Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2021-2022

Programming Exam 4

Due date: 23 April 2022, Saturday, 03:00

1 Problem Definition

In this final programming exam for Haskell, we will be practicing with some tree-style data structures mixed with lists. Should be fun and easy!

We're going to be helping an elite soldier ant, *Grrant*[1], prepare a raid on an unsuspecting common ant nest with the aim of foraging food for its own tribe of army ants.

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!
- `Data.List` and `Data.Maybe` are imported for you and could be useful in the final functions. Skim the docs 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!

[1] It's an ant, a grunt, and it's very angry!

2 Getting Familiar with the Data Structures

All the functions in this exam will be working on ant nests, which are composed of five types of different rooms:

```haskell
data Room = SeedStorage Int
          | Nursery Int Int
          | QueensChambers
          | Tunnel
          | Empty
          deriving Show
```

- Seed storage rooms contain seeds for nutrition.
- Nursery rooms contain eggs (first field) and larvae (second field).
- The queen's chambers contain the ant queen. There is at most one of these rooms in a nest. There can also be zero, implying an abandoned nest.
- Tunnel rooms are all connected to each other to form shortcuts. (Think of them like teleportation nodes!)
- Empty rooms contain nothing interesting.

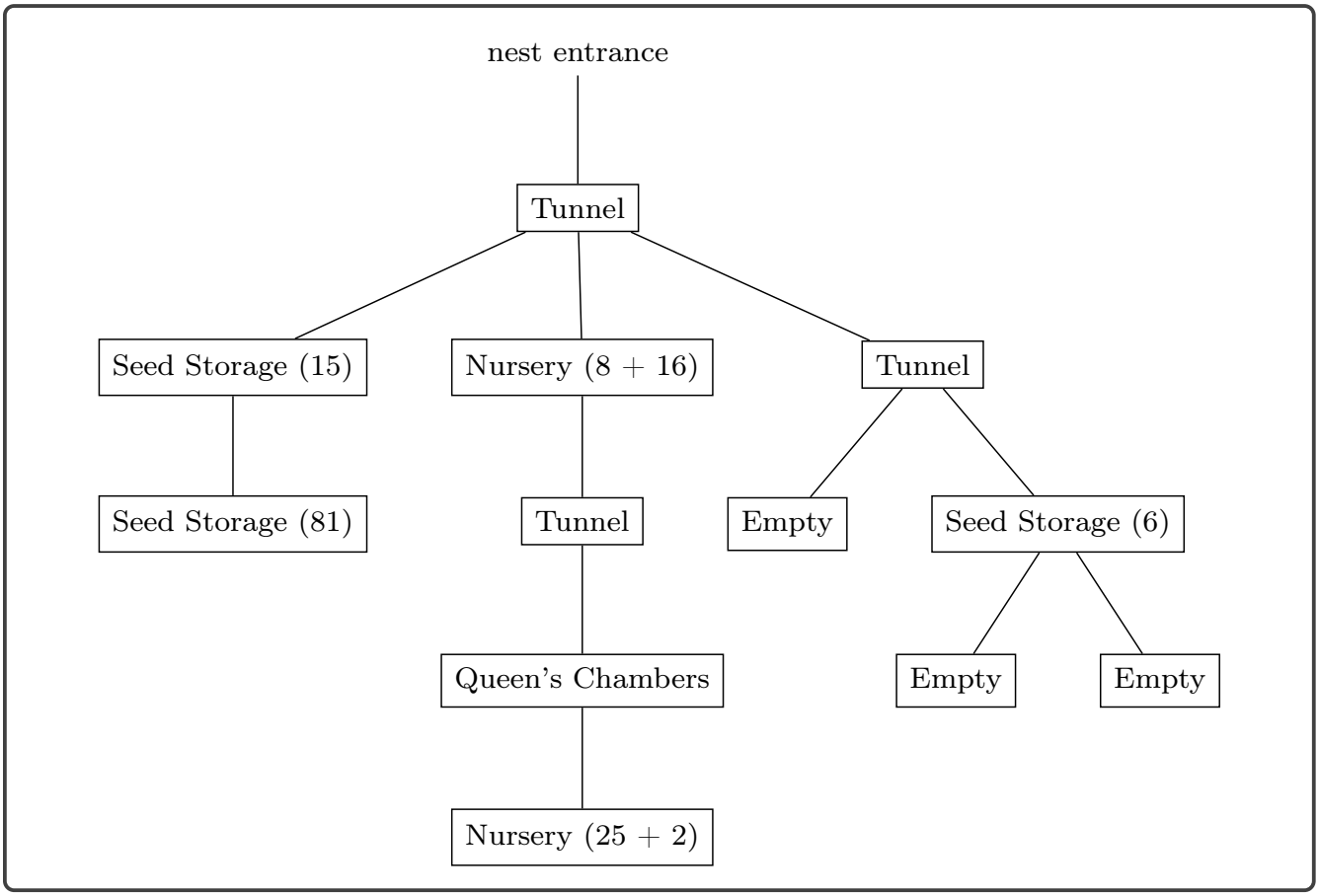
Nests are simply n-ary trees with rooms acting as tree nodes. Each node is defined by its own room and a list of its subtrees (deeper parts of the nest). The list of subtrees will be empty in case there are no deeper nodes:

```haskell
data Nestree = Nestree Room [Nestree] deriving Show
```

Here's an example nest (also present in your source code):

```haskell
exampleNest :: Nestree
exampleNest =
  Nestree Tunnel [
    Nestree (SeedStorage 15) [
      Nestree (SeedStorage 81) []
    ],
    Nestree (Nursery 8 16) [
      Nestree Tunnel [
        Nestree QueensChambers [
          Nestree (Nursery 25 2) []
        ]
      ]
    ],
    Nestree Tunnel [
      Nestree Empty [],
      Nestree (SeedStorage 6) [
        Nestree Empty [],
        Nestree Empty []
      ]
    ]
  ]
```

And this is how the example nest would look when drawn. Please note that `entrance` is not a node in the code! It is simply there to illustrate how the nest would look when entering from outside.



Helpful Script: You can use `format_nest.py` to view the inputs in an indented way if you get stuck one. Simply use it as `python format_nest.py < input_file.txt` or just run it and copy-paste the example input into it.

3 Functions to Implement

3.1 nestNutritionValue (20 points)

Grrant wants to know how nutritious the nest it is about to raid is to make the proper preparations. Given that the nutrition values of:

- Each seed is 3.
- Each egg is 7.
- And each larvae is 10.

Compute the total nutrition value of the given nest (no overflows are guaranteed). Here's the signature and some example runs:

```haskell
nestNutritionValue :: Nestree -> Int
```

```
*PE4> nestNutritionValue $ Nestree (SeedStorage 3) []
9
*PE4> nestNutritionValue $ Nestree Tunnel []
0
*PE4> nestNutritionValue $ Nestree (Nursery 3 5) []
71
*PE4> nestNutritionValue $ Nestree (Nursery 2 1) [
  Nestree Empty [],
  Nestree (SeedStorage 15) []
]
69
*PE4> nestNutritionValue $ Nestree Tunnel [
  Nestree (SeedStorage 80) [
    Nestree (Nursery 2 7) []
  ],
  Nestree (SeedStorage 15) [
    Nestree Empty [],
    Nestree Tunnel []
  ]
]
369
*PE4> nestNutritionValue exampleNest -- example above, also in the source code
717
```

Objective: Performing a very basic traversal of the tree.

Tip: Remember how to give multi-line input to `ghci` for copy-pasting the samples. Enter `:{` to the prompt and press `Enter`. Then, paste/enter whatever multi-line input you want to enter. Once you're done, enter `:}` and press `Enter` one last time.

3.2 pathNutritionValues (30 points)

So much to raid, so little time! To make the best use of its time, Grrant wants to analyse how much nutrition it can get out of raiding a single path in the nest. To help with the analysis, calculate how much nutritious value each **root-to-leaf path** contains from left-to-right.

For the example tree, the result would be `[288,411,0,18,18]`:

- The leftmost path is `Tunnel-SeedStorage(15)-SeedStorage(81)` having a nutrition value of 288.
- The next one is `Tunnel-Nursery(8+16)-Tunnel-QueensChambers-Nursery(25+2)` having a nutrition value of 411.
- The next one is `Tunnel-Tunnel-Empty` having 0 nutrition value.
- The next one is `Tunnel-Tunnel-SeedStorage(6)-Empty` (using the `Empty` on the left, under `SeedStorage(6)`), having a nutrition value of 18.
- And the final one is again `Tunnel-Tunnel-SeedStorage(6)-Empty`, but this time using the `Empty` on the right. Also 18.

Here's the signature and some example runs:

```
pathNutritionValues :: Nestree -> [Int]
```

```
*PE4> pathNutritionValues $ Nestree Empty []
[0]
*PE4> pathNutritionValues $ Nestree (SeedStorage 242) []
[726]
*PE4> pathNutritionValues $ Nestree Tunnel [
  Nestree (SeedStorage 10) [],
  Nestree (Nursery 2 7) [],
  Nestree Empty []
]
[30,84,0]
*PE4> pathNutritionValues $ Nestree Tunnel [
  Nestree (SeedStorage 10) [
    Nestree (Nursery 5 10) []
  ],
  Nestree Empty [
    Nestree Tunnel [],
    Nestree (SeedStorage 99) []
  ]
]
[165,0,297]
*PE4> pathNutritionValues exampleNest
[288,411,0,18,18]
```

Objective: Performing slightly more nuanced traversal of the tree :)

3.3 shallowestTunnel - (20 points)

Next, we're preparing for something more advanced. Grrant needs to know how quickly it can access the tunnel network inside the nest. Return the depth of the shallowest tunnel in a given nest with `Just`. The depth will be the number of edges from the entrance to the nearest tunnel. If there are no tunnels in the nest, return `Nothing`.

Here's the signature and basic examples:

```
shallowestTunnel :: Nestree -> Just Int
```

```
*PE4> shallowestTunnel $ Nestree Empty []
Nothing
*PE4> shallowestTunnel $ Nestree Tunnel []
Just 1
*PE4> shallowestTunnel $ Nestree Tunnel [Nestree Tunnel [Nestree Tunnel []]]
Just 1
*PE4> shallowestTunnel $ Nestree Empty [Nestree (Nursery 5 5) [Nestree Tunnel []]]
Just 3
*PE4> shallowestTunnel $ Nestree (SeedStorage 3) [
  Nestree Empty [
    Nestree Empty []
  ],
  Nestree Tunnel []
]
Just 2
*PE4> shallowestTunnel $ Nestree (SeedStorage 3) [
  Nestree Empty [
    Nestree Tunnel [] -- deeper than the other tunnel
  ],
  Nestree Tunnel []
]
Just 2
*PE4> shallowestTunnel exampleNest
Just 1
*PE4> shallowestTunnel exampleNestNoTunnel
Nothing
```

Objective: Familiarity with the ubiquitous `Maybe` type. Looking up docs. Combining the use of another basic type with tree traversal.

Hint: Look in the docs of `Data.Maybe`, there are a bunch of functions you should find useful!

3.4 pathToQueen (10 points)

Grrant knows that nurseries are concentrated around the Queen's Chambers. In in an attempt to further distill the raiding process, it wants to know the details of a direct path to the queen. Return the list of nodes from the entrance to the Queen's Chambers (not including the chambers) wrapped in a `Just`. Return `Nothing` if there is no Queen's Chambers in the nest. Remember that it is guaranteed there can be at most one.

