# Programming Paradigms 2020

# Coursework: Functional Programming using Haskell

# Rows of Fruit Game

With some minor typos corrected and highlighted.

## Introduction

In this game, you are given a row of fruits. Each turn you can either (i) remove one, (ii) duplicate one or (iii) exchange one with its neighbour. The goal is to transform the row into a target row of fruit in the shortest number of operations.

For example, suppose you start with [Apple, Pear, Banana, Orange] and want to get to [Apple, Pear, Apple, Banana]? Then this can be done in three steps:-

| Step | Operation | Row |
|---|---|---|
| | Start… | Apple, Pear, Banana, Orange |
| 1 | Remove 4$^{th}$ fruit | Apple, Pear, Banana |
| 2 | Duplicate 1$^{st}$ fruit | Apple, Apple, Pear, Banana |
| 3 | Exchange 2$^{nd}$ fruit with its right neighbour | Apple, Pear, Apple, Banana |

Table 1. Example operations on a row of fruit

Following the instructions below, you will write a program in Haskell that can solve this task.

Note that the instructions get progressively harder.

## Submission

Once you have completed the exercise you only need to submit your Haskell script with all the functions you have built in a `.hs` file, along with your written answers to Q13 and Q16 in a Word or PDF file only. Submit your two files via Moodle. Your script needs to load into GHCi (version 8+) without compilation error. You will lose marks if it does not. Your script will be tested and marked through function calls to check you have implemented them correctly.

The deadline for submission is 19 May 2020 at 16:00.

## Instructions

1. Create a data type to represent fruits:

```
data Fruit = Apple | Pear | Orange | Plum | Banana deriving
(Show, Eq)
```

*Enter this all on one line.*

You can add more types of fruit if you want. You can now use Fruits as objects in Haskell. So create a list of fruit using

```
f1 :: [Fruit]
f1 = [Pear, Apple, Plum]
```

2. Create a data type to express each of the different types of transformations: remove, duplicate and exchange:-

```
data Tran = Id | Rm | Dup | Exc deriving Show
```

Note that the first one Id is the identity operation – it does not do anything to the row of fruit. Each operation will be represented by a pair of type `(Tran, Int)`. Create the list of four operations:-

```
t1 ::  [(Tran, Int)]
t1 = [ (Rm,2), (Dup,0), (Id,0), (Exc,1) ]
```

We will use Haskell's convention for indexing lists: ie `0` corresponds to the first element, `1` to the second, and so on. So `(Rm,2)` removes fruit in position 2, `(Dup,0)` duplicates the fruit in position 0, `(Id,0)` does nothing and `(Exc,1)` exchanges the fruit in position 1 with the one in position 2.

3. You will now write a function `applyTran` that implements each of the transformations in lists of fruit. Here is its type declaration and the first two patterns to implement `Id` and `Rm`:-

```
applyTran :: (Tran,Int) -> [Fruit] -> [Fruit]
applyTran (Id,_) xs = xs
applyTran (Rm,n) xs = (take n xs) ++ (drop (n+1) xs)
```

Enter these into your Haskell script and also write the remaining function definitions to implement duplication `(Dup,n)` and exchange `(Exc,n)`.

***Hint:*** you should use `take`, `drop`, `++` and `!!`.

4. Here is a function `applySeries` that will execute a sequence (list) of transactions on a list of fruit and output the result as a list of fruit. The type definition is:-

```
applySeries :: [(Tran, Int)] -> [Fruit] -> [Fruit]
applySeries [] xs = xs
applySeries (tn:trans) xs = applySeries trans (applyTran tn xs)
```

Enter the function in your script and ensure it works. For example,

```
applySeries t1 f1
```

should give the answer:

```
[Pear,Apple,Pear]
```

*Double-check that you understand why this is the answer!*

5. Write another function `applyPar` that will apply a list of transactions *in parallel* to a list of fruit. That is, each is applied to the list of fruit separately and the output of the function is the list of list of fruits that are the outcome of the transactions.

