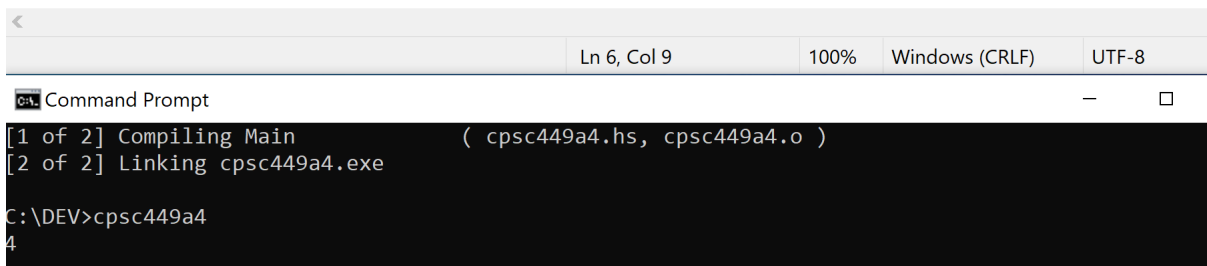


Q1

```
data Expr = Lit Integer | Add Expr Expr | Sub Expr Expr

size :: Expr -> Integer
size (Lit n) = 0
size (Add l r) = (size l) + (size r) + 1
size (Sub l r) = (size l) + (size r) + 1

main = do
  let expr = Add (Lit 1)(Sub(Lit 2)(Add (Lit 3) (Sub (Lit 4) (Lit 5))))
  let result = size expr
  print result
```



```
<
Ln 6, Col 9
100%
Windows (CRLF)
UTF-8
Command Prompt
[1 of 2] Compiling Main ( cpsc449a4.hs, cpsec449a4.o )
[2 of 2] Linking cpsec449a4.exe
C:\DEV>cpsec449a4
4
```

Defines data type Expr with Lit that takes Integer, Add and Sub takes Expr and Expr
Size takes Expr as an argument and return it as integer.

size (Lit n) = 0

Means that any number in Lit n will be returning 0 since we don't need to consider the numbers but only the operator.

size (Add l r) = (size l) + (size r) + 1

size (Sub l r) = (size l) + (size r) + 1

This returns left size of size(Lit n) + right side of size(Lit n) + 1 for both add and sub since if I do size (Sub l r) = (size l) - (size r) + 1, there will be a case where left side is smaller than right side, this produce the negative result which it doesn't work.

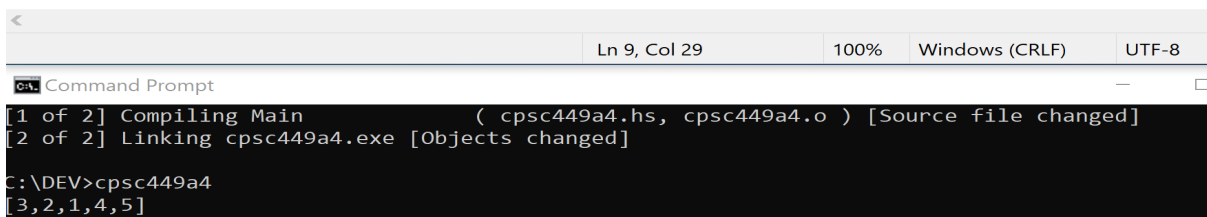
Since any number in size (Lit n) = 0, which it will always increase the count by 1 as Add or Sub is called in equation. Only be counting the number of calls of Add and Sub.

Q2

```
data NTree = NilT | Node Integer NTree NTree

collapse :: NTree -> [Integer]
collapse NilT = []
collapse (Node x l r) = collapse l ++ [x] ++ collapse r

main = do
  let tree = Node 1 (Node 2 (Node 3 NilT NilT) NilT) (Node 4 NilT (Node 5 NilT NilT))
  let result = collapse tree
  print result
```



```
<
Ln 9, Col 29 100% Windows (CRLF) UTF-8
Command Prompt
[1 of 2] Compiling Main ( cpsc449a4.hs, cpssc449a4.o ) [Source file changed]
[2 of 2] Linking cpssc449a4.exe [Objects changed]
C:\DEV>cpssc449a4
[3,2,1,4,5]
```

Defines data type NTree with NilT and Node. Node takes 3 arguments integer, NTree and NTree.

