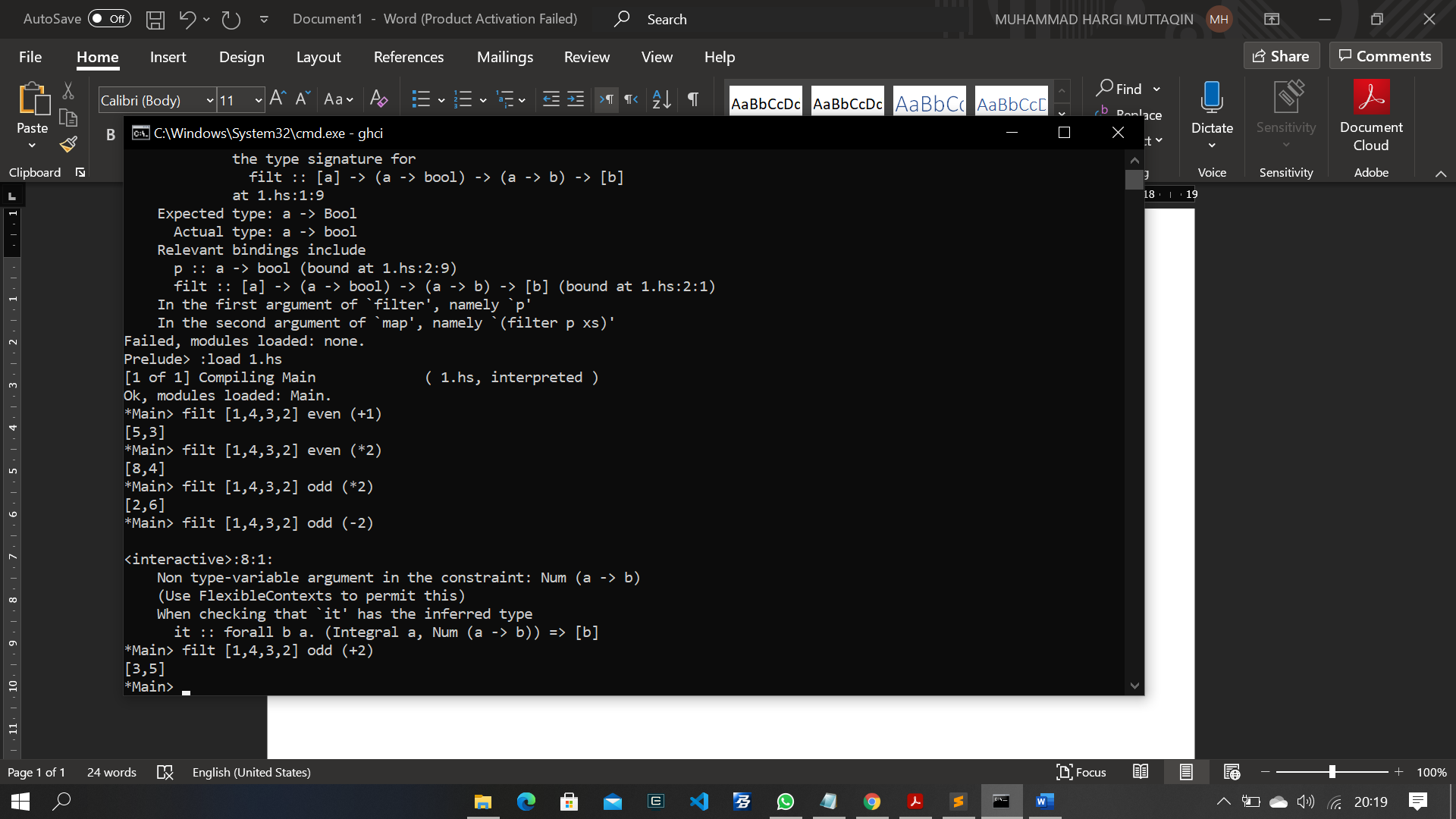
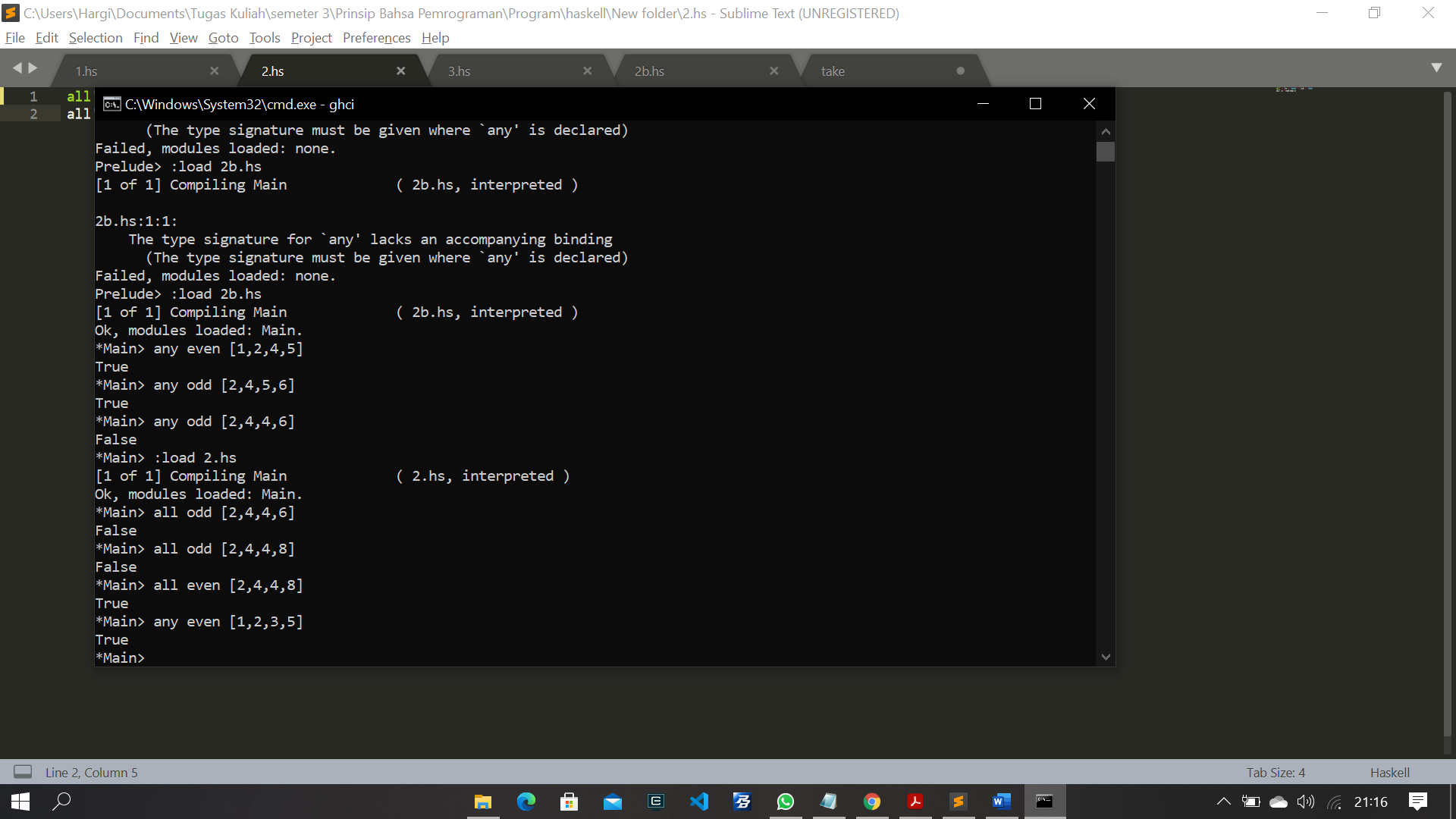
1. filt :: [a] -> (a -> Bool) -> (a -> b) -> [b]

filt xs p f = map f (filter p xs)

