UNIVERSITY OF JOHANNESBURG

FACULTY OF SCIENCE

HONOURS (Computer Science / IT)	APK CAMPUS
IT18X17 **FUNCTIONAL PROGRAMMING** **EXAMINATION EQUIVALENT ONLINE ASSESSMENT 2020** **PAPER B** **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 4 pages including the cover page

Iconic combinations

The Utopian Artisanal Electronic Entertainment Company and Heavy Industrial Concern is developing a retro style role playing game using fixed sized graphical assets of 128x128 pixels. There has been a recent pivot to adopting a functional programming style for their entire content production toolchain. You have been tasked with creating a tool which will combine images using a chroma-keying based approached to handle transparencies. Consider the combination of the following three images.

Art sourced from the liberated pixel cup under an open license by attribution: Zabin, Daneeklu, Jetrel, Hyptosis, Redshrike, Bertram.

Data format

- An image is represented as a grid / lattice of pixels. Pixels are represented as a 3-tuple of integers in the range 0 – 255 as per the RGB colour model: https://en.wikipedia.org/wiki/RGB_color_model
- Each cell in the grid should be uniquely and invariantly associated with a pair of coordinates reflecting their position within the grid.
- The RGB colour for magenta (R: 255, G: 0, B: 255) is special and represents transparency as per the chroma keying approach: https://en.wikipedia.org/wiki/Chroma_key
- All data provided to you is in the form of 128 x 128 pixel images. Images do store their dimensions and the merge operation is only defined on identically sized images.

Operations

- **Merge:**
  - When two images are merged their pixels a with corresponding coordinates are merged. This is done by averaging their colour components unless one of the operands is magenta.
- **Replace:**
  - All occurrences of one colour in an image are replaced with another.
- **Greyscale**
  - All pixels in an image are set to grey by setting each colour component to the average value of all three colour components.
- **getColourCoords**
  - All the coordinates containing a specified colour are returned.

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).
- Grayscale images may be output using either the P3 or P2 format at your discretion.

Contextual Values and Types

- Consider each cell as a wrapper around its colour within the computational context of having a location.
- Create your grid cell type so that it conforms to the `Functor` typeclass
- Ensure that your grid cell conforms to the **Functor rules**.
- **Bonus:**
  - Extend your grid cell so that it implements the Applicative Functor typeclass.
  - Do your images meet the requirement for potentially being Monoids? If so, describe in a comment why this either is or is not the case.

You will need to create a system in the Haskell language which reads in the three images and associated paramaters, 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: TileSet
- Fields:
  1. A pair representing the size of all of the images in pixels
  2. A list of lists of 3-tuples. Each list element represents an image, each 3-tuple represents the RGB values for each pixel. The coordinates of each pixel are not given but they are stored in row-major order.

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 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 of images • Implementation of the operations • Applying the per-image and per-pixel operations • Conversion to the PPM format	[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 functions using the Functor wrapper
	• Mapping of a function via the typeclass
	• Demonstrating compliance with the Functor laws
	• Bonus (5 marks)
	  ○ Extending the contextual type to an Applicative Functor
	  ○ Discussing the adherence to the properties of a Monoid
		[20]

```haskell
module DataStructures
( Pixel
, Dimensions
, Coordinate
, rgbChromaKey
, TileSet(..)
, GridCell(..)
, Image(..)
) where

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

rgbChromaKey = (255, 0, 255) :: (Int, Int, Int)

data TileSet = TileSet { dimensions :: Dimensions
                       , images :: [[Pixel]]
                       , outImage :: String
                       } deriving (Read, Show)

data GridCell t = Cell { value :: t
                       , position :: Coordinate
                       } | EmptyCell deriving (Read, Show)

data Image = Image { pixels :: [GridCell Pixel]
                   , size :: Dimensions
                   } deriving (Read, Show)
```

```haskell
module Process
( mergePixel
, mergeGridCell
, mergeImages
, buildImages
, toPPM
) where

import DataStructures
import Control.Applicative

mergePixel :: Pixel -> Pixel -> Pixel
mergePixel (r1, g1, b1) (r2, g2, b2)
    | (r1, g1, b1) == rgbChromaKey = (r2, g2, b2)
    | (r2, g2, b2) == rgbChromaKey = (r1, g1, b1)
    | otherwise = ((r1 + r2) `div` 2, (g1 + g2) `div` 2, (b1 + b2) `div` 2)

mergeGridCell :: GridCell Pixel -> GridCell Pixel -> GridCell Pixel
mergeGridCell EmptyCell _ = EmptyCell
mergeGridCell _ EmptyCell = EmptyCell
mergeGridCell c1 c2  =
    let
        coord = position c1
        p1 = value c1
        p2 = value c2
    in Cell {value = mergePixel p1 p2, position = coord}

mergeImages :: Image -> Image -> Image
mergeImages i1 i2 = Image { pixels = mp, size = size i1}
    where ps1 = pixels i1
          ps2 = pixels i2
          mp = zipWith mergeGridCell ps1 ps2

enGrid :: Dimensions -> [Pixel] -> [GridCell Pixel]
enGrid (rows, cols) ps = [ Cell {value = p, position = (c, r)} | r <- [0 .. rows - 1], c <- [0 .. cols - 1],

buildImages :: TileSet -> [Image]
buildImages ts = unPack $ images ts
    where
          unPack [] = []
          unPack (i:is) = Image { pixels = enGrid d i, size = d} : unPack is
          d = dimensions ts

toPixel :: GridCell Pixel -> String
toPixel EmptyCell = ""
toPixel cell = strRed ++ " " ++ strGreen ++ " " ++ strBlue ++ " "
    where (r, g, b) = value cell
          strRed = show r
          strGreen = show g
          strBlue = show b
```

```haskell
toPixelString :: [GridCell Pixel] -> String
toPixelString [] = ""
toPixelString (c:cs) = toPixel c ++ " " ++ toPixelString cs

toPPM :: Image -> String
toPPM i = strHeader ++ strPixels
    where strHeader = "P3\n" ++ strCols ++ " " ++ strRows ++ "\n255\n"
          s = size i
          strRows = show $ fst s
          strCols = show $ snd s
          strPixels = toPixelString (pixels i)

instance Functor GridCell where
    fmap f g = Cell {value = f v, position = p}
        where v = value g
              p = position g
```

```haskell
module InputOutput
( loadTileSet
, saveTextFile
, saveTileSet
, saveImage
) where

import DataStructures
import Process
import System.IO

loadTileSet :: String -> IO TileSet
loadTileSet strFilename = do
    hndFile <- openFile strFilename ReadMode
    strTileSet <- hGetContents hndFile
    let recTileSet = read strTileSet :: TileSet
    putStrLn $ "Loaded: " ++ strTileSet
    hClose hndFile
    return recTileSet

saveImage :: String -> Image -> IO ()
saveImage strFilename recImage= do
            let strImage = toPPM recImage
            saveTextFile strFilename strImage
```

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

saveTileSet :: String -> TileSet -> IO ()
saveTileSet strFilename recTileSet = do
        let strTileSet = show recTileSet
        saveTextFile strFilename strTileSet
```

```
import DataStructures
import Process
import InputOutput

main = do
    ts <- loadTileSet "TileSet.txt"
    let [a, b, c] = buildImages ts
    let i = mergeImages a b
    let iFin = mergeImages i c
    saveImage (outImage ts) iFin
    putStrLn "Done!"
```