
Haskell Programming Assignment: Various Computations

What's It All About?

Programming exercises that focus on functions, recursive list processing, list comprehensions, and higher order functions in Haskell.

Tasks

1. Working with GHCi, in a reasonable text editor, perform the 8 main tasks.
2. Craft a nicely structured document that contains representations of each of the 8 tasks that you are asked to do. Moreover, be sure to title the document, and place a “learning abstract” just after the title.
3. Post your document to your web work site.

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.

```
bash$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :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"]
"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.
bash$

```

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 as 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 as 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 $n \times n \times n$ 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

```
bash$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main                ( ha.hs, interpreted )
Ok, one module loaded.
>>> squareArea 10
100
>>> squareArea 12
144
>>> circleArea 10
314.1592653589793
>>> circleArea 12
452.3893421169302
>>> blueAreaOfCube 10
482.19027549038276
>>> blueAreaOfCube 12
694.3539967061512
>>> blueAreaOfCube 1
4.821902754903828
>>> map blueAreaOfCube [1..3]
[4.821902754903828,19.287611019615312,43.39712479413445]
>>> paintedCube1 1
0
>>> paintedCube1 2
0
>>> paintedCube1 3
6
>>> map paintedCube1 [1..10]
[0,0,6,24,54,96,150,216,294,384]
>>> paintedCube2 1
0
>>> paintedCube2 2
0
>>> paintedCube2 3
12
>>> map paintedCube2 [1..10]
[0,0,12,24,36,48,60,72,84,96]
>>> :quit
Leaving GHCi.
bash$
```

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

Please refer to the demo for clarification on these specifications, as needed. And, in writing these functions, please judiciously use a number of the functions that were featured in the first task of this programming challenge!

1. Define a function called `reverseWords`, taking one character string of words as its sole parameter, which returns a string containing those same words in reverse order.
2. Define a function called `averageWordLength`, taking one character string of words as its sole parameter, which returns the real number average word length of the words.

The given demo that you are to augment

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main                ( ha.hs, interpreted )
Ok, one module loaded.
>>> reverseWords "appa and baby yoda are the best"
"best the are yoda baby and appa"
>>> reverseWords "want me some coffee"
"coffee some me want"
>>> averageWordLength "appa and baby yoda are the best"
3.5714285714285716
>>> averageWordLength "want me some coffee"
4.0
>>> :quit
Leaving GHCi.
bash-3.2$
```

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

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main                ( ha.hs, interpreted )
Ok, one module loaded.
>>> list2set [1,2,3,2,3,4,3,4,5]
[1,2,3,4,5]
>>> list2set "need more coffee"
"ndmr cofe"
>>> isPalindrome ["coffee","latte","coffee"]
True
>>> isPalindrome ["coffee","latte","espresso","coffee"]
False
>>> isPalindrome [1,2,5,7,11,13,11,7,5,3,2]
False
>>> isPalindrome [2,3,5,7,11,13,11,7,5,3,2]
True
>>> collatz 10
[10,5,16,8,4,2,1]
>>> collatz 11
[11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
>>> collatz 100
[100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
>>> :quit
Leaving GHCi.
bash-3.2$
```

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

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main                ( ha.hs, interpreted )
Ok, one module loaded.
>>> count 'e' "need more coffee"
5
>>> count 4 [1,2,3,2,3,4,3,4,5,4,5,6]
3
>>> freqTable "need more coffee"
[( 'n',1),('d',1),('m',1),('r',1),(' ',2),('c',1),('o',2),('f',2),('e',5)]
>>> 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)]
>>> :quit
Leaving GHCi.
bash-3.2$
```

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

```
>>> tgl 5
15
>>> tgl 10
55
>>> triangleSequence 10
[1,3,6,10,15,21,28,36,45,55]
>>> triangleSequence 20
[1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210]
>>> vowelCount "cat"
1
>>> vowelCount "mouse"
3
>>> lcsim tgl odd [1..15]
[1,6,15,28,45,66,91,120]
>>> animals = ["elephant","lion","tiger","orangutan","jaguar"]
>>> lcsim length (\w -> elem ( head w ) "aeiou") animals
[8,9]
>>>
```

Task 7 - An Interesting Statistic: nPVI

This task 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 linguistics, 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.

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

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

c :: [Int]
c =

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]
```

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.

The type declaration that you are expected to include in your file?

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

Please mimic the following demo, and add to your presentation document:

```
>>> pairwiseValues a
[(2,5),(5,1),(1,3)]
>>> pairwiseValues b
[(1,3),(3,6),(6,2),(2,5)]
```



```
>>> pairwiseValues c
[(4,4),(4,2),(2,1),(1,1),(1,2),(2,2),(2,4),(4,4),(4,8)]
>>> pairwiseValues u
[(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2),(2,2)]
>>> pairwiseValues x
[(1,9),(9,2),(2,8),(8,3),(3,7),(7,2),(2,8),(8,1),(1,9)]
>>>
```

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.

The type declaration that you are expected to include in your file?

```
pairwiseDifferences :: [Int] -> [Int]
```

Please mimic the following demo, and add to your presentation document:

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

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.

