Laboratorium 5.

0) Przygotowanie środowiska

W konsoli/terminalu wpisujemy kolejno

```
$ cd
$ mkdir haskell-lab5
$ cd haskell-lab5
```

1) Elementarne operacje I/0

1. W konsoli GHCi wpisujemy kolejno

```
ghci> :i IO

ghci> :t getChar
ghci> getChar
a

ghci> let x = getChar
ghci> x
4
ghci> x
b
```

2. W konsoli GHCi wpisujemy kolejno

```
ghci> :t putChar
ghci> putChar a
ghci> :t putChar 'a'
ghci> putChar 'a'
```

3. W konsoli GHCi wpisujemy kolejno

```
ghci> :t getLine
ghci> getLine
"Hello"
ghci> getLine
Hello

ghci> let line = getLine
ghci> line
Hello
```

4. W konsoli GHCi wpisujemy kolejno

```
ghci> :t putStr
ghci> putStr "Hello"
ghci> :t putStrLn
ghci> putStrLn "Hello"
```

5. W konsoli GHCi wpisujemy kolejno

```
ghci> :t print
ghci> print 1
ghci> print a
ghci> print (1,2)
ghci> print [1..5]

ghci> newtype IntBox = MkIntBox Int
ghci> let ib1 = MkIntBox 1
ghci> print ib1

ghci> newtype IntBox = MkIntBox Int deriving Show
ghci> let ib1 = MkIntBox 1
ghci> print ib1
```

6. W konsoli GHCi wpisujemy kolejno

```
ghci> :t return 'a'
ghci> return 'a'
ghci> :t return 1
ghci> return 1
ghci> :t return "Hello"
ghci> return "Hello"

ghci> let hello = return "Hello"
ghci> hello
ghci> show "Hello"
ghci> show hello
ghci> :t "Hello"
ghci> :t hello
```

2) Łączenie (sekwencje) ‘akcji’ I/O — operatory `>>` (*then*) i `>>=` (*bind*), notacja *do*

1. W konsoli GHCi wpisujemy kolejno

```
ghci> :t (>>)
ghci> putChar 'a' >> putChar '\n'
```

2. W pliku `ex2.hs` wpisujemy

```
actSeq = putChar 'A' >> putChar 'G' >> putChar 'H' >>  putChar '\n'
```

generated by haroopad

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> :t actSeq
ghci> actSeq
```

3. W pliku `ex2.hs` dodajemy

```
doActSeq = do
  putChar 'A'
  putChar 'G'
  putChar 'H'
  putChar '\n'
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> :t putChar 'A'
ghci> :t putChar '\n'
ghci> :t doActSeq
ghci> doActSeq
```

4. W konsoli GHCi wpisujemy kolejno

```
ghci> :t (>>=)
ghci> :t getLine
ghci> :t putStrLn
ghci> :t getLine >>= putStrLn
```

5. W pliku `ex2.hs` dodajemy

```
echo1 = getLine >>= putStrLn

doEcho1 = do
  line <- getLine
  putStrLn line
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> :t echo1
ghci> echo1
abc

ghci> :t doEcho1
ghci> doEcho1
abc
```

6. W pliku `ex2.hs` dadajemy

```
echo2 = getLine >>= \line -> putStrLn $ line ++ "!"

doEcho2 = do
  line <- getLine
  putStrLn $ line ++ "!"
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> :t echo2
ghci> echo2
abc

ghci> :t doEcho2
ghci> doEcho2
abc
```

7. W pliku `ex2.hs` dadajemy

```
echo3 :: IO ()
echo3 =  getLine >>= \l1 -> getLine >>= \l2 -> putStrLn $ l1 ++ l2

dialog :: IO ()
dialog = putStr "What is your happy number? "
         >> getLine
         >>= \n -> let num = read n :: Int in
                   if num == 7
                   then putStrLn "Ah, lucky 7!"
                   else if odd num
                        then putStrLn "Odd number! That's most people's choice..."
                        else putStrLn "Hm, even number? Unusual!"
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie `echo3` i `dialog`