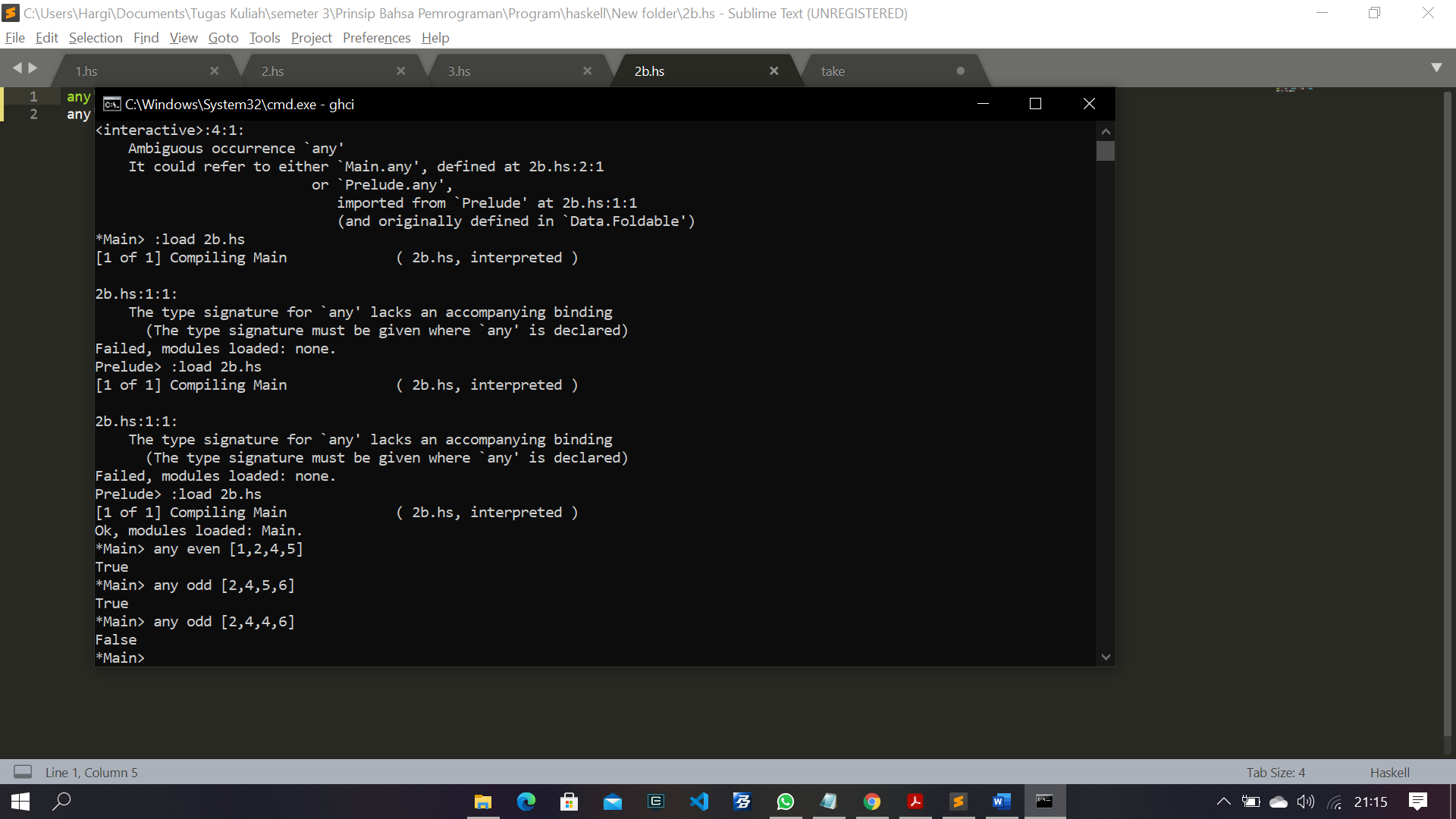


1. a. all' :: (a -> Bool) -> [a] -> Bool

all' p = and.map p 

b. any' :: (a -> Bool) -> [a] -> Bool

any' p = or.map p

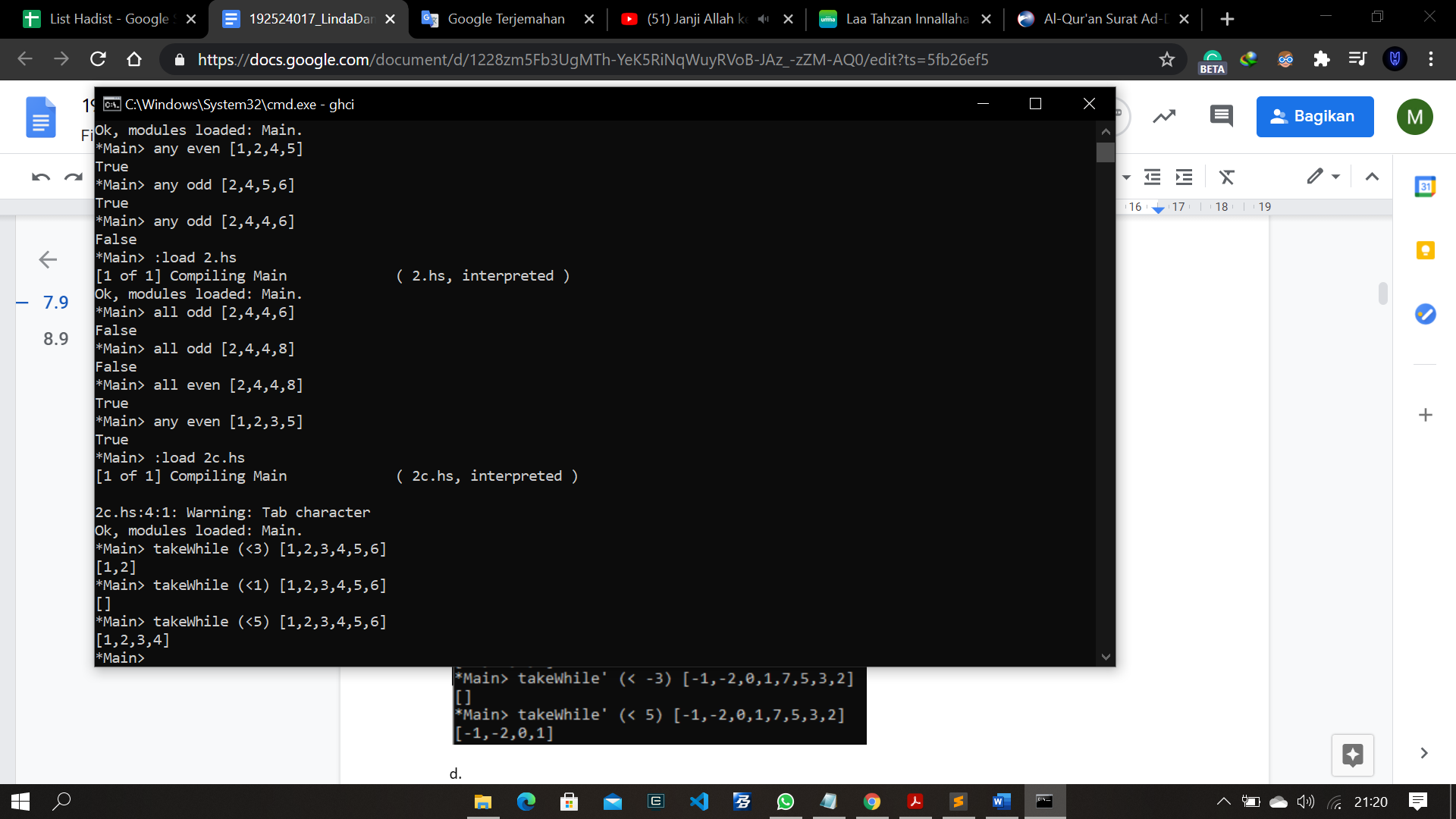


c. takeWhile' :: (a -> Bool) -> [a] -> [a]

takeWhile' \_ [] = []

takeWhile' p (x:xs) | p x = x : takeWhile' p xs

| otherwise = []

