



# **Programming and Paradigms Haskell Assignment**

**TU059**

**MSc in Computer Science  
(Advanced Software Development)**

**Lecturer name: Dr. Pierpaolo Dondio**

**Yuanshuo Du D22125495**

School of Computer Science  
TU Dublin – City Campus

**09/01/2024**

```
--Q1:

is_square :: Int -> Bool
is_square x
  | x < 0 = False
  | otherwise = (floor (sqrt (fromIntegral x)))^2 == x
```

Test `is_square 25`, the result is:

```
ghci> is_square 25
```

```
True
```

The `is_square` function takes an `Int` as a parameter `x` and returns a `Bool`.

It defines a function to check if a number is a perfect square.

There are two patterns used to match `x`:

- If `x` is less than 0, it returns `False` immediately.
- Otherwise, it calculates the square root of `x` by taking the floor of the square root.
- It then squares the result.
- And compares if it equals the original `x`. If so, it returns `True`, meaning `x` is a perfect square.

Q2:

```
--Q2:

import Data.Char (toLower, isLower)
import Data.List (sort, group)

freq_letter_pc :: String -> [(Char, Float)]
freq_letter_pc text =
  let lowercaseText = map toLower (filter isLower text)
      totalChars = fromIntegral (length lowercaseText)
      charGroups = group (sort lowercaseText)
  in
    [(char, (fromIntegral (length groupChars) / totalChars) ) |
groupChars@(char:_) <- charGroups]
```

Test:

```
ghci> freq_letter_pc "this is text"
[('e',0.1),('h',0.1),('i',0.2),('s',0.2),('t',0.3),('x',0.1)]
```

- The function takes a string as input and returns frequencies as `(Char, Float)` tuples
- It maps `toLower` to convert the whole string to lowercase
- Filters to only keep lowercase alphabetical characters
- Calculates the total number of characters
- Groups characters by sorting and adjoining duplicates
- Uses a list comprehension to iterate over the character groups
- For each group:

- Extracts the head character
- Calculates percentage by dividing group length by total characters

Q3:

```
type City = (Int, String, Int, Int) -- city_id, city_name, city_population, country_id
type Country = (Int, String) -- country_id, country_name

-- data
cities :: [City]
cities = [(1, "Paris", 7000000, 1), (2, "London", 8000000, 2), (3, "Rome", 3000000, 3),
          (4, "Edinburgh", 500000, 2), (5, "Florence", 50000, 3), (6, "Venice", 200000, 3),
          (7, "Lyon", 1000000, 1), (8, "Milan", 3000000, 3), (9, "Madrid", 6000000, 4),
          (10, "Barcelona", 5000000, 4)]

countries :: [Country]
countries = [(1, "UK"), (2, "France"), (3, "Italy"), (4, "Spain")]

-- Function a: get_city_above
get_city_above :: Int -> [String]
get_city_above n = [cityName | (_, cityName, cityPop, _) <- cities, cityPop >= n]

-- Function b: get_city
get_city :: String -> [String]
get_city countryName =
  let countryId = case lookupCountryId countryName of
    Just id -> id
    Nothing -> error "Country not found"
  in [cityName | (_, cityName, _, cityCountryId) <- cities, cityCountryId == countryId]

-- Helper function to lookup country_id by country_name
lookupCountryId :: String -> Maybe Int
lookupCountryId countryName = lookup countryName [(countryName', countryId) | (countryId,
countryName') <- countries]

-- Function c: num_city
num_city :: [(String, Int)]
num_city = [(countryName, countCities countryId) | (countryId, countryName) <-
countries]

-- Helper function to count cities in a country
countCities :: Int -> Int
countCities countryId = length [(cityName, _, _, cityCountryId) <- cities, cityCountryId
== countryId]
```

Explanation:

- get\_city\_above: Takes an integer n, returns a list of city names with population above n
- get\_city: Takes a country name string, returns list of cities in that country
- Looks up country id from name using lookupCountryId
- Filters cities list to ones matching that country id
- lookupCountryId: Helper to lookup country id from name in countries list

- num\_city: Returns list of (country name, number of cities) tuples
- Loops through countries to get name and id
- Counts cities for each id using countCities
- countCities: Helper to count number of cities matching a country id in cities list

Test result:

```
ghci> get_city_above 6000000
["Paris","London","Madrid"]
```

```
ghci> get_city "Italy"
["Rome","Florence","Venice","Milan"]
ghci> num_city
[("UK",2),("France",2),("Italy",4),("Spain",2)]
```