8. ***Zadania***:
   1. Napisać odpowiedniki `echo3` i `dialog` wykorzystujące notację `do`
   2. Napisać odpowiednik `twoQuestions` bez użycia notacji `do`

      ```
      twoQuestions :: IO ()
      twoQuestions = do
        putStr "What is your name? "
        name <- getLine
        putStr "How old are you? "
        age <- getLine
        print (name,age)
      ```

   3. (*opcjonalne*) Napisać 'akcję' `getLine'` odpowiadającą `getLine` z biblioteki `Prelude`

3) 'Akcje' I/O jako parametry lub wyniki funkcji oraz elementy struktur danych (*ćwiczenie opcjonalne*)

1. W pliku `ex3.hs` wpisujemy

```
nTimes :: Int -> IO () -> IO ()
nTimes 0 action = return ()
nTimes n action = do
  action
  nTimes (n-1) action

ioActionFactory :: Int -> String -> IO ()
ioActionFactory n = case n of
  1 -> \name -> putStrLn ("Good morning, " ++ name)
  2 -> \name -> putStrLn ("Good afternoon, " ++ name)
  3 -> \name -> putStrLn ("Good night, " ++ name)
  _ -> \name -> putStrLn ("Hello, " ++ name)

actionList :: [IO ()]
actionList = [ioActionFactory 1 "Ben",
              ioActionFactory 2 "Joe",
              ioActionFactory 3 "Ally"]

sequence'        :: [IO ()] -> IO ()
sequence' []     =  return ()
sequence' (a:as) =  do a
                       sequence' as
```

(zapisujemy zmiany, wczytujemy plik do GHCi)

2. W GHCi wpisujemy

```
ghci> nTimes 3 (putStrLn "Hathor")
ghci> sequence' [putStrLn "Hathor", putStrLn "Hathor", putStrLn "Hathor"]

ghci> sequence' actionList

ghci> :t sequence_
ghci> sequence_ actionList

ghci> :t sequence -- porównujemy wynik z poprzednim
ghci> sequence actionList

ghci> :t sequenceA  -- porównujemy wynik z poprzednimi
ghci> sequenceA actionList
```

3. W GHCi wpisujemy

```
ghci> foldr1 (>>) actionList
ghci> foldl1 (>>) actionList
ghci> foldr (>>) (return ()) actionList
```

4. ***Zadania***:
    1. Napisać odpowiednik `sequence'` wykorzystujący `foldr`
    2. Zmienić postać 1. agrumentu `foldr` : z `>>` na wyrażenie lambda
    3. Napisać odpowiednik `sequence'` wykonujący ‘akcje’ od ostaniej do pierwszej; rozważyć co najmniej dwa warianty, np. `foldr` na odwróconej liście i wykorzystanie `foldl`

4) Elementarne operacje na plikach i obsługa wyjątków I/O (*ćwiczenie opcjonalne*)

1. W pliku `ex4.hs` wpisujemy

```
import System.Environment
import System.IO

main = do
  (inFileName:outFileName:_) <- getArgs
  inHdlr <- openFile inFileName ReadMode
  outHdlr <- openFile outFileName WriteMode
  inpStr <- hGetContents inHdlr
  hPutStr outHdlr inpStr
  hClose inHdlr
  hClose outHdlr
```

   zapisujemy zmiany i sprawdzamy działanie (`runghc + cat`)

2. W terminalu/konsoli (nie GHCi) wpisujemy

```
$ runghc ex4.hs ex4.hs tmp.hs
$ cat tmp.hs
```

3. Modyfikujemy plik `ex4.hs` (zastępujemy poprzednią zawartość)

```
import System.Environment

main = do
  (inFileName:outFileName:_) <- getArgs
  inpStr <- readFile inFileName
  writeFile outFileName inpStr
```

   zapisujemy zmiany i sprawdzamy działanie (jw. `runghc + cat`)

4. Modyfikujemy plik `ex4.hs` (zastępujemy poprzednią zawartość)

```
import System.Environment
import qualified Data.ByteString as BStr

main = do
  (inFileName:outFileName:_) <- getArgs
  inpBStr <- BStr.readFile inFileName
  BStr.writeFile outFileName inpBStr
```

zapisujemy zmiany i sprawdzamy działanie (jw. `runghc + cat`)

