Haskell Programming Assignment: Various Computations

Learning Abstract

This is an introduction to Haskell. In this assignment we cover some built-in functions, write some numeric processing functions and preforming some String and List manipulations. We also cover HoF's and Lambda expressions.

Task 1: Mindfully Mimicking the Demo

```
>>>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?"
>>>:q
Leaving GHCi.
```

Task 2: Numeric Function Definitions

```
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- ha.hs
-- Numeric Function Definitions
-- Haskell Assignment 1 Task 2
-------------------------------------------------------------------------

--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.

squareArea :: Float -> Float
squareArea x = x * x

--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.

circleArea :: Float -> Float
circleArea x = pi * x * x

--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.

blueAreaOfCube :: Float -> Float
blueAreaOfCube x = (6 * squareArea x ) - (6 * circleArea (x/4))

--Imagine that a wooden cube is dissected into nxnxn 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.

paintedCube1 :: Int -> Int
paintedCube1 x = if (x < 3) then 0 else 6 * ((x - 2) * (x - 2))

--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.

paintedCube2 :: Int -> Int
paintedCube2 x = if (x < 3) then 0 else 12 * (x - 2)
```

```
C:\Users\stick\Documents\CSC344\Haskell Stuff>ghci
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
ghci> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main             ( ha.hs, interpreted )
Ok, one module loaded.
>>> blueAreaOfCube 10
482.19028
>>> squareArea 10
100.0
>>> squareArea 12
144.0
>>> circleArea 10
314.15927
>>> circleArea 12
452.38934
>>> blueAreaOfCube 10
482.19028
>>> blueAreaOfCube 12
694.354
>>> blueAreaOfCube 1
4.8219028
>>> map blueAreaOfCube [1..3]
[4.8219028,19.287611,43.397125]
```

```
>>> 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]
>>> :q
Leaving GHCi.
```

Task 3: Puzzlers

```haskell
--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.

reverseWords :: String -> String
reverseWords x = unwords (reverse (words x))

--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.

averageWordLength :: String -> Float
averageWordLength x = (fromIntegral (sum (map length (words x)))) /
(fromIntegral (length (words x)))
```

```
>>> :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"
>>> reverseWords "once in a blue moon"
"moon blue a in once"
>>> reverseWords "a storm in a teacup"
"teacup a in storm a"
>>> averageWordLength "appa and baby yoda are the best"
3.5714285
>>> averageWordLength "want me some coffee"
4.0
>>> averageWordLength "once in a blue moon"
3.0
>>> averageWordLength "a storm in a teacup"
3.0
>>> :q
Leaving GHCi.
```

Task 4: Recursive List Processors

```haskell
--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.

list2set :: Eq a => [a] -> [a]
list2set [] = []
list2set (x:xs) = if (elem x xs) then list2set xs else x : list2set xs

--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).

isPalindrome :: Eq a => [a] -> Bool
isPalindrome [] = True
isPalindrome (x:[]) = True
isPalindrome (x:xs) = if (x == last xs) then isPalindrome (init xs) else
False


--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.

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

```
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
ghci> :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]
>>> :q
Leaving GHCi.
```

Task 5: List Comprehensions

```haskell
--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.

count :: Eq a => a -> [a] -> Int
count x xs = length [y | y <- xs, y == x]

--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.

freqTable :: Eq a => [a] -> [(a, Int)]
freqTable xs = [(x, count x xs) | x <- list2set xs]
```

```
>>> count 'e' "need more coffee"
5
>>> count 'f' "need more coffee"
2
>>> count 4 [1,2,3,2,3,4,3,4,5,4,5,6]
3
>>> count 3 [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 "once in a blue moon"
[('c',1),('i',1),('a',1),('b',1),('l',1),('u',1),('e',2),(' ',4),('m',1),('o',3),('n',3)]
>>> 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)]
>>> freqTable [1,2,5,7,3,8,5,9,2,7]
[(1,1),(3,1),(8,1),(5,2),(9,1),(2,2),(7,2)]
>>> :q
Leaving GHCi.
```

Task 6: Higher Order Functions

```haskell
--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.)

tgl :: Int -> Int
tgl x = foldl (+) 0 [1..x]

--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.

triangleSequence :: Int -> [Int]
triangleSequence x = map tgl [1..x]

--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.

vowelCount :: String -> Int
vowelCount x = length (filter (\y -> y `elem` "aeiou") x)

