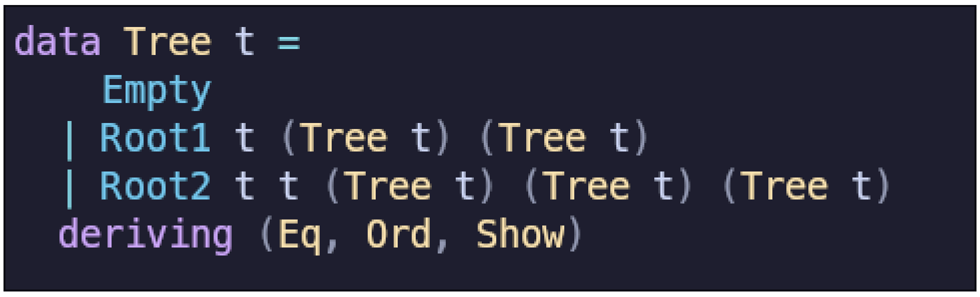
# CSC1048 Continuous Assessment: 2-3 Trees

Giuseppe Esposito 22702705

2024-2025

## 2-3 Trees Structure



```haskell
data Tree t =
    Empty
  | Root1 t (Tree t) (Tree t)
  | Root2 t t (Tree t) (Tree t) (Tree t)
  deriving (Eq, Ord, Show)
```

image 1.0

As described in the docs for this exercise a 2-3 Trees is either:

- an empty Tree.
- a node with just one value and 2 branches (**left:** all element <= value, **right**: all element > value).
- a node with 2 value amd 3 branches
  (**left**: all element <= first value,
  **mid**: first value < all element<= second value;
  **right**: all element > second value).

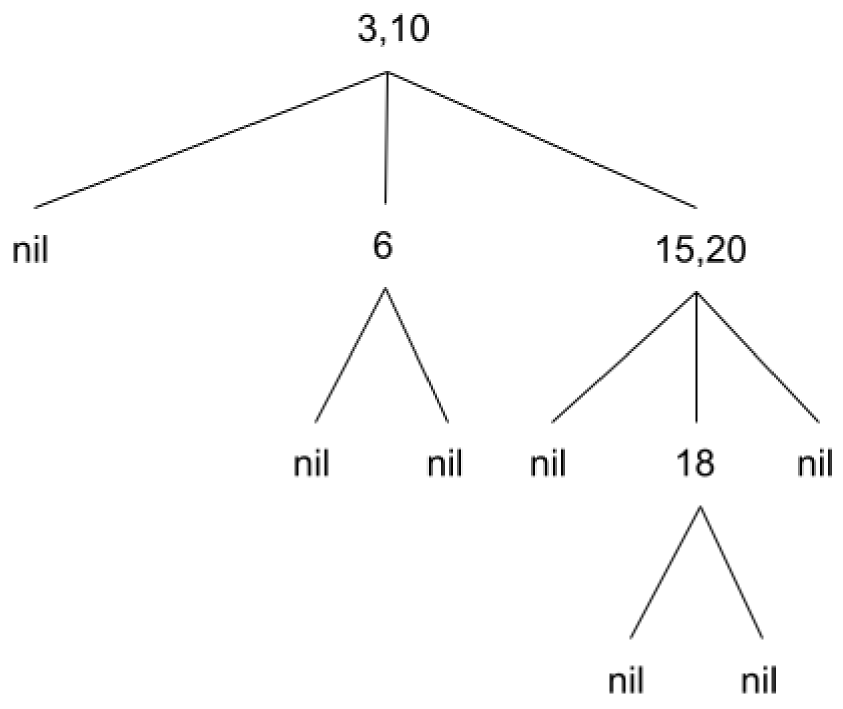
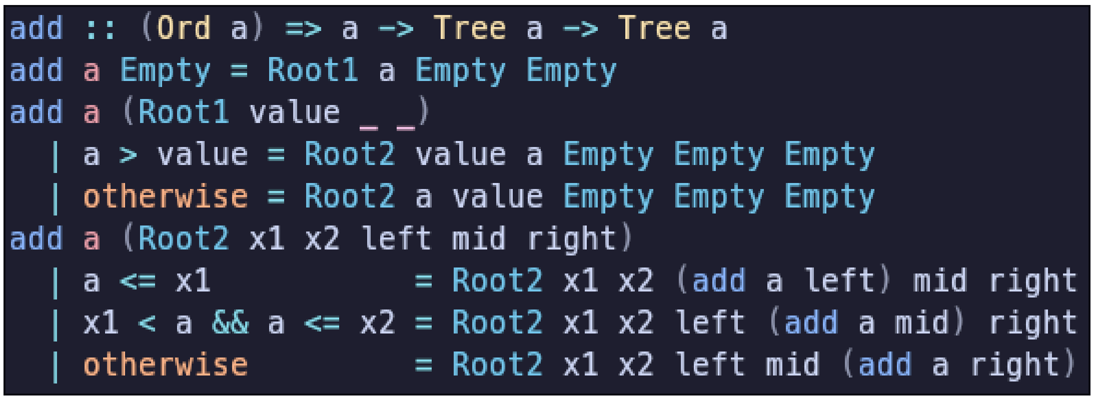


