MIDDLE EAST TECHNICAL UNIVERSITY, NORTHERN CYPRUS CAMPUS
CNG242 Programming Language Concepts – Spring 2021 – Lab 4: Haskell

1. New Type Definition (Disjoint Union)

```
data Decision = Yes | No | Maybe
data Weekday = Monday | Tuesday | Wednesday | Thursday | Friday
data SpringDate = March Int | April Int | May Int deriving (Show, Eq, Ord)
data SchoolMember = Student [Char] [Char] Int | Teacher [Char] [Char] Int deriving
(Show, Eq, Ord)
```

2. Typeclasses

Deriving members of **Show** class can be presented as strings.

```
Prelude> show 242
"242"
Prelude> show [2,4,2]
"[2,4,2]"
```

Example (Without Show)

```
Prelude> data Decision = Yes | No | Maybe
Prelude> :t Yes
Yes :: Decision
Prelude> Yes

<interactive>:18:1: error:
    * No instance for (Show Decision) arising from a use of `print'
      …
```

Example (With Show)

```
Prelude> data Decision = Yes | No | Maybe deriving Show
Prelude> :t Yes
Yes :: Decision
Prelude> Yes
Yes
```

Eq is used to support equality check between deriving types. Types that derive **Eq** can implement == and /= operators for equality testing.

Example (Without Eq)

```
Prelude> data Decision = Yes | No | Maybe
Prelude> Yes == No

<interactive>:28:1: error:
    * No instance for (Eq Decision) arising from a use of `=='
      …
```

Example (With Eq)
Prelude> data Decision = Yes | No | Maybe deriving Eq
Prelude> Yes == No
False
Prelude> Yes == Yes
True
Prelude> Yes /= No
True

Ord is used to check ordering between deriving types. Types that derive **Ord** can implement >, <, >=, <= and compare function. Only the members of **Eq** can be the members of **Ord**.

Example (Without Ord)
Prelude> data Decision = Yes | No | Maybe
Prelude> Yes > No
<interactive>:42:1: error:

* No instance for (Ord Decision) arising from a use of `>'
…

Example (With Ord)
Prelude> data Decision = Yes | No | Maybe deriving (Eq, Ord)
Prelude> Yes > No
False

Example 2 (With Ord)
Prelude> data Decision = No | Yes | Maybe deriving (Eq, Ord)
Prelude> Yes > No
True

Read typeclass takes a string and returns a type that is a deriving member of itself. It works like the opposite of Show typeclass.
Prelude> read "200" + 42
242
Prelude> read "[2,4]" ++ [2]
[2,4,2]

Example (Without Read)
Prelude> data Decision = Yes | No | Maybe deriving Eq
Prelude> read "Yes" == Yes

<interactive>:53:1: error:
* No instance for (Read Decision) arising from a use of `read'
…

Example (With Read)
Prelude> data Decision = Yes | No | Maybe deriving (Eq, Read)
Prelude> read "Yes" == Yes
True

3. Usage of Disjoint Union Types

```
--------------------disjointExample.hs--------------------
data Decision = Yes | No | Maybe deriving (Show, Eq, Ord)
data SchoolMember = Student [Char] [Char] Int | Teacher [Char] [Char] [Char] | TA
[Char] [Char] deriving (Show, Eq, Ord)

askQuestion x = if x == "Are you taking CNG 242?"  then Yes
                else if x== "Do you want to fail CNG 242?" then No
                else Maybe

react Yes = "Nice!"
react No = "Why?"
react Maybe = "To what?"

accept Yes = "OK!"
accept _ = "You should accept!"

toString (Student name surname id) = "I am a student in METU NCC, my name is " ++
name ++ ", my surname is " ++ surname ++ ", and my student ID is " ++ (show id)
toString (Teacher name surname department) = "My name is " ++ name ++ " " ++
surname ++ ". I am the instructor of CNG 242 in " ++ department ++ " department"
toString (TA name message) = "You have a message from your TA, " ++ name ++ ": "
++ message
------------------------------------------------------------------
```