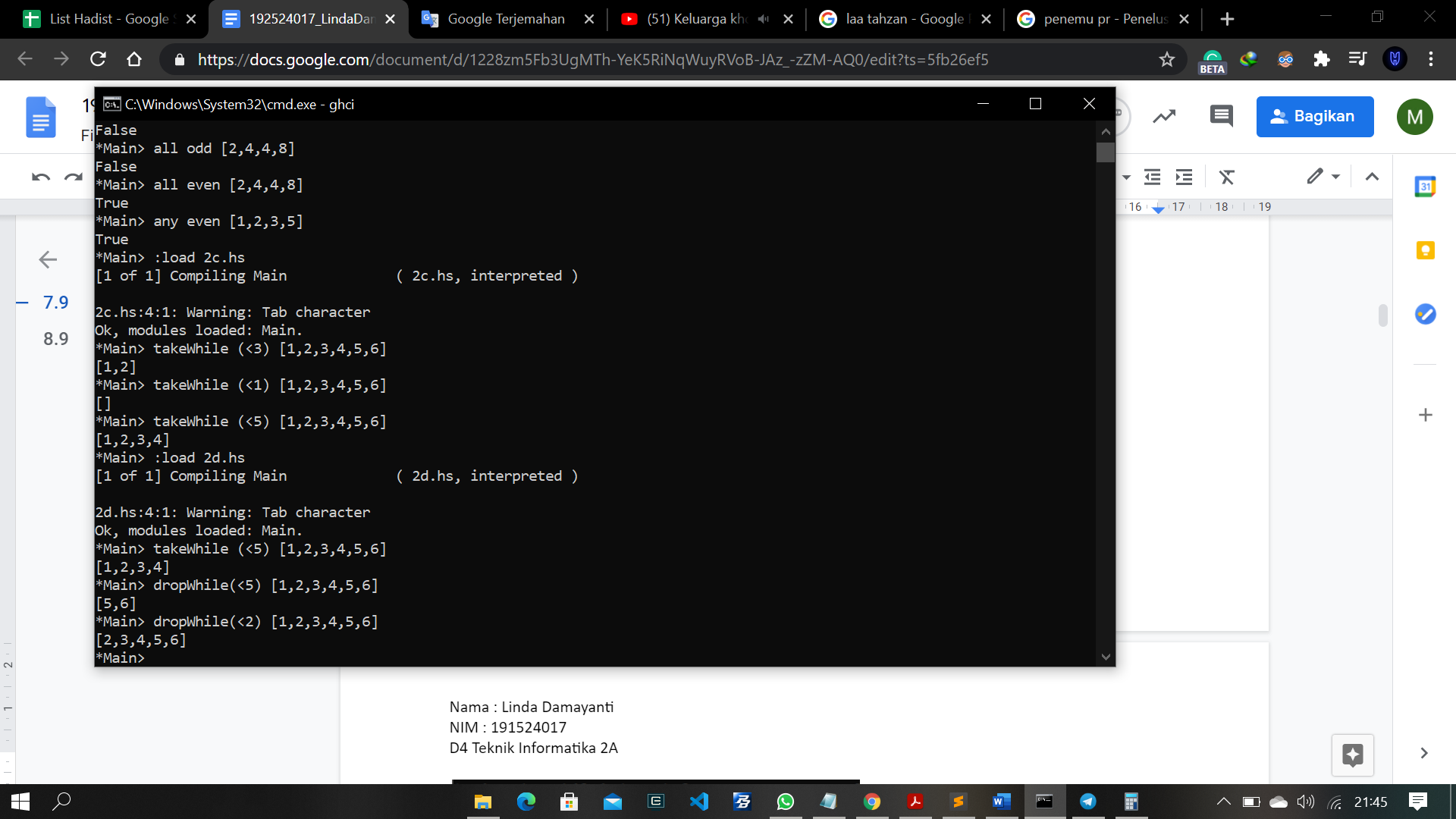


d. dropWhile' :: (a -> Bool) -> [a] -> [a]

dropWhile' \_ [] = []

dropWhile' p (x:xs) | p x = x : dropWhile' p xs

| dropWhile = x:xs

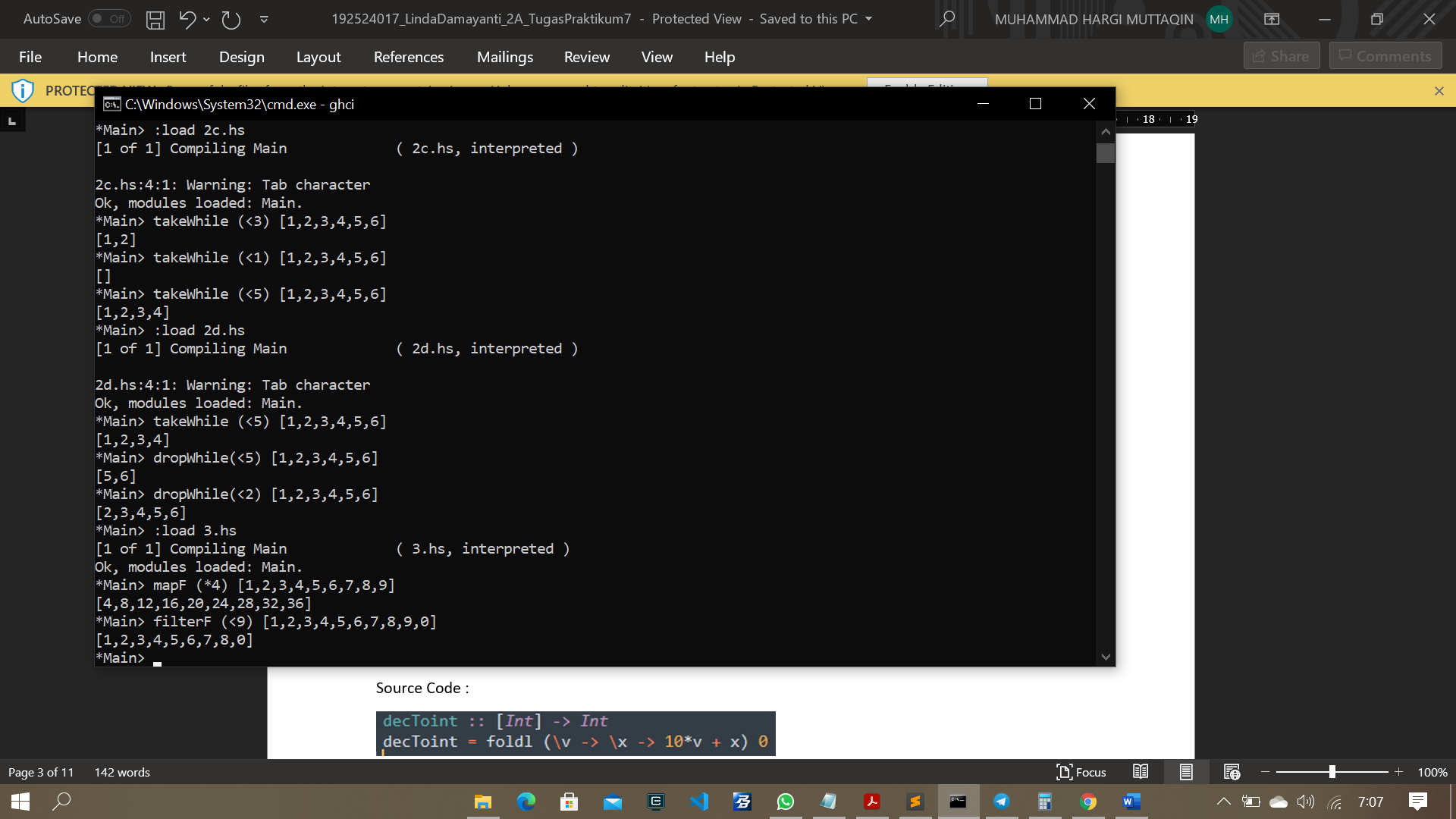


1. mapF :: (a -> b) -> [a] -> [b]

mapF f = foldr ((:).f) []

