ECM2418 Computer Languages and Representations Continuous Assessment 1: Functional Programming

Dr David Wakeling

Handed out	Handed in
Thursday 8th October 2020 (T1:03)	Thursday 5th November 2020 (T1:07)

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

The solutions that you produce should all run on

```
https://www.tutorialspoint.com/compile_haskell_online.php
```

All students are reminded of the University regulations on academic honesty and plagiarism.

Question 1

A *puzzle* requires one to find six-digit numbers where:

1. all digits are different;
2. alternate digits are even and odd, or odd and even;
3. alternate digits differ by more than two;
4. the first and middle pairs of digits form numbers that are both multiples of the last.

One solution is 496307, and another is 692703.

A Haskell program to find all solutions to this puzzle has the following structure

```
main :: IO ()
main
  =  putStrLn (show (filter isSolution possibles))
```

Question 1.1

Show a Haskell function "rule1" that given a tuple representing a six-digit number, returns true if all digits are different, so that

```
rule1 (4,9,6,3,0,7)
   ===>  True
```

```
rule1 :: (Int, Int, Int, Int, Int, Int) -> Bool
rule1 (x1, x2, x3, x4, x5, x6)
  =  xs == nub xs
     where
     xs = [ x1, x2, x3, x4, x5, x6 ]
```

(10 marks)

Question 1.2

Show a Haskell function "rule2" that given a tuple representing a six-digit number, returns true if alternate digits are even and odd, or odd and even, so that

```
rule2 (4,9,6,3,0,7)
   ===>  True
```

```
rule2 :: (Int, Int, Int, Int, Int, Int) -> Bool
rule2 ( x1, x2, x3, x4, x5, x6 )
  =   ( evens [ x1, x3, x5 ] && odds  [ x2, x4, x6 ] )
  ||  ( odds  [ x1, x3, x5 ] && evens [ x2, x4, x6 ] )

evens :: [ Int ] -> Bool
evens
  =  all even

odds :: [ Int ] -> Bool
odds
  =  all odd
```

(10 marks)

Question 1.3

Show a Haskell function "`rule3`" that given a tuple representing a six-digit number, returns true if alternate digits differ by more than two, so that

```
rule3 (4,9,6,3,0,7)
   ===>  True

rule3 :: (Int, Int, Int, Int, Int, Int) -> Bool
rule3 (x1, x2, x3, x4, x5, x6)
  =   abs( x1 - x2 ) > 2
  &&  abs( x2 - x3 ) > 2
  &&  abs( x3 - x4 ) > 2
  &&  abs( x4 - x5 ) > 2
  &&  abs( x5 - x6 ) > 2
```

(10 marks)

Question 1.4

Show a Haskell function "`rule4`" that given a tuple representing a six-digit number, returns true if the first and middle pairs of digits form numbers that are both multiples the number formed by the last pair of digits.

```
rule4 (4,9,6,3,0,7)
   ===>  True
```

(because 49 and 63 are both multiples of 7).

```
rule4 :: (Int, Int, Int, Int, Int, Int) -> Bool
rule4 ( x1, x2, x3, x4,  0,  0 )
  =  False
rule4 ( x1, x2, x3, x4, x5, x6 )
  =  x1x2 `mod` x5x6 == 0 && x3x4 `mod` x5x6 == 0
     where
     x1x2 = 10 * x1 + x2
     x3x4 = 10 * x3 + x4
     x5x6 = 10 * x5 + x6
```

(10 marks)

Question 1.5

Show a Haskell value "possibles" that gives a list of all possible tuples representing six-digit numbers, so that

```
possibles
   ===>  [ (0,0,0,0,0,0), (0,0,0,0,0,1), ... (9,9,9,9,9,9) ]

-- students not taught list comprehensions.
possibles :: [ (Int, Int, Int, Int, Int, Int) ]
possibles
  =  concatMap (\a ->
       concatMap (\b ->
         concatMap (\c ->
           concatMap (\d ->
             concatMap (\e ->
               concatMap (\f -> [ (a, b, c, d, e, f) ])
                     xs)
                   xs)
                 xs)
               xs)
             xs)
           xs
     where xs = [0..9]
```

(10 marks)

Question 1.6

Show a Haskell function "isSolution" that takes a tuple representing a six-digit number and returns true if it is a solution to the puzzle, so that

```
isSolution (1,2,3,4,5,6)
  ===>  False
isSolution (4,9,6,3,0,7)
  ===>  True

isSolution :: (Int, Int, Int, Int, Int, Int) -> Bool
isSolution t
  =  rule1 t && rule2 t && rule3 t && rule4 t
```

(10 marks)

Question 2

The late John Conway was a mathematician who became known outside the world of mathematics for his invention of the *Game of Life*. The game involves configuring a cellular automaton and then observing how it evolves according to three rules. The cells are organised as a two-dimensional grid, and are either live or dead. The three rules of evolution are:

- Any live cell with two or three live neighbours survives.
- Any dead cell with three live neighbours becomes alive.
- All other live cells die, and all other dead cells stay dead.

The next generation is created by applying the rules simultaneously to every cell in the current generation, so that births and deaths occur simultaneously. One famous pattern of cells, the *glider*, shows how the rules of evolution are applied. See Figure 1.

