soon as their need becomes apparent. Delay in making a request involves the risk of losing marks if the request is refused.

Extensions are usually granted only in cases of Compassionate and Compelling Circumstances in accordance with the assessment of Compassionate and Compelling Circumstances Procedure. Generally, extensions will be limited to a maximum of five University Business Days. A Student requiring an extension for a period of time in excess of this should consider applying for a Deferred Assessment as per Section 4.4 of the assessment procedure.

Applications for extensions must be made through your **Student Centre**. An Assignment submitted after the deadline without an approved extension of time will be penalised. The penalty for late submission without a pre-approved extension is a reduction by 5% of the maximum mark applicable for the assignment, for **each *calendar* day** or part day that the assignment is late. An assignment submitted more than **one (1) week** after the deadline will have a mark of zero recorded for that assignment.

The Examiner may refuse to accept assignments for assessment purposes after marked assignments and/or feedback have been released.

Non-submission of Assignments and Passing Grades[3]

To be assured of receiving a passing grade a student must obtain at least 50% of the total weighted marks available for the course and have satisfied any Secondary Hurdles (if applicable).

Supplementary assessment may be offered where a student has undertaken all of the required summative assessment items and has passed the Primary Hurdle but failed to satisfy the Secondary Hurdle (Supervised), or has satisfied the Secondary Hurdle (Supervised) but failed to achieve a passing Final Grade by 5% or less of the total weighted Marks.

To be awarded a passing grade for a supplementary assessment item (if applicable), a student must achieve at least 50% of the available marks for the supplementary assessment item as per the Assessment Procedure (point 4.4.2).

[2] See http://policy.usq.edu.au/documents.php?id=14749PL for the full assessment regulations.
[3] See the University Assessment Procedure policy document

The offer of Supplementary Assessment normally will only be made if the Student has undertaken all possible Summative Assessment Items for the Course (i.e. the assignments).

Student Responsibilities

The Assessment Procedure Section 4.2.2, also outlines the following student responsibilities:

- If requested, Students must be capable of providing a copy of Assignments submitted. Copies should be dispatched to the University within 24 hours of receipt of a request being made.
- Students are responsible for submitting the correct Assignment.
- Assignment submissions must contain evidence of student effort to address the requirements of the Assignment. In the absence of evidence of Student effort to address the requirements of the assignment, no Mark will be recorded for that Assessment Item.
- A Student may re-submit an Assignment at any time up to the deadline. A request to re-submit after the deadline is dealt with in accordance with section 4.4 'Deferred, Supplementary and Varied Assessment and Special Consideration' of these procedures.

Academic Integrity (AI)

Academic misconduct is unacceptable and includes plagiarism, collusion and cheating:

- *plagiarism*: involves the use of another person's work without full and clear referencing and acknowledgement;
- *cheating*: involves presenting another person's work as your own — this includes ***contract cheating***.
- *collusion*: is a specific type of cheating, that occurs when two or more students fail to abide by directions from the examiner regarding the permitted level of collaboration on an assessment.

All are seen by the University as acts of misconduct for which you can be penalised. For further details go to the Library's site on Academic Integrity.

Task 1 (7 marks)

Topics:	Functional programming; recursive programming.
Submission:	Part of single `.txt` document containing all of your Haskell code and comments for the assignment.
Notes:	You must use the function names as specified and any specified type signatures. You may write helper functions that the specified function calls. Not all solutions *necessarily* require a recursive function. However, if the task requests it specifically, the function must be recursive.

Background

You may have noticed how many of our day-to-day numbers involve the number 12. The duodecimal (or dozenal) system[4] is a base 12 number system, in the same way as hexadecimal is base 16.

If we use `A` to represent `10` and `B` to represent `11`, counting to 12 in duodecimal is:

```
0 1 2 3 4 5 6 7 8 9 A B 10 ...
```

In our tasks here in Haskell:

- `A` represents `10` and `B` represents `11`
- a duodecimal number will be represented by a string beginning with `0d`[5], for example decimal `22` equals duodecimal `0d1A`

Task 1.1 (4 marks) Write a *recursive function*:

```
duoToDec ::  String -> Int
```

to convert a duodecimal number to decimal.