Q4

```
--Q4:
eucl_dist :: Floating a => [a] -> [a] -> a
-- eucl_dist x y = sqrt $ sum (map (\(a,b) -> (a-b)^2) (zip x y))
eucl_dist x y = sqrt $ sum (map (\(a,b) -> (a-b)^2) (zip x y))
```

Explanation:

- It calculates the Euclidean distance between two lists of floating point numbers.
- It takes two lists as input - x and y.
- It uses zip to pair up corresponding elements from x and y.
- It maps over the zipped pairs, calculating the squared difference between each pair  $(a - b)^2$ .
- It sums the results of the squared differences.
- It takes the square root of the sum, to calculate the final Euclidean distance.
- The type is `Floating a => [a] -> [a] -> a`, so it works on any lists of numbers that implement `Floating`.

Test result:

```
ghci> eucl_dist [0,0,0] [1,1,1]
1.7320508075688772
```

Q5

```
-- Function to compute the Euclidean distance between two frequency distributions
eucl_dist2 :: Floating a => [a] -> [a] -> a
eucl_dist2 x y
  | length x /= length y = error "Vectors must have the same number of elements"
```

```

    | otherwise = sqrt $ sum $ map (\(a, b) -> (a - b) ^ 2) (zip x y)
-- Function to identify the language of a given text
get_lang :: String -> IO ()
get_lang text = do
    let lowerText = map toLower text
        eng_freq =
[8.12,1.49,2.71,4.32,12.02,2.30,2.03,5.92,7.31,0.10,0.69,3.98,2.61,6.95,7.68,1.82,0.
11,6.02,6.28,9.10,2.88,1.11,2.09,0.17,2.11,0.07]
        pt_freq =
[12.21,1.01,3.35,4.21,13.19,1.07,1.08,1.22,5.49,0.30,0.13,3.00,5.07,5.02,10.22,3.01,
1.10,6.73,7.35,5.07,4.46,1.72,0.05,0.28,0.04,0.45]
        text_freq = map (\c -> fromIntegral (length $ filter (== c) lowerText) /
fromIntegral (length lowerText) * 100) ['a'..'z']
        dist_to_eng = eucl_dist text_freq eng_freq
        dist_to_pt = eucl_dist text_freq pt_freq
    putStrLn $ if dist_to_eng < dist_to_pt
        then "The text is in English"
        else "The text is in Portuguese"
main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            contents <- readFile filename
            get_lang contents
        _ -> putStrLn "Usage: ./lang filename"

```

Explanation:

- eucl\_dist2 function calculates Euclidean distance between two frequency distributions, handling different lengths.
- get\_lang function identifies language of given text:
  - Calculates frequency distribution of lowercase text
  - Defines English and Portuguese frequency distributions
  - Calculates distances from text to English and Portuguese distributions
  - Compares distances and prints language with smaller distance
- main function:
  - Gets filename from arguments
  - Reads file contents
  - Passes contents to get\_lang function
  - Prints usage if no filename provided

Test result:

```
ghci> portugueseText = "Olá mundo, este é um texto de amostra em português"
ghci> get_lang portugueseText
The text is in Portuguese
ghci> englishText = "Hello world, this is a sample english text"
ghci> get_lang englishText
The text is in English
```

Successfully get the language

Q6

```
import System.IO
import Data.Char
import System.Environment

main = do
    [filename, index] <- getArgs
    contents <- readFile filename
    writeFile (filename ++ ".chp") (encrypt (read index :: Int) contents)

    -- Decrypt the file
    let decryptedContents = c_decrypt (read index :: Int) contents
    writeFile (filename ++ ".decrypted") decryptedContents

let2int :: Char -> Int
let2int c = ord c - ord 'a'

int2let :: Int -> Char
int2let n = chr (ord 'a' + n)

shift :: Int -> Char -> Char
shift n c
    | isLower c = int2let ((let2int c + n) `mod` 26)
    | otherwise = c

encrypt :: Int -> String -> String
encrypt n s = [shift n (toLower c) | c <- s]

decrypt :: Int -> String -> String
decrypt n s = encrypt (-n) s

c_decrypt :: Int -> String -> String
c_decrypt n s = decrypt n s
```

This code implements a simple Caesar cipher encryption and decryption of a text file:

- It takes a filename, encryption key (index) and text as arguments
- It reads the file contents
- It encrypts the text using encrypt, shifting each lowercase letter by the key
- It writes the encrypted text to a new file
- It decrypts the encrypted text back using the same key in c\_decrypt
- It writes the decrypted text to another new file

The key steps are:

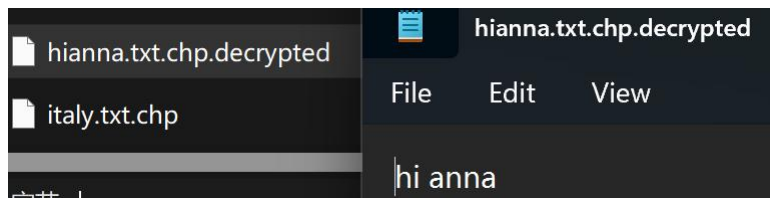
- encrypt shifts each letter by the key using modular arithmetic
- decrypt reuses encrypt with negative key to decrypt
- c\_decrypt is an alias for decrypt for clarity
- It handles I/O of files and command line args

Test result:

First use encrypt.hs to encrypt the hianna.txt.

```
D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>ghc c_decrypt.hs
[1 of 2] Compiling Main          ( c_decrypt.hs, c_decrypt.o )
[2 of 2] Linking c_decrypt.exe
```

```
\Haskell>c_decrypt.exe hianna.txt.chp 3
```



Successfully generate the decrypted file hianna.txt.chp.decrypted, and see the content, successfully decrypt the content.

## Q7

### Part1:

```
import Data.Char (toLower, isAlpha)
import Data.List (nub, sort)
import System.IO

-- Function to extract distinct unique words from a given text
extractWords :: String -> [String]
extractWords text = nub $ filter (not . null) $ map (map toLower . filter isAlpha)
$ words text

-- Function to read novels and create a dictionary file
buildDictionary :: IO ()
buildDictionary = do
    prideText <- readFile "pride.txt"
    ulyssesText <- readFile "ulysses.txt"
    dorianText <- readFile "dorian.txt"

    let dictionary = sort $ extractWords $ prideText ++ " " ++ ulyssesText ++
" " ++ dorianText
```

```

    writeFile "dict.txt" $ unlines dictionary
main :: IO ()
main = buildDictionary

```

- extractWords takes a text, splits it into words, filters non-alphanumeric characters, lowercases letters, and removes duplicates to extract unique words
- buildDictionary function:
  - Reads in 3 novel text files
  - Combines text and calls extractWords to create a dictionary of unique words
  - Sorts the dictionary
  - Writes the dictionary to a file "dict.txt"
- main simply calls buildDictionary

#### Test result:

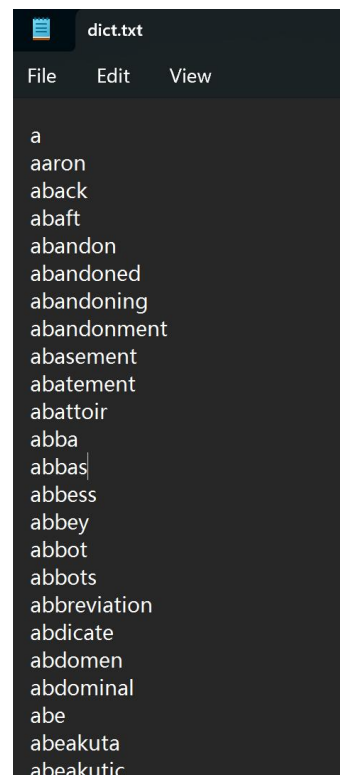
```

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>ghc Dict.hs
[1 of 2] Compiling Main          ( Dict.hs, Dict.o ) [Source file changed]
[2 of 2] Linking Dict.exe [Objects changed]

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>Dict.exe

```

Run the Dict.exe generate the file :



```

dict.txt
File Edit View

a
aaron
aback
abaft
abandon
abandoned
abandoning
abandonment
abasement
abatement
abattoir
abba
abbas
abbess
abbey
abbot
abbots
abbreviation
abdicate
abdomen
abdominal
abe
abeakuta
abeakutic

