Haskell Programming Assignment Specification

In this assignment I learned how program many things in haskel. I learned list development through recursion, I learned how higher order functions work, list comprehension, and more.

Task 1 - Mindfully Mimicking the Demo

Please engage in a Haskell session with the REPL that mimics the following session. Then, incorporate the session into your presentation document.

```
ndavis20@altair:~/public_html/CSC344WorkSite/.homeWork/files_to_send/assignment_5$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :set promp ">>>"
Some flags have not been recognized: promp, >>>
Prelude> :set prompt ">>>"
>>>:set prompt ">>> "
>>> length [2,3,5,7]
4
>>>  words "need more coffee"
["need","more","coffee"]
>>>  unwords ["need","more","coffee"]
"need more coffee"
>>> reverse "need more coffee"
"eeffoc erom deen"
>>> reverse ["need","more","coffee"]
["coffee","more","need"]
>>> head ["need","more","coffee"

<interactive>:9:29: error:
    parse error (possibly incorrect indentation or mismatched brackets)
>>> head ["need","more","coffee"]
"need"
>>> tail ["need","more","coffee"]
["more","coffee"]
>>> last ["need","more","coffee"]
"coffee"
>>> init ["need","more","coffee"]
["need","more"]
>>> take 7 "need more coffee"
"need mo"
>>> drop 7 "need more coffee"
"re coffee"
>>> ( \x -> length x > 5 ) "Friday"
True
>>>  ( \x -> length x > 5 ) "uhoh"
False
>>> ( \x -> x /= ' ' ) 'Q'
True
>>> ( \x -> x /= ' ' ) ' '
False
>>> filter ( \x -> x /= ' ' ) "Is the Haskell fun yet?"
"IstheHaskellfunyet?"
>>> :quit
Leaving GHCi.
ndavis20@altair:~/public_html/CSC344WorkSite/.homeWork/files_to_send/assignment_5$
```

Task 2 - Numeric Function Definitions

This task requires that you write 5 function definitions, and that you then demo them by recreating a given demo. For this task, please add to your presentation document (1) a text containing the 5 function definitions, and (2) a text containing a demo which is a recreation of the one that is provided. **Constraint: Strive to make each function definition as meaningful as possible, by choosing meaningful names for values (most of them, anyway), and by writing code that illuminates, for even the most casual reader, the process by which the final value is obtained.**

Function specifications

Please refer to the demo for clarification on these specifications, as needed.

1. Define a function called squareArea, taking one real number representing the side length of a square is its sole parameter, which returns the area of the square with the given side length.
2. Define a function called circleArea, taking one real number representing the radius of a circle is its sole parameter, which returns the area of the circle with the given radius.

3. Imagine a cube, each face of which is blue with a centered white dot of radius one-fourth the side length of the cube. Define a function called blueAreaOfCube, taking the length of one edge of the cube as its sole parameter, which returns the blue area of the cube.
4. Imagine that a wooden cube is dissected into n x n x n little cubes. For such a cube, take n to be the **order** of the cube. Now, suppose that such a cube of order n is painted blue, and then taken apart. How many of the little cubes would have just one of its faces painted? Define a function called paintedCube1, taking the order of a dissected, painted cube as its sole parameter, which returns the number of little cubes that would have just one blue face.
5. Again, imagine the painted cube scenario. Define a function called paintedCube2, taking the order of a dissected, painted cube as its sole parameter, which returns the number of little cubes that would have exactly two blue faces.

The given demo that you are to recreate

```
squareArea side = side * side

circleArea radius = pi * radius * radius

blueAreaOfCube edge = totalCubeArea - totalCircleArea
   where totalCubeArea = (squareArea edge) * 6
         radius = edge / 4
         totalCircleArea = (circleArea radius) * 6

paintedCube1 1 = 0
paintedCube1 order = (face - parameter) * 6
   where face = order * order
         parameter = ((order * 4) - 4)

paintedCube2 1 = 0
paintedCube2 order = paramCornersRemoved * 3
   where parameter = (( order * 4) - 4)
         paramCornersRemoved = parameter - 4
```