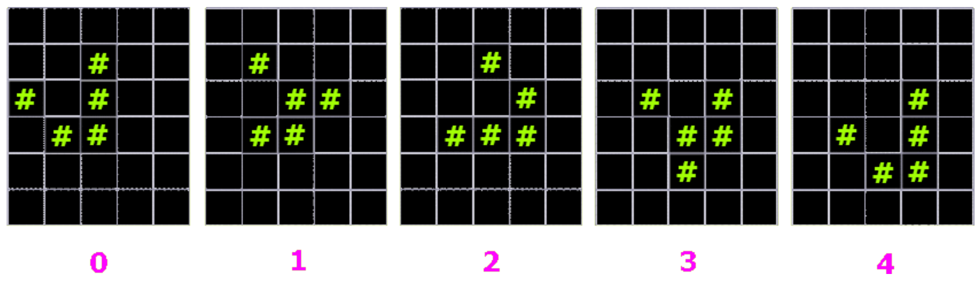


Figure 1: The glider pattern.

A Haskell program to pretty-print the first eight generations of the glider on a six-by-six grid has the following structure

```
main :: IO ()
main
  =  putStrLn (pretty (take 8 (visualisation 5 5 (evolution glider))))
```

Question 2.1

Show a Haskell function “`pretty`” that may be used to pretty-print a possibly-infinite sequence of generations, so that

```
pretty [ [ [ ’a’,’b’ ], [ ’c’,’d’ ] ]
       , [ [ ’e’,’f’ ], [ ’g’,’h’ ] ]
```

```
          , [ [ 'i','j' ], [ 'k','l' ] ] ]
   ===>  "ab\ncd\nef\ngh\nij\nkl\n"
```

(note letters are used for clarity, and '`\n`' is the newline character).

```
pretty :: [ [ [ Char ] ] ] -> [ Char ]
pretty
  =  concatMap unlines
```

(10 marks)

Question 2.2

Given the type

```
type Point
  = ( Int, Int )
```

and definition

```
glider :: [ Point ]
glider
  =  [ (0, 2), (1, 3), (2, 1), (2, 2), (2, 3) ]
```

show a Haskell function that may be used to visualise a possibly-infinite sequence of generations on a grid of a given width and height, so that

```
visualisation 5 5 [ glider ]
  ===>  [[ "......", "..#...", "#.#...", ".##...", "......", "......" ]]
```

(note that this corresponds to Panel 1 of Figure 1, origin top-left).

```
visualisation :: Int -> Int -> [ [ Point ] ] -> [ [ [ Char ] ] ]
visualisation w h
  =  map (visualise w h)

visualise :: Int -> Int -> [ Point ] -> [ [ Char ] ]
visualise w h gen
  =  map (\y -> map (\x -> hashDot (isAlive gen (x,y))) [0..w]) [0..h]
```

```
isAlive :: [ Point ] -> Point -> Bool
isAlive pts pt
  =  pt `elem` pts

hashDot :: Bool -> Char
hashDot True  = '#'
hashDot False = '.'
```

(10 marks)

Question 2.3

Show a Haskell function "`evolution`" that uses the "`iterate`" function to produce a potentially-infinite sequence of generations of live cells according to the three rules of evolution, so that

```
evolution [ (0, 2), (1, 3), (2, 1), (2, 2), (2, 3) ]
  ===>  [ [ (0, 2), (1, 3), (2, 1), (2, 2), (2, 3) ]
        , [ (1, 3), (2, 2), (2, 3), (1, 1), (3, 2) ]
        , [ (1, 3), (2, 3), (3, 2), (2, 1), (3, 3) ]
        , ...
```

```
evolution :: [ Point ] -> [ [ Point ] ]
evolution
  =  iterate evolve

evolve :: [ Point ] -> [ Point ]
evolve gen
  =  nub (survivors gen ++ births gen)

survivors :: [ Point ] -> [ Point ]
survivors gen
  =  filter (survives gen) gen

survives :: [ Point ] -> Point -> Bool
survives gen pt
  =  countLiveNeighbours gen pt `elem` [2,3]

births :: [ Point ] -> [ Point ]
births gen
  =  concatMap (born gen) gen
```

```
born :: [ Point ] -> Point -> [ Point ]
born gen pt
  =  filter (\qt -> countLiveNeighbours gen qt `elem` [3])
            (neighbours pt)

neighbours :: Point -> [ Point ]
neighbours (x,y)
  =  [ (x-1, y-1), (x, y-1), (x+1, y-1)
     , (x-1, y  ),           (x+1, y  )
     , (x-1, y+1), (x, y+1), (x+1, y+1)
     ]

liveNeighbours :: [ Point ] -> Point -> [ Point ]
liveNeighbours gen pt
  =  filter (isAlive gen) (neighbours pt)

countLiveNeighbours :: [ Point ] -> Point -> Int
countLiveNeighbours gen pt
  =  length (liveNeighbours gen pt)
```

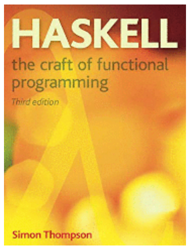
(20 marks)

Assessment Materials and Criteria

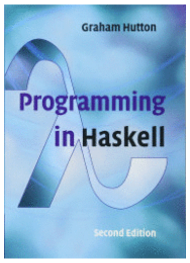
You should submit a ".zip" file to the EBART system.

Your work will be assessed on the grounds of clarity (is what it computes obvious?) and correctness (is what it computes right?). Clarity is all about the appropriate use of type signatures, meaningful variable names, higher-order functions, and functions from the standard prelude. Correctness is all about computing the outputs from inputs, as described in each question.

Readings



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



R. Bird, *Programming in Haskell (Second Edition)*, Cambridge University Press, 2016, ISBN 978-1316626229.