UNIVERSITY OF JOHANNESBURG

FACULTY OF SCIENCE

HONOURS (Computer Science / IT)	APK CAMPUS
IT18X17 **FUNCTIONAL PROGRAMMING** **EXAMINATION EQUIVALENT ONLINE ASSESSMENT 2020** **PAPER A** **MEMORANDUM**	
2020–06	

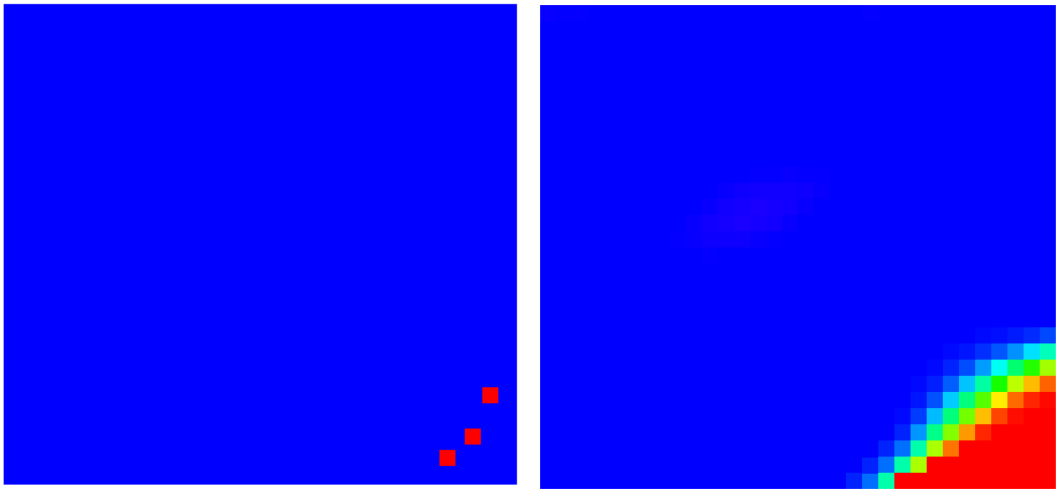
EXAMINER		**Prof. DA Coulter**	
EXTERNAL MODERATOR		**Prof. B van der Merwe (SUN)**	
TIME (EXCLUDING SUBMISSION)	**2 HOURS**	**MARKS**	**100**

Please read the following instructions carefully

1. You must complete this assignment yourself within the prescribed time limits.
2. You are bound by all university regulations please special note of those regarding assessment, plagiarism, and ethical conduct.
3. You may refer to the lectures, notes, official Haskell wiki, prescribed textbook, and any web resources directly linked to from this document during the assessment. You may not directly use any code from any source.
4. You must complete and submit the *"Honesty Declaration : Online Assessment"* document along with your submission to EVE. No submissions without an accompanying declaration will be marked.
5. Your code together with the declaration must be submitted in a zip archive named in the following format.
   STUDENTNUMBER_SURNAME_INITIALS_SUBJECTCODE_ASSESSMENT
   e.g. `202012345_COULTER_DA_IT18X17_EXAM.zip`
6. Additional time for submission is allowed for as per the posted deadlines on EVE. If you are experiencing technical difficulties related to submission please contact me as soon as possible.
7. No communication concerning this test is permissible during the assessment session except with Academy staff members. I am available via email (dcoulter@uj.ac.za) and on the *"Prof Coulter - Postgraduate Matters"* Discord server throughout the assessment.
8. This paper consists of 5 pages including the cover page

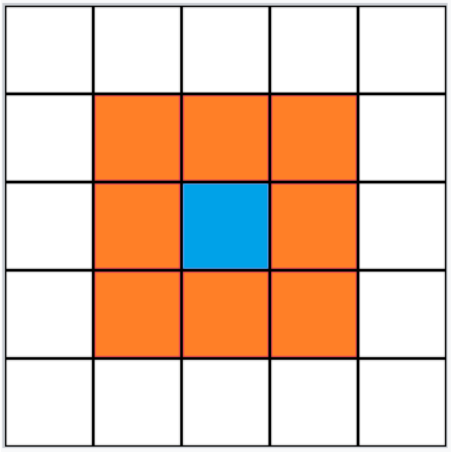
Mapping the HeatMap

You will need to develop a software system in the Haskell Programming Language which implements a very simple simulation of the diffusion of heat (note the actual heat transfer equations used by physicists and engineers are far more complicated). You will need to create the required data types using only the functionality available as part of the core Haskell libraries (no third-party code may be used).