Sample Run:

```
[1 of 1] Compiling Main (…\disjointExample.hs, interpreted)
Ok, one module loaded.
*Main> askQuestion "Are you ok?"
Maybe
*Main> askQuestion "Are you taking CNG 242?"
Yes
*Main> react No
"Why?"
*Main> react Maybe
"To what?"
*Main> accept No
"You should accept!"
*Main> accept Maybe
"You should accept!"
*Main> :t accept
accept :: Decision -> [Char]
*Main> :react
*Main> :t askQuestion
askQuestion :: [Char] -> Decision
*Main> toString (Teacher "Enver" "Ever" "Computer Engineering")
"My name is Enver Ever. I am the instructor of CNG 242 in Computer Engineering
department"
*Main> toString (TA "Zekican" "Hello World!")
"You have a message from your TA, Zekican: Hello World!"
```

4. Polymorphic Data Types

Example (Non-polymorphic Disjoin Union)
data Distance = Kilometres Float | Miles Float deriving Show ⟵ Float only

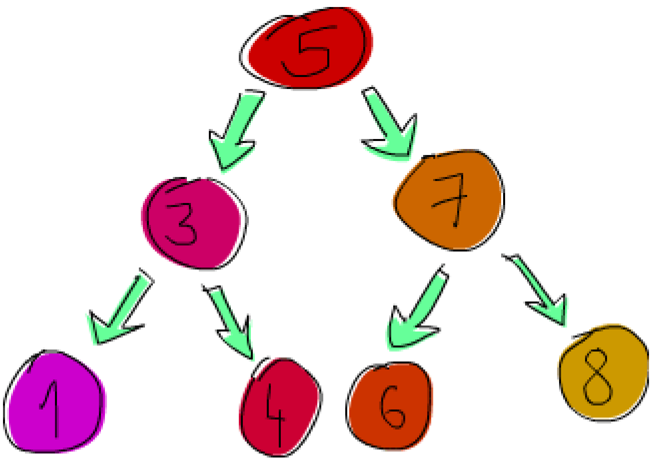
```
Prelude> Kilometres 5.5
Kilometres 5.5
Prelude> Miles 8
Miles 8.0
Prelude> Kilometres "12"

<interactive>:45:12: error:
   * Couldn't match expected type `Float' with actual type `[Char]'
     ...
```

Example (Polymorphic Disjoint Union)
data Distance a = Kilometres a | Miles a deriving Show

```
Prelude> Kilometres 5.5
Kilometres 5.5
Prelude> Miles 8
Miles 8
Prelude> Kilometres "12"
Kilometres "12"
Prelude> Miles "13.5"
Miles "13.5"
```

5. Recursive Data Types with Tree Example

data Tree = EmptyTree | Node Integer Tree Tree deriving (Show, Eq, Ord)



[1]

- insertElement function inserts a new element to the given binary tree.

[1] http://learnyouahaskell.com/making-our-own-types-and-typeclasses

```
---------------------------------------------------Tree.hs---------------------------------------------------
data Tree = EmptyTree | Node Integer Tree Tree deriving (Show, Eq, Ord)
insertElement x EmptyTree = Node x EmptyTree EmptyTree      -- BASE CASE
insertElement x (Node a left right) = if x == a             -- DO NOTHING
                                      then (Node x left right)
                                      else if x < a         -- INSERT TO LEFT
                                      then (Node a (insertElement x left) right)
                                      else                  -- INSERT TO RIGHT
                                      Node a left (insertElement x right)
---------------------------------------------------------------------------------------------------------------
[1 of 1] Compiling Main (…\Tree.hs, interpreted)
Ok, one module loaded.
*Main> insertElement 5 EmptyTree
Node 5 EmptyTree EmptyTree
*Main> x = insertElement 5 EmptyTree
*Main> x
Node 5 EmptyTree EmptyTree
*Main> y = insertElement 10 x
*Main> y
Node 5 EmptyTree (Node 10 EmptyTree EmptyTree)
*Main> x = insertElement 3 y
*Main> x
Node 5 (Node 3 EmptyTree EmptyTree) (Node 10 EmptyTree EmptyTree)
```