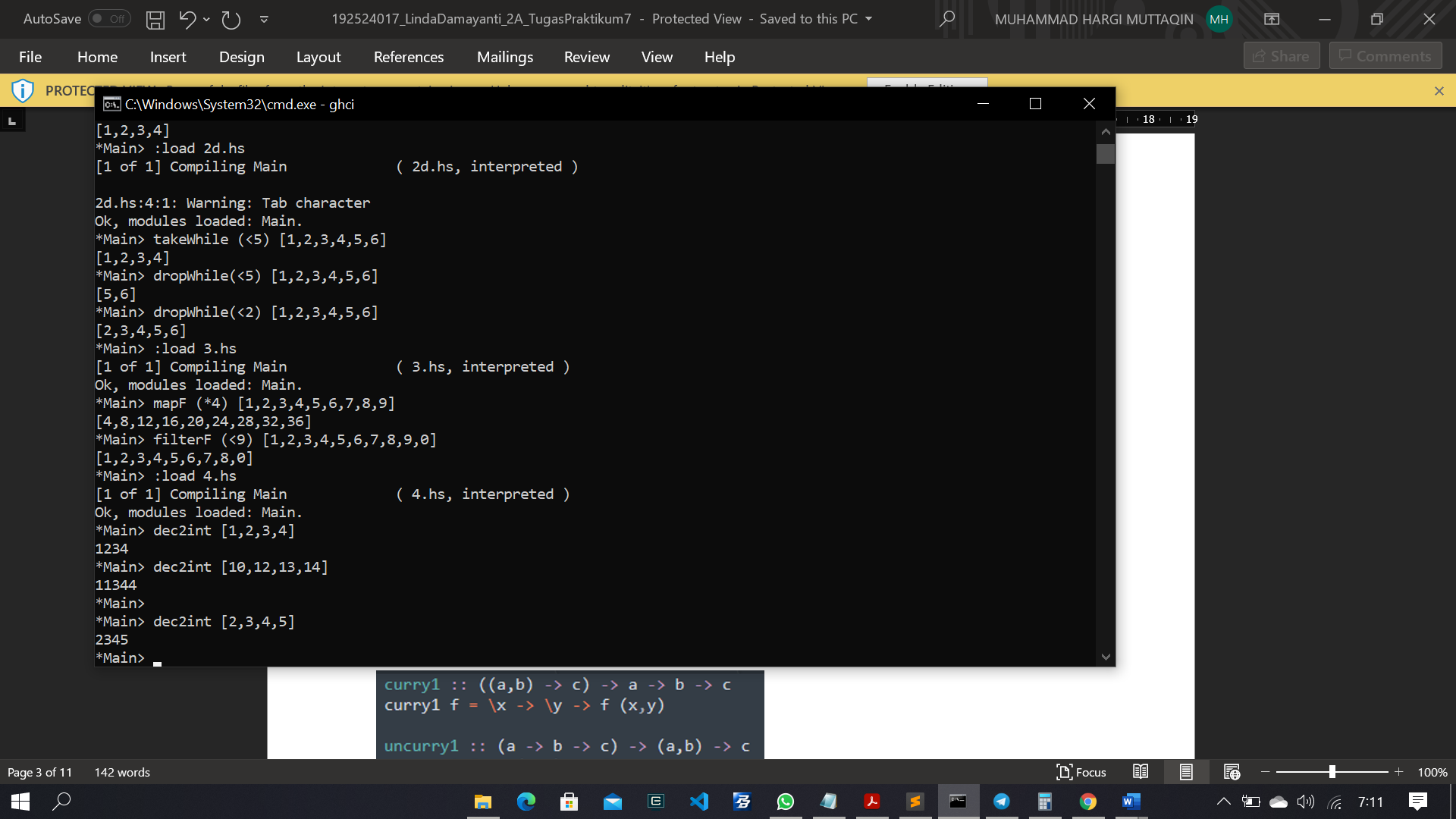
filterF :: (a -> Bool) -> [a] -> [a]

filterF p = foldr (\x xs -> if p x then x : xs else xs) []



1. dec2int :: [Int] -> Int

dec2int = foldl (\v -> \x -> 10\*v + x) 0

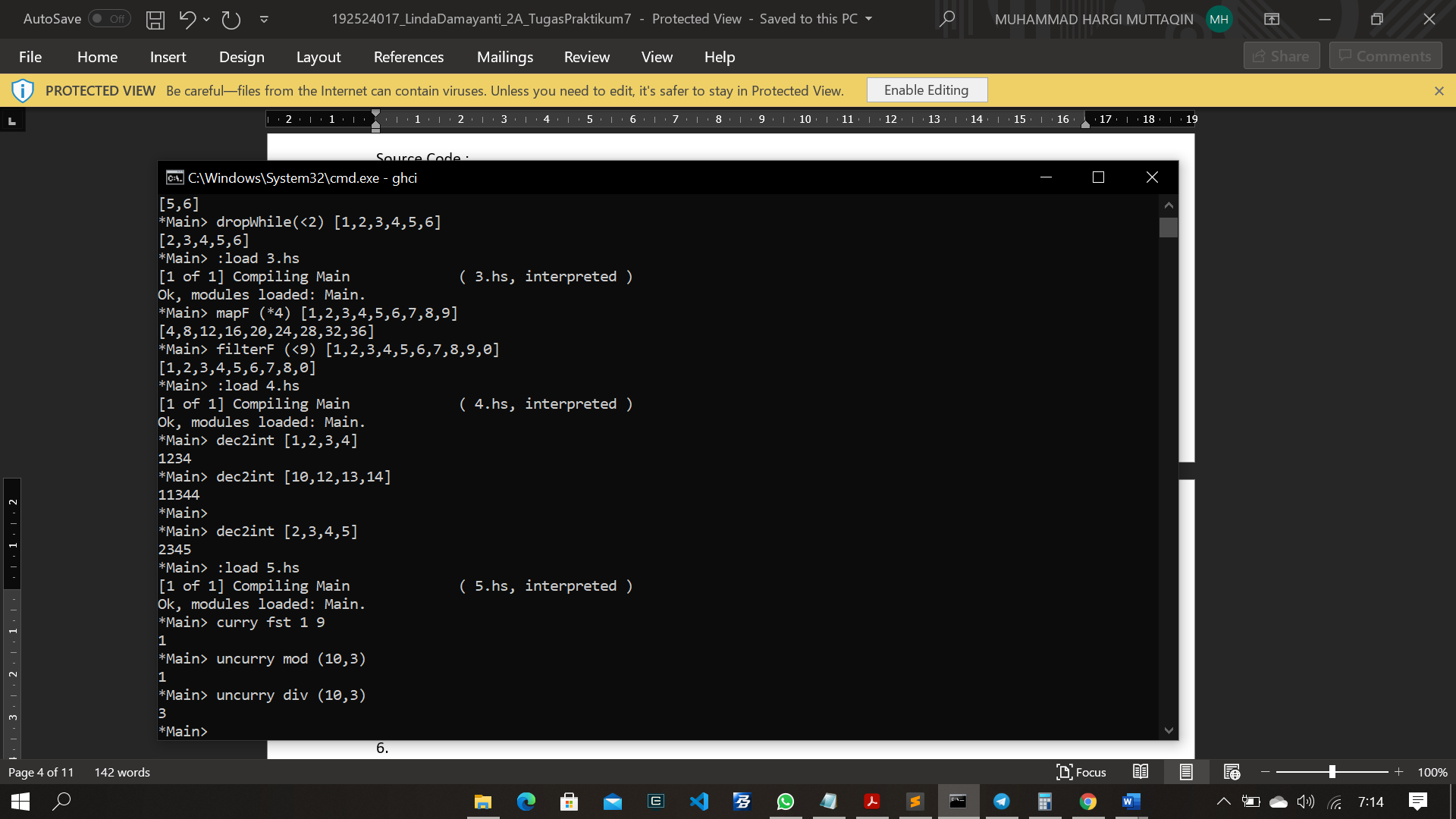


1. curry' :: ((a,b) -> c) -> a -> b -> c

curry' f = \x y -> f (x, y)

uncurry' :: (a -> b -> c) -> ((a, b) -> c)

uncurry' f = \(x, y) -> f x y



1. unfold p h t x | p x = []

| otherwise = h x : unfold p h t (t x)

nt2bin :: Int -> [Int]

int2bin = unfold (==0) (`mod` 2) (`div` 2)

chop8 :: [Int] -> [[Int]]

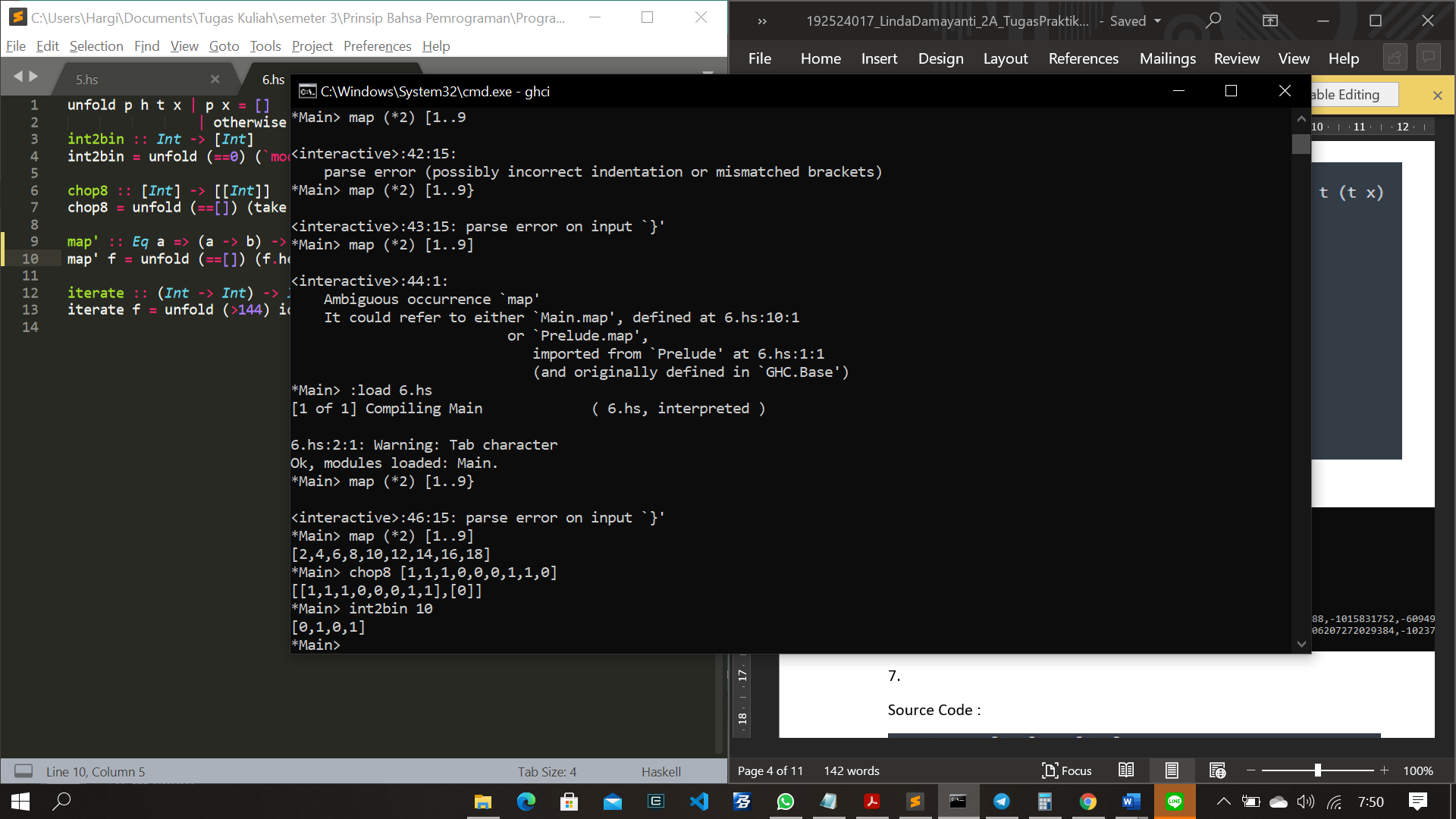
chop8 = unfold (==[]) (take 8) (drop 8)

map' :: Eq a => (a -> b) -> [a] -> [b]

map' f = unfold (==[]) (f.head) (tail)

iterate :: (Int -> Int) -> Int -> [Int]

iterate f = unfold (>144) id f



1. parity :: [Int] -> [Int]

parity xs | even (sum xs) = xs ++ [0]

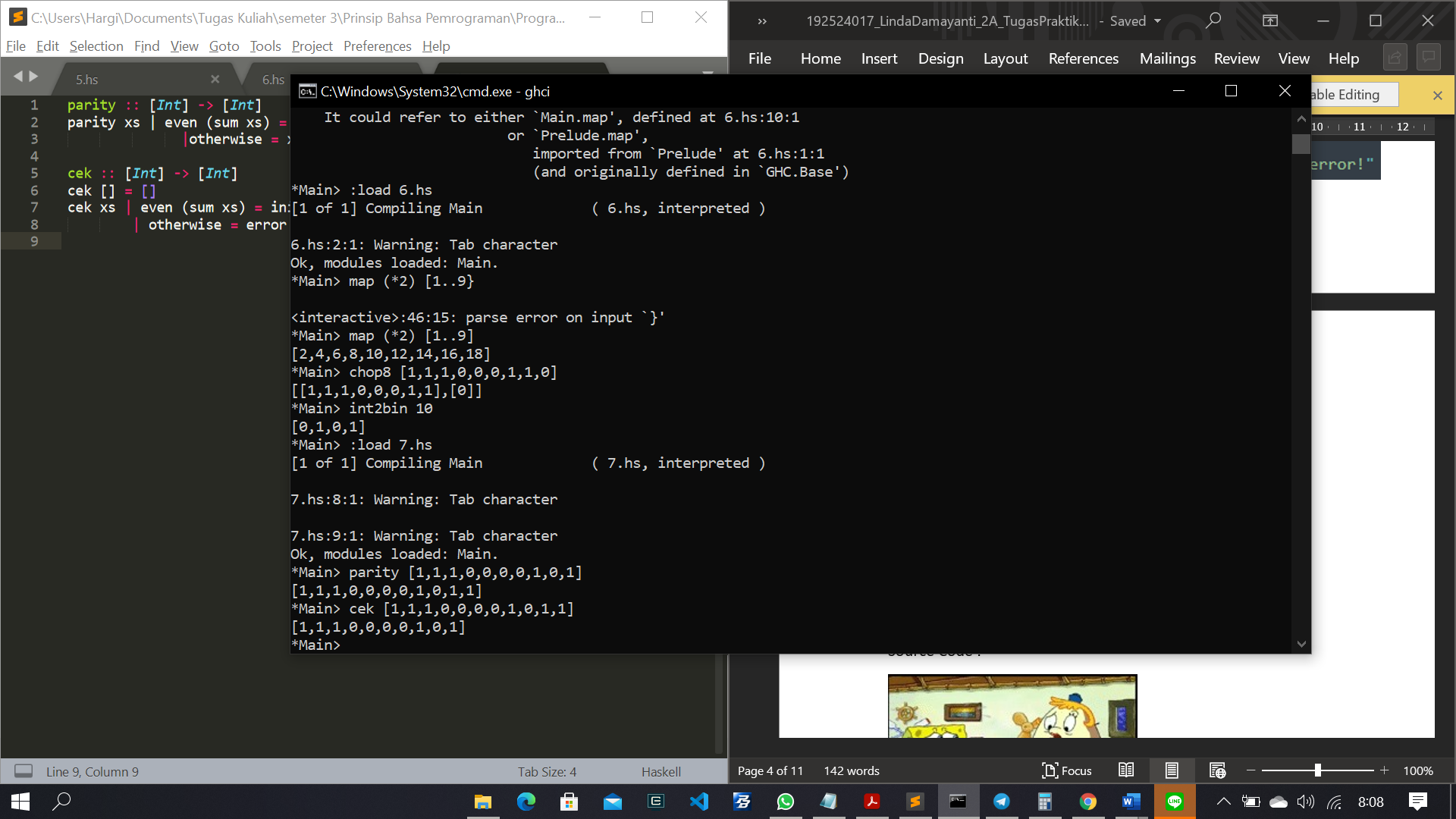
|otherwise = xs ++ [1]

cek :: [Int] -> [Int]

cek [] = []

cek xs | even (sum xs) = init xs

| otherwise = error "Error"