Simulation setup

- The simulation will take the form of applying transformations to a numerical grid / lattice of values.
- Each cell in the grid should be uniquely and invariantly associated with a pair of coordinates reflecting their position within the grid.
- The value at each cell in the grid represents the amount of heat at that location (this can be a real value in the range [0.0, 1.0] or natural number up to some maximum as you see fit).
- Each value in the grid is initially set to some uniform value such as zero except for a set of hotspot cells which are set to the maximum value you have chosen.
- Every cell is uniquely and invariantly associated with a set of Moore neighbours (i.e. adjacent and diagonals cells) as illustrated below from the point of view of an interior cell. You may handle edge cells by either reducing the size of their neighbourhood or wrapping around as you see fit.



Update Rules

- At every iteration of the simulation each cell's value is set to contain the average of its neighbours from the previous iteration.

Visualisation

- The final state of the heat map must be output as an image in the Portable Pixel Map (PPM) format (see the Input and Output section of the marksheet for more details).
- You must create an appropriate set of functions for translating the heat map's intensity values to a visual representation. You may elect to simply translate the intensity value to a grayscale value (using the P2 variant of the format) or to an RGB Colour (using the P3 variant). Colour will be more difficult, but the extra effort will be rewarded.
- **Bonus**:
  - In order to convert from an intensity value to an RGB colour we will actually assume that the colour is in HSV (Hue, Saturation, Value) format but keep the saturation and value amounts set at one. The HSV Colour model is cyclical so, in this case, the H value ranges from 0 to 360. Also note that because of this the colours at 0 are close to the colours at 360 (consider this when choosing an appropriate value for the initialisation of the heat map).
  - **Hint** the improved `mod'` function from `Data.Fixed` operates better with real values.
  - The formula for conversion from HSV to RGB is given below:

When $0 \le H < 360$, $0 \le S \le 1$ and $0 \le V \le 1$:

$$C = V \times S$$

$$X = C \times (1 - |(H / 60°) \bmod 2 - 1|)$$

$$m = V - C$$

$$(R', G', B') = \begin{cases} (C, X, 0) & , 0° \le H < 60° \\ (X, C, 0) & , 60° \le H < 120° \\ (0, C, X) & , 120° \le H < 180° \\ (0, X, C) & , 180° \le H < 240° \\ (X, 0, C) & , 240° \le H < 300° \\ (C, 0, X) & , 300° \le H < 360° \end{cases}$$

$$(R,G,B) = ((R'+m)\times 255, (G'+m)\times 255, (B'+m)\times 255)$$

sourced from: https://www.rapidtables.com/convert/color/hsv-to-rgb.html

Contextual Values and Types

- Consider your heat map as a wrapper around its cells in some computational context.
- Create your heat map so that it conforms to the `Functor` typeclass
- Ensure that your heat map conforms to the **Functor rules**.
- In physics heat death occurs when all energy in a closed system (such as your simulation and the possibly the universe) is evenly distributed so no work can be done. Create a heat death function which sets every cell's value in your heat map to the same value. Apply this function by mapping it onto every cell in the grid.

You will need to create a system in the Haskell language which reads in the parameters for such a simulation, generates, and then outputs the resulting image in the PPM format Make use of as wide a variety of topics / techniques as you can within the confines of the problem.

Each of the following sub-questions can be answered independently. For example, if you are struggling with input then you can hardcode the values and work on subsequent sub-questions in the interim. Place each sub-question's code into separate `.hs` files.

a) **Data structures** [20]

You have been provided with a human-readable, serialized instance of a data structure defining the configuration of such a simulation:

- Name: Configuration
- Fields:
  1. A list of integer pairs called `points` which represents the initial hotspots in your simulation
  2. An integer pair called `dimensions` representing the number of rows and columns in your simulation
  3. An integer called iterations representing the number of turns to run your simulation for.
  4. A string called `imageFile` representing the name of image file to be created representing the final state of your simulation.