6. Lambda Abstractions

```
Prelude> square x = x * x
Prelude> square 2
4
Prelude> (\x -> x * x) 2
4
Prelude> (\x y -> (x + y)/2) 5 7
6.0
Prelude> example q p = (\x y -> ([a | a<-x,a<'o',a/='a'],[ b | b<-y,b>=2])) q p
Prelude> example "congrats" [(-2),2,1,(-3),4,(-4),2,0]
("cng",[2,4,2])
```

Practical Exercises:

1. Write a Haskell function that takes a list and a number and replicate the elements of a list a given number of times [3].

Sample Run:

```
*Main> repli "abc" 3
"aaabbbccc"
*Main> repli "tb" 2
"ttbb"
```

2. Write a Haskell function that takes a list and eliminate consecutive duplicates of list elements. If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed [3].

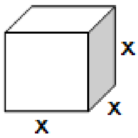
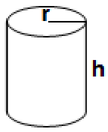
Sample Run:

```
*Main> compress "aaaabccaadeeee"
"abcade"
```

3. Implement first and second exercise using lambda abstraction.

4. Define a new type for `ThreeDShapes`. It should have a data constructor for `Cube` and `Cylinder`. `Cube` should have side length and `Cylinder` should have radius and height. You need to implement two functions related to this type.

- `volume` function calculates the volume of the 3D shape
- `surfaceArea` function calculates the surface area of the 3D shape.

Volume and Surface Area Formulas for Cube and Cylinder		
Cube		Volume = x^3 Surface Area = $6x^2$
Cylinder		Volume = $\pi r^2 h$ Surface Area = $2\ \pi rh + 2\pi r^2$

5. Modify your data type in exercise four so that it works with both integers and floating-point numbers.

For question 6,7,8 and 9 you can use the given Tree example and insertElement function or you can implement your own Tree type and use it.

6. Write a Haskell function which takes a list of numbers and generates a binary tree. Hint: You can use the insertElement function

Sample Run:

```
*Main> inserter [1,2,3]
Node 3 (Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree) EmptyTree
*Main> inserter [5]
Node 5 EmptyTree EmptyTree
*Main> inserter []
EmptyTree
*Main> inserter [1,2,3,4,5]
Node 5 (Node 4 (Node 3 (Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree) EmptyTree) EmptyTree) EmptyTree
*Main> inserter [12,4,2,6,3,5,7,8]
Node 8 (Node 7 (Node 5 (Node 3 (Node 2 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 6 EmptyTree EmptyTree)) EmptyTree) (Node 12 EmptyTree EmptyTree)
```

7. Write a Haskell function which returns the minimum value in the given Tree.

Sample Run:

```
*Main> x = inserter [3,2,4]
*Main> x
Node 4 (Node 2 EmptyTree (Node 3 EmptyTree EmptyTree)) EmptyTree
*Main> minOf x
2
*Main> minOf (Node 5 (Node 3 EmptyTree EmptyTree) EmptyTree
3
```

8. Write a Haskell function that checks if a given Tree is empty or not.

Sample Run:

```
*Main> t = EmptyTree
*Main> isEmpty t
True
*Main> t = Node 3 EmptyTree (Node 7 EmptyTree EmptyTree)
*Main> isEmpty t
False
```

9. Write a Haskell function that searches a given element inside a given Tree. It should return True if the element is found.

Sample Run:

```
*Main> t = Node 3 EmptyTree (Node 7 EmptyTree EmptyTree)
*Main> searchElement 4 t
False
*Main> searchElement 7 t
True
```

References:

1. Learn You a Haskell <http://learnyouahaskell.com/chapters>
2. Types and Typeclasses – Learn You a Haskell <http://learnyouahaskell.com/types-and-typeclasses>
3. A Gentle Introduction to Haskell <http://www.haskell.org/tutorial/index.html>
4. H-99: Ninety-Nine Haskell Problems <https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems>