ASSIGNMENT - 1

NAME : J.SAMPAT SRIVATSAV

ROLL NUMBER : AM.EN.U4CSE19125

BATCH : C.S.E – B

Data.List

```
adhinene@sampat: ~/ppl/assignment
Prelude> import Data.List
Prelude Data.List> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
Prelude Data.List> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
Prelude Data.List> intercalculate " " ["hey","there","guys"]

<interactive>:5:1: error:
    • Variable not in scope: intercalculate :: [Char] -> [[Char]] -> t
    • Perhaps you meant ‘intercalate’ (imported from Data.List)
Prelude Data.List> intercalate " " ["hey","there","guys"]
"hey there guys"
Prelude Data.List> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
Prelude Data.List> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
Prelude Data.List> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
Prelude Data.List> concat ["foo","bar","car"]
"foobarcar"
Prelude Data.List> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
Prelude Data.List> and $ map (>4) [5,6,7,8]
True
```

adhinene@sampat: ~/ppl/assignment

```
Prelude Data.List> and $ map (==4) [4,4,4,3,4]
False
Prelude Data.List> or $ map (==4) [2,3,4,5,6,1]
True
Prelude Data.List> or $ map (>4) [1,2,3]
False
Prelude Data.List> any (==4) [2,3,5,6,1,4]
True
Prelude Data.List> all (>4) [6,9,10]
True
Prelude Data.List> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
Prelude Data.List> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
Prelude Data.List> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
Prelude Data.List> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
Prelude Data.List> splitAt 3 "heyman"
("hey","man")
Prelude Data.List> splitAt 100 "heyman"
("heyman","")
```

adhinene@sampat: ~/ppl/assignment

```
Prelude Data.List> splitAt (-3) "heyman"
("","heyman")
Prelude Data.List> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
Prelude Data.List> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
Prelude Data.List> takeWhile (/=' ') "This is a sentence"
"This"
Prelude Data.List> dropWhile (/=' ') "This is a sentence"
" is a sentence"
Prelude Data.List> dropwhile (<3) [1,2,2,2,3,4,5,4,3,2,1]

<interactive>:30:1: error:
    • Variable not in scope:
        dropwhile :: (Integer -> Bool) -> [Integer] -> t
    • Perhaps you meant ‘dropWhile’ (imported from Prelude)
Prelude Data.List> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
Prelude Data.List> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
Prelude Data.List> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
Prelude Data.List> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

adhinene@sampat: ~/ppl/assignment

```
Prelude Data.List> inits "w00t"
["","w","w0","w00","w00t"]
Prelude Data.List> tails "w00t"
["w00t","00t","0t","t",""]
Prelude Data.List> let w="w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
Prelude Data.List> "cat" `IsInfixOf` "im a cat burglar"

<interactive>:38:7: error:
    • Data constructor not in scope: IsInfixOf :: [Char] -> [Char] -> t
    • Perhaps you meant variable ‘isInfixOf’ (imported from Data.List)
Prelude Data.List> "cat" `isInfixOf` "im a cat burglar"
True
Prelude Data.List> "Cat" `isInfixOf` "im a cat burglar"
False
Prelude Data.List> "cats" `isInfixOf` "im a cat burglar"
False
Prelude Data.List> "hey" `isPrefixOf` "hey there!"
True
Prelude Data.List> "hey" `isPrefixOf` "oh hey there!"
False
Prelude Data.List> "there!" `isSuffixOf` "oh hey there!"
True
```

adhinene@sampat: ~/ppl/assignment

```
Prelude Data.List> "there!" `isSuffixOf` "oh hey there!"
True
Prelude Data.List> "there!" `isSuffixOf` "oh hey there"
False
Prelude Data.List> find (>4) [1,2,3,4,5,6]
Just 5
Prelude Data.List> find (>9) [1,2,3,4,5,6]
Nothing
Prelude Data.List> :t find
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
Prelude Data.List> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
Prelude Data.List> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
Prelude Data.List> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
Prelude Data.List> findIndex (==4) [5,3,2,1,6,4]
Just 5
Prelude Data.List> findIndex (==7) [5,3,2,1,6,4]
Nothing
Prelude Data.List> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
Prelude Data.List> delete 'h' "hey there ghang!"
"ey there ghang!"
```