5. W konsoli GHCi wpisujemy

```
ghci> import Control.Exception
ghci> :i Exception
ghci> :i SomeException
ghci> :t try
ghci> :t throw
ghci> :t catch
ghci> :t handle
```

6. W konsoli GHCi wpisujemy

```
ghci> data MyException = MkMyException deriving Show
ghci> instance Exception MyException
ghci> throw MkMyException  `catch` \e -> putStrLn ("Caught " ++ show (e :: MyExcepti
```

7. Modyfikujemy plik `ex4.hs` (zastępujemy poprzednią zawartość)

```
import System.Environment
import System.IO.Error
import Control.Exception

riskyAction :: IO ()
riskyAction = do (fileName:_) <- getArgs
                 contents <- readFile fileName
                 putStrLn contents

exHdlr :: IOError -> IO ()
exHdlr = \ex -> if isDoesNotExistError ex
                then putStrLn "The file doesn't exist!"
                else ioError ex

main :: IO ()
main = do
  result <- try riskyAction
  case result of
```

generated by haroopad

```
    Left ex -> exHdlr ex
    Right _ -> putStrLn "Operation completed"
```

zapisujemy zmiany i sprawdzamy działanie, np. w konsoli/terminalu

```
$ runghc ex4.hs ex4.hs
$ runghc ex4.hs non-existent.hs
```

8. Modyfikujemy funkcję `main`

```
main = catch riskyAction exHdlr
```

zapisujemy zmiany i sprawdzamy jej działanie (np. jw.)

9. Modyfikujemy funkcję `main`

```
main = handle exHdlr riskyAction
```

zapisujemy zmiany i sprawdzamy jej działanie (np. jw.)

10. Modyfikujemy funkcję `main`

```
main = riskyAction `catch` exHdlr
```

zapisujemy zmiany i sprawdzamy jej działanie (np. jw.)

11. ***Zadania***:
    1. Napisać program obliczający dla podanego pliku następujące wskaźniki:
       - liczbę linii,
       - liczbę wyrazów,
       - liczbę znaków,
       - liczbę różnych wyrazów
       - liczbę linii o długości większej niż 80 znaków
    2. Dodać do programu obsługę błędów (co najmniej: nieprawidłowa liczba argumentów wywołania, nieistniejący plik wejściowy)
    3. Napisać program obliczający liczbę wystąpień zadanego słowa w pliku (dodać obsługę błędów)
    4. Napisać program łączący zawartość plików podanych w argumentach wywołania (dodać obsługę błędów)
    5. Zmodyfikować funkcję `exHdlr` : zamianst predykatu `isDoesNotExistError` (i wyrażenia warunkowego) użyć `IOErrorType` i mechanizmu dopasowania wzorców
    6. Przeanalizować zawartość `Control.Exception` i `System.IO.Error`

5) Funktory 1: operatory `fmap` , `(<$>)` i `(<$)`

1. W konsoli GHCi wpisujemy

```
ghci> :i Functor
ghci> :t fmap (+1)

ghci> :i Either
```

```
ghci> fmap (+2) (Left 3)
ghci> fmap (+2) (Right 3)

ghci> :i []
ghci> fmap (*2) [1..5]
ghci> fmap (*2) []

ghci> :i Maybe
ghci> fmap (+1) (Just 3)
ghci> fmap (+1) Nothing

ghci> :i IO
ghci> import Data.Char
ghci> fmap toUpper getChar
a
ghci> fmap (map toUpper) getLine
abcde

ghci> :t fmap (+1) (*10)
ghci> fmap (+1) (*10) 1

ghci> fmap (+1) (0,0)
ghci> fmap (+1) (0,0,0)
```