image 1.1

We can represent this tree in haskell code thanks to our structure in this way:
“mytree = Root2 **3 10** Empty (Root1 **6** Empty Empty) (Root2 **15 20** Empty (Root1 **18** Empty Empty) Empty)”

## 2-3 Trees Add Function



```haskell
add :: (Ord a) => a -> Tree a -> Tree a
add a Empty = Root1 a Empty Empty
add a (Root1 value _ _)
   | a > value = Root2 value a Empty Empty Empty
   | otherwise = Root2 a value Empty Empty Empty
add a (Root2 x1 x2 left mid right)
   | a <= x1             = Root2 x1 x2 (add a left) mid right
   | x1 < a && a <= x2 = Root2 x1 x2 left (add a mid) right
   | otherwise           = Root2 x1 x2 left mid (add a right)
```

image 1.2

While adding an element to a 2-3 Tree we can have 3 possible case, add an element to:

1) An empty Tree (Base case)
2) A node with just one value (Base case)
3) A node with two value (Recursive call)

In the first case we just need to return an One value node with empty as left and right branch.
In the second case we need to evaluate if the element that we want to add is greater or less than the value of the node, to understand which will be the first value and which will be the second one, then we just return a two value node with all Empty branches.
In the third case we use recursion to return a new tree with the new value added, by evaluating in which of the 3 branches (left, mid and right) the new element must be added.