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load ha.hs
[1 of 1] Compiling Main             ( ha.hs, interpreted )
Ok, one module loaded.
*Main> sq
sqrt        squareArea
*Main> squareArea 10
100
*Main> squareArea 12
144
*Main> circleArea 10
314.1592653589793
*Main> circleArea 12
452.3893421169302
*Main> blueAreaOfCube 10
482.19027549038276
*Main> blueAreaOfCube 12
694.3539967061512
*Main> blueAreaOfCube 1
4.821902754903828
*Main> map blueAreaOfCube [1..3]
[4.821902754903828,19.287611019615312,43.39712479413445]
*Main> paintedCube1 1
0
*Main> paintedCube1 2
0
*Main> paintedCube1 3
6
*Main> map paintedCube1 [1..10]
[0,0,6,24,54,96,150,216,294,384]
*Main> paintedCube2 1
0
*Main> paintedCube2 2
0
*Main> paintedCube2 3
12
*Main> map paintedCube2 [1..10]
[0,0,12,24,36,48,60,72,84,96]
*Main> :quit
Leaving GHCi.
```

Task 3 - Puzzlers

This task requires that you write 2 function definitions, and that you then demo them by creating a “proper” demo. For this task, please add to your presentation document (1) a text containing the 2 function definitions, and (2) a text containing the demo that you are asked to create. What is a proper demo with respect to this task? Run each of the 2 functions with the applications that I provide in my sample demo, and then, for each of the 2 functions, add 2 applications of your own invention. Thus, a proper demo will have 4 applications for each of the 2 functions.

Function specifications

```
reverseWords string = (reverse string)

averageWordLength string = avg
   where list = (words string)
         numOfWords = fromIntegral (length list)
         lengthList = (map length list)
         total = fromIntegral (foldl (+) 0 lengthList)
         avg = total / numOfWords
```

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load ha.hs
[1 of 1] Compiling Main             ( ha.hs, interpreted )
Ok, one module loaded.
*Main> reverseWords "appa and baby yoda are the best"
"tseb eht era adoy ybab dna appa"
*Main> reverseWords "want me some coffee"
"eeffoc emos em tnaw"
*Main> averageWordLength "appa and baby yoda are the best"
3.5714285714285716
*Main> averageWordLength "want me some coffee"
4.0
*Main> :quit
```

Task 4 - Recursive List Processors

This task requires that you write 3 recursive function definitions, and that you then demo them by recreating a given demo. For this task, please add to your presentation document (1) a text containing the 3 function definitions, and (2) a text containing a recreation of the demo that I have provided

Function specifications

Please refer to the demo for clarification on these specifications, as needed.

1. Define a **recursive** function called list2set, taking one list of objects as its sole parameter, which returns a list of the objects in the given list, but with all duplicates removed.
2. Define a **recursive** function called isPalindrome, taking one list of objects as its sole parameter, which returns true if the list of objects is palindromic (reads the same forwards as it does backwards).
3. Define a **recursive** function called collatz, taking one positive integer value as its sole parameter, which returns the Collatz sequence corresponding to the given value as a list. (Recall that the Collatz sequence was introduced during the “Racket” portion of this course.

The given demo that you are to recreate

```haskell
-------------------------------------------------------
----------------------- TASK 4 ------------------------
-------------------------------------------------------

list2set [] = []
list2set list = finalList
   where thisHead = take 1 list
         newList = filter ( \x -> x /= (head thisHead)) list
         currentList = thisHead ++ (list2set newList)
         finalList = currentList

isPalindrome [] = True
isPalindrome [_] = True
isPalindrome list =  boolean
   where firstElement = (head list)
         lastElement = (last list)
         boolean = firstElement == lastElement && (isPalindrome (tail (init list)))

collatz :: Int -> [Int]
collatz 1 = []
collatz num =
   if (even num) then
       [num] ++ (collatz (num `div` 2))
   else
       [num] ++ (collatz ((num * 3) + 1))

```

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>  list2set [1,2,3,2,3,4,3,4,5]