2. W konsoli GHCi wpisujemy

```
ghci> :t ($)
ghci> :i ($)
ghci> (+2) $ 3

ghci> :t (<$>)
ghci> :i (<$>)

ghci> (+2) $ (Right 3)
ghci> (+2) <$> (Right 3)

ghci> (*2) <$> [1..5]

ghci> (+1) <$> (Just 3)

ghci> toUpper <$> getChar

ghci> (map toUpper) <$> getLine
abcde

ghci> (+1) <$> (*10) $ 1

ghci> (+1) <$> (0,0)
```

generated by haroopad

3. W konsoli GHCi wpisujemy

```
ghci> :i Functor
ghci> :t (<$)

ghci> 1 <$ Left 2
ghci> 1 <$ Right 2

ghci> 'a' <$ [1..5]
ghci> 'a' <$ []

ghci> 'a' <$ Just 1
ghci> 'a' <$ Nothing

ghci> 42 <$ getLine
abcd

ghci> 1 <$ (*10) $ 5
ghci> :t  1 <$ (*10)

ghci> 1 <$ (0,0)
ghci> 1 <$ (0,0,0)
```

W jakich sytuacjach może być przydatny operator `(<$)` ?

6) Funktory 2: dołączanie typów użytkownika do klasy `Functor`

1. W pliku `ex6.hs` wpisujemy

```
newtype Box a = MkBox a deriving Show

instance Functor Box where
  fmap f (MkBox x) = MkBox (f x)
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> :i Box
ghci> :i MkBox

ghci> fmap (^2) (MkBox 3)
ghci> 1 <$ MkBox 3
```

2. Modyfikujemy definicję `Box a`

```
newtype Box a = MkBox a deriving (Show, Functor)
```

3. Zapisujemy zmiany, wczytujemy plik do GHCi (`:r`) i analizujemy komunikat błędu
4. W pierwszej linii pliku `ex6.hs` dodajemy rozszerzenie

```
{-# LANGUAGE DeriveFunctor #-}
```

5. Zapisujemy zmiany, wczytujemy plik do GHCi (`:r`) i analizujemy komunikat błędu
6. Usuwamy (lub 'zakomentowujemy') definicję

```
instance Functor Box where
  fmap f (MkBox x) = MkBox (f x)
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie `fmap` , np.

```
ghci> fmap (^2) (MkBox 3)
ghci> 1 <$ MkBox 3
```

7. W pliku `ex6.hs` dodajemy

```
data MyList a = EmptyList
              | Cons a (MyList a) deriving Show

instance Functor MyList where
  fmap _ EmptyList     = EmptyList
  fmap f (Cons x mxs) = Cons (f x) (fmap f mxs)
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie `fmap` , np.

```
ghci> fmap (*2) EmptyList
ghci> let lst1 = Cons 1 (Cons 2 (Cons 3 (Cons 4 EmptyList)))
ghci> fmap id lst1
ghci> fmap (const 1) lst1
ghci> fmap (*2) lst1
ghci> fmap odd lst1
```

8. ***Zadania***:
    1. Sprawdzić możliwość automatycznego wygenerowania instancji `Functor` dla typu `MyList` (klauzula *deriving*)
    2. Napisać własną implementację funktora (`instance Functor`), a następnie sprawdzić możliwość jej automatycznego wygenerowania dla drzewa binarnego zdefiniowanego jako

    ```
    data BinTree a = EmptyBT | NodeBT a (BinTree a) (BinTree a) deriving (Show)
    ```

    3. (*opcjonalne*) Napisać implementacje funktora (`instance Functor`) dla następujących typów:

    ```
    newtype Pair b a = Pair { getPair :: (a,b) } -- fmap should change the first el
    data Tree2 a = EmptyT2 | Leaf a | Node (Tree2 a) a (Tree2 a) deriving Show
    data GTree a = Leaf a | GNode [GTree a] deriving Show
    ```

generated by haroopad

4. (*opcjonalne*) Napisać implementację funktora (`instance Functor`) dla funkcji `a -> b`