Additionally you must create types and type aliases as you see fit to represent the elements of your problem domain.

b) **Input and Output** [20]

You must write a function / set of functions which will read an instance of the dataset parameters from a specified file as well as save your result in the PPM format.

PPM is a text-based image format with the following structure:

```
P3
NUM_COLS NUM_ROWS
255
RED GREEN BLUE RED GREEN BLUE …
.
.
.
RED GREEN BLUE RED GREEN BLUE …
```

The first line indicates that this will be a colour image (P1 is black and white while P2 is greyscale). The second line indicates the image dimensions. The third line indicates the maximum value of any colour component. The remainder of the image is simply the red, green, and blue colour component values for each pixel given in sequence.

For example, a 5x4 yellow (255, 255, 0) image would look as follows:

```
P3
5 4
255
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
```

For more information see: https://en.wikipedia.org/wiki/Netpbm

The file extension should be `.ppm`. You have been provided with the GNU Image Manipulation Program (GIMP) which is capable of displaying PPM images.

c)	**Processing** Although the exact problem decomposition is up to you the following aspects are important: • Creation and initialisation of the heatmap • Implementation of the neighbourhood function • Applying the heat transfer function • Conversion to the PPM format • Bonus: Mapping from intensity to Colour (5 marks)	[20]
d)	**Language Features Used** This section of the marksheet is not meant to be prescriptive. Marks are awarded for language features used correctly in pursuit of the solution of the problem.	[20]
e)	**Contextual Values and Types** • Implementation of the Functor typeclass • Implementation of the heat death function • Mapping of the heat death function onto the heat map • Demonstrating compliance with the Functor laws	[20]

```haskell
module DataStructures
( Coordinate
, Dimensions
, Configuration(..)
, GridCell(..)
, HeatMap(..)
) where

type Coordinate = (Int, Int)
type Dimensions = (Int, Int)

data Configuration = Configuration { points :: [Coordinate]
                                   , dimensions :: Dimensions
                                   , iterations :: Int
                                   , imageFile :: String
                                   } deriving (Read, Show)

data GridCell = GridCell { value :: Int
                         , coord :: Coordinate
                         } | EmptyCell deriving (Read, Show)

data HeatMap cell = Grid [cell] | EmptyMap deriving (Read, Show)
```

```haskell
module Process
( buildHeatMap
, buildGrid
, distance
, neighbours
, toPPM
, heatDeath
, updateHeatMap
) where

import DataStructures
import Data.Fixed
import Control.Applicative

distance :: GridCell -> GridCell -> Int
distance cellA cellB =
    let
        (r1, c1) = coord cellA
        (r2, c2) = coord cellB
        deltaRows = abs (r1 - r2)
        deltaCols = abs (c1 - c2)
    in max deltaRows deltaCols
```

```haskell
neighbours :: GridCell -> [GridCell] -> [GridCell]
neighbours _ [] = []
neighbours cell cells = extractCells dists
    where d = distance cell
          close p = (d p) <= 1
          dists = [(d c, c) | c <- cells, close c, coord c /= coord cell]
          extractCells [] = []
          extractCells ((d,c):ds) = c : extractCells ds
```

```haskell
buildGrid :: Configuration -> [GridCell]
buildGrid cfg = let
                    rows = fst $ dimensions cfg
                    cols = snd $ dimensions cfg
                    ps = points cfg
                    rs = [0 .. rows - 1]
                    cs = [0 .. cols - 1]