<interactive>:1:2: error:
    Variable not in scope: list2set :: [Integer] -> t
Prelude> :load ha.hs
[1 of 1] Compiling Main             ( ha.hs, interpreted )
Ok, one module loaded.
*Main>  list2set [1,2,3,2,3,4,3,4,5]
[1,2,3,4,5]
*Main>  list2set "need more coffee"
"ned morcf"
*Main>  isPalindrome ["coffee","latte","coffee"]
True
*Main>  isPalindrome ["coffee","latte","espresso","coffee"]
False
*Main> isPalindrome [1,2,5,7,11,13,11,7,5,3,2]
False
*Main>  isPalindrome [2,3,5,7,11,13,11,7,5,3,2]
True
*Main> collatz 10
[10,5,16,8,4,2]
*Main> collatz 11
[11,34,17,52,26,13,40,20,10,5,16,8,4,2]
*Main> collatz 100
[100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2]
*Main> :quit
Leaving GHCi.
ndavis20@altair:~/public_html/CSC344WorkSite/.homeWork/files_to_send/assignment_5$
```

Task 5 - List Comprehensions

This task requires that you write 2 function definitions by using list comprehensions, and that you then demo them by creating a “proper” demo. For this task, please add to your presentation document (1) a text containing the 2 function definitions, and (2) a text containing the demo that you are asked to create. What is a proper demo with respect to this task? Run each of the 2 functions with the 2 applications that I provide in my sample demo, then add 2 applications of your own invention for each of the functions. Thus, your demo will have 4 applications for each of the 2 functions.

Function specifications

Please refer to the demo for clarification on these specifications, as needed.

1. Define a function called count, taking an object and a list of objects of the same type as parameters, which returns the number of times the object occurs in the list. **Constraint: Make good use of a list comprehension in defining this function.**
2. Define a function called freqTable, taking a list of objects as its sole parameter, which returns a list of ordered pairs, each consisting of an element of the list together with the number of times the element occurs in the list. **Constraint: Make good use of a list comprehension in defining this function.** Hint: Use the list2set function from your previous task, and the count function from this task.

The given demo that you are to augment

```
54 ------------------------------------------------------------
55 ------------------------ TASK 5 --------------------------
56 ------------------------------------------------------------
57
58 count a b =
59   length [ occurences | occurences <- filter (\occurences -> occurences == a) b]
60
61 repeatList 0 myList = []
62 repeatList num myList = finalList
63    where finalList = [myList] ++ (repeatList (num - 1) myList)
64
65 freqTable objList = myTable
66    where myTable = zip singleList myCount
67          singleList = list2set objList
68          myCount = zipWith (count) singleList (repeatList (length singleList) objList)
69
```

```
*Main> count 'e' "need more coffee"
5
*Main> count 4 [1,2,3,2,3,4,3,4,5,4,5,6]
3
*Main> freqTable "need more coffee"
[('n',1),('e',5),('d',1),(' ',2),('m',1),('o',2),('r',1),('c',1),('f',2)]
*Main> freqTable [1,2,3,2,3,4,3,4,5,4,5,6]
[(1,1),(2,2),(3,3),(4,3),(5,2),(6,1)]
*Main> :quit
Leaving GHCi.
```

Task 6 - Higher Order Functions

This task requires that you write 4 function definitions that feature higher order programming, and that you then demo them by creating a “proper” demo. For this task, please add to your presentation document (1) a text containing the 4 function definitions, and (2) a text containing the demo that you are expected to create. What is a proper demo with

respect to this task? Run each of the 4 functions with the applications that I provide in my sample demo, and then add 2 applications of your own invention. Thus, your proper demo will have 4 applications for each of the 4 functions.

Function specifications

Please refer to the demo for clarification on these specifications, as needed.

1. Define a function called tgl, taking a positive Int value, which returns the triangular number corresponding to the given value. That is, it returns the sum of the numbers from 1 to the given value. **Constraint: Do so using the** foldl **function.** (Do **not** use the sum function.)
2. Define a function called triangleSequence, taking a positive Int value, which returns the list of triangular numbers from 1 to the given number. **Constraint: Do so using the** map **function, along with the** tgl **function.**
3. Define a function called vowelCount, taking a string of lower case letters, which returns the number of vowels in the given string. **Constraint: Do so using the** filter **function, along with a lambda function of your own design which returns True only if a given character is a lower case vowel.**
4. Using the map function and the filter function, define a function called lcsim (for "list comprehension simulation") taking three parameters, a function for mapping, a predicate for filtering, and a list of elements, which returns the same value as the following list comprehension:

   [f x | x <- xs, p x]

The given demo that you are to augment

```
70 ------------------------------------------------------------
71 ------------------------ TASK 6 ----------------------------
72 ------------------------------------------------------------
73
74 tgl 0 = 0
75 tgl input = foldl (+) (input) [(tgl (input - 1))]
76
77 triangleSequence input = (map tgl [0,1..input])
78
79 vowelCount myString = length (filter ( \x -> x `elem` ['a','e','i','o','u'])myString)
80
81 lcsim function mapping elements = map function (filter mapping elements)
82
```

```
Ok, one module loaded.
*Main> tgl 5
15
*Main> tgl 10
55
*Main> triangleSequene 10

<interactive>:4:1: error:
    • Variable not in scope: triangleSequene :: Integer -> t
    • Perhaps you meant 'triangleSequence' (line 77)
*Main> triangleSequence 10
[0,1,3,6,10,15,21,28,36,45,55]
*Main> triangleSequence 20
[0,1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210]
*Main> vowelCount "cat"
1
*Main> vowelCount "mouse"
3
*Main> lcsim tgl odd [1..15]
[1,6,15,28,45,66,91,120]
*Main>  animals = ["elephant","lion","tiger","orangatan","jaguar"]
*Main> lcsim length (\w -> elem ( head w ) "aeiou") animals
[8,9]
*Main>
```

Task 7 - An Interesting Statistic: nPVI

This tasks invites you to implement the "normalized pairwise variability index" (nPVI) by making good use of zip and map. This statistic has been used extensively in the field of lingustics, and is also used to significant effect in the field of music cognition. In case you find yourself with a bit of time, and the inclination to see an impressive application of nPVI, you might like to spend some time with the following paper:

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1063.772&rep=rep1&type=pdf

What is the nPVI? As the name implies, it is a measure of the pairwise variability of terms in a sequence of numeric terms. In mathematical notation, the nPVI is defined by the following expression:

$$nPVI = \left(\frac{100}{m-1}\right)\sum_{k=1}^{m-1}\left|\frac{d_k - d_{k+1}}{(d_k + d_{k+1})/2}\right|$$

Should that seem like a lot to unpack, no worries, the plan is for you to reconstruct the nPVI expression in Haskell by writing a sequence of functions that are consistent with the obvious deconstruction of the expression, the last of which actually computes the nPVI for a sequence of integral values.

Please be aware of the fact that, for this little exercise in Haskell programming, the type of a function will habitually be expressed prior to function definition, and, moreover, the type for each function will be very narrowly construed.

Task 7a - Test data

Please establish a file called npvi.hs within which to place your code for this task. Then, add a reasonable opening comment to your file.

Please add the following lines of code to your file, with ease of testing in mind. And then, load the file, and make sure that the variables are properly bound. Add this bit of demo to your presentation document.

```
-----
--a--
-----

-- Test data
a :: [Int]
a = [2,5,1,3]

b :: [Int]
b = [1,3,6,2,5]

c :: [Int]
c = [4,4,2,1,1,2,2,4,4,8]

u :: [Int]
u = [2,2,2,2,2,2,2,2,2,2]

x :: [Int]
x = [1,9,2,8,3,7,2,8,1,9]
```

```
[1 of 1] Compiling Main             ( npvi.hs, interpreted )
Ok, one module loaded.
*Main> a
[2,5,1,3]
*Main> b
[1,3,6,2,5]
*Main> c
[4,4,2,1,1,2,2,4,4,8]
*Main> u
[2,2,2,2,2,2,2,2,2,2]
*Main> x
[1,9,2,8,3,7,2,8,1,9]
*Main>
```

Task 7b - The pairwiseValues function

Write the function called pairwiseValues, taking a list of Int values as its sole parameter, which produces a list of pairs of Int values, such that each element of the given list is paired with its successor. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the zip function and the tail function from the standard prelude, (3) keep your code (excluding the type declaration) to just one line.