```

Successfully get the dictionary

**Q7:**



## Part 2:

```
import System.IO
import System.Environment
import Data.Char (toLower, isAlpha, ord, chr)
import Data.List (nub, maximumBy)
import Data.Ord (comparing)

-- Function to extract distinct unique words from a given text
extractWords :: String -> [String]
extractWords text = nub $ filter (not . null) $ map (map toLower . filter isAlpha) $ words
text

-- Function to decrypt a message with a given index
decrypt :: Int -> String -> String
decrypt n s = [shift (-n) c | c <- s]
  where
    shift n c
      | toLower c `elem` ['a'..'z'] = int2let ((let2int (toLower c) + n) `mod` 26)
      | otherwise = c
    let2int :: Char -> Int
    let2int c = ord c - ord 'a'
    int2let :: Int -> Char
    int2let n = chr (ord 'a' + n)

-- Function to count the number of English words in a text
countEnglishWords :: [String] -> String -> Int
countEnglishWords dictionary text = length $ filter (`elem` dictionary) $ extractWords
text

-- Function to guess the index by maximizing the number of English words
guessIndex :: [String] -> String -> Int
guessIndex dictionary encryptedText =
  fst $ maximumBy (comparing snd) [(i, countEnglishWords dictionary $ decrypt i
encryptedText) | i <- [0..25]]

main :: IO ()
main = do
  [filename] <- getArgs
  encryptedText <- readFile filename
  dictionary <- lines <$> readFile "dict.txt"
  let index = guessIndex dictionary encryptedText
  putStrLn $ "Guessed Index: " ++ show index
  -- Use decrypt function
  let decryptedContents = decrypt index encryptedText
  writeFile (filename ++ ".decrypted") decryptedContents
```

- extractWords gets unique words from a text

- decrypt decrypts a message with a given index
- countEnglishWords counts words in given text that are in the dictionary
- guessIndex tries indexes from 0-25, returns the one with most dictionary words
- The main function:
  - Reads encrypted text file and dictionary
  - Calls guessIndex to get the index with most words
  - Prints the guessed index
  - Decrypts the text using the index
  - Writes the decrypted text to a new file

Test Result:

```
D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>ghc guessIndex.hs
[1 of 2] Compiling Main             ( guessIndex.hs, guessIndex.o )
[2 of 2] Linking guessIndex.exe
```

```
D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>guessIndex.exe italy.txt.chp
Guessed Index: 8
```



Italy officially the italian republic italian is a country in europe located in the heart of the mediterranean sea italy  
 switzerland austria slovenia san marino and vatican city italy has a largely temperate seasonal and mediterranean  
 inhabitants it is the fourth most populous european member state and the most populous country in southern e  
 location in europe and the mediterranean italy has historically been home to a myriad of peoples and cultures i  
 tribes and italic peoples dispersed throughout the italian peninsula and insular italy beginning from the classica  
 greeks established settlements in the south of italy with etruscans and celts inhabiting the centre and the north  
 known as the latins formed the roman kingdom in the fifth century bc which eventually became a republic that  
 neighbours in the first century bc the roman empire emerged as the dominant power in the mediterranean basi  
 political and religious centre of western civilisation the legacy of the roman empire is widespread and can be ob  
 civilian law republican governments christianity and the latin scriptduring the early middle ages italy endured se  
 invasions but by the th century numerous rival citystates and maritime republics mainly in the northern and cer  
 prosperity through shipping commerce and banking laying the groundwork for modern capitalism these mostly

Successfully decrypt the italy.txt.chp to normal English.

Q8:

```
import System.Random

-- Function to check if a point (x, y) is inside the quarter circle
isInsideQuarterCircle :: Double -> Double -> Bool
isInsideQuarterCircle x y = x^2 + y^2 <= 1

-- Function to perform the Monte Carlo simulation and estimate the area
monteCarloQuarterCircle :: Int -> IO Double
monteCarloQuarterCircle numPoints = do
  gen <- newStdGen
```

```

    let points = take numPoints $ randomPoints gen
        insidePoints = filter (\(x, y) -> isInsideQuarterCircle x y) points
        ratio = fromIntegral (length insidePoints) / fromIntegral numPoints
    return $ ratio * 1 -- Area of the square is 1
-- Function to generate random points in the square
randomPoints :: StdGen -> [(Double, Double)]
randomPoints gen = zip (randoms gen) (randoms (snd (split gen)))
main :: IO ()
main = do
    let numPoints = 1000000 -- Adjust the number of points as needed
    estimatedArea <- monteCarloQuarterCircle numPoints
    putStrLn $ "Estimated Area of the Quarter Circle: " ++ show estimatedArea