- If the duodecimal number does not begin with `0d` the function should produce a program error
- If the duodecimal number contains invalid duodecimal digits (i.e. not `0-9, A, B` it should produce a program error
- You must use only only the Haskell prelude (no library functions)
- You may use helper functions; recursion may occur in calling or called functions or both

[4] https://en.wikipedia.org/wiki/Duodecimal
[5] `0d` = 'zero d'

Example output (Hugs98):

```
Main> duoToDec "0d1A"
22
Main> duoToDec "0d01A"
22
Main> duoToDec "0d9"
9
Main> duoToDec "0d10"
12
Main> duoToDec "10"

Program error: Not a valid duodecimal number: must start with '0d'

Main> duoToDec "0d2C"

Program error: Not a valid duodecimal number: contains invalid digit
```

Task 1.2 (3 marks) Write the complementary *recursive function*:

```
decToDuo ::  Int -> String
```

to convert a decimal number to a duodecimal number.

- No error checking is required; assume input to the function is a valid `Int`
- The duodecimal must begin with the characters `0d`
- You must use only only the Haskell prelude (no library functions)
- You may use helper functions; recursion may occur in calling or called functions or both
- **Hint:** To convert an `Int` to a `String` you can use
  - `show x - where x is an Int`

  but use it wisely and think through where to use it.

Example output (Hugs98):

```
Main> show 3
"3"
Main> decToDuo 12
"0d10"
Main> decToDuo 10
"0dA"
```

```
Main> decToDuo 77
"0d65"
Main>
```

End of Task 1

Task 2 (13 marks)

Topics:	Functional programming; recursive programming.
Submission:	Part of single `.txt` document containing all of your Haskell code and comments for the assignment.
Notes:	You must use the function names as specified and any specified type signatures. You may write helper functions that the specified function calls. Not all solutions *necessarily* require a recursive function. However, if the task requests it specifically, the function must be recursive.

Background

A *random walk*[6] is a path that consists of a series of random steps. Figure 1[7] shows five lattice random walks that all start from a central point. Each step is to an adjacent point. As noted, some of the paths appear shorter than the eight steps because the path backtracks!

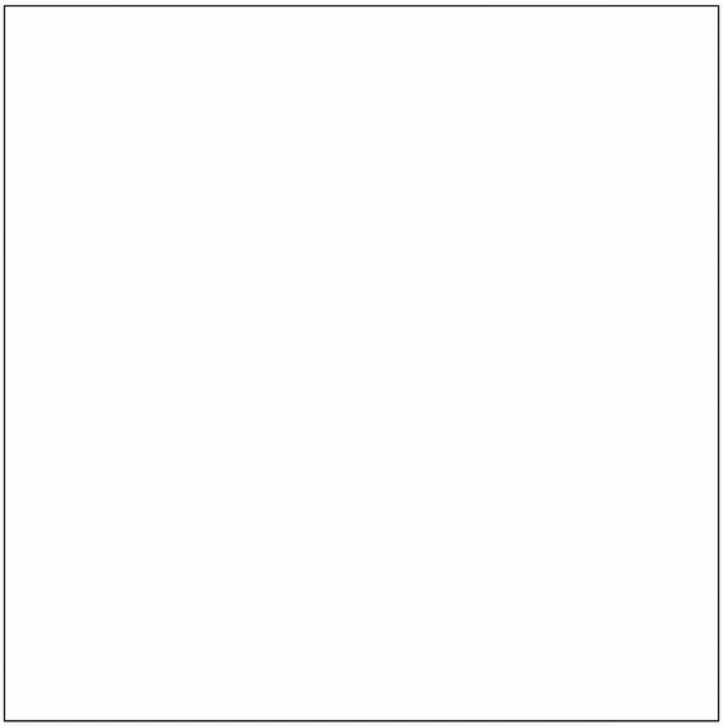

Figure 1: Five eight-step random walks from a central point. Some paths appear shorter than eight steps where the route has doubled back on itself[7,8]

[6] https://en.wikipedia.org/wiki/Random_walk

[7] This is an animation and should display in Adobe Reader, Okular and other PDF readers. If the image is not present, follow the link in the image credit.