```
-----
--b--
-----

pairwiseValues :: [Int] -> [(Int,Int)]
pairwiseValues myList = zip myList (tail myList)

-----
```

```
*Main> pairwiseValues a
[(2,5),(5,1),(1,3)]
*Main>  pairwiseValues b
[(1,3),(3,6),(6,2),(2,5)]
*Main> pairwiseValues c
[(4,4),(4,2),(2,1),(1,1),(1,2),(2,2),(2,4),(4,4),(4,8)]
*Main> pairwiseValues u
[(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2)]
*Main> pairwiseValues x
[(1,9),(9,2),(2,8),(8,3),(3,7),(7,2),(2,8),(8,1),(1,9)]
*Main>
```

Task 7c - The pairwiseDifferences function

Write the function called pairwiseDifferences, taking a list of Int values as its sole parameter, which produces a list Int values consisting of pairwise differences of each element in the list with its successor. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the map function together with the lambda function (\(x,y) -> x - y) and your previously written pairwiseValues function, (3) keep your code (excluding the type declaration) to just one line.

```
28 -----
29 --c--
30 -----
31
32 pairwiseDifferences :: [Int] -> [Int]
33 pairwiseDifferences myList = map ( \(x,y) -> x - y ) (pairwiseValues myList)
34
```

```
*Main> pairwiseDifferences a
[-3,4,-2]
*Main> pairwiseDifferences b
[-2,-3,4,-3]
*Main> pairwiseDifferences c
[0,2,1,0,-1,0,-2,0,-4]
*Main> pairwiseDifferences u
[0,0,0,0,0,0,0,0,0]
*Main> pairwiseDifferences x
[-8,7,-6,5,-4,5,-6,7,-8]
*Main>
```

Task 7d - The pairwiseSums function

Write the function called pairwiseSums, taking a list of Int values as its sole parameter, which produces a list Int values consisting of pairwise sums of each element in the list with its successor. Please (1) place the type of this function in

your file prior to your code which defines the function, (2) make good use of the map function together with the appropriate lambda function and your previously written pairwiseValues function, (3) keep your code (excluding the type declaration) to just one line.

```
35 -----
36 --d--
37 -----
38
39 pairwiseSums :: [Int] -> [Int]
40 pairwiseSums myList = map ( \(x,y) -> x + y ) (pairwiseValues myList)
41
42 -----
```

```
*Main>  pairwiseSums a
[7,6,4]
*Main>  pairwiseSums b
[4,9,8,7]
*Main>  pairwiseSums c
[8,6,3,2,3,4,6,8,12]
*Main>  pairwiseSums u
[4,4,4,4,4,4,4,4,4]
*Main>  pairwiseSums x
[10,11,10,11,10,9,10,9,10]
*Main>
```

Task 7e - The pairwiseHalves function

In preparation for defining the featured function of this task, add the following lines of code to your file, and then test the half function.

```
half :: Int -> Double half number = ( fromIntegral
number ) / 2
```

Write the function called pairwiseHalves, taking a list of Int values as its sole parameter, which produces a list Double values by dividing each value in the input list by 2. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the map function together with the half function, (3) keep your code (excluding the type declaration) to just one line.

```
-----
--e--
-----

half :: Int -> Double
half number = ( fromIntegral number ) / 2

pairwiseHalves :: [Int] -> [Double]
pairwiseHalves myList = map half myList
```

```
*Main> pairwiseHalves [1..10]
[0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
*Main> pairwiseHalves u
[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0]
*Main> pairwiseHalves x
[0.5,4.5,1.0,4.0,1.5,3.5,1.0,4.0,0.5,4.5]
*Main>
```

Task 7f - The pairwiseHalfSums function

Write the function called pairwiseHalfSums, taking a list of Int values as its sole parameter, which produces a list Double values by dividing each pairwise sum by 2. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the pairwiseSums function and the pairwiseHalves function, (3) keep your code (excluding the type declaration) to just one line.

```
52 -----
53 --f--
54 -----
55
56 pairwiseHalfSums :: [Int] -> [Double]
57 pairwiseHalfSums myList = pairwiseHalves (pairwiseSums myList)
58
```

```
*Main> pairwiseHalfSums a
[3.5,3.0,2.0]
*Main> pairwiseHalfSums b
[2.0,4.5,4.0,3.5]
*Main> pairwiseHalfSums c
[4.0,3.0,1.5,1.0,1.5,2.0,3.0,4.0,6.0]
*Main> pairwiseHalfSums u
[2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0]
*Main> pairwiseHalfSums x
[5.0,5.5,5.0,5.5,5.0,4.5,5.0,4.5,5.0]
*Main>
```

Task 7g - The pairwiseTermPairs function

Write the function called pairwiseTermPairs, taking a list of Int values as its sole parameter, which produces a list pairs corresponding to the numerators/denominator in the summation of the nPVI formula. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the zip function, and the pairwiseDifference along with the pairwiseHalfSums function, (3) keep your code (excluding the type declaration) to just one line.

