ECM2418 Computer Languages and Representations
Continuous Assessment 1: Functional Programming

Dr David Wakeling

Handed out	Handed in
Monday 8th October 2018 (T1:03)	Friday 26 October 2018 (T1:05)

This Continuous Assessment is worth 15% of the module mark.

Work is to be submitted via BART at the Student Services Desk of the Harrison Building by midday on Friday 26th October 2018. **All students are reminded of the University regulations on academic honesty and plagiarism.**

You should make use of higher-order functions, and functions from the Haskell standard prelude whenever possible. In all cases, partial marks will be awarded for partial solutions.

Question 1

The twenty six upper-case letters of the alphabet and their Morse code equivalents are shown in Table 1. The Morse code for each character is followed by a *gap* that is not shown in Table 1.

Question 1.1

Show a Haskell data type suitable for representing a Morse code dot, dash or gap.

Letter	Morse code	Letter	Morse code
A	*dot dash*	N	*dash dot*
B	*dash dot dot dot*	O	*dash dash dash*
C	*dash dot dash dot*	P	*dot dash dash dot*
D	*dash dot dot*	Q	*dash dash dot dash*
E	*dot*	R	*dot dash dot*
F	*dot dot dash dot*	S	*dot dot dot*
G	*dash dash dot*	T	*dash*
H	*dot dot dot dot*	U	*dot dot dash*
I	*dot dot*	V	*dot dot dot dash*
J	*dot dash dash dash*	W	*dot dash dash*
K	*dash dot dash*	X	*dash dot dot dash*
L	*dot dash dot dot*	Y	*dash dot dash dash*
M	*dash dash*	Z	*dash dash dot dot*

Figure 1: The twenty-six upper-case letters and their Morse code equivalents.

```
data Morse
  =  Dot
  |  Dash
  |  Gap
     deriving Show
```

(5 marks)

Question 1.2

Show a Haskell function suitable for encoding words using the twenty six upper-case letters of the alphabet as Morse code. For example

```
encode "DOG" ===>
```
$\langle$ *dash, dot, dot, gap, dash, dash, dash, gap, dash, dash, dot, gap* $\rangle$

```
encode :: String -> [Morse]
encode
  =  concatMap enc

enc :: Char -> [Morse]
enc 'A'
  = [ Dot,  Dash, Gap ]
enc 'B'
```

```
  = [ Dash, Dot,  Dot,  Dot,  Gap ]
enc 'C'
  = [ Dash, Dot,  Dash, Dot,  Gap ]
enc 'D'
  = [ Dash, Dot,  Dot,  Gap ]
enc 'E'
  = [ Dot,  Gap ]
enc 'F'
  = [ Dot,  Dot,  Dash, Dot,  Gap ]
enc 'G'
  = [ Dash, Dash, Dot,  Gap ]
enc 'H'
  = [ Dot,  Dot,  Dot,  Dot,  Gap ]
enc 'I'
  = [ Dot,  Dot,  Gap ]
enc 'J'
  = [ Dot,  Dash, Dash, Dash, Gap ]
enc 'K'
  = [ Dash, Dot,  Dash, Gap ]
enc 'L'
  = [ Dot,  Dash, Dot,  Dot,  Gap ]
enc 'M'
  = [ Dash, Dash, Gap ]
enc 'N'
  = [ Dash, Dot,  Gap ]
enc 'O'
  = [ Dash, Dash, Dash, Gap ]
enc 'P'
  = [ Dot,  Dash, Dash, Dot,  Gap ]
enc 'Q'
  = [ Dash, Dash, Dot,  Dash, Gap ]
enc 'R'
  = [ Dot,  Dash, Dot,  Gap ]
enc 'S'
  = [ Dot,  Dot,  Dot,  Gap ]
enc 'T'
  = [ Dash, Gap ]
enc 'U'
  = [ Dot,  Dot,  Dash, Gap ]
enc 'V'
  = [ Dot,  Dot,  Dot,  Dash, Gap ]
enc 'W'
  = [ Dot,  Dash, Dash, Gap ]
enc 'X'
  = [ Dash, Dot,  Dot,  Dash, Gap ]
```

```
enc 'Y'
  = [ Dash, Dot,  Dash, Dash, Gap ]
enc 'Z'
  = [ Dash, Dash, Dot,  Dot,  Gap ]
```

(10 marks)

Question 1.3

Show a Haskell function suitable for decoding Morse code to the twenty six upper-case letters, so that

decode ⟨ *dash, dot, dot, gap, dash, dash, dash, gap, dash, dash, dot, gap* ⟩ `===>`
`"DOG"`

```
decode :: [ Morse ] -> String
decode ( Dot  : Dash : Gap : ms )
  =  'A' : decode ms
decode ( Dash : Dot  : Dot  : Dot  : Gap : ms )
  =  'B' : decode ms
decode ( Dash : Dot  : Dash : Dot  : Gap : ms )
  =  'C' : decode ms
decode ( Dash : Dot  : Dot  : Gap  : ms )
  =  'D' : decode ms
decode ( Dot  : Gap  : ms )
  =  'E' : decode ms
decode ( Dot  : Dot  : Dash : Dot  : Gap : ms )
  =  'F' : decode ms
decode ( Dash : Dash : Dot  : Gap  : ms )
  =  'G' : decode ms
decode ( Dot  : Dot  : Dot  : Dot  : Gap : ms )
  =  'H' : decode ms
decode ( Dot  : Dot  : Gap  : ms )
  =  'I' : decode ms
decode ( Dot  : Dash : Dash : Dash : Gap : ms )
  =  'J' : decode ms
decode ( Dash : Dot  : Dash : Gap  : ms )
  =  'K' : decode ms
decode ( Dot  : Dash : Dot  : Dot  : Gap : ms )
  =  'L' : decode ms
decode ( Dash : Dash : Gap  : ms )
  =  'M' : decode ms
```

```
decode ( Dash : Dot  : Gap  : ms )
  =  'N' : decode ms
decode ( Dash : Dash : Dash : Gap : ms )
  =  'O' : decode ms
decode ( Dot  : Dash : Dash : Dot  : Gap : ms )
  =  'P' : decode ms
decode ( Dash : Dash : Dot  : Dash : Gap : ms )
  =  'Q' : decode ms
decode ( Dot  : Dash : Dot  : Gap  : ms )
  =  'R' : decode ms
decode ( Dot  : Dot  : Dot  : Gap  : ms )
  =  'S' : decode ms
decode ( Dash : Gap  : ms )
  =  'T' : decode ms
decode ( Dot  : Dot  : Dash : Gap  : ms )
  =  'U' : decode ms
decode ( Dot  : Dot  : Dot  : Dash : Gap : ms )
  =  'V' : decode ms
decode ( Dot  : Dash : Dash : Gap  : ms )
  =  'W' : decode ms
decode ( Dash : Dot  : Dot  : Dash : Gap : ms )
  =  'X' : decode ms
decode ( Dash : Dot  : Dash : Dash : Gap : ms )
  =  'Y' : decode ms
decode ( Dash : Dash : Dot  : Dot  : Gap : ms )
  =  'Z' : decode ms
decode []
  =  []

main :: IO ()
main
=  putStr (decode (encode "ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
```