```
adhinene@sampat: ~/ppl/assignment
Prelude Data.List> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
Prelude Data.List> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
Prelude Data.List> "Im a big baby" \\ "big"
"Im a  baby"
Prelude Data.List> "hey man" `union` "man what's up"
"hey manwt'sup"
Prelude Data.List> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude Data.List> [1..7] `intersect` [5..10]
[5,6,7]
Prelude Data.List> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
Prelude Data.List> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

Data.Char

```
adhinene@sampat: ~/ppl/assignment
Prelude Data.List Data.Char> all isAlphaNum "bobby283"
True
Prelude Data.List Data.Char> all isAlphaNum "eddy the fish!"
False
Prelude Data.List Data.Char> generalCategory 'A'
UppercaseLetter
Prelude Data.List Data.Char> generalCategory 'a'
LowercaseLetter
Prelude Data.List Data.Char> generalCategory '.'
OtherPunctuation
Prelude Data.List Data.Char> generalCategory '9'
DecimalNumber
Prelude Data.List Data.Char> words "hey guys its me"
["hey","guys","its","me"]
```

```
adhinene@sampat: ~/ppl/assignment
Prelude Data.List Data.Char> ord 'a'
97
Prelude Data.List Data.Char> chr 97
'a'
Prelude Data.List Data.Char> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
Prelude Data.List Data.Char> map digitToInt "34538"
[3,4,5,3,8]
Prelude Data.List Data.Char> map digitToInt "FF85AB"
[15,15,8,5,10,11]
Prelude Data.List Data.Char> intToDigit 15
'f'
Prelude Data.List Data.Char> intToDigit 15
'f'
Prelude Data.List Data.Char> intToDigit 5
'5'
```

Example 1

```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2-x1) * (abs $ y2-y1)
```

```
Prelude Data.List Data.Char> :load ex1.hs
[1 of 1] Compiling Main             ( ex1.hs, interpreted )
Ok, one module loaded.
*Main Data.List Data.Char> :t Circle
Circle :: Float -> Float -> Float -> Shape
*Main Data.List Data.Char> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
*Main Data.List Data.Char> surface $ Circle 10 20 10
314.15927
*Main Data.List Data.Char> surface $ Circle 0 0 100 100

<interactive>:96:11: error:
    • Couldn't match expected type ‘Integer -> Shape’
                  with actual type ‘Shape’
    • The function ‘Circle’ is applied to four arguments,
      but its type ‘Float -> Float -> Float -> Shape’ has only three
      In the second argument of ‘($)’, namely ‘Circle 0 0 100 100’
      In the expression: surface $ Circle 0 0 100 100
*Main Data.List Data.Char> surface $ Rectangle 0 0 100 100
10000.0
*Main Data.List Data.Char>
```

Example 2

```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2-x1) * (abs $ y2-y1)
```

```
adhinene@sampat: ~/ppl/assignment
*Main Data.List Data.Char> Circle 10 20 5
Circle 10.0 20.0 5.0
*Main Data.List Data.Char> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
*Main Data.List Data.Char>
```

Example 3

```haskell
data Person = Person String String Int Float String String deriving (Show)
```

```
adhinene@sampat: ~/ppl/assignment
Prelude Data.List Data.Char> :load ex3.hs
[1 of 1] Compiling Main             ( ex3.hs, interpreted )
Ok, one module loaded.
*Main Data.List Data.Char> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-
2928" "Chocolate"
*Main Data.List Data.Char> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
*Main Data.List Data.Char>
```

Example 4

```haskell
data Person = Person String String Int Float String String deriving (Show)

firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

```
adhinene@sampat: ~/ppl/assignment
*Main Data.List Data.Char> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-
2928" "Chocolate"
*Main Data.List Data.Char> firstName guy
"Buddy"
*Main Data.List Data.Char> height guy
184.2
*Main Data.List Data.Char> flavor guy
"Chocolate"
*Main Data.List Data.Char>
```

Example 5

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

```
adhinene@sampat: ~/ppl/assignment
*Main Data.List Data.Char> :t flavor
flavor :: Person -> String
*Main Data.List Data.Char> :t firstName
firstName :: Person -> String
*Main Data.List Data.Char>
```

Example 6

```
adhinene@sampat: ~/ppl/assignment
Prelude Data.List Data.Char> data Car = Car {company :: String, model :: String,
 year :: Int} deriving (Show)
Prelude Data.List Data.Char> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
Prelude Data.List Data.Char>
```