The simple examples, as always:

```
pathToQueen :: Nestree -> Maybe [Room]
```

```
*PE4> pathToQueen $ Nestree (SeedStorage 333) []
Nothing
*PE4> pathToQueen $ Nestree QueensChambers []
Just []
*PE4> pathToQueen $ Nestree (Nursery 7 7) [
  Nestree (SeedStorage 33) [],
  Nestree QueensChambers []]
Just [Nursery 7 7]
*PE4> pathToQueen $ Nestree Empty [ -- there can be other rooms below the queen's!
  Nestree (Nursery 9 10) [
    Nestree QueensChambers [
      Nestree (Nursery 6 8) []
    ]
  ]
]
Just [Empty,Nursery 9 10]
*PE4> pathToQueen exampleNest
Just [Tunnel,Nursery 8 16,Tunnel]
*PE4> pathToQueen exampleNestNoTunnel
Just [Empty,Nursery 8 16,Empty]
```

Objective: Doing something slightly different from what you've done previously. Realising that there are patterns of similarity, but not having time to capture them in an abstract way. Ha! Think about it later :)

Hint: This should be similar to `shallowestTunnel`.

3.5 quickQueenDepth (20 points)

It's time for Grrant to put its master plan in action: using tunnels to get to the queen quickly. Calculate how many edges it would take to go from the entrance to the queen by using a tunnel and return the value wrapped in a `Just`. If there are no tunnels on the path to the queen or if the direct path is shorter than a path through the nearest tunnel, `Just` return the number of edges in the direct path. If there is no queen, return `Nothing`. Remember that there is no going *upwards* after exiting a tunnel, Grrant is always moving downwards!

Consider the example tree: one edge from the entrace to the first tunnel, through which Grrant can skip over the nursery by exiting from the tunnel right above the queen. One more edge from there to the Queen's Chambers for a total of 2 edges.

If there were no tunnels, the path through the nursery would take 4 edges.

Example runs and signature:

```
quickQueenDepth :: Nestree -> Maybe Int
```

```
*PE4> quickQueenDepth $ Nestree (SeedStorage 9999) []
Nothing
*PE4> quickQueenDepth $ Nestree QueensChambers  []
Just 1
*PE4> quickQueenDepth $ Nestree Empty [
  Nestree (SeedStorage 16) [
    Nestree (Nursery 3 5) [
      Nestree Tunnel [ -- leave here
        Nestree QueensChambers []
      ]
    ],
    Nestree Tunnel [] -- enter here to skip one edge
  ]
]
Just 4
*PE4> -- Same, but no other tunnel => direct path
*PE4> quickQueenDepth $ Nestree Empty [
  Nestree (SeedStorage 16) [
    Nestree (Nursery 3 5) [
      Nestree Tunnel [ -- leave here
        Nestree QueensChambers []
      ]
    ],
    Nestree Empty [] -- useless room
  ]
]
Just 5
*PE4> -- Same, but tunnels don't shorten the direct path
*PE4> quickQueenDepth $ Nestree Empty [
  Nestree (SeedStorage 16) [
    Nestree (Nursery 3 5) [
      Nestree Tunnel [ -- leave here
        Nestree QueensChambers []
      ]
    ],
    Nestree Empty [
      Nestree (SeedStorage 36) [
        Nestree Tunnel [] -- deep!
      ]
    ]
  ]
]
Just 5
*PE4> quickQueenDepth exampleNest
Just 2
*PE4> quickQueenDepth exampleNestNoTunnel
Just 4
```

Objective: Writing a final function you can be proud of by combining simpler building blocks. Possibly getting annoyed by the constant unwieldy use of `Maybe`. Remembering to take a look at the advanced solution that will be posted next week as a teaser for better Haskell :)

Hint: Use `shallowestTunnel` and `pathToQueen`. A bit of manipulation on their results and voila!

4 Regulations

1. **Implementation and Submission:** The template file named "PE4.hs" is available in the Virtual Programming Lab (VPL) activity called "PE4" 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.