One might try a table-based solution, such as this, but it is hard to make it elegant.

```
table :: [(Char,[Morse])]
table
  =  [ ('A', [ Dot,  Dash, Gap ])
     , ('B', [ Dash, Dot,  Dot,  Dot,  Gap ])
     , ('C', [ Dash, Dot,  Dash, Dot,  Gap ])
     , ('D', [ Dash, Dot,  Dot,  Gap ])
     , ('E', [ Dot,  Gap ])
     , ('F', [ Dot,  Dot,  Dash, Dot,  Gap ])
     , ('G', [ Dash, Dash, Dot,  Gap ])
```

```
      , ('H', [ Dot,  Dot,  Dot,  Dot,  Gap ])
      , ('I', [ Dot,  Dot,  Gap ])
      , ('J', [ Dot,  Dash, Dash, Dash, Gap ])
      , ('K', [ Dash, Dot,  Dash, Gap ])
      , ('L', [ Dot,  Dash, Dot,  Dot,  Gap ])
      , ('M', [ Dash, Dash, Gap ])
      , ('N', [ Dash, Dot,  Gap ])
      , ('O', [ Dash, Dash, Dash, Gap ])
      , ('P', [ Dot,  Dash, Dash, Dot,  Gap ])
      , ('Q', [ Dash, Dash, Dot,  Dash, Gap ])
      , ('R', [ Dot,  Dash, Dot,  Gap ])
      , ('S', [ Dot,  Dot,  Dot,  Gap ])
      , ('T', [ Dash, Gap ])
      , ('U', [ Dot,  Dot,  Dash, Gap ])
      , ('V', [ Dot,  Dot,  Dot,  Dash, Gap ])
      , ('W', [ Dot,  Dash, Dash, Gap ])
      , ('X', [ Dash, Dot,  Dot,  Dash, Gap ])
      , ('Y', [ Dash, Dot,  Dash, Dash, Gap ])
      , ('Z', [ Dash, Dash, Dot,  Dot,  Gap ])
      ]

elbat :: [([Morse],Char)]
elbat
  =  map swap table

encode1 :: String -> [Morse]
encode1
  =  concatMap (\c -> search c table)

decode1 :: [Morse] -> String
decode1
  =  map (\ms -> search (ms ++ [Gap]) elbat) . filter (not . null)
                                             . splitOn [Gap]

search :: Eq a => a -> [(a,b)] -> b
search k kvs
  =  fromJust (lookup k kvs)
```

(10 marks)

Question 2

At one time, the *The Times* newspaper used to publish a Suko puzzle in the form of a grid. See Figure 2. Given the integers t_1, t_2, t_3 and t_4, the reader was challenged to

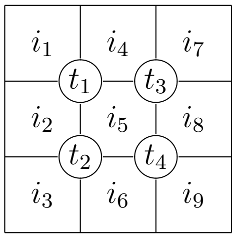


Figure 2: The Suko grid.

arrange the integers i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, and i_9 between 1 and 9, so that

$$\begin{aligned} t_1 &= i_1 + i_2 + i_4 + i_5 \\ t_2 &= i_2 + i_3 + i_5 + i_6 \\ t_3 &= i_4 + i_5 + i_7 + i_8 \\ t_4 &= i_5 + i_6 + i_8 + i_9 \end{aligned}$$

The solution to one particular Suko puzzle is shown in Figure 3.

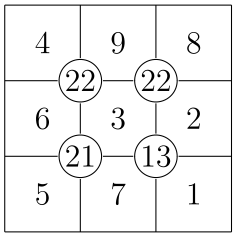


Figure 3: A Suko solution.

Question 2.1

Show a Haskell function that given a tuple of four integers, t_1, t_2, t_3 and t_4, returns a solution list of nine integers i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, and i_9 for the Suko puzzle. Thus,

```
    suko (22, 21, 22, 13)
        ===> [4,6,5,9,3,7,8,2,1]
```

```
module Main where
import Data.List

main :: IO ()
main
  = putStr (show (suko (22,21,22,13)))

suko :: (Int,Int,Int,Int) -> [Int]
suko spec
  =  head (filter (isSolution spec) (permutations [1..9]))

isSolution :: (Int,Int,Int,Int) -> [Int] -> Bool
isSolution (t1,t2,t3,t4) [i1,i2,i3,i4,i5,i6,i7,i8,i9]
  =   t1 == i1 + i2 + i4 + i5
  &&  t2 == i2 + i3 + i5 + i6
  &&  t3 == i4 + i5 + i7 + i8
  &&  t4 == i5 + i6 + i8 + i9
```

This question is all about *software structure*. One third of the marks are for a producer of possible solutions, and two thirds of the marks are for a consumer of those solutions. The producer generates permutations of the digits between 1 and 9. The consumer filters those permutations with a separate testing predicate.