```
-----
--g--
-----

pairwiseTermPairs :: [Int] -> [(Int,Double)]
pairwiseTermPairs myList = zip (pairwiseDifferences myList) (pairwiseHalfSums myList)

-----
```

```
*Main>  pairwiseTermPairs a
[(-3,3.5),(4,3.0),(-2,2.0)]
*Main>  pairwiseTermPairs b
[(-2,2.0),(-3,4.5),(4,4.0),(-3,3.5)]
*Main> pairwiseTermPairs c
[(0,4.0),(2,3.0),(1,1.5),(0,1.0),(-1,1.5),(0,2.0),(-2,3.0),(0,4.0),(-4,6.0)]
*Main> pairwiseTermPairs u
[(0,2.0),(0,2.0),(0,2.0),(0,2.0),(0,2.0),(0,2.0),(0,2.0),(0,2.0),(0,2.0)]
*Main> pairwiseTermPairs x
[(-8,5.0),(7,5.5),(-6,5.0),(5,5.5),(-4,5.0),(5,4.5),(-6,5.0),(7,4.5),(-8,5.0)]
*Main>
```

ion

Task 7h - The pairwiseTerms function

In preparation for defining the featured function of this task, add the following lines of code to your file, and then test the term function, which simply transforms a given "(numerator,denomenator) pair" into an evaluated term for the summation operation.

term :: (Int,Double) -> Double term ndPair = abs (fromIntegral (fst ndPair) / (snd ndPair))

Write the function called pairwiseTerms, taking a list of Int values as its sole parameter, which produces a list Double values corresponding to the terms in the summation of the nPVI formula. Please (1) place the type of this function in your file prior to your code which defines the function, (2) make good use of the map function together with the term function and the pairwiseTermPairs function, (3) keep your code (excluding the type declaration) to just one line.

```
66 -----
67 --h--
68 -----
69
70 term :: (Int,Double) -> Double
71 term ndPair = abs ( fromIntegral ( fst ndPair ) / ( snd ndPair ) )
72
73 pairwiseTerms :: [Int] -> [Double]
74 pairwiseTerms myList = map term (pairwiseTermPairs myList)
75
```

```
*Main> pairwiseTerms a
[0.8571428571428571,1.3333333333333333,1.0]
*Main> pairwiseTerms b
[1.0,0.6666666666666666,1.0,0.8571428571428571]
*Main> pairwiseTerms c
[0.0,0.6666666666666666,0.6666666666666666,0.0,0.6666666666666666,0.0,0.6666666666666666,0.0,0.6666666666666666]
*Main> pairwiseTerms u
[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
*Main> pairwiseTerms x
[1.6,1.2727272727272727,1.2,0.9090909090909091,0.8,1.1111111111111112,1.2,1.5555555555555556,1.6]
*Main>
```

Task 7i - The nPVI function

Simply incorporate my code for the nPVI type declaration and function definition into your file.

```
76 -----
77 --i--
78 -----
79
80 nPVI :: [Int] -> Double
81 nPVI xs = normalizer xs * sum ( pairwiseTerms xs )
82    where normalizer xs = 100 / fromIntegral ( ( length xs ) - 1 )
83
```