--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]

lcsim :: (a -> b) -> (a -> Bool) -> [a] -> [b]
lcsim f p xs = map f (filter p xs)
```

```
C:\Users\stick\Documents\CSC344\Haskell Stuff>ghci
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
ghci> :set prompt ">>> "
>>> :load ha
[1 of 1] Compiling Main             ( ha.hs, interpreted )
Ok, one module loaded.
>>> tgl 5
15
>>> tgl 10
55
>>> tgl 20
210
>>> tgl 7
28
>>> 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]
>>> triangleSequence 12
[1,3,6,10,15,21,28,36,45,55,66,78]
>>> triangleSequence 18
[1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171]
>>> vowelCount "cat"
1
>>> vowelCount "mouse"
3
>>> vowelCount "trumpet"
2
>>> vowelCount "saxophone"
4
>>> lcsim tgl odd [1..15]
[1,6,15,28,45,66,91,120]
>>> animals = ["elephant","lion","tiger","orangatan","jaguar"]
>>> lcsim length (\w -> elem ( head w ) "aeiou") animals
[8,9]
>>> lcsim tgl even [1..15]
[3,10,21,36,55,78,105]
>>> pie = ["apple","cherry","pumpkin","key-lime","pecan"]
>>> lcsim length (\w -> elem ( head w ) "aeiou") pie
[5]
```

Task 7: An Interesting Statistic: nPVI

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- npvi.hs
-- an implementation of the "normalized pairwise variability index"
-- (nPVI)
-- Haskell Assignment 1 Task 7
--------------------------------------------------------------------------

-- 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]

{-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.-}

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

{-Write the function called pairwiseDifferences, taking a list of Int values
as its sole parameter, which produces a list
  of Int values, such that each element of the list is the absolute value of
the difference between the corresponding
  elements of the given list and 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.-}

pairwiseDifferences :: [Int] -> [Int]
pairwiseDifferences x = map ( \(x,y) -> x - y ) (pairwiseValues x)

{-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.-}

pairwiseSums :: [Int] -> [Int]
pairwiseSums x = map ( \(x,y) -> x + y ) (pairwiseValues x)

{-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.-}

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

pairwiseHalves :: [Int] -> [Double]
pairwiseHalves x = map half x

{-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.-}

pairwiseHalfSums :: [Int] -> [Double]
pairwiseHalfSums x = pairwiseHalves (pairwiseSums x)

{-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.-}

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

{-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.-}

term :: (Int,Double) -> Double
```

```haskell
term ndPair = abs ( fromIntegral ( fst ndPair ) / ( snd ndPair ) )

pairwiseTerms :: [Int] -> [Double]
pairwiseTerms x = map term (pairwiseTermPairs x)

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

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

```
>>> 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]
```

```
>>> 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]
>>> 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]
>>> 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
[2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0]
>>> pairwiseHalfSums x
[5.0,5.5,5.0,5.5,5.0,4.5,5.0,4.5,5.0]
```

```
>>> 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),(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)]
>>> 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]
>>> pairwiseTerms x
[1.6,1.2727272727272727,1.2,0.9090909090909091,0.8,1.1111111111111112,1.2,1.5555555555555556,
1.6]
```

```
>>> 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

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- File: ditdah.hs
-- Data: Late November, 2021
-- Line: Simulation of a Morse code encoder.

--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- Code to build the dictionary of Morse code symbols for the lower
-- case letters.

--------------------------------------------------------------------------
-- Names for the representation of "dot" (spoken dit) and "dash"
-- (spoken dah)

dit = "-"
dah = "---"

--------------------------------------------------------------------------
-- A special version of append. It will add one space between two
-- strings in a binary string append function.

(+++) x y = x ++ " " ++ y

--------------------------------------------------------------------------
-- Entries for a dictionary of Morse code symbols corresponding to the
-- lower case letters.

a = ('a',dit+++dah)
b = ('b',dah+++dit+++dit+++dit)
c = ('c',dah+++dit+++dah+++dit)
d = ('d',dah+++dit+++dit)
e = ('e',dit)
f = ('f',dit+++dit+++dah+++dit)
g = ('g',dah+++dah+++dit)
h = ('h',dit+++dit+++dit+++dit)
i = ('i',dit+++dit)
j = ('j',dit+++dah+++dah+++dah)
k = ('k',dah+++dit+++dah)
l = ('l',dit+++dah+++dit+++dit)
m = ('m',dah+++dah)
n = ('n',dah+++dit)
o = ('o',dah+++dah+++dah)
p = ('p',dit+++dah+++dah+++dit)
q = ('q',dah+++dah+++dit+++dah)
r = ('r',dit+++dah+++dit)
s = ('s',dit+++dit+++dit)
t = ('t',dah)
u = ('u',dit+++dit+++dah)
v = ('v',dit+++dit+++dit+++dah)
w = ('w',dit+++dah+++dah)
x = ('x',dah+++dit+++dit+++dah)
```