                    coords = [(r, c) | r <- rs, c <- cs]
                    val c = if c `elem` ps then 360 else 240
                in [GridCell {value = val c, coord = c} | c <- coords]
```

```haskell
buildHeatMap :: Configuration -> HeatMap GridCell
buildHeatMap cfg = Grid g
    where g  = buildGrid cfg
```

```haskell
toPixel :: Int -> String
toPixel h = strRed ++ " " ++ strGreen ++ " " ++ strBlue
    where hdiv = fromIntegral h / 60
          hmod = hdiv `mod'` 2 - 1
          xsub = abs $ hmod
          x = 1 - xsub
          rgb
            | h >= 0 && h < 60 = (1, x, 0)
            | h >= 60 && h < 120 = (x, 1, 0)
            | h >= 120 && h < 180 = (0, 1, x)
            | h >= 180 && h < 240 = (0, x, 1)
            | h >= 240 && h < 300 = (x, 0, 1)
            | h >= 300 && h <= 360 = (1, 0, x)
          rgb255 (r, g, b) = (round $ r * 255, round $ g * 255, round $ b * 255)
          (red, green, blue) = rgb255 $ rgb
          strRed = show red
          strGreen = show green
          strBlue = show blue
toPixels :: [Int] -> String
toPixels [] = ""
toPixels (v:vs) = toPixel v ++ " " ++ toPixels vs

extractValues :: [GridCell] -> [Int]
extractValues [] = []
extractValues (c:cs) = value c : extractValues cs

toPPM :: Configuration -> HeatMap GridCell -> String
toPPM cfg (Grid grid) = strHeader ++ strPixels
    where strHeader = "P3\n" ++ strCols ++ " " ++ strRows ++ "\n255\n"
          strRows = show $ fst $ dimensions cfg
          strCols = show $ snd $ dimensions cfg
          strPixels = toPixels $ extractValues grid
```

```haskell
updateGrid :: [GridCell] -> [GridCell]
updateGrid [] = []
updateGrid cells@(c:cs) = c' : updateGrid cs
    where ns = neighbours c cs
          a = n / d
          n = fromIntegral $ sum $ extractValues ns
          d = fromIntegral $ length ns
          c' = GridCell {value = round a, coord = coord c}
```

```haskell
setCell :: Int -> GridCell -> GridCell
setCell _ EmptyCell = EmptyCell
setCell v cell = GridCell {value = v, coord = coord cell}

heatDeath :: HeatMap GridCell -> HeatMap GridCell
heatDeath hm = fmap (setCell 0) hm


updateHeatMap :: HeatMap GridCell -> HeatMap GridCell
updateHeatMap EmptyMap = EmptyMap
updateHeatMap (Grid grid) = Grid g
    where g = updateGrid grid


instance Functor HeatMap where
    fmap f EmptyMap = EmptyMap
    fmap f (Grid grid) = Grid g
        where g = map f grid
```

```haskell
import DataStructures
import InputOutput
import Process

runSim :: HeatMap GridCell -> Int -> HeatMap GridCell
runSim EmptyMap _ = EmptyMap
runSim hm 0 = hm
runSim hm n = run hm
    where run = foldl1 (.) (replicate n updateHeatMap)

main = do
    cfg <- loadConfig "Config.txt"
    let strConfig = show cfg
    putStrLn $ strConfig
    let hm = buildHeatMap cfg
    let (Grid cs) = hm
    showNeighbours cs
    let hm' = runSim hm (iterations cfg)
    let strPPM = toPPM cfg hm'
    let strFilename = imageFile cfg
    saveTextFile strFilename strPPM
```

```haskell
module InputOutput
( loadConfig
, saveConfig
, saveTextFile
) where

import DataStructures
import System.IO

loadConfig :: String -> IO Configuration
loadConfig strFilename = do
    hndFile <- openFile strFilename ReadMode
    strConfig <- hGetContents hndFile
    let recConfig = read strConfig :: Configuration
    putStrLn $ "Loaded: " ++ strConfig
    hClose hndFile
    return recConfig

saveTextFile :: String -> String -> IO ()
saveTextFile strFilename strContents = do
    hndFile <- openFile strFilename WriteMode
    hPutStrLn hndFile strContents
    hClose hndFile

saveConfig :: String -> Configuration -> IO ()
saveConfig strFilename recConfig = do
    let strConfig = show recConfig
    saveTextFile strFilename strConfig
```