7) Funktory aplikatywne 1: operatory `pure` , `(<*>)` , `(*>)` i `(<*)`

1. W konsoli GHCi wpisujemy

```
ghci> :i Applicative

ghci> fmap (+1) (Just 1)
ghci> (+1) <$> (Just 1)
ghci> (+) <$> (Just 1) (Just 2) -- analizujemy opis błędu
ghci> :t (+) <$> (Just 1) -- w czym tkwi problem?

ghci> (+) <$> (Just 1) <*> (Just 2)
ghci> pure (+) <*> (Just 1) <*> (Just 2)
ghci> :t pure (+) <*> (Just 1)

ghci> (\x y z -> x + y + z) <$> Just 1 <*> Just 2 <*> Just 3
ghci> pure (\x y z -> x + y + z) <*> Just 1 <*> Just 2 <*> Just 3
```

2. W konsoli GHCi wpisujemy

```
ghci> :i Applicative
ghci> :t pure

ghci> pure 1 :: Either Int Int
ghci> pure 1 :: Either a Int
ghci> pure 1 :: Either a Double

ghci> pure 1 :: [Int]
ghci> pure 1 :: [Double]

ghci> pure 1 :: Maybe Int
ghci> pure 1 :: IO Int

ghci> pure 1 :: (->) r Int

ghci> pure 1 :: ((,) a Int)
ghci> pure 1 :: Monoid a => ((,) a Int)
```

3. W konsoli GHCi wpisujemy

```
ghci> :t (<*>)

ghci> :i Either
ghci> pure (+1) <*> Left 0
ghci> pure (+1) <*> Right 0
ghci> Left (+1) <*> Left 0
ghci> Left (+1) <*> Right 0
```

generated by haroopad

```
ghci> Right (+1) <*> Right 0
ghci> :t pure (+1) <*> Left 0
ghci> :t pure (+1) <*> Right 0
ghci> :t Left (+1) <*> Left 0
ghci> :t Left (+1) <*> Right 0
ghci> :t Right (+1) <*> Right 0

ghci> :i []
ghci> pure (*2) <*> [1..5]
ghci> :t pure (*2)
ghci> :t pure (*2) :: [Int->Int]
ghci> [(+1), (*2)] <*> [1,2,3]
ghci> (*) <$> [1,2,3] <*> [100,101,102]

ghci> import Control.Applicative
ghci> :i ZipList
ghci> pure (+) <*> ZipList [1,2,3] <*> ZipList [100,100,100]
ghci> (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]

ghci> let timesList = fmap (*) [1..5]
ghci> :t timesList
ghci> fmap (\f -> f 3) timesList
ghci> (\f -> f 3) <$> timesList
ghci> ($ 3) <$> timesList
ghci> (:) <*> (\x -> [x]) $ 2

ghci> (++) <$> Just "Abra" <*> Just "kadabra"
ghci> (++) <$> Just "Abra" <*> Nothing
ghci> (++) <$> Nothing <*> Just "Abra"
ghci> pure (\x y z -> (x,y,z)) <*> Just 1 <*> Just 2 <*> Just 3
ghci> (\x y z -> (x,y,z)) <$> Just 1 <*> Just 2 <*> Just 3

ghci> (++) <$> getLine <*> getLine
abc
def
ghci> (++) <$> (fmap reverse getLine) <*> getLine
abc
def
ghci> :t getLine
ghci> :t fmap reverse getLine
ghci> (+) <$> (fmap read) getLine <*> (fmap read) getLine

ghci> (+) <$> (+1) <*> (*100) $ 5
ghci> (+) <$> (^2) <*> (^3) $ 3
```

4. W konsoli GHCi wpisujemy

```
ghci> Left 1 *> Left 2
ghci> Right 1 *> Right 2
```

```
ghci> [1..2] *> [11..15]

ghci> Just 1 *> Just 2
ghci> Nothing *> Just 2

ghci> getLine *> getLine
abc
def

ghci> (+1) *> (*100) $ 5
```

W jakich sytuacjach może być przydatny operator `(*>)` ?

5. W konsoli GHCi wpisujemy

```
ghci> Left 1 <* Left 2
ghci> Right 1 <* Right 2

ghci> [1..2] <* [11..15]

ghci> Just 1 <* Just 2
ghci> Just 1 <* Just 2
ghci> Just 2 <* Nothing

ghci> getLine <* getLine
abc
def

ghci> (+1) <* (*100) $ 5
```

W jakich sytuacjach może być przydatny operator `(<*)` ?