   **Important: Specify the function type first, then build the function body.**

If you have written the function correctly, the following command

```
applyPar t1 f1
```

will give the answer

```
[ [Pear,Apple], [Pear,Pear,Apple,Plum], [Pear,Apple,Plum],
[Pear,Plum,Apple] ]
```

**6.** Write a function `transSeq` that will take a type of transaction `t` of type `Tran` and an integer `n` and will output the list of transaction pairs, `[ (t,0), … , (t,n) ]`.
**Important: for this exercise, use list comprehension to write this function.**

Test it using the command

```
transSeq Exc 3.
```

7. Write a function `transSeqAll` with one argument, an integer `n`, that outputs the list of *all possible* operations, except `Id`, that can legitimately be made on a list of length `n`.

So, for example, `(Exc,1)` is a legitimate operation on `[Apple,Pear,Pear]`, but `(Dup,4)` is not since there is no index position 4 in the list, nor is `(Exc 2)` because there is no neighbour in position 3 to exchange with it (*remember: in Haskell, lists are indexed from 0,1,2,etc*).

*Hint: make use of the* `transSeq` *function*.

❖ ❖ ❖

Now we can think about how to search for the shortest sequence of operations to go from one row of fruit to another. We will do this through a ***search tree*** where each node is a row of fruit and each branch is a single operation (ie a possible move in the game). The root node is the starting row of fruit. The depth of the tree will correspond to the length of a sequence of operations. Hence the best search method is a **breadth-first** search of the tree (as opposed to depth-first).

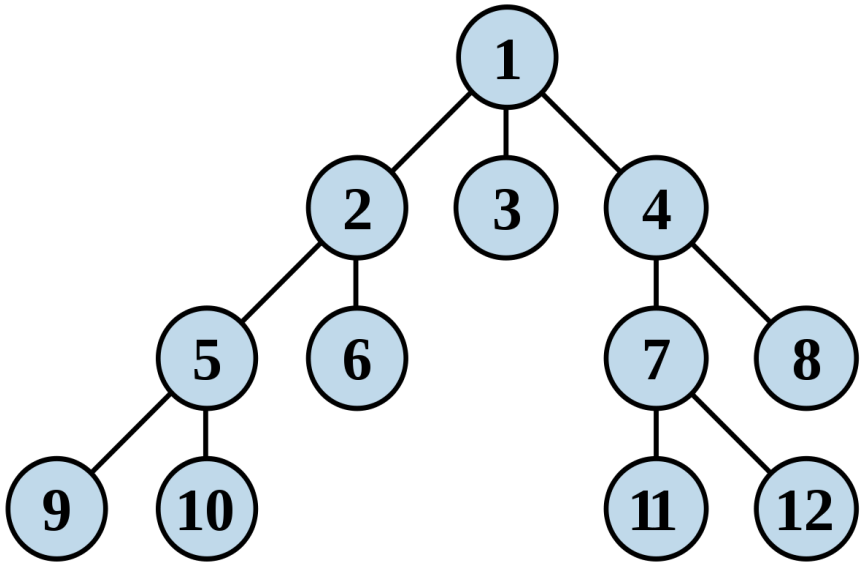


Figure 1. Example of breadth-first search of a search tree. Search begins at root node 1 and moves in number sequence, ending at node 12.
(source: nl.wikipedia.org)

The tree will be represented as a list of nodes. Each node will tell us where its parent node is, the transformation and the outcome after that transformation is made. Therefore, in Haskell each node will be represented by a 3-tuple with the following type:

```
(Int, (Tran,Int), [Fruit])
```

To control the search, the position at which the search has got to needs to be stored. Hence the pair

`(pos, tree)` of type `( Int,[(Int, (Tran,Int), [Fruit])] )`

will be used for this purpose (*remember: a tree is just a list of nodes*).

8. Now a search can begin at position 0 of a tree that contains just the root node. This is constructed as follows:

```
initTree :: [Fruit] -> (Int, [(Int, (Tran,Int), [Fruit])])
initTree xs = (0, [(-1, (Id,0), xs)] )
```

Implement this tree in your script. Notice that the root node has no parent so a position -1 is given and there is no transformation at the root node, so the `Id` transformation is used.