In the collapse, it takes NTree and return it as integer

NilT = [] has empty value. Node takes integer x and NTree l and NTree r which x will be integer to be saved and l and r will be left side of tree and right side of tree respectively.

From Node 1 (Node 2 (Node 3 NilT NilT) NilT) (Node 4 NilT (Node 5 NilT NilT))

We have 1 as parent in our tree, and Node takes 3 arguments so it takes 1 as integer to be saved as parent in tree and child node (Node 2 (Node 3 NilT NilT) NilT) for left side and child node (Node 4 NilT (Node 5 NilT NilT)) for right side. It works same way for other nodes as well. If we recursively keep doing this, we will eventually get.

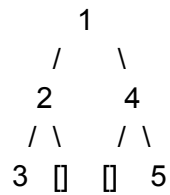
(Node 2 (Node 3 NilT NilT) NilT), node 2 will be parent node for node 3, node 3 takes left side of node 2. This is left part of node for node 1.

```
  2
 / \
3 []
```

(Node 4 NilT (Node 5 NilT NilT)). This is right part for node 2.

```
  4
 / \
[] 5
```

Node 1 (Node 2 (Node 3 NilT NilT) NilT) (Node 4 NilT (Node 5 NilT NilT))



If we list from left to right it will be 3 2 1 4 5.

Q3

```

data GTree a = Leaf a | GNode [GTree a]

numberOfLeaves :: GTree a -> Int
numberOfLeaves (Leaf _) = 1
numberOfLeaves (GNode []) = 0
numberOfLeaves (GNode xs) = sum [numberOfLeaves x | x <- xs]

depth :: GTree a -> Int
depth (Leaf _) = 0
depth (GNode []) = 0
depth (GNode xs) = foldr (\x f -> max (depth x) f) 0 xs + 1

findVal :: (a -> a -> Bool) -> a -> GTree a -> Bool
findVal eq x (Leaf y) = eq x y
findVal eq x (GNode xs) = foldr (\y f -> f || findVal eq x y) False xs

gt = GNode [(Leaf 1), GNode [(Leaf 2), (Leaf 3), (Leaf 4), (Leaf 5)]]

main = do
  let leaves = numberOfLeaves gt
  let treeDepth = depth gt
  let val = findVal (==) 6 gt
  let val1 = findVal (==) 5 gt
  print leaves
  print treeDepth
  print val
  print val1

```

< Ln 19, Col 10

Command Prompt

```

C:\DEV>cpsc449a4
5
2
False
True

```

```

numberOfLeaves (Leaf _) = 1
numberOfLeaves (GNode []) = 0

```

```
numberOfLeaves (GNode xs) = sum [numberOfLeaves x | x <- xs]
```

If there is a leaf, we will be counting that as 1, if it's empty, we won't count so it remains 0. We will then add all the number of leaves.

```
depth (Leaf _) = 0
depth (GNode []) = 0
depth (GNode xs) = foldr (\x f -> max (depth x) f) 0 xs + 1
```

We need to find the maximum depth. x is the element of xs and initial value for depth is 0. It will be finding max depth level of subtrees with depth x so add one to the each depth.

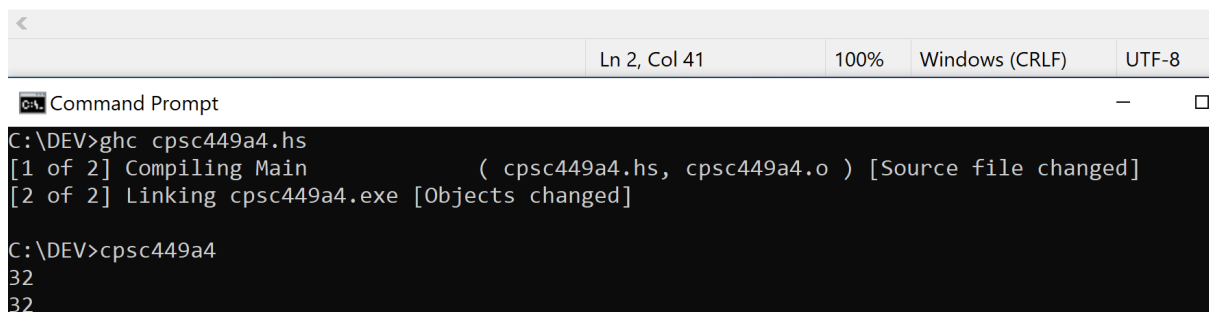
```
findVal :: (a -> a -> Bool) -> a -> GTree a -> Bool
findVal eq x (Leaf y) = eq x y
findVal eq x (GNode xs) = foldr (\y f -> f || findVal eq x y) False xs
```