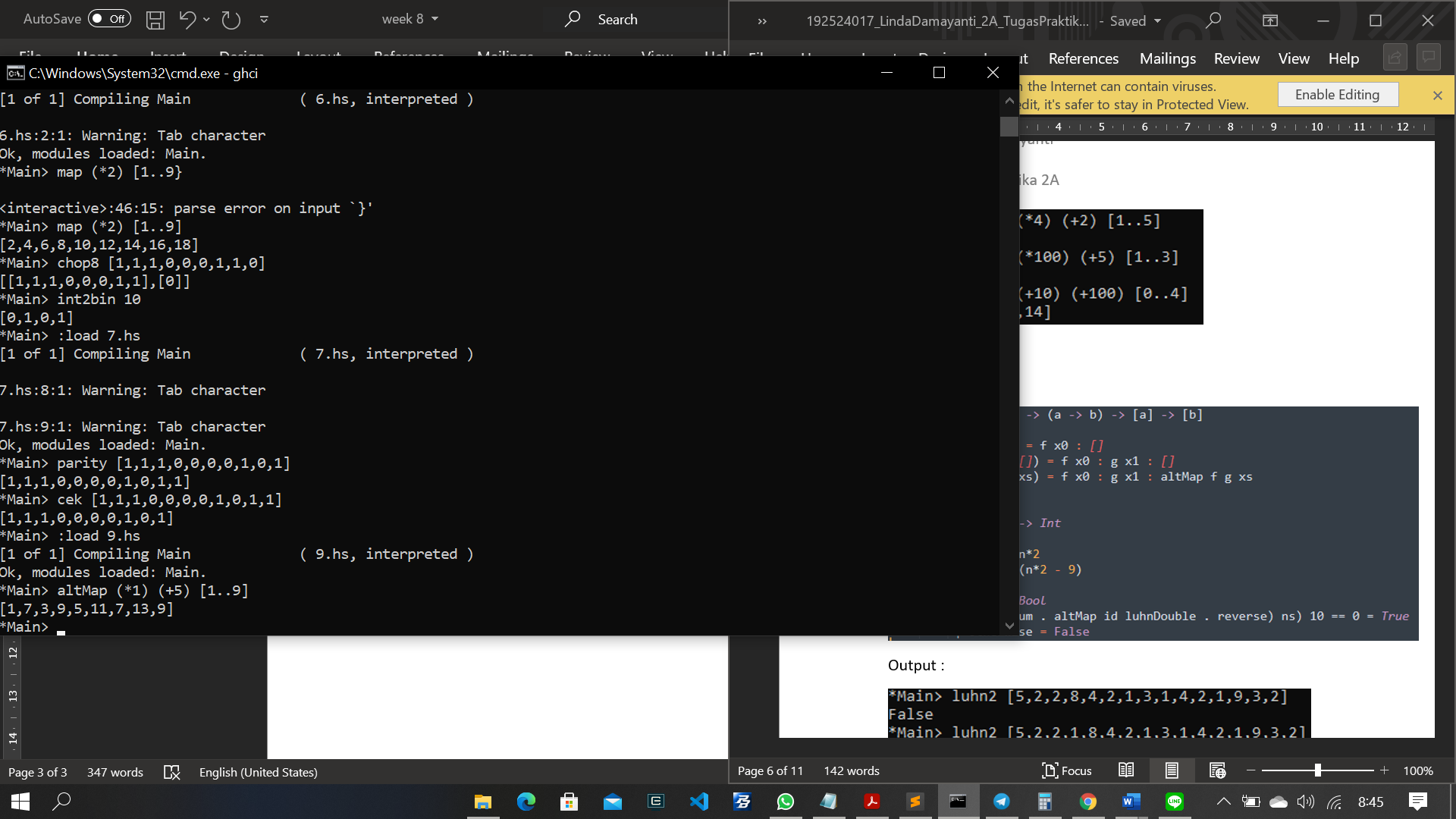
1. –
2. altMap :: (a->b) -> (a -> b) -> [a] -> [b]

altMap \_ \_ [] = []

altMap f \_ (x0 : []) = f x0 : []

altMap f g (x0:x1:[]) = f x0 : g x1 : []

altMap f g (x0:x1:xs) = f x0 : g x1 : altMap f g xs



1. altMap :: (a->b) -> (a -> b) -> [a] -> [b]

altMap \_ \_ [] = []

altMap f \_ (x0 : []) = f x0 : []

altMap f g (x0:x1:[]) = f x0 : g x1 : []

altMap f g (x0:x1:xs) = f x0 : g x1 : altMap f g xs

luhnDouble :: Int -> Int

luhnDouble n

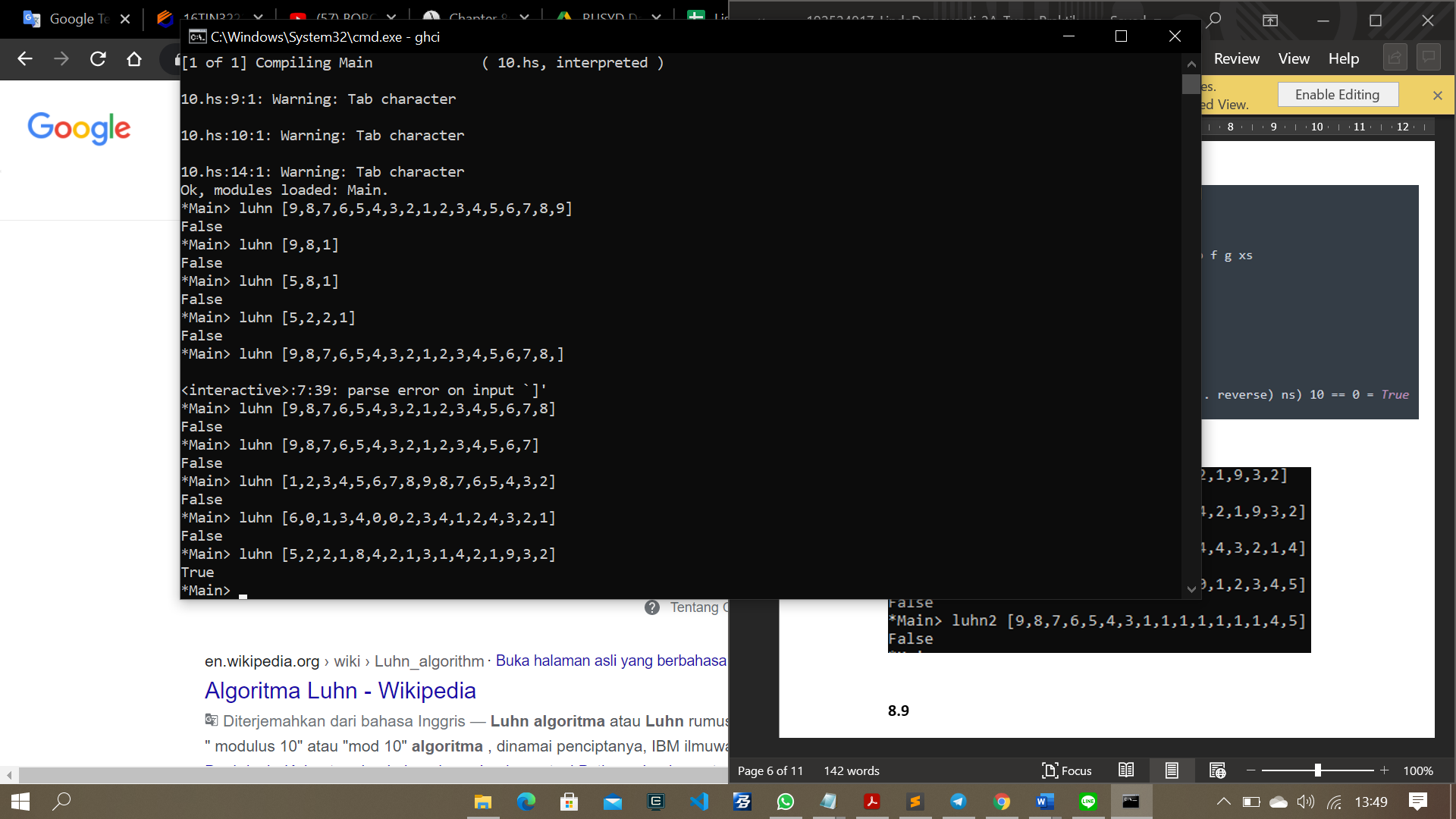
| n\*2 < 9 = n\*2

| otherwise = (n\*2 - 9)

