1.

isVal :: [[Int]] -> String

isVal [] = "True"

isVal [\_] = "True"

isVal (x:y:xs)

| all (\row -> length row == length x) (y:xs) =

let rowCount = length (x:y:xs)

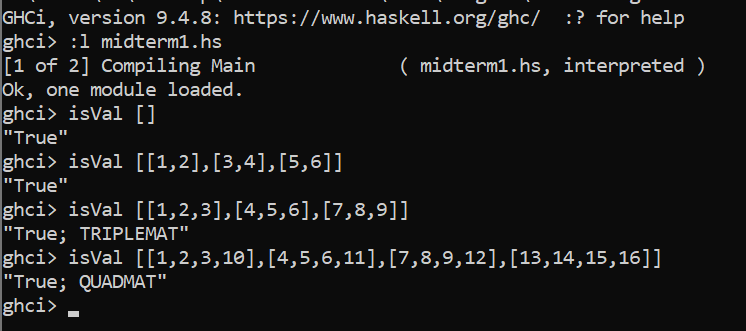
colCount = length x

in if rowCount == 3 && colCount == 3 then "True; TRIPLEMAT"

else if rowCount == 4 && colCount == 4 then "True; QUADMAT"

else "True"

| otherwise = "False"



2.

partitionList::(a->Bool)->[a]->([a],[a])

partitionList \_ [] =([],[])

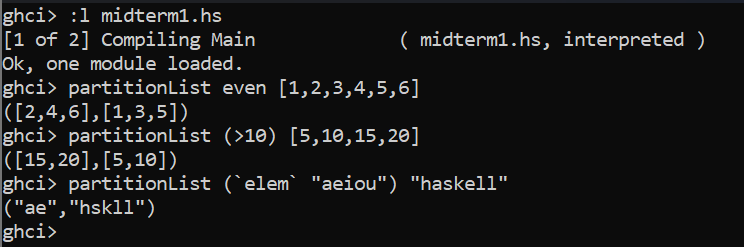
partitionList p (x:xs)

| p x = (x:c,d)

| otherwise = (c,x:d)

where

(c,d)=partitionList p xs



4.

updateElement :: [Int] -> Int -> Int -> [Int]

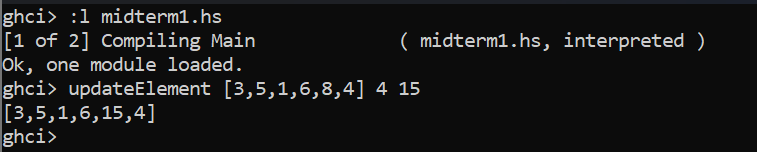
updateElement xs i v = replaceElement xs i v 0

replaceElement :: [Int] -> Int -> Int -> Int -> [Int]

replaceElement [] \_ \_ \_ = []

replaceElement (x:xs) i v y

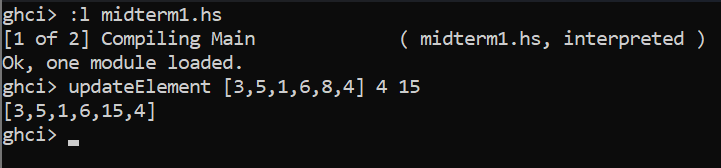
| y == i = v : replaceElement xs i v (y + 1)

| otherwise = x : replaceElement xs i v (y + 1)  
  


Non recursive method

updateElement :: [Int] -> Int -> Int -> [Int]

updateElement xs i v = take i xs ++ [v] ++ drop (i+1) xs



5.

compose :: (b -> c) -> (a -> b) -> (a -> c)

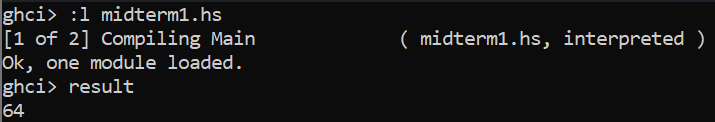
compose f g = \x -> f (g x)

composedFunction :: Int -> Int

composedFunction = compose (\x -> x \* x) (\x -> x + 5)

result :: Int

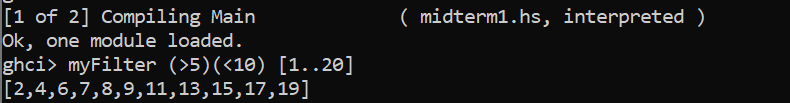
result = composedFunction 3



6.

myFilter:: (Int->Bool)->(Int->Bool)->[Int]->[Int]

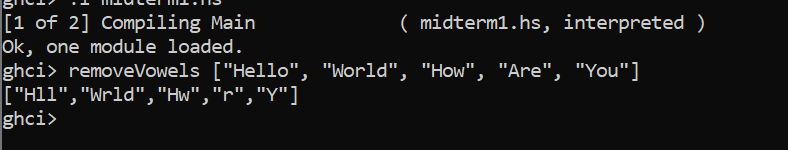
myFilter p1 p2 xs = [x|x <- xs,(odd x && p1 x) || (even x && p2 x)]



7.

removeVowels :: [String] -> [String]

removeVowels = map (filter (`notElem` "AEIOUaeiou"))



8.

rotateList :: [a] -> Int -> [a]

rotateList xs n

| n >= 0 = drop n xs ++ take n xs

| otherwise = drop len xs ++ take len xs

where len = n + length xs