8) Funktory aplikatywne 2: dołączanie typów użytkownika do klasy `Applicative`

1. W pliku `ex7.hs` wpisujemy

```
newtype Box a = MkBox a deriving Show

instance Applicative Box where
  pure = MkBox
  (MkBox f) <*> w = fmap f w
```

2. Zapisujemy zmiany, wczytujemy plik do GHCi (`:r`) i analizujemy komunikat błędu
3. W pliku `ex7.hs` dodajemy

```
instance Functor Box where
  fmap f (MkBox x) = MkBox (f x)
```

generated by haroopad

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi wpisujemy

```
ghci> pure (*2) <*> MkBox 3
ghci> (*2) <$> MkBox 3
ghci> (+) <$> MkBox 1 <*> MkBox 2
ghci> (++) <$> MkBox "abc" <*> MkBox "def"
ghci> (\x y z -> (z,y,x)) <$> MkBox (Just 1) <*> MkBox (Just 2) <*> MkBox (Just 3)
```

4. ***Zadania***:
   1. Napisać implementacje funktora aplikatywnego (`instance Applicative`) dla typu

      ```
      newtype MyTriple a = MyTriple (a,a,a) deriving Show
      ```

   2. (*opcjonalne*) Napisać implementację funktora aplikatywnego (`instance Applicative`) dla funkcji `a -> b`
   3. (*opcjonalne*) Napisać implementacje funktora aplikatywnego (`instance Applicative`) dla typu

      ```
      data Tree2 a = EmptyT2 | Leaf a | Node (Tree2 a) a (Tree2 a) deriving Show
      ```

      Uwaga: rozważyć dwa warianty — jak w przypadku list: iloczyn kartezjański i ZipList
   4. (*opcjonalne*) Przeanalizować działanie funkcji `liftA` , `liftA2` i `liftA3` z `Control.Applicative`

9) `Monoid` , `Foldable` , `Traversable` (*ćwiczenie opcjonalne*)

1. W konsoli GHCi wpisujemy

```
ghci> :i Monoid

ghci> mempty

ghci> mempty :: [a]
ghci> [1,2,3] `mappend` [4,5,6]

ghci> :i Ordering
ghci> mempty :: Ordering
ghci> EQ `mappend` EQ
ghci> LT `mappend` LT
ghci> LT `mappend` EQ
ghci> LT `mappend` GT
ghci> EQ `mappend` LT
ghci> EQ `mappend` GT
ghci> GT `mappend` LT
ghci> GT `mappend` EQ
ghci> GT `mappend` GT

ghci> mempty :: (Monoid a => Maybe a)
```

generated by haroopad

```
ghci> Nothing `mappend` Nothing
ghci> Nothing `mappend` Just "Haskell"
ghci> Just "Haskell" `mappend` Nothing
ghci> Just "Me" `mappend` Just " and Haskell"

ghci> mempty :: IO String
ghci> getLine `mappend` getLine
```

2. W konsoli GHCi wpisujemy

```
ghci> :i Foldable
```

3. W pliku `ex8.hs` wpisujemy

```
data Tree2 a = EmptyT2 | Leaf a | NodeT2 (Tree2 a) a (Tree2 a) deriving Show

instance Foldable Tree2 where
  foldMap f EmptyT2        = mempty
  foldMap f (Leaf x)       = f x
  foldMap f (NodeT2 l k r) = foldMap f l `mappend` f k `mappend` foldMap f r
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie `foldMap` dla kilku drzew

4. W pliku `ex8.hs` dodajemy

```
instance Functor Tree2 where
  fmap f EmptyT2        = EmptyT2
  fmap f (Leaf x)       = Leaf $ f x
  fmap f (NodeT2 l x r) = NodeT2 (fmap f l)
                                 (f x)
                                 (fmap f r)

instance Traversable Tree2 where
  traverse f EmptyT2        = pure EmptyT2
  traverse f (Leaf x)       = Leaf <$> f x
  traverse f (NodeT2 l x r) = NodeT2 <$> traverse f l
                                     <*> f x
                                     <*> traverse f r
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie `traverse` dla kilku drzew