The type declaration that you are expected to include in your file?

```
pairwiseSums :: [Int] -> [Int]
```

Please mimic the following demo, and add to your presentation document:

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

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.

The type declaration that you are expected to include in your file?

```
pairwiseHalves :: [Int] -> [Double]
```

Please mimic the following demo, and add to your presentation document:

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

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.

The type declaration that you are expected to include in your file?

```
pairwiseHalfSums :: [Int] -> [Double]
```

Please mimic the following demo, and add to your presentation document:

```
>>> :reload
[1 of 1] Compiling Main                ( nPVI.hs, interpreted )
Ok, one module loaded.
>>> pairwiseHalfSums a
[3.5,3.0,2.0]
>>> pairwiseHalfSums b
[2.0,4.5,4.0,3.5]
>>> pairwiseHalfSums c
[4.0,3.0,1.5,1.0,1.5,2.0,3.0,4.0,6.0]
>>> pairwiseHalfSums u
[2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0]
>>> pairwiseHalfSums x
[5.0,5.5,5.0,5.5,5.0,4.5,5.0,4.5,5.0]
>>>
```

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.

The type declaration that you are expected to include in your file?

```
pairwiseTermPairs :: [Int] -> [(Int,Double)]
```

Please mimic the following demo, and add to your presentation document:

```
>>> pairwiseTermPairs a
[(-3,3.5),(4,3.0),(-2,2.0)]
>>> pairwiseTermPairs b
[(-2,2.0),(-3,4.5),(4,4.0),(-3,3.5)]
>>> 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)]
>>> 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)]
>>> 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)]
>>>
```

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,denominator) 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.

The type declaration that you are expected to include in your file?

```
pairwiseTerms :: [Int] -> [Double]
```

Please mimic the following (slightly edited) demo, and add to your presentation document:

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

Task 7i - The nPVI function

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

```
nPVI :: [Int] -> Double
nPVI xs = normalizer xs * sum ( pairwiseTerms xs )
  where normalizer xs = 100 / fromIntegral ( ( length xs ) - 1 )
```

Load the file, mimic the following demo, and add to your presentation document:

```
>>> nPVI a
106.34920634920636
>>> nPVI b
88.09523809523809
>>> nPVI c
```

```
37.03703703703703
>>> nPVI u
0.0
>>> nPVI x
124.98316498316497
>>>
```

Task 8 - Historic Code: The Dit Dah Code

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

| | | | |
|---|---------|---|-----------|
| A | • — | U | • • — |
| B | — • • • | V | • • • — |
| C | — • — • | W | • — — |
| D | — • • | X | — • • — |
| E | • | Y | — • — — |
| F | • • — • | Z | — — • • |
| G | — — • | | |
| H | • • • • | | |
| I | • • | | |
| J | • — — — | | |
| K | — • — | | |
| L | • — • • | | |
| M | — — | | |
| N | — • | | |
| O | — — — | | |
| P | • — — • | | |
| Q | — — • — | | |
| R | • — • | | |
| S | • • • | | |
| T | — | | |
| | | | |
| | | 1 | • — — — — |
| | | 2 | • • — — — |
| | | 3 | • • • — — |
| | | 4 | • • • • — |
| | | 5 | • • • • • |
| | | 6 | — • • • • |
| | | 7 | — — • • • |
| | | 8 | — — — • • |
| | | 9 | — — — — • |
| | | 0 | — — — — — |

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 successful interactions into just one complete demo, and include the demo in your presentation document.

Subtask 8a

1. Ask Haskell to evaluate `dit`.
2. Ask Haskell to evaluate `dah`.
3. Figure out how to use the infix binary operator `+++`, and apply it once.
4. Ask Haskell to evaluate `m`.
5. Ask Haskell to evaluate `g`.
6. Ask Haskell to evaluate `h`.
7. Ask Haskell to evaluate `symbols`.

Subtask 8b

1. Study the functions `assoc` and `find`, until you have an idea about what they do. The `assoc` function is more than a half century old, transcending languages with its significance, regardless of syntactic variation. The `find` function is a convenience function that makes good use of the `assoc` function.
2. Twice, apply the `assoc` function with a lowercase letter as first argument and the value of `symbol` as its second argument.
3. Twice, with a different lowercase letter as input each time, apply the `find` function.

Subtask 8c

1. Find a way to apply the `addletter` function.
2. Find a way to apply the `addword` function.
3. Find a way to apply the `droplast3` function.
4. Find a way to apply the `droplast7` function.

Subtask 8d

1. Ask Haskell to evaluate `encodeletter 'm'`.
2. Two more times, each time with a different argument, apply the `encodeletter` function.
3. Ask Haskell to evaluate `encodeword "yay"`.
4. Two more times, each time with a different argument, apply the `encodeword` function.
5. Ask Haskell to evaluate `encodemessage "need more coffee"`.
6. Two more times, each time with a different argument, apply the `encodemessage` function.

Due Date

Please complete your work on this assignment, and post your work to your web work site, by sometime on Friday, May 5, 2023.