**Module 3:Declarative Programming Paradigm: Functional Programming**

## 4..Inbuilt Type Class in haskell

Ans In Haskell, every statement is considered as a mathematical expression and the category of this expression is called as a Type. You can say that "Type" is the data type of the expression used at compile time.

To learn more about the Type, we will use the ":t" command. In a generic way, Type can be considered as a value, whereas Type Class can be the considered as a set of similar kind of Types.

5.Define List and Tuple in Haskell.

List Types

A list is sequence of values of the same type:

[False,True,False] :: [Bool]

[’a’,’b’,’c’,’d’] :: [Char]

In general:

[T] is the type of lists with elements of type T.

The type of a list says nothing about its length:

The type of the elements is unrestricted. For example, we can have lists of lists:

[[’a’],[’b’,’c’]] :: [[Char]]

Tuple Types

A tuple is a sequence of values of different types:

(False,True) :: (Bool,Bool)

(False,’a’,True) :: (Bool,Char,Bool)

6.Problem solving with a list.

7.Define functions in Haskell. Give example

A function is a mapping from values of one type to values of another type:

The argument and result types are unrestricted. For example, functions with multiple

arguments or results are possible using lists or tuples:

add :: (Int,Int) → Int

add (x,y) = x+y

zeroto :: Int → [Int]

zeroto n = [0..n]

8.Generate list based on the generator function provided. (list comprehension)

9.Write haskell modules ( refer to examples take in class including recursion, conditional (if else), )

10.What is pattern matching in Haskell, explain with examples?

Many functions have a particularly clear definition using pattern matching on their arguments.

not :: Bool → Bool

not False = True

not True = False

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example

(&&) :: Bool → Bool → Bool

True && True = True

True && False = False

False && True = False

False && False = False

can be defined more compactly by

True && True = True

\_ && \_ = False

However, the following definition is more efficient, as it avoids evaluating the second

argument if the first argument is False:

False && \_ = False

True && b = b

The underscore symbol \_ is the wildcard pattern that matches any argument value.

11.What are guards in Haskell, explain with examples

List comprehensions can use guards to restrict the values produced by earlier generators

[x | x ← [1..10], even x]

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

factors :: Int → [Int]

factors n = [x | x ← [1..n]

, n `mod` x == 0]

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a

function that decides if a number is prime:

prime :: Int → Bool

prime n = factors n == [1,n]

Using a guard we can now define a function that returns the list of all primes up to a given limit:

primes :: Int → [Int]

primes n = [x | x ← [1..n], prime x]

> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]

As an alternative to conditionals, functions can also be defined using guarded equations.

abs n | n ≥ 0 = n

| otherwise = -n

As previously, but using guarded equations

Guarded equations can be used to make definitions involving multiple conditions easier to

read:

signum n | n < 0 = -1

| n == 0 = 0

| otherwise = 1

The catch all condition otherwise is defined in the prelude by otherwise = True.