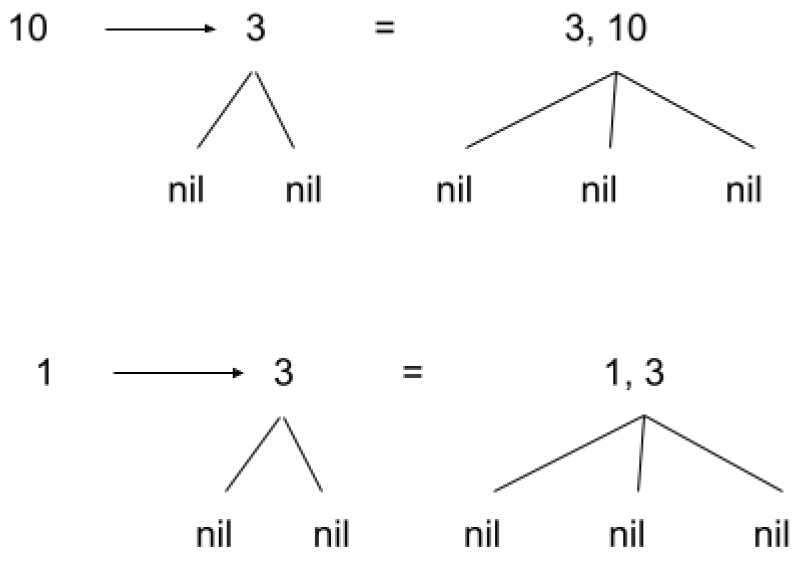
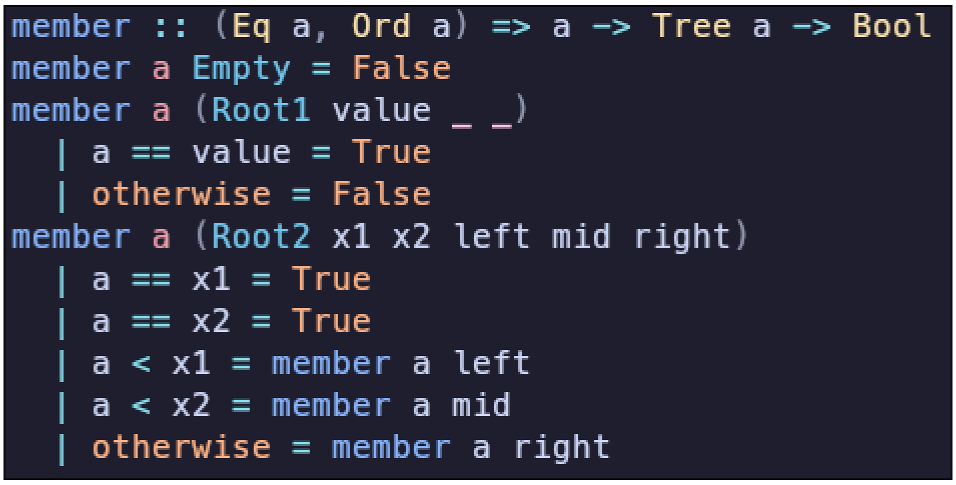


image 1.3

In the docs for this exercise there is the first example of the (image 1.3) where you add 10 to a node with just 3,
I'm assuming that if we add a number that is less than 3 we got the same result as in the figure.
This is way when we add a number to a one node value there is no recursive call (so it is a base case), because at least in my design node with just one value can not have children, they need to become first a two value node and then they can have it.

Now we can use the add function to represent the tree of the image 1.1:
"mytree = add **18** (add **20** (add **15** (add **6** (add **10** (add **3** Empty)))))"

## 2-3 Trees Member Function



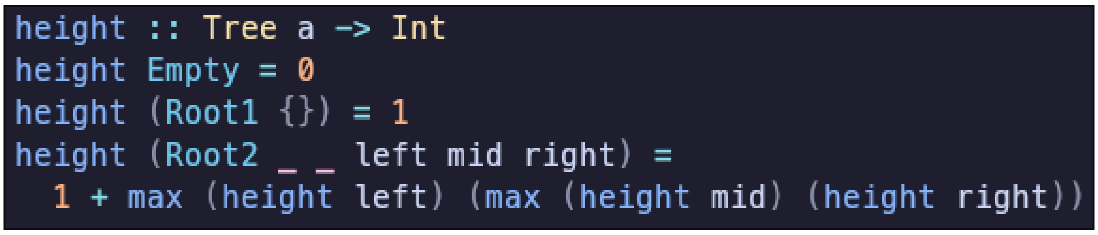
```haskell
member :: (Eq a, Ord a) => a -> Tree a -> Bool
member a Empty = False
member a (Root1 value _ _)
  | a == value = True
  | otherwise = False
member a (Root2 x1 x2 left mid right)
  | a == x1 = True
  | a == x2 = True
  | a < x1 = member a left
  | a < x2 = member a mid
  | otherwise = member a right
```

image 1.4

If we want to check if a is a member a Tree a we have 3 cases:

- The tree is Empty: so we just return False
- The node has just one value: we check is that value is equal to a and return True is yes and False otherwise
- The node has two values: first we check if either of the two values is equal to a, and if yes we return True otherwise we search with a recursive call in the right branch
  (**left** if: a < first value,
  **mid** if: first value < a < second value,
  **right** if: a > second value)

## 2-3 Trees Height



```haskell
height :: Tree a -> Int
height Empty = 0
height (Root1 {}) = 1
height (Root2 _ _ left mid right) =
  1 + max (height left) (max (height mid) (height right))
```

image 1.5

The assumption here is that an Empty tree has 0 of height and a node with one value has 1 of height (because the fact that they can not have children explained before).
In the recursive case (the one with a node with 2 values), we just need to add one to the max height between the height of the left, mid and right branch.