```
*Main> nPVI a
106.34920634920636
*Main> nPVI b
88.09523809523809
*Main> nPVI c
37.03703703703703
*Main> nPVI u
0.0
*Main> nPVI x
124.98316498316497
*Main>
```

Task 8 - Historic Code: The Dit Dah Code

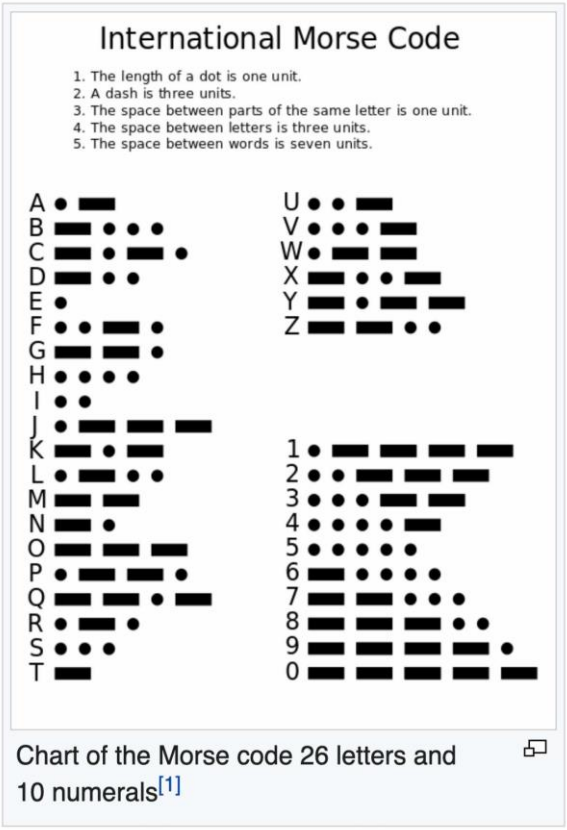

Chart of the Morse code 26 letters and 10 numerals[1]

This task does not require that you write function definitions. Rather, it asks you to read some code, display some variable bindings, and write expressions to illuminate the behavior of a collection of functions.

Haskell programmers seem to enjoy playing with famous codes when showcasing the language. No matter that the Caesar cipher is mostly thought of as a cognitive toy of some historical interest. It still appears as a programming example in a number of texts devoted to learning to program in Haskell. The present task honors another historically significant code, one that once served as a very useful technology, Morse code.

The idea is for you to download a file called ditdah.hs, study it, load it into a Haskell process, and perform the following subtasks. By doing so, perhaps you will learn a little something more about Haskell programming.

Please incorporate your successfull interactions into just one complete demo, and include the demo in your presentation document.

Subtask 8a

```
Prelude> :load ditdah.hs
[1 of 1] Compiling Main             ( ditdah.hs, interpreted )
Ok, one module loaded.
*Main> :t dit
dit :: [Char]
*Main> :t dah
dah :: [Char]
*Main> "tell"+++"me"
"tell me"
*Main> :t m
m :: (Char, [Char])
*Main> :t g
g :: (Char, [Char])
*Main> :t h
h :: (Char, [Char])
*Main> :t symbols
symbols :: [(Char, [Char])]
*Main>
```

Subtask 8b

```
*Main> assoc 'b' symbols
('b',"--- - - -")
*Main> assoc 'x' symbols
('x',"--- - - ---")
*Main> find 'b'
"--- - - -"
*Main> find 'x'
"--- - - ---"
*Main>
```

Subtask 8c

```
*Main> addletter "b" "v"
"b   v"
*Main> addword  "b" "v"
"b       v"
*Main> droplast3 "drop the last 3 of this"
"drop the last 3 of t"
*Main> droplast7 "drop the last 7 of this"
"drop the last 7 "
*Main>
```

Subtask 8d

```
*Main> :t encodeletter 'm'
encodeletter 'm' :: [Char]
*Main> encodeletter 'm'
"--- ---"
*Main> encodeletter 'g'
"--- --- -"
*Main> encodeword "yay"
"--- - --- ---   - ---   --- - --- ---"
*Main> :t encodeword "yay"
encodeword "yay" :: [Char]
*Main> encodeword "word"
"- --- ---   --- --- ---   - --- -   --- - -"
*Main> encodeword "yay"
"--- - --- ---   - ---   --- - --- ---"
*Main> :t en
encodeFloat     encodemessage   enumFrom        enumFromThenTo
encodeletter    encodeword      enumFromThen    enumFromTo
*Main> :t encodemessage "need more coffee"
encodemessage "need more coffee" :: [Char]
*Main> encodemessage "billy bob"
"--- - - -   - -   - --- - -   - --- - -   --- - --- ---       --- - - -   --- --- ---   --- -
 - -"
*Main>
*Main> encodemessage "howdy"
"- - - -   --- --- ---   - --- ---   --- - -   --- - --- ---"
*Main>
```

Due Date

Please complete your work on this assignment, and post your work to your web work site, by sometime on Friday, December 10, 2021.