```
y = ('y',dah+++dit+++dah+++dah)
z = ('z',dah+++dah+++dit+++dit)

-------------------------------------------------------------------------
-- The dictionary for Morse code symbols corresponding to lower case
-- letters.

symbols = [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- Mindful of the fact that the symbols variable is an association
-- list, assoc performs association list lookup, and is used in a
-- function to find the Morse code string corresponding to a particular
-- lower case letter.

assoc key alist = head [ (k,v) | (k,v) <- alist, k == key ]

find letter = snd $ assoc letter symbols

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- Helping functions for the Morse code encoders for a letter, a word,
-- and a message.

addletter x y = x ++ "   " ++ y

addword x y = x ++ "       " ++ y

droplast3 w = reverse ( drop 3 ( reverse w ) )

droplast7 w = reverse ( drop 7 ( reverse w ) )

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- Morse code encoders for a letter, a word, and a message.

encodeletter x = find x -- snd ( assoc x symbols )

encodeword w = droplast3 almost
   where almost = foldr addletter "" ( map encodeletter w )

encodemessage m = droplast7 almost
   where almost = foldr addword "" ( map encodeword ( words m ) )
```

```
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
ghci> :set prompt ">>> "
>>> :load ditdah.hs
[1 of 1] Compiling Main             ( ditdah.hs, interpreted )
Ok, one module loaded.
>>> dit
"-"
>>> dah
"---"
>>> dah +++ dit
"--- -"
>>> m
('m',"--- ---")
>>> g
('g',"--- --- -")
>>>
>>> h
('h',"- - - -")
>>> symbols
[('a',"- ---"),('b',"--- - - -"),('c',"--- - --- -"),('d',"--- - -"),('e',"-"),('f',"- - ---
-"),('g',"--- --- -"),('h',"- - - -"),('i',"- -"),('j',"- --- --- ---"),('k',"--- - ---"),('l
',"- --- - -"),('m',"--- ---"),('n',"--- -"),('o',"--- --- ---"),('p',"- --- --- -"),('q',"--
- --- - ---"),('r',"- --- -"),('s',"- - -"),('t',"---"),('u',"- - ---"),('v',"- - - ---"),('w
',"- --- ---"),('x',"--- - - ---"),('y',"--- - --- ---"),('z',"--- --- - -")]
```

```
>>> assoc 'a' symbols
('a',"- ---")
>>> assoc 'b' symbols
('b',"--- - - -")
>>> find 'x'
"--- - - ---"
>>> find 'z'
"--- --- - -"
```

```
>>> addletter "J" "ava"
"J   ava"
>>> addword "hello" "world"
"hello       world"
>>> droplast3 "hello"
"he"
>>> droplast7 "hello world"
"hell"
```

```
>>> encodeletter 'm'
"--- ---"
>>> encodeletter 'c'
"--- - --- -"
>>> encodeletter 's'
"- - -"
>>> encodeword "yay"
"--- - --- ---   - ---   --- - --- ---"
>>> encodeword "hello"
"- - - -   -   - --- - -   - --- - -   --- --- ---"
>>> encodeword "world"
"- --- ---   --- --- ---   - --- -   - --- - -   --- - -"
>>> encodemessage "need more coffee"
"--- -   -   -   --- - -       --- ---   --- --- ---   - --- -   -       --- - --- -   --- --
- ---   - - --- -   - - --- -   -   -"
>>> encodemessage "once in a blue moon"
"--- --- ---   --- -   --- - --- -   -       - -   --- -       - ---       --- - - -   - ---
- -   - - ---   -       --- ---   --- --- ---   --- --- ---   --- -"
>>> encodemessage "a storm in a teacup"
"- ---       - - -   ---   --- --- ---   - --- -   --- ---       - -   --- -       - ---
  ---   -   - ---   --- - --- -   - - ---   - --- --- -"
```