We will be searching value in GTree by comparing each leaf. It takes 3 arguments so when GTree is leaf node, it compares x and y that if x is equal to y then it returns true otherwise false. We will be checking that whether x is in subtree of y. With or operator, if the value is find in any subtree, it will be returning true.

Q4

```
scalarProduct xs ys = sum[x*y | (x,y) <- zip xs ys]
scalarProductzipWith xs ys = sum(zipWith(*) xs ys)

main = do
  let xs = [1, 2, 3]
  let ys = [4, 5, 6]
  let result1 = scalarProduct xs ys
  let result2 = scalarProductzipWith xs ys
  print result1
  print result2
```



```
C:\DEV>ghc cpssc449a4.hs
[1 of 2] Compiling Main             ( cpssc449a4.hs, cpssc449a4.o ) [Source file changed]
[2 of 2] Linking cpssc449a4.exe [Objects changed]

C:\DEV>cpssc449a4
32
32
```

```
scalarProduct xs ys = sum[x*y | (x,y) <- zip xs ys]
```

Zip takes two list and returns it like it pairs with first element of first list with first element of second list. Second element of first list with second element of second list.

zip creates list $xs = [1,2,3]$ and $ys = [4,5,6]$ and pairs it like $([1,4][2,5][3,6])$. The element of xs and ys are x and y . It now multiplies the element that has been paired which equals to $[4,10,18]$ then sum the numbers in list which equals to 32.

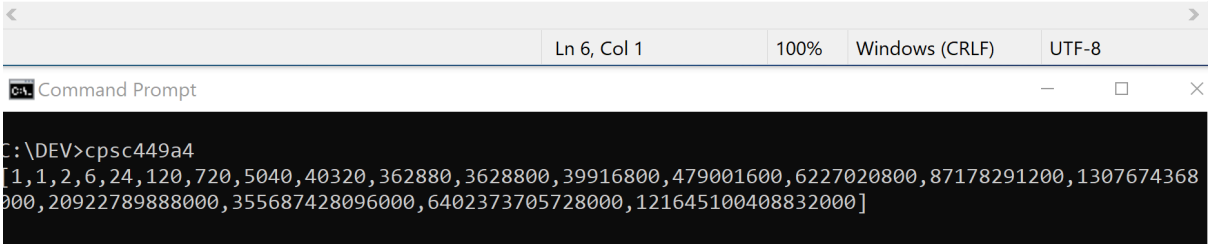
zipWith

zipWith takes two list and multiplies elements in two list respectively then sum the element in list. From $xs = [1,2,3]$ and $ys = [4,5,6]$, multiplies first element of first list with first element of second list. It would result $[4,10,18]$ and then we sum the elements in the list which it will be 32.

Q5

```
factorial = 1 : zipWith (*) factorial[1..]
```

```
main = do
  let result = take 20 factorial
  print result
```



```
C:\DEV>cpsc449a4
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800,87178291200,1307674368000,20922789888000,355687428096000,6402373705728000,121645100408832000]
```

```
factorial = 1 : zipWith (*) factorial[1..]
```

Factorial has first element of 1 in list and we have factorial list which it has infinite starting from 1. We use zipWith with multiply that it will multiply the two list. First element is 1, second element will be $[1*1] = [1]$. Third element will be $[1*2] = [2]$. Fourth is $[2*3] = 6$

Now we have list of $[1,1,2,6]$ and we will be multiply factorial[1....]. This would get us, zipWith (*) $[1,1,2,6] [1,2,3,4]$ which would result $[1*1,1*2,2*3,6*4] = [1,2,6,24]$. Since first element is 1, list will be represented as $[1,1,2,6,24,120...]$