NAMAL COLLEGE

UNIVERSITY of BRADFORD

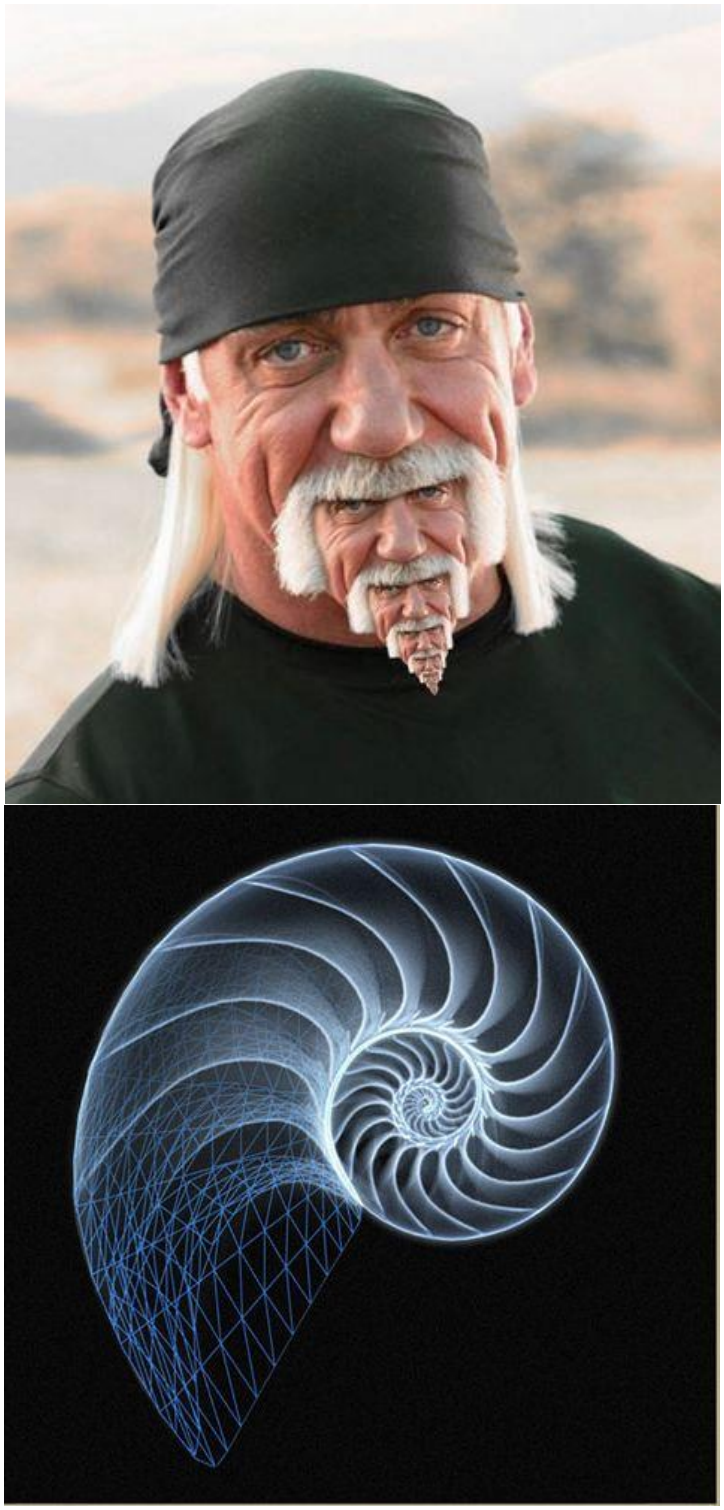
Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