luhn :: [Int] -> Bool

luhn ns | mod ((sum. altMap id luhnDouble . reverse) ns) 10 == 0 = True

| otherwise = False



8.9

1. type Nat = Int

addN :: Nat -> Nat -> Nat

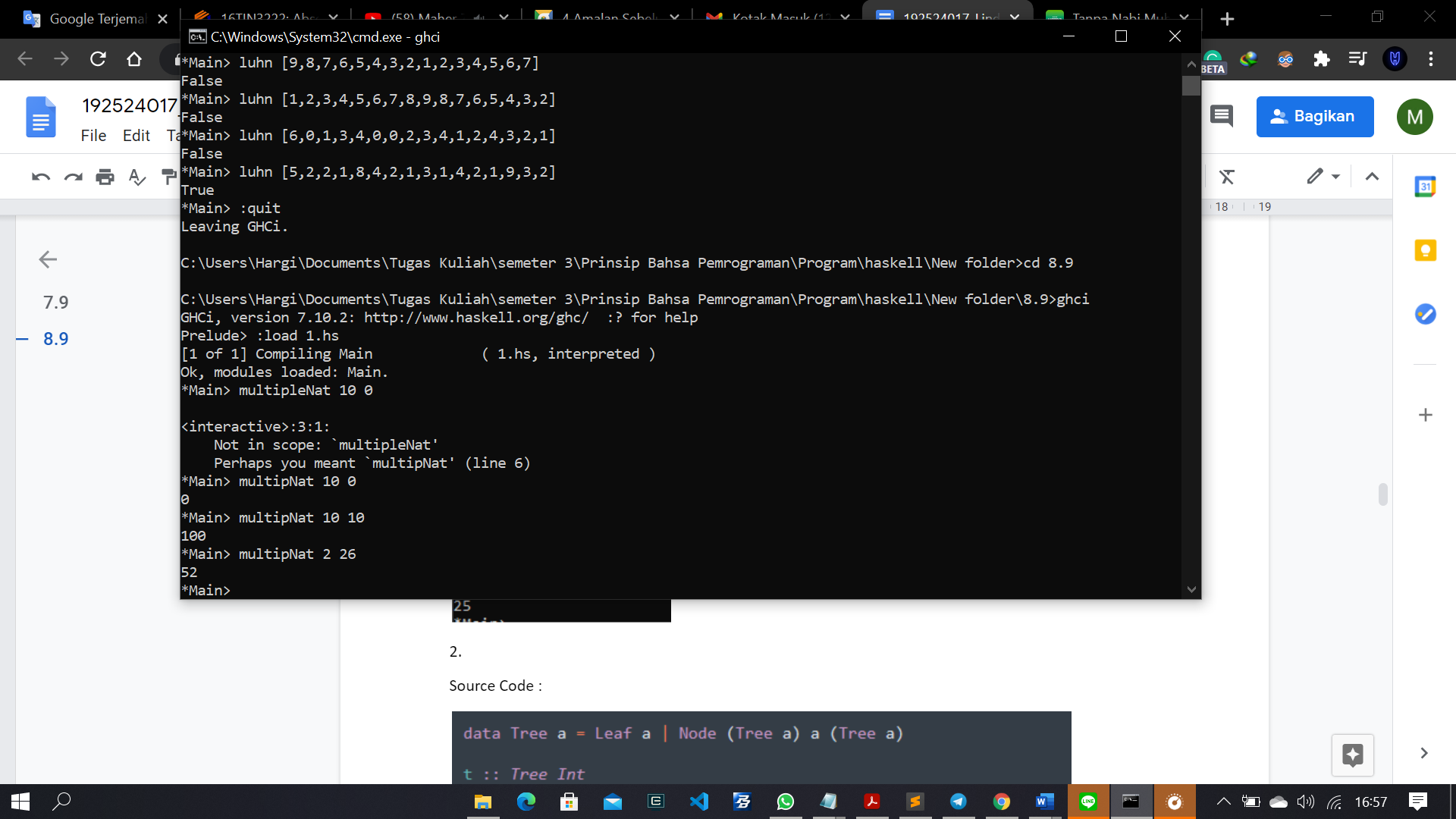
addN m n = m + n

multipNat :: Nat -> Nat -> Nat

multipNat 0 n = 0

multipNat n 0 = 0

multipNat m n = addN n (multipNat (m-1)n)



2. data Tree a = Leaf a | Node (Tree a) a (Tree a)

t :: Tree Int

t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))

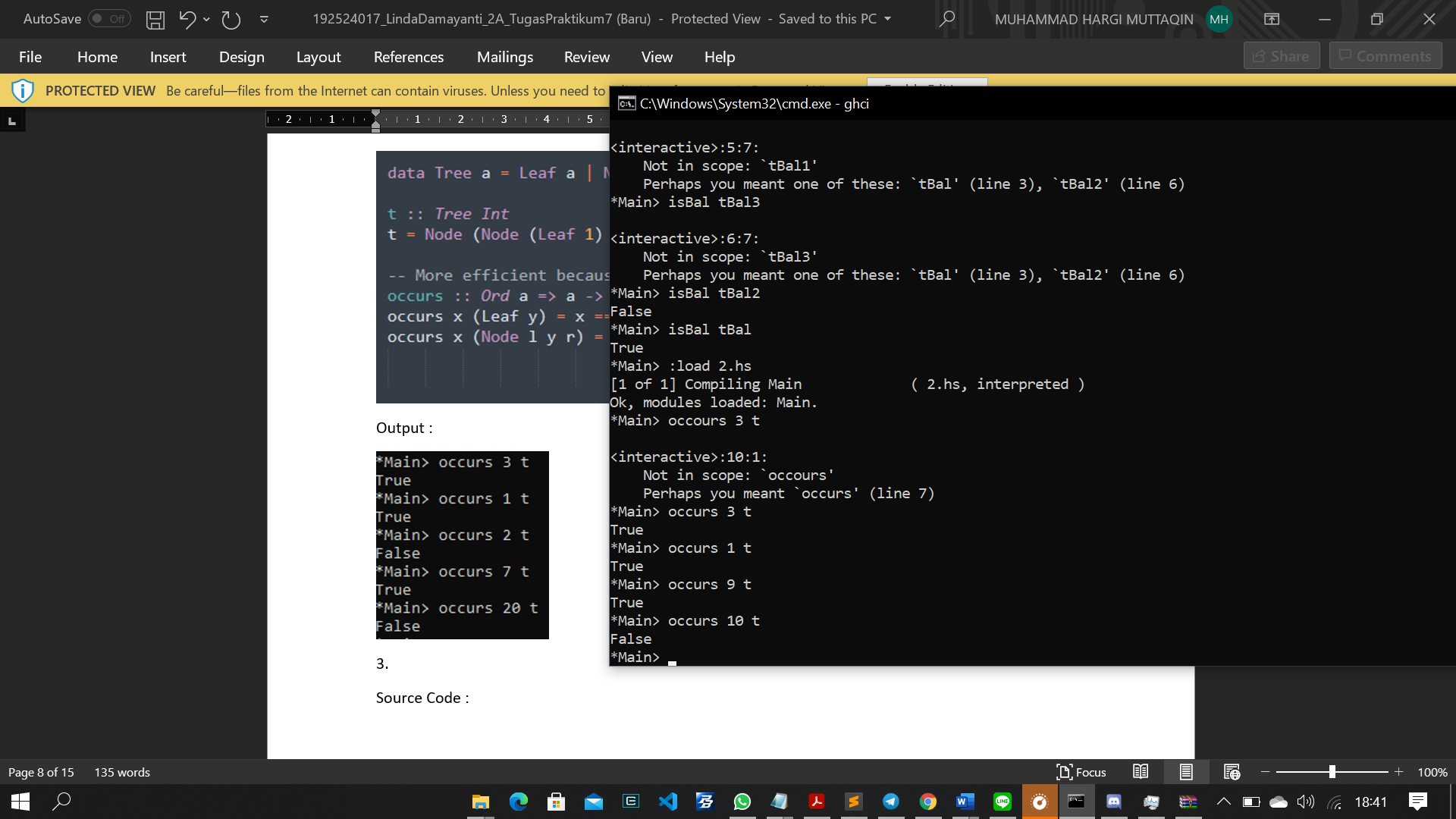
occurs :: Ord a => a -> Tree a -> Bool

occurs x (Leaf y) = x == y

occurs x (Node l y r) = case compare x y of LT -> occurs x l

EQ -> True

GT -> occurs x r



3. data TreeBal a = LeafBal a | NodeBal (TreeBal a) (TreeBal a) deriving Show

tBal :: TreeBal Int

tBal = NodeBal (NodeBal (LeafBal 1) (LeafBal 4))

(NodeBal (LeafBal 6) (LeafBal 9))

tBal2 :: TreeBal Int

tBal2 = NodeBal

(NodeBal

(LeafBal 4)

(NodeBal

(LeafBal 13)

(LeafBal 0)))

(NodeBal

(NodeBal

(LeafBal 333)

(NodeBal

(LeafBal 6)

(LeafBal 9)))

(LeafBal 42))

isBal :: TreeBal Int -> Bool

isBal (LeafBal \_) = True

isBal (NodeBal l r) = (diffLeaves <= 1) && isBal l && isBal r

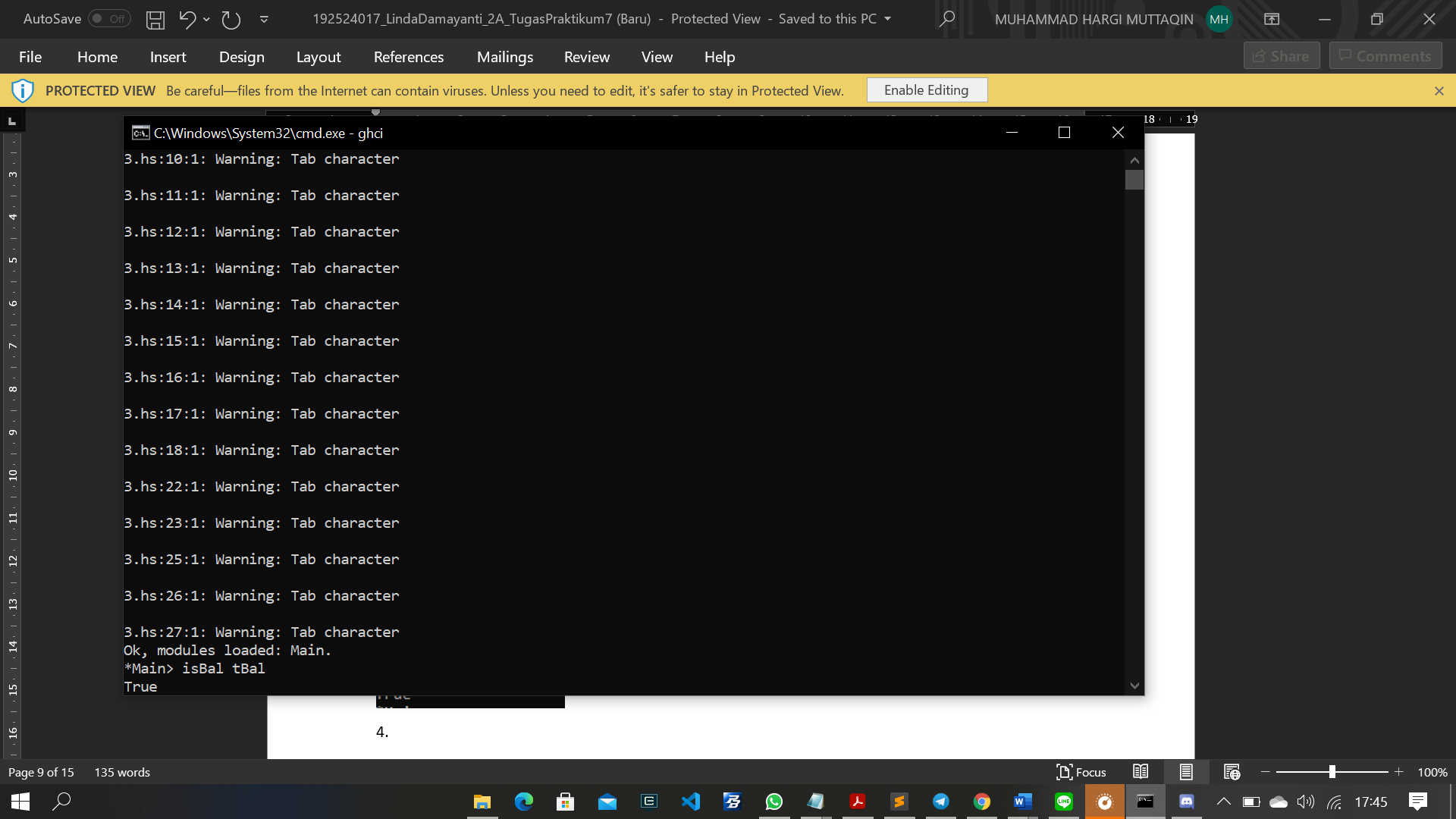
where

diffLeaves = abs (numLeaves l - numLeaves r)

numLeaves :: TreeBal Int -> Int

numLeaves (LeafBal \_) = 1

numLeaves (NodeBal l r) = numLeaves l + numLeaves r



4. data TreeBal a = LeafBal a | NodeBal (TreeBal a) (TreeBal a) deriving Show

balance :: [a] -> TreeBal a

balance [] = error " Tidak Boleh Kosong"

balance [y] = LeafBal y

balance ys = NodeBal (balance (fst (halveL ys))) (balance (snd (halveL ys)))

halveL :: [a] -> ([a],[a])

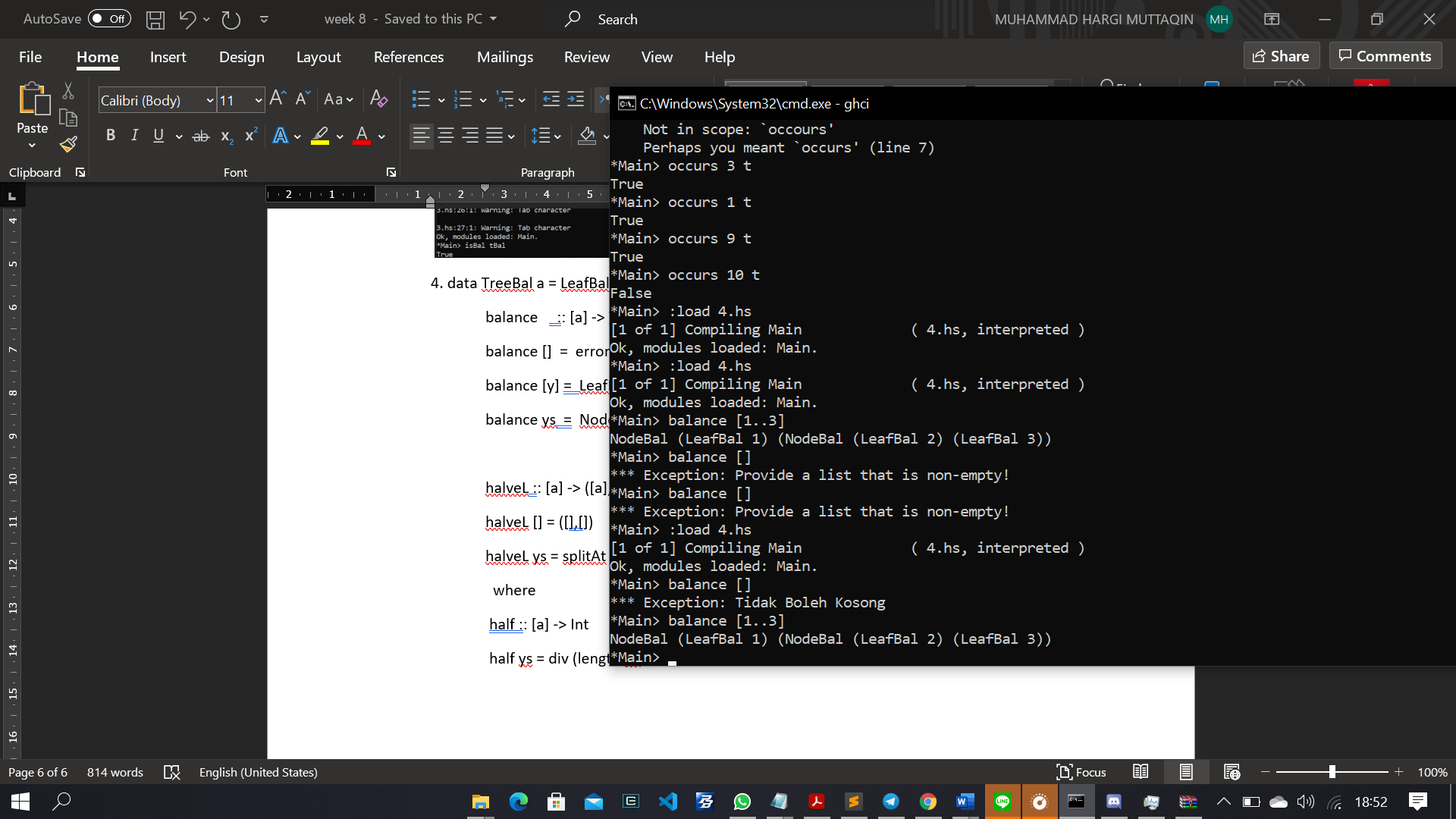
halveL [] = ([],[])

halveL ys = splitAt (half ys) ys

where

half :: [a] -> Int

half ys = div (length ys) 2



5. data Expr' = Val' Int | Add' Expr' Expr'

folde :: (Int -> i) -> (i -> i -> i) -> Expr' -> i

folde f g (Val' x) = f x

folde f g (Add' x y) = g (folde f g x) (folde f g y)



6. data Expr' = Val' Int | Add' Expr' Expr'

folde :: (Int -> i) -> (i -> i -> i) -> Expr' -> i

folde f g (Val' x) = f x

folde f g (Add' x y) = g (folde f g x) (folde f g y)

evalE :: Expr' -> Int

evalE (Val' k) = k

evalE (Add' e1 e2) = folde toEnum (+) (Add' e1 e2)

numVals :: Expr' -> Int

numVals (Val' k) = 1

numVals (Add' e1 e2) = numVals e1 + numVals e2