The height of the tree (image 1.1) is 3 following these constraints.

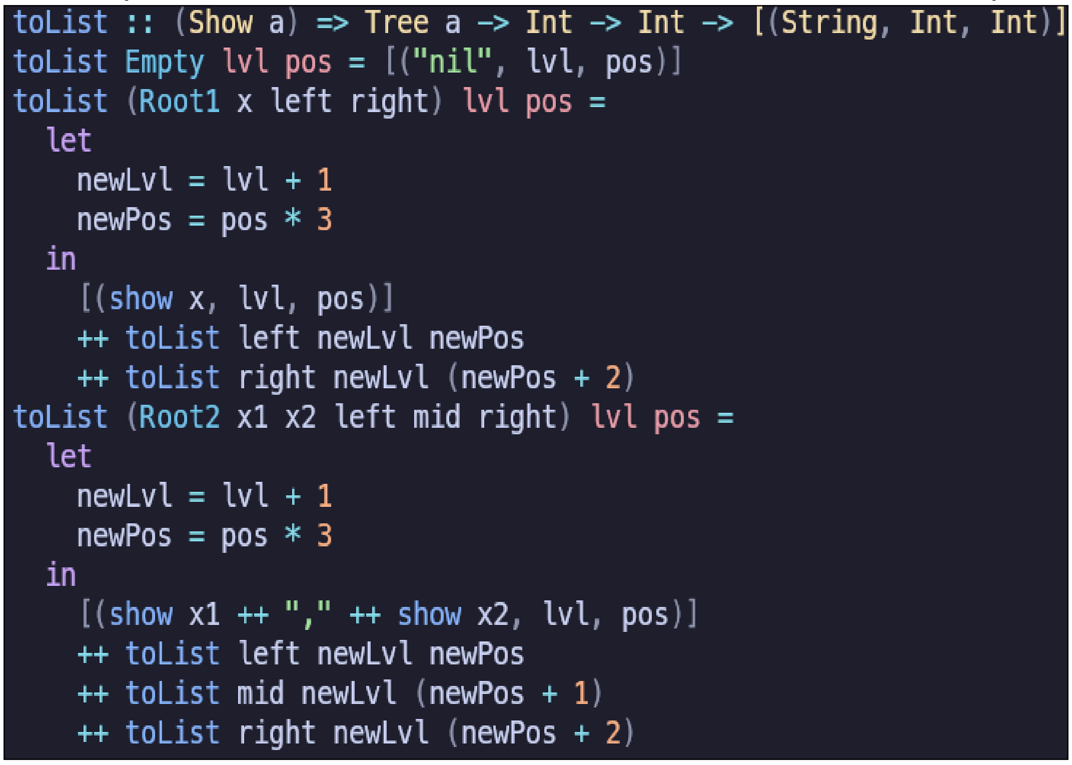
Due to the fact that max is a binary operator (takes only two arguments) in haskell, to find the max of 3 Integers, first we find the max between mid and right and then compare it with the left value.

## 2-3 Trees PrettyPrinting

This was quite a difficult task so I tried to split it into as many small problems as possible.

The idea behind this is that for each level **(max_possible_nodes = 3 ^ current_level)**,
and each node has a position **(position = 0 - (max_possible_nodes - 1) )**
To make the math work every node must have 3 children so even if we encounter a node with just one value we pretend that it has 3 children, but one is going to be hidden.

