Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2019-2020

Homework 1

Due date: 8 March 2020, Sunday, 23:55

1 Objectives

This assignment aims to help you practice Haskell basics by having you implement a set of simple functions over two dimensional grids, which will be represented by nested lists.

2 Problem Definition

In this assignment you will be working with grids, which we will represent as one-deep nested lists of any type, i.e. `[[a]]`. Note that each nested list (i.e. *row*) will have the same number of elements. Here are a few basic examples:

```
numGrid  = [[1, 7, 2], [28, 0, -42], [11, 200, 33]]
unitGrid = [[0]]
boolGrid = [[True, False, False],
            [False, True, False],
            [False, False, True]]
```

The homework is split into three parts in order of increasing difficulty. [1] In each part, you will implement one or more functions. The split is only logical, every function will be tested and graded independently of each other.

3 Functions to Implement

3.1 The Fellowship of the Grid

3.1.1 form - 25 points

The `form` function is your bread and butter for creating grids from flat lists. It takes a list and a shape (a (height, width) tuple) as arguments, and then creates a grid with the given shape from the elements of the list in row major order. The function signature and a few example runs are shown below.

[1] Not to worry: easiest, easier and easy.

```haskell
form :: [a] -> (Int, Int) -> [[a]]
-- form list shape -> grid with values from list
```

```
ghci> form [1,2,3] (1, 3)
[[1,2,3]]
ghci> form [1,2,3] (3, 1)
[[1],[2],[3]]
ghci> form [1..10] (2, 5)
[[1,2,3,4,5],[6,7,8,9,10]]
ghci> form [1..10] (5, 2)
[[1,2],[3,4],[5,6],[7,8],[9,10]]
ghci> form "abcdefghi" (3, 3)
["abc","def","ghi"]
ghci> form [True,False,True,False,True,False,True,False,True] (3, 3)
[[True,False,True],[False,True,False],[True,False,True]]
```

Note that no erroneous input will be provided. By this, we mean that it is guaranteed that the length of the input list will always be equal to the product of the height and width. Implementing some sort of error checking could be beneficial for your own debugging, however.

3.1.2 constGrid - 5 points

`constGrid` is an even more reduced form of the `form` function. Given a single, constant value and a shape, the goal is to create a grid with the given shape, filled with the given constant value.

```haskell
constGrid :: a -> (Int, Int) -> [[a]]
-- constGrid x shape -> grid filled with x
```

```
ghci> constGrid False (1, 1)
[[False]]
ghci> constGrid False (1, 3)
[[False,False,False]]
ghci> constGrid 0 (5, 5)
[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0]]
ghci> constGrid 'a' (2, 4)
["aaaa","aaaa"]
```

3.1.3 flatten - 5 points

`flatten` is like the inverse of `form`. Given a grid, the function will put the rows together again to create a one dimensional list.

```haskell
flatten :: [[a]] -> [a]
-- flatten grid -> list with values from grid
```

```
ghci> flatten [[1, 7], [3, 5], [0, -2]]
[1,7,3,5,0,-2]
ghci> flatten $ form [5..16] (3, 4)
[5,6,7,8,9,10,11,12,13,14,15,16]
ghci> flatten [["hi"]]
["hi"]
```

3.1.4 access - 5 points

Given a grid and a tuple of two zero-indexed indices (i, j), `access` will simply return the value at the given position in the provided grid.

```haskell
access :: [[a]] -> (Int, Int) -> a
-- access grid position -> grid element at position
```

```
ghci> access [[1, 2, 3], [4, 5, 6], [7, 8, 9]] (0, 0)
1
ghci> access [[1, 2, 3], [4, 5, 6], [7, 8, 9]] (0, 2)
3
ghci> access [[1, 2, 3], [4, 5, 6], [7, 8, 9]] (2, 2)
9
```

3.2 The Two Signatures

3.2.1 slice - 10 points

The `slice` function takes a grid and two sets of tuples (i1, i2) and (j1, j2). With these inputs, the slice function will extract the subgrid containing rows in the range (i1, i2) and columns in the range (j1, j2). The start indices (i1 and j1) are inclusive, while the end indices are exclusive (in Python style).

```
slice :: [[a]] -> (Int, Int) -> (Int, Int) -> [[a]]
-- slice grid rowInds colInds -> subgrid between given inds
```

```
ghci> grid = form [25..49] (5, 5)
ghci> grid
[[25,26,27,28,29],[30,31,32,33,34],[35,36,37,38,39],[40,41,42,43,44],
[45,46,47,48,49]]
ghci> putStrLn $ intercalate "\n" $ map show grid -- fancier print
[25,26,27,28,29]
[30,31,32,33,34]
[35,36,37,38,39]
[40,41,42,43,44]
[45,46,47,48,49]
ghci> slice grid (0, 2) (0, 1)
[[25],[30]]
ghci> slice grid (1, 3) (1, 3)
[[31,32],[36,37]]
ghci> slice grid (1, 4) (1, 4)
[[31,32,33],[36,37,38],[41,42,43]]
ghci> slice grid (1, 4) (1, 5)
[[31,32,33,34],[36,37,38,39],[41,42,43,44]]
ghci> -- intercalate can be found in Data.List. From now on:
ghci> -- fancyPrint = putStrLn . intercalate "\n" . map show
ghci> -- you do not need to know putStrLn etc., the only purpose
ghci> -- of the function is to illustrate via line-by-line printing
```