(25 marks)

Question 3

One string may be transformed into another by a sequence of edit operations that either insert a character, delete a character, copy a character, or change one character to another. For example,

```
transform "fish" "chips" ===>
```
$\langle$ *Insert 'c', Change 'f' 'h', Copy 'i', Change 's' 'p', Change 'h' 's'* $\rangle$

Question 3.1

Show a Haskell data type suitable for representing edit operations.

```
data Edit
  =  Insert Char
  |  Delete Char
```

```
      |  Copy   Char
      |  Change Char Char
      deriving Show
```

(5 marks)

Question 3.2

Show a Haskell function suitable for computing the cost of a sequence of edit operations, assuming that the cost of copying a character is 0, and that the cost of all other operations is 1. For example,

cost $\langle$ *insert 'c', change 'f' 'h', copy 'i', change 's' 'p', change 'h' 's'* $\rangle$
    ===> 4

```
    cost :: [Edit] -> Int
    cost
      =  length . filter (not . isCopy)

    isCopy :: Edit -> Bool
    isCopy (Copy c)
      =  True
    isCopy e
      =  False
```

(10 marks)

Question 3.3

Show a Haskell function suitable for computing the lowest-cost sequence of edit operations needed to transform one string into another. For example,

transform "exeter" "exmouth" ===>
  $\langle$ *copy 'e', copy 'x', insert 'm', insert 'o', change 'e' 'u', copy 't',*
   delete 'e', change 'r' 'h' $\rangle$

```
    transform :: String -> String -> [Edit]
    transform [] ys
    =  map Insert ys
  transform xs []
    =  map Delete xs
  transform (x:xs) (y:ys)
    |  x == y     =  Copy x : transform xs ys
    |  otherwise  =  head (orderBy cost
                            [  Delete x   : transform xs (y:ys)
                            ,  Insert y   : transform (x:xs) ys
                            ,  Change x y : transform xs ys
                            ])



  orderBy :: Ord b => (a -> b) -> [a] -> [a]
  orderBy metric
    =  sortBy (\x y -> compare (metric x) (metric y))


  main :: IO ()
  main
    =  putStrLn (show (transform "exeter" "exmouth"))
```

A more efficient function considers all pairs of substrings (seen as keys), calculates transformations from one element of the pair one to the other (seen as values), records these (key, value) pairs on a blackboard, and uses the blackboard to help calculate further transformations. This dynamic programming approach is not a better implementation — it is a better algorithm.

```
  dynamic :: String -> String -> [Edit]
  dynamic xs ys
    =  search (xs,ys) blackboard
       where

       blackboard :: [((String,String), [Edit])]
       blackboard
         =  zip keys values

       keys :: [(String,String)]
       keys
         =  concat (map (\is -> map (\js -> (is,js)) (tails ys)) (tails xs))

       values :: [[Edit]]
       values
         =  map (\(is,js) -> f is js) keys
```

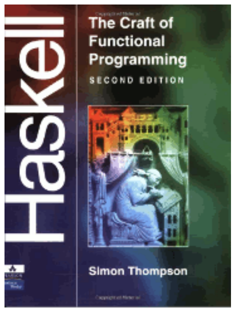
```
f :: String -> String -> [Edit]
f [] ys
  =  map Insert ys
f xs []
  =  map Delete xs
f (x:xs) (y:ys)
  |  x == y     =  Copy x : search (xs,ys) blackboard
  |  otherwise  =  head (orderBy cost
                     [  Delete x   : search (xs, (y:ys)) blackboard
                     ,  Insert y   : search ((x:xs), ys) blackboard
                     ,  Change x y : search (xs,ys)      blackboard
                     ])

search :: Eq a => a -> [(a,b)] -> b
search k kvs
  =  fromJust (lookup k kvs)
```

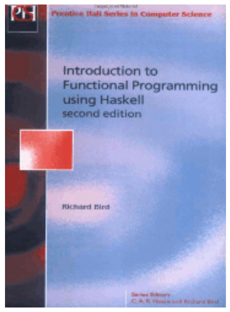
(35 marks)

The marks here will be awarded as follows: upto 15 for a functioning, but inefficient solution that can transform short strings (say upto 10 characters) in reasonable time [1]; upto 35 for a functioning and efficient solution that can transform longer strings (say upto 20 characters) in reasonable time.

Readings



S. Thompson, *The Craft of Functional Programming (Third Edition)*, Addison Wesley, 2011, ISBN 978-0201882957.


Prentice Hall Series in Computer Science
Introduction to Functional Programming using Haskell
second edition
Richard Bird

R. Bird, *Introduction to Functional Programming Using Haskell (Second Edition)*, Prentice Hall, 1998, ISBN 978-0134843469.

[1] Reasonable time is defined to be no more than a few seconds when using a Red Room computer.