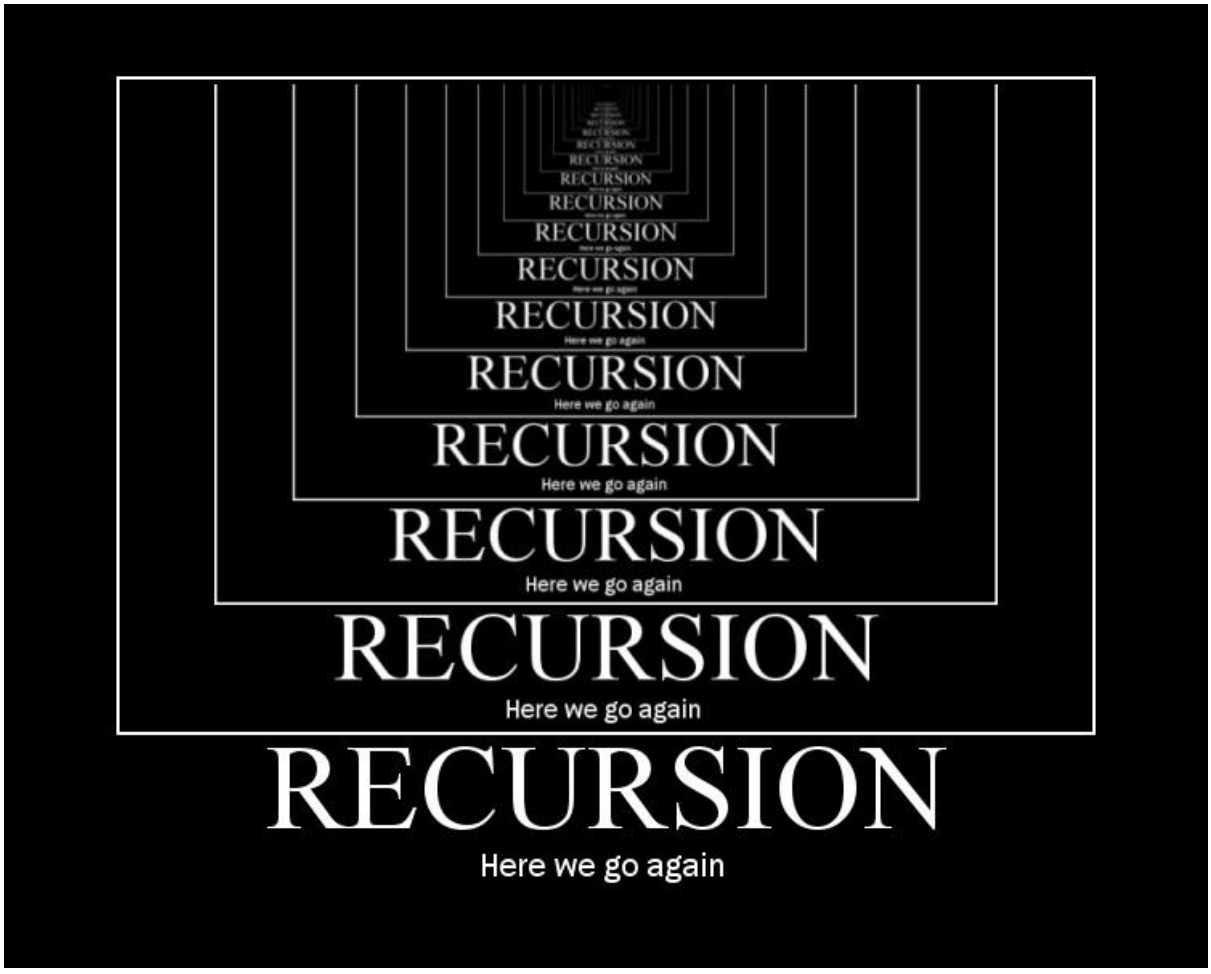
October 26, 2016 Week 5 Lab2

LAB MANUAL: RECURSION

In this tutorial you will learn about

- Recursion in Haskell





Recursion is a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively e.g. Fibonacci, factorial etc.

Let's define a function **`enumFromTo`** which will work exactly same as the following line does

```
[1..5]
```

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n
              | m > n = []
              | m <= n = m : enumFromTo (m+1) n
```

How `enumFromTo` works?

```
enumFromTo 1 3
= 1 : enumFromTo 2 3
= 1 : (2 : enumFromTo 3 3)
= 1 : (2 : (3 : enumFromTo 4 3))
```

```
= 1 : (2 : (3 : []))
= [1,2,3]
```

Exercise

1. Write a function **`enumFrom`** which works as an infinite list e.g `[1..]`.
2. Write down a function **`sumFac`** which will give sum of factorial of all number up to the given input. For example,

```
Hugs> sumFac 5
154
As 154 = factorial 5 + factorial 4 + factorial 3 + factorial 2 + factorial 1 + factorial 0
= 120 + 24 + 6 + 2 + 1 + 1
```

3. Write down definitions of **`remainder`** and **`divide`** functions using minus `(-)` operator. For example, remainder when 37 is divided by 10 will be calculated as :

```
remainder 37 10
remainder 27 10
remainder 17 10
remainder  7 10
```

   10 is subtracted from 37 which results 27, that is again subtracted from 10 until the value is less than 10 to give actual result.

   Similarly, divide is the number of times the subtraction operation is performed.

4. Following is the definition of **`multiplication (*)`** operator using **`addition (+)`** :

```
(*) :: Int -> Int -> Int
m*0 = 0
m*n = m+ (m*(n-1))
```

   By the same way, implement **`power (^)`** operator using **`multiplication (*)`**. Name your operator **`^^^`** to avoid confusion with built in operator.

5. Write down your definitions for following built in functions on Lists
   a. init
   b. replicate
6. Write a function `zip` that takes two lists as input and returns a list of pairs in which each pair contains one member each from the input lists. Its types signature will be as follows

```
zip :: [a] -> [b] -> [(a,b)]
```

7. How your zip function is behaving if the input lists are of the same length?
8. How is it behaving when the input lists are of different lengths?

9. Write the following function in a .hs file

```
pairs xs = zip xs (tail xs)
```

10. What happens when we call the above function e.g. `pairs "words"`?
11. The built in function `zip` take two lists as arguments and produces a list of pairs. Write down a function **`zip3`** that will take three lists and produces a list of triples. For example,

```
Hugs> zip3 [1,2,3,4] ['a','b','c','d'] [True,False,True]
[(1,'a',True),(2,'b',False),(3,'c',True)
```

12. Write a function **`dotProduct`** to compute the scalar product of two vectors.
13. Write a function **`myOR`** that will take a list of Booleans and perform OR operation on the values. For example,

```
Hugs> myOR [True,False,False]
True
```

14. Write a function to **`qsort`** a List in descending order using quick sort algorithm.
15. Write a function to **`msort`** a List in ascending order using merge sort algorithm.
16. Write a function to **`isort`** a List in ascending order using insert sort algorithm.