## First step is traverse the tree and link each node to his level and position



```haskell
toList :: (Show a) => Tree a -> Int -> Int -> [(String, Int, Int)]
toList Empty lvl pos = [("nil", lvl, pos)]
toList (Root1 x left right) lvl pos =
  let
    newLvl = lvl + 1
    newPos = pos * 3
  in
    [(show x, lvl, pos)]
    ++ toList left newLvl newPos
    ++ toList right newLvl (newPos + 2)
toList (Root2 x1 x2 left mid right) lvl pos =
  let
    newLvl = lvl + 1
    newPos = pos * 3
  in
    [(show x1 ++ "," ++ show x2, lvl, pos)]
    ++ toList left newLvl newPos
    ++ toList mid newLvl (newPos + 1)
    ++ toList right newLvl (newPos + 2)
```

image 1.5

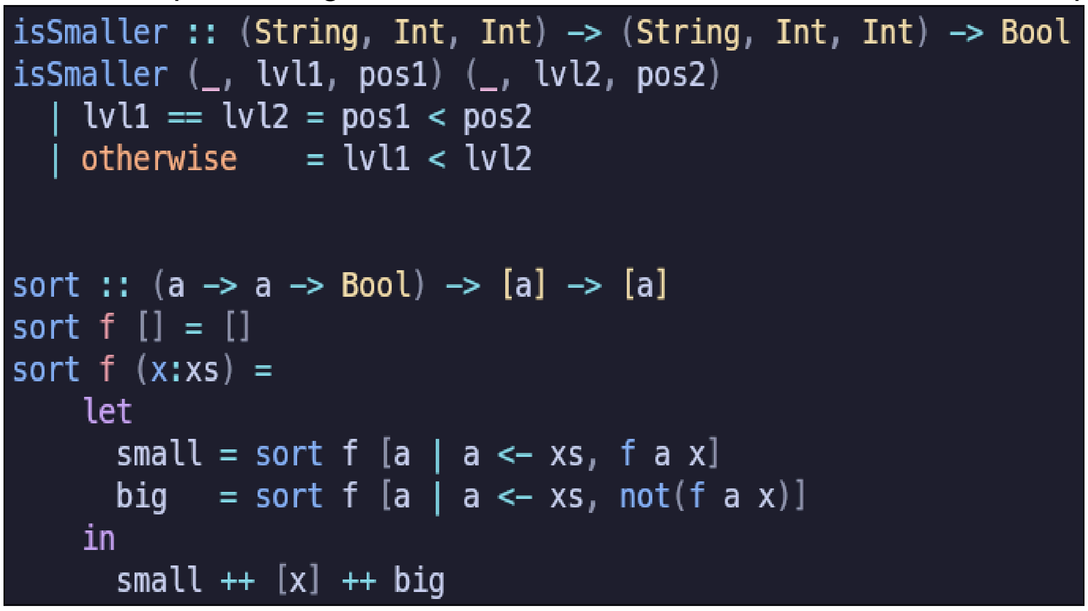
This function is supposed to be called like: toList tree 0 0, in this we start the level and the current position for the first element that is going to be at level 0 and position 0.

In each function call we add to the result the string representation of the current node with the current level and position, and then we call the function on all the branches (left, mid, right).
It is important to correctly update the newLvl and newPos:

- **newLvl** is just the previous one plus one
- **newPos** is the previous one times three (because you need to count all the possible children of the node before this on the current level) + either 0,1,2 (0 for the left, 1 for the mid, 2 for the right).

An example output of calling toList on the tree in the image 1.1
ghci> **toList mytree 0 0**
[("3,10",0,0),("nil",1,0),("6",1,1),("nil",2,3),("nil",2,5),("15,20",1,2),("nil",2,6),("18",2,7),("nil",3,21),("nil",3,23),("nil",2,8)]

Second step is sorting this list based first on the level and then on the position



