Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2021-2022

Programming Exam 3

Due date: 16 April 2022, Saturday, 23:59

1 Problem Definition

Now that we've had our fun with robots, it's time to move on to something else! In this third programming examination, we will be practicing slightly more advanced concepts. In the first part, we will be working on a custom data type, including a tiny amount of functional abstraction. In the second part, we will move one step further by defining another custom data type and dealing with some operations 2D lists.

The core idea revolves around simulating a bunch of soldiers getting into formation! All functions will be tested independently, but are designed to *build up*. Thus, either using or copying and modifying previous functions functions when defining new functions should make life easier.

1.1 General Specifications

- The signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define any number of helper function(s) as you need.
- You can (and sometimes even should) use previous functions inside your new function definitions. The exercises are designed to build up!
- You are allowed to import `Data.List` and `Data.Maybe` for this exam, if you want to. The `Prelude` is more than enough though! Skim the official docs for the `Prelude` and be on the lookout for useful built-ins!

1.2 Quick VPL Tips

- Evaluation is fast. If evaluation seems to hang for more than a few seconds, your code is entering an infinite loop or has an abnormal algorithmic complexity. Or you've lost your connection, which is *much* less likely!
- Although the run console does not support keyboard shortcuts, it should still be possible to copy and paste using the right-click menu (Tested on latest versions of Firefox and Chrome).

- Get familiar with the environment. Press the plus shapes button on the top-left corner to see all the options. You can download/upload files, change your font and theme, switch to fullscreen etc. Useful stuff!

2 Part I - Warming up with Abstractions

2.1 bubbleHumans (15 points)

Imagine you are a soldier standing in single file with some others. Suddenly you get the order to order yourselves by height, the aim is having the tallest at the front and the shortest at the back.

Now, as you can image it would not be very practical to try to apply something like mergesort or quicksort in this case since you do not have a view of the entire file. Instead you simply check the person in front of you, and swap with them in case you are taller. Otherwise you stay, possibly waiting for someone in the back to swap with you if they are taller than you are. Everyone in the line repeats this step until no one can swap any more and voila, the whole file is sorted! This is similar to bubblesort, but instead of bubbling a single value up at each step, multiple values swap once in parallel every time. We will henceforth call each step *parallel bubbling*.

The goal in this function is simulating **a single step** of this process given a list of numbers: Imagine they are the soldiers' heights and the first element is the front of the file. All values having a smaller value on their immediate left will be swapped. This is unless the value on the left is already swapping, of course: Earlier elements in the list have priority. An example for clarification:

```
[1, 3, 5, 4, 2, 6]
```

Examine the list: First observe that 3 will swap with 1 since it's larger. Then, 5 could swap with 3; but 3 is already swapping with 1 so 5 has to wait. 4 sees that it is smaller than 5 and 2 is smaller than 4, so they do not move. Finally, 6 is larger than 2 so it swaps with 2. This is the result of the first step, with 1-3 and 2-6 swapping:

```
[3, 1, 5, 4, 6, 2]
```

Here are some example runs that should clarify further:

```
bubbleHumans :: [Int] -> [Int]
```

```
*PE3> bubbleHumans []
[]
*PE3> bubbleHumans [1, 2, 3]
[2,1,3]
*PE3> bubbleHumans [1, 2, 3, 4]
[2,1,4,3]
*PE3> bubbleHumans [1, 3, 5, 4, 2, 6]
[3,1,5,4,6,2]
*PE3> bubbleHumans [3, 1, 5, 4, 6, 2]
[3,5,1,6,4,2]
*PE3> bubbleHumans [3, 5, 1, 6, 4, 2]
[5,3,6,1,4,2]
*PE3> bubbleHumans [5, 3, 6, 1, 4, 2]
[5,6,3,4,1,2]
*PE3> bubbleHumans [5, 6, 3, 4, 1, 2]
[6,5,4,3,2,1]
PE3> bubbleHumans [10, -5, 2, 7, 16, 29, 13, 2, 44, 58, 32, 24, 10, 50, 242]
[10,2,-5,16,7,29,13,44,2,58,32,24,50,10,242]
```

Objective: Warming up using previous knowledge by implementing a simple algorithm on lists.

Hint: *Parallel* is only in terms of time, when thinking about the simulation of humans swapping, since multiple pairs will swap at the same time. Your implementation of it will of course be sequential, swapping from the start of the list towards the end in order.

2.2 bubblePrivates (25 points)

Now it's time to put our new knowledge to use. This function will do the exact same thing as before, i.e. perform a single parallel bubbling step. But instead of using simple numbers, we will introduce a new data type for the soldiers standing in file:

```
data Private = Private { idNumber :: Int, height :: Int, timeToSwap :: Int }
               deriving Show
```

Hint: Check the recitation PDF/video if you've forgotten *record syntax*, which is used here in the definition of `Private`.

Every private (lowest ranked soldier) has its own ID number, its height and also the time it takes for them to perform a single swap, depending on how fast or slow they are. Please note that all fields are positive, i.e. **idNumber** > 0, **height** > 0 and **timeToSwap** > 0 for all given privates.

Privates do not always have to order themselves by height; that is only for when they are preparing to march! When taking counts for example, they need to be ordered by ID. To allow some flexibility, our function will take a key function as argument, along with a bool specifying whether the order should be ascending or descending (`True` for descending, `False` for ascending). Note that privates **having the same key value should not swap**.

Here's the function signature and some example runs. Do note that the key function can be arbitrarily complex, anything taking a private and turning it to an integer will work:

```
bubblePrivates :: (Private -> Int) -> Bool -> [Private] -> [Private]
```

```
*PE3> pri = Private -- shorten to not spill
*PE3> -- Same list for first 3 examples, manually formatted for clarity
*PE3> [pri 1 181 5, pri 3 173 17, pri 2 170 9]
[Private {idNumber = 1, height = 181, timeToSwap = 5}
,Private {idNumber = 3, height = 173, timeToSwap = 17}
,Private {idNumber = 2, height = 170, timeToSwap = 9}]
*PE3> bubblePrivates height True [pri 1 181 5, pri 3 173 17, pri 2 170 9]
[Private {idNumber = 1, height = 181, timeToSwap = 5}
,Private {idNumber = 3, height = 173, timeToSwap = 17}
,Private {idNumber = 2, height = 170, timeToSwap = 9}]
*PE3> bubblePrivates height False [pri 1 181 5, pri 3 173 17, pri 2 170 9]
[Private {idNumber = 3, height = 173, timeToSwap = 17}
,Private {idNumber = 1, height = 181, timeToSwap = 5}
,Private {idNumber = 2, height = 170, timeToSwap = 9}]
*PE3> bubblePrivates idNumber False [pri 1 181 5, pri 3 173 17, pri 2 170 9]
[Private {idNumber = 1, height = 181, timeToSwap = 5}
,Private {idNumber = 2, height = 170, timeToSwap = 9}
,Private {idNumber = 3, height = 173, timeToSwap = 17}]
*PE3> bubblePrivates (negate . height) True [pri 42 193 15, pri 51 175 9,
pri 15 166 2, pri 6 182 14, pri 54 179 17, pri 59 190 12, pri 7 175 4,
pri 24 189 6, pri 55 182 9, pri 34 167 4]
[Private {idNumber = 51, height = 175, timeToSwap = 9},Private {idNumber =
42, height = 193, timeToSwap = 15},Private {idNumber = 15, height = 166,
timeToSwap = 2},Private {idNumber = 54, height = 179, timeToSwap =
17},Private {idNumber = 6, height = 182, timeToSwap = 14},Private {idNumber
= 7, height = 175, timeToSwap = 4},Private {idNumber = 59, height = 190,
timeToSwap = 12},Private {idNumber = 55, height = 182, timeToSwap =
9},Private {idNumber = 24, height = 189, timeToSwap = 6},Private {idNumber
= 34, height = 167, timeToSwap = 4}]
```

Objective: Understanding custom data types, adding extra complexity to previously straightforward (hopefully!) code.

2.3 sortPrivatesByHeight (20 pts)

Great job so far, assuming you're keeping up! This time we will finalize our job of sorting the privates via parallel bubbling. There's a catch though: we're interested in how long it takes for everyone to get sorted. We'll only sort the privates by height, since we've already dealt with abstracting a little.

Computing how long the sort takes is simple: When two privates swap, this process takes as much time as is necessary for the slower of the two privates. So, the time a single parallel bubbling step takes is determined by the slowest private that swaps in that step. Sum up the time for each bubbling step, and we get the total sorting time! [1]

Here are a bunch of example runs with the signature, the function returns the sorted result list along with

[1] In a perfect simulation this would be a bit more complicated, since faster privates could move on to the next parallel bubbling step without waiting for the slower ones to finish swapping. In our implementation each step is *entirely independent*. Everyone will wait for everyone else to finish swapping once, before swapping a second time.

the time it took to sort together in a 2-tuple:

```
sortPrivatesByHeight :: [Private] -> ([Private], Int)
```

```
*PE3> pri = Private
*PE3> -- Again some manual formatting for clarity
*PE3> sortPrivatesByHeight []
([],0)
*PE3> sortPrivatesByHeight [pri 8 178 10, pri 10 166 3]
([Private {idNumber = 8, height = 178, timeToSwap = 10}
,Private {idNumber = 10, height = 166,timeToSwap = 3}]
,0)
*PE3> sortPrivatesByHeight [pri 10 166 3, pri 8 178 10]
([Private {idNumber = 8, height = 178, timeToSwap = 10}
,Private {idNumber = 10, height = 166,timeToSwap = 3}]
,10)
*PE3> sortPrivatesByHeight [pri 1 170 5, pri 3 173 17, pri 2 181 9, pri 5 190 2]
([Private {idNumber = 5, height = 190, timeToSwap = 2}
,Private {idNumber = 2, height = 181, timeToSwap = 9}
,Private {idNumber = 3, height = 173, timeToSwap = 17}
,Private {idNumber = 1, height = 170, timeToSwap = 5}]
,56)
*PE3> sortPrivatesByHeight [pri 15 193 15, pri 51 175 9, pri 6 166 2, pri
55 182 14, pri 42 179 17, pri 7 190 12, pri 54 175 4, pri 34 189 6, pri 24
182 9]
([Private {idNumber = 15, height = 193, timeToSwap = 15},Private {idNumber
= 7, height = 190, timeToSwap = 12},Private {idNumber = 34, height = 189,
timeToSwap = 6},Private {idNumber = 55, height = 182, timeToSwap =
14},Private {idNumber = 24, height = 182, timeToSwap = 9},Private {idNumber
= 42, height = 179, timeToSwap = 17},Private {idNumber = 51, height = 175,
timeToSwap = 9},Private {idNumber = 54, height = 175, timeToSwap =
4},Private {idNumber = 6, height = 166, timeToSwap = 2}],99)
```

Objective: Combining repetition and accumulation, a fairly common task in Haskell. Good luck!

Hint: You do not absolutely have to bother with a smart algorithm that detects when the privates are sorted and stops then. Applying the bubble step a certain amount of times will guarantee a sorted result. Think about how many times though; applying it n times for a list of size n like the original bubblesort will not be enough since values may sometimes be blocked due to the value on their left currently swapping. Hmm...

3 Part II - Squeezing your Brain Muscles

So far we can sort privates standing in single file, very cool! Military units march in columns composed of multiple files though. So this time we will be dealing with privates standing in a 2D grid-like area, where each cell of the grid can either contain a private or be empty.

```
data Cell = Empty | Full Private deriving Show
type Area = [[Cell]]
```

3.1 ceremonialFormation (25 points)

We're finally reducing the abstraction, the privates are now standing in the grid-like area chatting, smoking, lollygagging and whatnot. The first order of the day is getting them into ceremonial formation, where each file is sorted by height. The function will receive as input a 2D list of `Cell`s, representing the privates standing in an area. An example grid follows (the left side is the front, each inner list is a single file):

```
[[Full (Private 15 193 15), Empty, Full (Private 51 175 9), Full (Private 6 166 2)],
 [Empty, Full (Private 55 182 14), Full (Private 42 179 17), Full (Private 7 190 1)],
 [Full (Private 54 175 4), Empty, Empty, Full (Private 24 182 9)]]
```

A quick review to make sure the layout is clear:

- An `Area` will be a list of *files*.
- Each file will be a single list like in Part I, except they are now composed of `Cell`s that may contain privates instead of just `Private`s.

Grids are guaranteed to be rectangular, i.e. each inner list will have the same length. Getting the privates into formation will be easy, since they are all very sensible and are already standing in their own file. [2] The only thing you need to take care of is sorting every file independently, and making sure the privates move into empty spots; leaving all empty spots together at the back (the right side) of the file. Swapping with an empty spot **takes the timeToSwap of the private moving into the empty spot**.

Return the area after everyone has gotten into ceremonial formation, along with the time it took to get into formation. When thinking about time, remember that each file is sorting themselves **concurrently and independently**. Check the example results to make sure you understand this. The inner lists can be empty, but the input will never be `[]` alone, some inner lists are guaranteed. Here's the signature and some example runs:

[2] Actually getting into the correct formation would be 2D, with height decreasing from both front to back and right to left, but we're taking it easy today, thankfully :)

```
ceremonialFormation :: Area -> (Area, Int)
```

```
*PE3>
*PE3> -- Got some manual formatting again, watch the indentation!
*PE3> -- Same input in different orders to understand time
*PE3> ceremonialFormation [[Full (Private 20 185 5), Full (Private 22 177 6)]
                          ,[Empty, Full (Private 16 175 16)]]
([[Full (Private {idNumber = 20, height = 185, timeToSwap = 5})
  ,Full (Private {idNumber = 22, height = 177, timeToSwap = 6})]
 ,[Full (Private {idNumber = 16, height = 175, timeToSwap = 16})
  ,Empty]]
,16)
*PE3> ceremonialFormation [[Full (Private 20 185 5), Full (Private 22 177 6)]
                          ,[Full (Private 16 175 16), Empty]]
([[Full (Private {idNumber = 20, height = 185, timeToSwap = 5})
  ,Full (Private {idNumber = 22, height = 177, timeToSwap = 6})]
 ,[Full (Private {idNumber = 16, height = 175, timeToSwap = 16})
  ,Empty]]
,0)
*PE3> ceremonialFormation [[Full (Private 22 177 6), Full (Private 20 185 5)]
                           ,[Full (Private 16 175 16), Empty]]
([[Full (Private {idNumber = 20, height = 185, timeToSwap = 5})
  ,Full (Private {idNumber = 22, height = 177, timeToSwap = 6})]
 ,[Full (Private {idNumber = 16, height = 175, timeToSwap = 16})
  ,Empty]]
,6)
```

```
*PE3> ceremonialFormation [[Full (Private 22 177 6), Full (Private 20 185 5)]
                          ,[Empty, Full (Private 16 175 16)]]
([[Full (Private {idNumber = 20, height = 185, timeToSwap = 5})
  ,Full (Private {idNumber = 22, height = 177, timeToSwap = 6})]
 ,[Full (Private {idNumber = 16, height = 175, timeToSwap = 16})
  ,Empty]]
,16)
*PE3> ceremonialFormation
 [[Full (Private 15 193 15), Empty, Full (Private 51 175 9), Full (Private 6 166 2)]
 ,[Empty, Full (Private 55 182 14), Full (Private 42 179 17), Full (Private 7 190 1)]
 ,[Full (Private 54 175 4), Empty, Empty, Full (Private 24 182 9)]]
([[Full (Private {idNumber = 15, height = 193, timeToSwap = 15})
  ,Full (Private {idNumber = 51, height = 175, timeToSwap = 9})
  ,Full (Private {idNumber = 6, height = 166, timeToSwap = 2})
  ,Empty]
 ,[Full (Private {idNumber = 7, height = 190, timeToSwap = 1})
  ,Full (Private {idNumber = 55, height = 182, timeToSwap = 14})
  ,Full (Private {idNumber = 42, height = 179, timeToSwap = 17})
  ,Empty]
 ,[Full (Private {idNumber = 24, height = 182, timeToSwap = 9})
  ,Full (Private {idNumber = 54, height = 175, timeToSwap = 4})
  ,Empty
  ,Empty]]
,35)
```

Objective: Dealing with yet more complexity. Combining basic building blocks beautifully and skillfully layering levels of complexity is an art that takes a lifetime to perfect... The more practice the better, even if it is short!

3.2 swapPrivates (15 pts)

Oh, snap! The platoon commander suddenly arrives... And he starts swapping privates around randomly to make the formation more suitable to his liking. Our task seems simple this time around: Given two private ID numbers, find them in the area and swap their position, returning the modified area. If one or both of the IDs cannot be found in the area, the area will not change. It is of course guaranteed that both IDs will be positive and different. Remember that all privates in an area are **guaranteed to have unique ID numbers**. It is possible for the privates that should be swapped to be in the same file (inner list), different files or for one or both of them to not exist at all. Anything goes!

Here's the signature and some example runs:

```
swapPrivates :: Int -> Int -> Area -> Area
```

```
*PE3> example = [[Full (Private 1 170 3), Full (Private 4 175 2), Empty,
                  Full (Private 3 182 9)]]
*PE3> example -- a 1x4 grid example
[[Full (Private {idNumber = 1, height = 170, timeToSwap = 3})
 ,Full (Private {idNumber = 4, height = 175, timeToSwap = 2})
 ,Empty
 ,Full (Private {idNumber = 3, height = 182, timeToSwap = 9})]]
*PE3> swapPrivates 1 3 example
[[Full (Private {idNumber = 3, height = 182, timeToSwap = 9})
 ,Full (Private {idNumber = 4, height = 175, timeToSwap = 2})
 ,Empty
 ,Full (Private {idNumber = 1, height = 170, timeToSwap = 3})]]
*PE3> swapPrivates 2 3 example
[[Full (Private {idNumber = 1, height = 170, timeToSwap = 3})
 ,Full (Private {idNumber = 4, height = 175, timeToSwap = 2})
 ,Empty
 ,Full (Private {idNumber = 3, height = 182, timeToSwap = 9})]]
*PE3> swapPrivates 4 3 example
[[Full (Private {idNumber = 1, height = 170, timeToSwap = 3})
 ,Full (Private {idNumber = 3, height = 182, timeToSwap = 9})
 ,Empty
 ,Full (Private {idNumber = 4, height = 175, timeToSwap = 2})]]
*PE3> -- Now a 3x2 grid
*PE3> swapPrivates 28 18 [[Full (Private 9 193 19), Full (Private 28 191 14)],
                          [Full (Private 24 184 12), Full (Private 43 175 11)],
                          [Full (Private 18 168 15), Empty]]
[[Full (Private {idNumber = 9, height = 193, timeToSwap = 19})
 ,Full (Private {idNumber = 18, height = 168, timeToSwap = 15})]
,[Full (Private {idNumber = 24, height = 184, timeToSwap = 12})
 ,Full (Private {idNumber = 43, height = 175, timeToSwap = 11})]
,[Full (Private {idNumber = 28, height = 191, timeToSwap = 14})
 ,Empty]]
```

Objective: Recognizing that functional list structures are not ideal for solving every problem, but still being able to cleverly manipulate them. It's important to use the right tool for the job, and now we're seeing how it goes when using the wrong tool.

Grading: Half of the test cases will have only a single file, i.e. a grid of shape $1 \times n$ instead of $m \times n$. You can start by solving focusing on that case only if you want to.

Hint: This problem is definitely difficult. Think long and hard, there's no shame in not doing it if you cannot find time! `find` from `Data.List` could help you *find* a bit more time...

4 Regulations

1. **Implementation and Submission:** The template file named "PE3.hs" is available in the Virtual Programming Lab (VPL) activity called "PE3" on `OdtuClass`. At this point, you have two options:
   - You can download the template file, complete the implementation and test it with the given

sample I/O on your local machine. Then submit the same file through this activity.

- You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submitting a file.

The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it.

There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

3. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official testcases to determine your ***actual*** grade after the deadline.