9. Implement the following functions that output each of the three components of a 3-tuple node:

```
nodeParent :: (Int, (Tran,Int), [Fruit]) -> Int
nodeParent (parent,_,_) = parent
nodeTran :: (Int, (Tran,Int), [Fruit]) -> (Tran,Int)
nodeTran (_,trans,_) = trans
nodeOutcome :: (Int, (Tran,Int), [Fruit]) -> [Fruit]
nodeOutcome (_,_,xs) = xs
```

   You can use these functions in the functions you write later.

10. Write a function `processCurrent` that takes a `(pos, tree1)` pair and outputs an updated `(pos+1, tree2)` pair, with the additional nodes when all possible transformations from the node pos are made added to the end of `tree1` to form `tree2`.

*Hint 1: make use of* `transSeqAll` *to get a list of all possible transactions from the fruit list in a given node.*

*Hint 2: you will probably need at least one auxiliary function to do this.*

Notice that the position counter is moved on one, `pos+1`, which is the next node to search using the breadth first search strategy.

Test `processCurrent` by running `processCurrent (initTree f1)`. This should include the root and all nodes following branches from the root node. Check that it does this.

11. Write a function `findFruits` that takes a list of fruit `f1` and a list of nodes and outputs the index position of the first node whose outcome is exactly equal to `f1`. If none of the nodes have `f1` as an outcome, then output -1.

12. Write a function `processRepeat` that takes three arguments, as follows:-

```
processRepeat maxiter f2 nt
```

where `maxiter` is an integer, `f2` is a list of fruit and `nt` is a `(pos,tree)` pair. This function should repeatedly execute `processCurrent` and expanding the tree until either `f2` is found amongst the nodes, as the tree expands, or the size of the tree (measured as number of nodes) exceeds `maxiter`. Output the pair

```
(f2_pos, (pos',tree'))
```

where `(pos',tree')` is the output of the last execution of `processCurrent` and `f2_pos` is the position at which `f2` was found. If `f2` was not found then output with `f2_pos=-1`.

*Hint: This requires iterative calls of* `processCurrent`. *In Haskell, this is done using recursion: ie* `processRepeat` *needs to refer to itself.*

13. Here is the function that makes use of your functions to solve the task of moving from list of fruits `f1` to `f2` in the shortest number of operations:-

```
findTransFrom :: Int -> [Fruit] -> [Fruit] -> (Int, (Int, [(Int,
(Tran,Int), [Fruit])]))
findTransFrom maxiter f1 f2 = processRepeat maxiter f2 (initTree
f1)
```

Implement it, ensuring that your `processRepeat` is consistent with it. Test it by running:

```
search1 = findTransFrom 10 f1 [Pear, Plum, Pear, Apple]
search1
```

What does the output of this function call mean? Has it found the solution? If so, what is it? If not, why not?

14. Implement the function `transPath` given below that shows the path of operations and list of fruit at each stage of the solution (if one has been found).

```
transPath :: (Int, (Int, [(Int, (Tran,Int), [Fruit])])) ->
[((Tran,Int), [Fruit])]
transPath search1 = reverse (transPath2 (fst search1) (snd (snd
search1)) )
  where
    transPath2 pos stree1
      | pos == -1 = []
      | otherwise = (nodeTran n1, nodeOutcome n1) : (transPath2
(nodeParent n1) stree1)
      where
        n1 = stree1!!pos
```

Ensure that it works with your code. In particular,

```
transPath search1
```

should work and show the list of operations moving from `f1` to `f2`, along with the row of fruit at each stage, like Table 1.

15. You now have all the code written for the computer to play the game. Test it with each of these examples:-

```
f2 = [Pear, Plum, Pear, Apple]
p1 = findTransFrom 1000 f1 f2
p2 = findTransFrom 10000 [Apple, Orange, Plum] [Apple, Orange,
Orange]
p3 = findTransFrom 10000 [Apple, Orange, Plum] [Plum, Orange,
Apple, Plum]
```

In each case, use your `transPath` function to show the solution, if there is one.

16. Finally, output the result of running your program against the problem given in Table 1. Is it the same solution as given in Table 1?

END OF COURSEWORK