Once again, there will be no erroneous input. Empty and out of range slices will also not be tested, we will assume that they are *undefined behavior* and leave it up to your implementation. Thus, it is guaranteed that ($0 \leq$ i1 $<$ i2 $\leq$ number of rows), and ($0 \leq$ j1 $<$ j2 $\leq$ number of columns) for a given grid. Also, the grids will be small enough that you do not have to worry about overflowing/underflowing the `Int` type.

3.2.2 vcat - 5 points

Given two grids, `vcat` will return a new grid with the two input grids concatenated vertically.

```
vcat :: [[a]] -> [[a]] -> [[a]]
-- vcat grid1 grid2 -> grid1 and grid2 stacked vertically
```

```
ghci> grid = [[1, 2, 3], [4, 5, 6]]
ghci> vcat grid grid
[[1,2,3],[4,5,6],[1,2,3],[4,5,6]]
ghci> fancyPrint $ vcat grid grid
[1,2,3]
[4,5,6]
[1,2,3]
[4,5,6]
```

3.2.3 hcat - 5 points

The horizontal version of `vcat`, `hcat` will return a new grid with the two input grids concatenated horizontally.

```
hcat :: [[a]] -> [[a]] -> [[a]]
-- hcat grid1 grid2 -> grid1 and grid2 stacked horizontally
```

```
ghci> grid = [[1, 2, 3], [4, 5, 6]] -- same as the vcat example
ghci> hcat grid grid
[[1,2,3,1,2,3],[4,5,6,4,5,6]]
ghci> fancyPrint $ hcat grid grid
[1,2,3,1,2,3]
[4,5,6,4,5,6]
```

3.2.4 without - 10 points

`without` is a function similar to slice in that it takes a grid and two sets of indices (i1, i2) and (j1, j2) as input. However, this time the function removes the rows and columns in the given indices and returns a new grid with those rows and columns removed. How exactly? See the illustrations.

```
without :: [[a]] -> (Int, Int) -> (Int, Int) -> [[a]]
-- without grid rowInds colInds -> grid with given rows and cols removed
```

```
ghci> grid = form [25..49] (5, 5) -- same as the slice example
ghci> fancyPrint grid
[25,26,27,28,29]
[30,31,32,33,34]
[35,36,37,38,39]
[40,41,42,43,44]
[45,46,47,48,49]
ghci> without grid (0, 1) (0, 1)
[[31,32,33,34],[36,37,38,39],[41,42,43,44],[46,47,48,49]]
ghci> without grid (0, 0) (0, 1)
[[26,27,28,29],[31,32,33,34],[36,37,38,39],[41,42,43,44],[46,47,48,49]]
ghci> without grid (0, 0) (0, 0)
[[25,26,27,28,29],[30,31,32,33,34],[35,36,37,38,39],[40,41,42,43,44],
[45,46,47,48,49]]
ghci> without grid (0, 3) (0, 0)
[[40,41,42,43,44],[45,46,47,48,49]]
ghci> without grid (0, 3) (0, 3)
[[43,44],[48,49]]
ghci> without grid (2, 3) (1, 4)
[[25,29],[30,34],[40,44],[45,49]]
```

The same constraints as `slice` apply on the indices, with the exception that indices may be equal (i1 = i2 or j1 = j2), which means that no rows or columns should be removed. There will be no cases that attempt to remove all the rows or columns of the grid (i.e. (i2 - i1 $<$ number of rows) and (j2 - j1 $<$ number of columns)).

3.3 Return of the Non-trivial

3.3.1 matches2d - 30 points

Our weightiest function, `matches2d`, takes a grid and a pattern (also a grid) as inputs, and searches for the pattern grid in the original grid in sliding window fashion; returning every position (corresponding to the position of the top-left corner of the pattern) in which the pattern was found.

```
matches2d :: Eq a => [[a]] -> [[a]] -> [(Int, Int)]
-- matches2d grid pattern -> indices where the grid matches the pattern
```

```
ghci> grid = constGrid 0 (3, 3)
ghci> fancyPrint grid
[0,0,0]
[0,0,0]
[0,0,0]
ghci> pattern = [[0]]
ghci> matches2d grid pattern
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)]
ghci> pattern = [[1]]
ghci> matches2d grid pattern
[]
ghci> grid = [[0,0,0,0],[0,1,1,0],[0,1,1,0],[0,0,0,0]]
ghci> fancyPrint grid
[0,0,0,0]
[0,1,1,0]
[0,1,1,0]
[0,0,0,0]
ghci> matches2d grid [[1, 1], [1, 1]]
[(1,1)]
ghci> matches2d grid [[1, 1], [0, 0]]
[(2,1)]
ghci> matches2d grid [[1, 0], [0, 0]]
[(2,2)]
ghci> grid = [[1,0,1,0,1],[0,1,0,1,0],[1,0,1,0,1],[0,1,0,1,0],[1,0,1,0,1]]
ghci> fancyPrint grid
[1,0,1,0,1]
[0,1,0,1,0]
[1,0,1,0,1]
[0,1,0,1,0]
[1,0,1,0,1]
ghci> pattern = [[0,1,0],[1,0,1],[0,1,0]]
ghci> fancyPrint pattern
[0,1,0]
[1,0,1]
[0,1,0]
ghci> matches2d grid pattern
[(0,1),(1,0),(1,2),(2,1)]
```

The order in which you output the positions does not matter as they will be sorted during testing. Obviously, the same positions should not be repeated. If the pattern is larger than the grid, the result will be empty.

Time Complexity of the Solution
We expect your solution to this problem to have a proper running time. A naive imperative language implementation (with four nested loops: grid rows, grid columns, pattern rows, pattern columns) would perform $pq(m-p+1)(n-q+1)$ comparisons for an m by n grid and a p by q pattern. Your solution should be of similar order. In particular, you should not repeatedly and unnecessarily traverse the grid. Remember that lists are good for iterating through, but not random access. **15 out of the 30 points** for this function will come from time limited test cases.
Here is an example test case. Look to achieve a running time of under 60 seconds on any of the ineks, *without compiler optimizations*, for this case. This does not mean that a function achieving a running time of 55 seconds will certainly get through all the other test cases, you should easily be able to go a few times lower than that.

```
ghci> g = form [1..10000000] (1000, 10000) :: [[Int]]
ghci> -- force g to be fully evaluated first to create the grid
ghci> length $ intercalate [0] g
10000999
ghci> -- the line below should take < 60 sec on an inek, ideally much lower
ghci> matches2d g [[300, 301], [10300, 10301]]
[(0,299)]
```

4 Tips

- Maximizing the use of higher order functions from the `Prelude` will make your job easier. You can also import `Data.List` or any other standard library module you want to use, but you should not really need anything else.
- As explained in the function descriptions, there will be no erroneous input, or dubious edge cases producing empty grids. Focus on cleanly solving the problem at hand using a functional style.
- Remember that there is no running time constraint on the functions apart from `matches2d`. They should still terminate quickly (under a second, but tests will allow up to a minute) for inputs with a size smaller than or equal to (1000, 1000).

5 Submission

Submission will be done via cengclass. Submit a single Haskell source file named `HW1.hs`. Keep the signatures and module declaration the same as the template file.

6 Grading

Each function will be tested and graded **independently**. However, you still need to make sure your code file **as a whole** can be compiled by `ghc`; so if there are functions you do not intend to implement, make them return a default value you like, or keep the `undefined`s as in the template. The types of the functions **must** obey the provided specifications.
If your file has a different name than expected or you have multiple files, 5 points will be deducted from your final grade. If your code does not compile (but the compilation errors can be fixed trivially), this will also result in a 5 point deduction. These deductions will *not* reduce your grade below zero.

7 Regulations

1. **Programming Language:** You must write your code in Haskell. Your submission has to be compilable with `ghc/ghci` on the ineks.

2. **Late Submission:** You have been given 10 late days in total and at most 3 of them can be used for one assignment. After 3 days, submissions will no longer be accepted and you will receive a grade of 0, no matter the circumstances. The extended deadline is therefore 11 March 2020, Wednesday, 23:55 **sharp**.

3. **Cheating: We have a zero tolerance policy on cheating**. People involved in cheating will be punished according to the university regulations.

4. **Newsgroup:** You must follow the course category on COW for discussions and possible updates on a daily basis.

5. **Evaluation:** Your code will be evaluated automatically using the "black-box" testing technique, so make sure to obey the specifications.