[8] Image credit: https://en.wikipedia.org/wiki/File:Random_Walk_Simulator.gif CC BY-SA 4.0

On page 12 is the `main` routine of a program that performs a random walk and an example output.

If you examine the output, you will see that in this random walk, each "step" is forward/back **and** left/right, represented by a tuple of type `(Int, Int)`:

```
(left/right, front/back)
```

where:

```
left/right:  -1 = "left", 0 = "no movement", 1 = "right"
front/back:  -1 = "back", 1 = "front"
```

For example, `(0,1)` represents no left/right, forward one step; `(-1,-1)` represents one step left, one step back.

As you can see, there are several functions being called by the main module. You need to write these functions as instructed.

You do not need to make the entire code work. Each function you write can be tested independently! This is part of the paradigm of functional programming - writing functions which do a single task predictably and reliably!

Task 2.1 (2 marks) In the code, the generator `g` produces a string of random integers between `1` and `10` representing the left/right component of a "step". The probability of the walker taking a "left (-1)" step is 20%; for no movement left/right (0) is 60%, and for "right (1) is 20%.

Write a *single recursive function*:

```
leftRight ::  [Int] -> [Int]
```

which takes the array of random numbers from the generator and returns an array of equal length where:

if input array element is:	*output array element is:*
1,2	-1
3,4,5,6,7,8	0
9,10	1

- Only write a single function (no helper functions!)
- You must use only only the Haskell prelude (no library functions)

Example Output (Hugs98):

```
Main> leftRight [9,6,6,5,3,4,8,4]
[1,0,0,0,0,0,0,0]

Main> leftRight [6,9,8,1,10,4,5,8]
[0,1,0,-1,1,0,0,0]
```

Task 2.2 (2 marks) In the code, the generator `h` produces a string of random integers between `1` and `10` representing the front/back component of a "step". The probability of the walker taking a "back (-1)" step is 30%; for a "front (1)" step it is 70%.

Write two functions.

The *first* function:

```
frontBack ::  [Int] -> [Int]
```

takes the array of random numbers from the generator and returns an array of equal length where:

if input array element is:	*output array element is:*
1,2,3	-1
4,5,6,7,8,9,10	1

You *must* use the Haskell `map` function which will apply your *second* function[9]:

```
frontBack'
```

to each element of the input array. Note that you must determine and use an appropriate type signature for this second function.

- You must use only only the Haskell prelude (no library functions)

Example output (Hugs98):

```
Main> frontBack [7,2,7,1,10,10,10,1]
[1,-1,1,-1,1,1,1,-1]

Main> frontBack [6,6,9,4,5,2,7,8]
[1,1,1,1,1,-1,1,1]
```

[9]Note the name of the function is "`frontBack-prime`", where the `'` is the apostrophe character

Task 2.3 (2 marks) Write a *recursive function*:

```
takeStep ::  [Int] -> [Int] -> [ (Int, Int) ]
```

which takes the output lists from the `leftRight` and `frontBack` and combines them into a list of tuples describing each "step".

- You must use only only the Haskell prelude (no library functions)
- You may write helper functions if required

Example output (Hugs98):

```
Main> takeStep [1,0,0,0,0,0,0,0] [1,-1,1,-1,1,1,1,-1]
[(1,1),(0,-1),(0,1),(0,-1),(0,1),(0,1),(0,1),(0,-1)]

Main> takeStep [0,1,0,-1,1,0,0,0] [1,1,1,1,1,-1,1,1]
[(0,1),(1,1),(0,1),(-1,1),(1,1),(0,-1),(0,1),(0,1)]
```

Task 2.4 (3 marks) Now that the "steps" have been found, we want to find the destination (final coordinates)! The final coordinates can be found by summing the left-/right terms in each step tuple and summing the front/back terms in each step tuple!