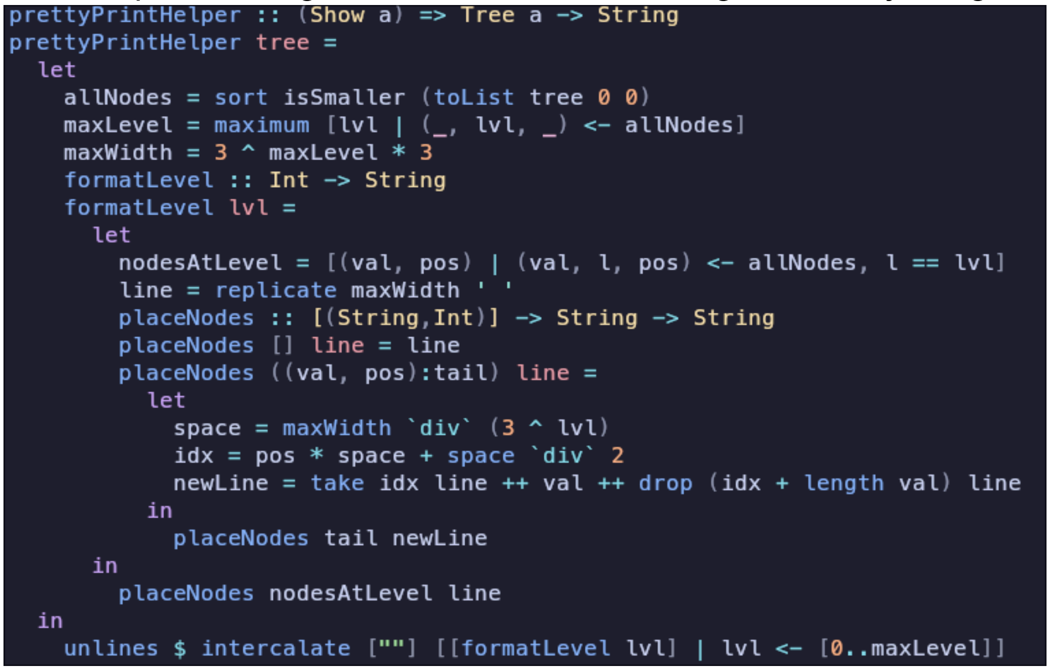
```haskell
isSmaller :: (String, Int, Int) -> (String, Int, Int) -> Bool
isSmaller (_, lvl1, pos1) (_, lvl2, pos2)
  | lvl1 == lvl2 = pos1 < pos2
  | otherwise    = lvl1 < lvl2


sort :: (a -> a -> Bool) -> [a] -> [a]
sort f [] = []
sort f (x:xs) =
    let
      small = sort f [a | a <- xs, f a x]
      big   = sort f [a | a <- xs, not(f a x)]
    in
      small ++ [x] ++ big
```

image 1.6

This is basically the code from lab 5. Only the function is a bit changed to meet the requirements of the list.
The sort function is an implementation of quicksort that uses the isSmaller function to split the list.
So we can get the list of all the nodes in the tree sorted in this way:

```
ghci> sort isSmaller (toList mytree 0 0)
[("3,10",0,0),("nil",1,0),("6",1,1),("15,20",1,2),("nil",2,3),("nil",2,5),("nil",2,6),("18",2,7),("nil",2,8),("nil",3,21),("nil",3,23)]
```

Third step formatting each level into list of string and then joining them



```haskell
prettyPrintHelper :: (Show a) => Tree a -> String
prettyPrintHelper tree =
  let
    allNodes = sort isSmaller (toList tree 0 0)
    maxLevel = maximum [lvl | (_, lvl, _) <- allNodes]
    maxWidth = 3 ^ maxLevel * 3
    formatLevel :: Int -> String
    formatLevel lvl =
      let
        nodesAtLevel = [(val, pos) | (val, l, pos) <- allNodes, l == lvl]
        line = replicate maxWidth ' '
        placeNodes :: [(String,Int)] -> String -> String
        placeNodes [] line = line
        placeNodes ((val, pos):tail) line =
          let
            space = maxWidth `div` (3 ^ lvl)
            idx = pos * space + space `div` 2
            newLine = take idx line ++ val ++ drop (idx + length val) line
          in
            placeNodes tail newLine
      in
        placeNodes nodesAtLevel line
  in
    unlines $ intercalate [""] [[formatLevel lvl] | lvl <- [0..maxLevel]]
```