```

- isInsideQuarterCircle checks if a point (x,y) is inside the quarter circle
- randomPoints generates random points within the containing square
- monteCarloQuarterCircle:
  - Generates random points
  - Filters only inside points
  - Calculates ratio of inside/total points
  - Estimates area as ratio \* area of containing square
- main:
  - Calls monteCarloQuarterCircle with numPoints
  - Prints estimated area

Test Result:

```

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>cabal install --lib random
Resolving dependencies...

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>ghc randomPoints.hs
Loaded package environment from C:\Users\Steven.du\AppData\Roaming\ghc\x86_64-mingw32-9.4.8\envi
lt
[1 of 2] Compiling Main             ( randomPoints.hs, randomPoints.o )
[2 of 2] Linking randomPoints.exe

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>randomPoints.exe
Estimated Area of the Quarter Circle: 0.785917

```

**Q9:**

```

import System.Environment (getArgs)

-- Higher-order function to compute the sum of the first n elements of a series
math_series :: (Float -> Float) -> Int -> Float
math_series series n = sum $ take n $ map series [1..]
-- Define the sample_series function for 1/2^k

```

```

sample_series :: Float -> Float
sample_series k = 1 / (2**k)
-- Define the series  $\sum (-1)^{(k+1)} * (4/(2k-1))$ 
pi_series :: Float -> Float
pi_series k = (-1)**(k+1) * (4 / (2*k - 1))
main :: IO ()
main = do
    args <- getArgs
    case args of
        [seriesName, nStr] -> do
            let n = read nStr :: Int
            result = case seriesName of
                "sample_series" -> math_series sample_series n
                "pi_series" -> math_series pi_series n
                _ -> error "Unknown series function"
            putStrLn $ "Sum of the first " ++ show n ++ " elements of " ++ seriesName
            ++ " series: " ++ show result
        _ -> putStrLn "Usage: ./program series_name number_of_terms"

```

Here is a concise explanation:

- math\_series is a higher-order function that takes a series function and n, and returns the sum of the first n terms
- sample\_series and pi\_series define specific series
- main:
  - Gets series name and n from command line
  - Calls math\_series with the appropriate series function based on name
  - Prints result

The key aspects are:

- math\_series is reusable for any series using partial application
- sample\_series and pi\_series define example series
- main parses args and calls math\_series to compute the sum for the given series

Test Result:

```

D:\TUD course\Master of Computer Science\Programming paradigm\Haskell>math_series.exe pi_series 2
Sum of the first 2 elements of pi_series series: 2.6666665

```

Q10:

```

-- Function to approximate the integral of a given function
integral :: (Float -> Float) -> Float -> Float -> Int -> Float
integral f x1 x2 n = sum [f (x1 + fromIntegral i * delta) * delta | i <- [0..n-1]]
    where
        delta = (x2 - x1) / fromIntegral n

-- Example function f: f x = 0.5 * x

```

```

exampleFunction :: Float -> Float
exampleFunction x = 0.5 * x
main :: IO ()
main = do
    let result1 = integral exampleFunction 1 20 9
        result2 = integral exampleFunction 1 20 10000
    putStrLn $ "Integral result with 9 intervals: " ++ show result1
    putStrLn $ "Integral result with 10000 intervals: " ++ show result2

```

Here is a concise explanation:

- integral is a function that approximates the integral of a function  $f$  from  $x_1$  to  $x_2$
- It divides the range into  $n$  intervals of size  $\delta$
- It evaluates  $f$  at the left endpoint of each interval
- It multiplies the function value by the interval size  $\delta$
- It sums these areas to approximate the integral
- exampleFunction defines a sample function  $f(x) = 0.5x$
- main:
  - Calls integral twice with exampleFunction
  - First with 9 intervals, second with 10000
  - Prints the results

Test Result:

Successfully get right result

```

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>ghc integral.hs
Loaded package environment from C:\Users\Steven.du\AppData\Roaming\ghc\x86_64-mingw32-9.4.8\environments\defau
lt
[1 of 2] Compiling Main             ( integral.hs, integral.o )
[2 of 2] Linking integral.exe

D:\TUD course\Master of Computer Science\Programing paradigm\Haskell>integral.exe
Integral result with 9 intervals: 89.72223
Integral result with 10000 intervals: 99.74098

```