Write a *recursive function*:

```
walkEnd ::  [(Int,Int)] -> (Int,Int)
```

which takes the takes the list of steps and produces a single tuple.

- You must use only only the Haskell prelude (no library functions)
- You may write helper functions if required

Example output (Hugs98):

```
Main> walkEnd  [(1,1),(0,-1),(0,1),(0,-1),(0,1),(0,1),(0,1),(0,-1)]
(1,2)

Main> walkEnd [(0,1),(1,1),(0,1),(-1,1),(1,1),(0,-1),(0,1),(0,1)]
(1,6)
```

Task 2.5 (4 marks) The final part to the program is to take the list of steps and produce a list which describes the *path* taken, starting at coordinates `[(0,0)]`.

Write a recursive function:

```
walkPath ::  Int -> [(Int,Int)] -> [(Int,Int)]
```

which takes the takes the number of steps, the list of steps and produces a list of coordinates showing the path taken.

- You must use only only the Haskell prelude (no library functions)
- You may write helper functions if required
- This is a tricky problem which you may need to think through...

Example output (Hugs98):

```
Main> walkPath 8 [(1,1),(0,-1),(0,1),(0,-1),(0,1),(0,1),(0,1),(0,-1)]
[(0,0),(1,1),(1,0),(1,1),(1,0),(1,1),(1,2),(1,3),(1,2)]

Main> walkPath 8 [(0,1),(1,1),(0,1),(-1,1),(1,1),(0,-1),(0,1),(0,1)]
[(0,0),(0,1),(1,2),(1,3),(0,4),(1,5),(1,4),(1,5),(1,6)]
```

End of Task 2

Code for main program described in Task 2 on next page

Random Walk Program for Task 2:

```
import System.Random
import RandomSteps

main = do

    putStrLn "Enter the number of steps for random walk:  "
    inputInt <- getLine
    let steps = (read inputInt :: Int)

    g <- newStdGen  -- leftRight random sequence
    h <- newStdGen  -- frontBack random sequence
    let leftRightSteps = take steps (randomRs (1,10) g :: [Int])
    let frontBackSteps = take steps (randomRs (1,10) h :: [Int])

    putStrLn ""

    putStrLn $ "Random Sequence g (leftRight):  " ++ show (leftRightSteps)
    putStrLn $ "Random Sequence h (frontBack):  " ++ show (frontBackSteps)

    putStrLn ""

    putStrLn $ "Steps Taken:  " ++ show (takeStep (leftRight $ leftRightSteps)  (frontBack $ frontBackSteps))
    putStrLn $ "Path Walked:  " ++ show (walkPath steps (takeStep (leftRight $ leftRightSteps)  (frontBack $ frontBackSteps)))
    putStrLn $ "Final Coordinates:  " ++ show (walkEnd $ takeStep (leftRight $ leftRightSteps)  (frontBack $ frontBackSteps))
```

Example output (Hugs98):

```
Main> main
Enter the number of steps for random walk:
8

Random Sequence g (leftRight):  [9,6,6,5,3,4,8,4]
Random Sequence h (frontBack):  [7,2,7,1,10,10,10,1]

Steps Taken:  [(1,1),(0,-1),(0,1),(0,-1),(0,1),(0,1),(0,1),(0,-1)]
Path Walked:  [(0,0),(1,1),(1,0),(1,1),(1,0),(1,1),(1,2),(1,3),(1,2)]
Final Coordinates:  (1,2)
```

Note

The more steps the code is given, the larger the "front" component should be, while the left/right should be around zero.

Looking at the probabilities, this is pretty obvious. Some final coordinates of runs of 1000 steps:

```
Final Coordinates:  (18,412)
Final Coordinates:  (-33,372)
Final Coordinates:  (-9,468)
Final Coordinates:  (-21,416)
```

End of Assignment 2

You should have one (1) `.txt` file containing only valid Haskell comments and code.

These files should then be uploaded using the Assignment 2 submission link on the Study Desk.

This assignment specification was typeset using LaTeX