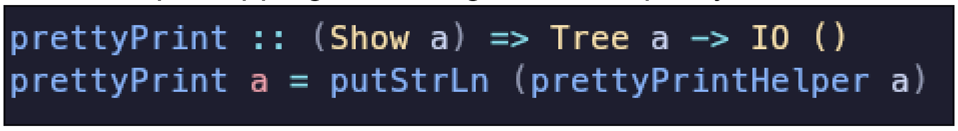
image 1.7

We get **allNodes** by using the previous function,
**maxLevel** by extracting the max from the list,
**maxWidth** by multiplying the number of nodes in the max level (3 ^ maxLevel) with the width of each node (I choose 3 but it is arbitrary)
Then we return the string from the **unlines function**, which has joined the list of string create using **formatLevel** function.

**formatLevel** takes the current level and output the string with the nodes of that level
First it finds all the nodes at that level with list comprehension.
Then it creates a string line long as the maxWidth and "place" the nodes in it by using the placeNodes function.

**placeNodes** to build the string has two cases:
Base case when the list is empty we can return the line built
Otherwise we create another line we the current node insert:
the **space** for each element is the max width divided by the number of nodes on the current level (3 ^ lvl)
The **idx** where we want to insert the current node is the number of the nodes before the current * their space + the space of the current node divided by 2.
**newLine** just take everything up to idx in line add the value of the current node and then add the rest of the line.

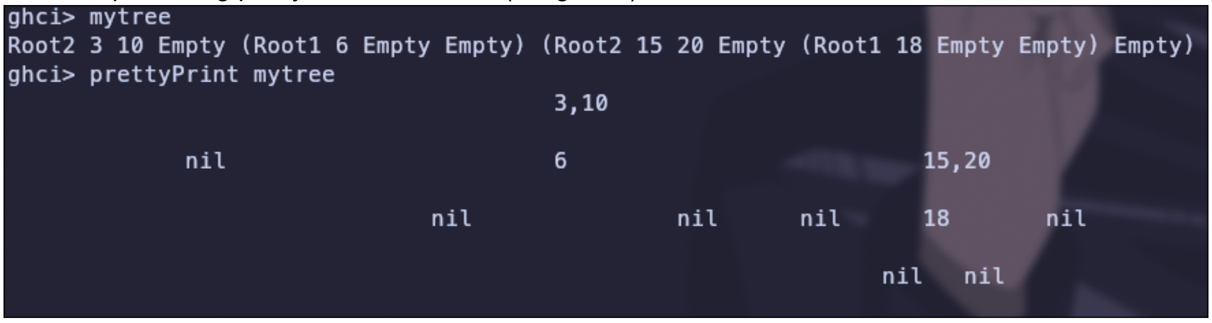
## Fourth step wrapping all the login into the prettyPrint function



```haskell
prettyPrint :: (Show a) => Tree a -> IO ()
prettyPrint a = putStrLn (prettyPrintHelper a)
```

This function is not perfect but it gets the job done at least decently. The problem is the fact that if you have a single node under a double it is not perfectly aligned.

For example calling prettyPrint on our tree (image 1.1) result in:



```
ghci> mytree
Root2 3 10 Empty (Root1 6 Empty Empty) (Root2 15 20 Empty (Root1 18 Empty Empty) Empty)
ghci> prettyPrint mytree
                                        3,10

             nil                        6                          15,20

                               nil               nil      nil      18       nil

                                                                nil   nil
```

## References

Everything in this file is considered my own work with the except of the following resources:

"Prelude," *Haskell.org*, 2024. https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#v:unlines (accessed Nov. 26, 2024).
Haskell Tutorial, "Haskell Tutorial," *YouTube*, Aug. 02, 2015. https://youtu.be/02_H3LjqMr8?feature=shared (accessed Nov. 26, 2024).

L. Haskell, "Learning Haskell for Dummies - Lesson 3 - Strings," *YouTube*, Oct. 27, 2019. https://youtu.be/k9lgXBbP2oM?feature=shared (accessed Nov. 26, 2024)