FUNCTIONAL PROGRAMMING USING HASKELL

-Raghavarapu Venkateswara Rao (IMT2015036)
-Ghantasala Oormila (IMT2015015)
-Alavala Deepika (IMT2015006)

Haskell : Haskell was made by some really smart guys (with PhDs). Work on Haskell began in 1987 when a committee of researchers got together to design the language. The Haskell Report, which defines a stable version of the language, was published in 1999.

Features:

- Haskell is a purely functional programming language (opposed to imperative programming). In Imperative programming, a program follows a sequence of steps, changing the state of the program. Where as in purely functional PL, we dont specify how to do but rather tell what to accomplish.
- No side effects: Cannot assign a variable different values at different parts of the program. So, functions don't have side effects. They only perform some operations and return a result.
- Referential transparency : It is a property of a functional programming language which makes it easier to reason about a program's behaviour and that an expression evaluates to same result in any context.
- Lazy : It won't execute a function and compute things unless it is forced to show the results. This allows us to create infinite data structures.
- Statically typed : Type checking done at compile time. A lot of possible errors are caught at compile time.
- It's type system has a type inference. Programmer need not explicitly name the type of every variable, it is intelligent enough to figure out about it. Type inference also allows the code to be more general, a piece of code can be used for any type it is valid for.
- Elegant and Concise : Programs written in Haskell are usually shorter compared to other imperative equivalents. And shorter programs contain lesser bugs and are easily maintainable.
- GHC (Glassgow Haskell Compiler) : Most widely used Haskell Compiler. To load a file into the GHCi (GHC interface), give
  :l filename. Use 'let' keyword to define variables and functions in GHCi

Functions:

- Infix functions : sandwiching function name with arguments. If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks. Ex: "div 90 10" vs "90 `div` 10"
- Function names must not start with an Uppercase letter. They can contain ' (apos) in them. ' is a valid character for function names. Usually used to denote either a strict version of a function (i.e., one that isn't lazy), or a slightly modified version of afunction or variable with a similar name.
- Function application has the highest precedence of all the operations.
- Ex:
```
doubleSmallNumber' x = if x > 100
                       then x
                       else x*2
```
- When a function doesn't take any parameters, we usually call it a definition or a name.

Lists:

- Homogeneous data structures.
- A string in Haskell is a list of characters.
- Operations on Lists:
  - To concat two lists : **++**
  - Cons Operator: **:** to add an element to front of the list:

  Ex: 2:3:4:[]~[2,3,4]
  - To get an element of a list by index : **!!**

    Ex: [1,2,3] !! 1 = 2
  - Lists within a list can be of different lengths, but they can't be of different types since lists are homogeneous.
  - Lists can be compared if the elements they contain can be compared. <,<=,> and >= can be applied to compare lists. They are compared in lexicographical order i.e, the heads are compared, if they are equal then the second elements and so on.
  - **head** takes a list and returns its head. The head of a list is basically its first element.

    Ex: head [1,2,3,4] = 1
  - **tail** takes a list and returns its tail. In other words, it is a list without its head.

    Ex: tail [1,2,3,4] = [2,3,4]
  - **init** takes a list and returns everything except its last element.

    Ex: init [1,2,3,4] = [1,2,3]
  - **length** returns the length of the list. Ex: length [1,2] = 2.
  - **null** l : checks if the given list is empty. Returns true is empty, else false. It is good to use this instead of saying l == [].
  - **reverse** returns the reverse of a list. Ex: reverse [1,2,3] = [3,2,1]
  - **take** n l : Returns n elements from the begining of the list l.

    Ex: take 3 [1,2,3,4,5] = [1,2,3]
  - **drop** n l : drops 'n' elements from the beginning of the list l.

    Eg : drop 2 [1,2,3,4] = [3,4]
  - **maximum** [1,2,3,4,5] = 5
  - **minimum** [1,2,3,4,5] = 1
  - **sum** [1,2,3,4,5] = 15
  - **product** [1,2,3,4,5] = 120
  - 4 `**elem**` [1,2,3,4,5] = True – checks if 4 is in the given list
- **Ranges :**
  - [1..20] - same as writing all the numbers from 1 to 20.
  - ['a'..'z'] - same as writing all characters from 'a' to 'z' as a string
  - [2,4..8] = [2,4,6,8]
  - [2,4..] - infinite list! Haskell will not try to evaluate the entire infinite list immediately since it is lazy (**take** can be used to get elements till a given index)
- **List Comprehension :** Building more specific sets out of general sets

  $S = \{ 2 \cdot x \mid x \in \mathbb{N},\ x \leq 10 \}$
- x is the **variable**, N is the **input set** and x <= 10 is the **predicate.**
- [x*2 | x <- [1..10]] - Returns a list with elements double the numbers from 1 to 10. [x <- [1..10]] means that x takes on the value of each element that is drawn from [1..10]. The part before ' | ' is the output of list comprehension. Predicates to the right of ' | ', at the end.

- Weeding out parts of lists using predicates is also called filtering. Like,
  `[ x | x <- [50..100], x `mod` 7 == 3]`
- When drawing values from several lists, every combination of elements from these lists is reflected in the resulting list.
  Eg : `[ x*y | x <- [2,5,10], y <- [8,10,11]] :: [16,20,22,40,50,55,80,100,110]`
  ```
  length' xs = sum [1 | _ <- xs]
  removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
  ```

- Nested list comprehensions : `[ [ x | x <- xs, even x ] | xs <- xxs]`
- (w,h) <- xs : generator

Tuples:

- Pair : tuple of size 2. Triple : tuple of size 3. Both are of different types.
- Tuples are of a fixed size and size is treated as a part of its type.
- No appending to tuple.
- Compare tuples only of same size.
- No sigleton tuple (Since, singleton tuple is simply like(same properties) the value it contains. So no new type for singleton tuple).
- Fst (8, 11) --> 8
- snd (8,11) --> 11 (only on pairs and not on any tuples)
- 'zip' to produce a list of pairs : zip [1,2,3] [1,2,3] --> [(1,1),(2,2),(3,3)] (the 2 lists can be of different types). If lists are of different types only as much of the longer list is used as needed and the rest is ignored. So you can zip finite lists with infinite lists.
- Using triples : let rightTriangles' = [(a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a],
  $a^2 + b^2 == c^2$, a+b+c == 24].

TYPE :

- :t <exp> - gives the type of the expression (<exp> :: Type).
- Explict types are always denoted with the first letter uppercase.
- Functions are also expressions.
- [Char] : list of characters (String is same as [Char])
- (True, 'a') :: (Bool, Char).

- Each tuple length has its own type.

- Functions with explicit type declarations.

  Eg : <function name> :: type1 -> type2.....-> type where type1, type2,..... are the parameter types and type is return type.

- Int : 2^{-63} to 2^{63} (For a 64-bit machine)

- Integers : not bounded

- Float is a real floating-point number with single precision. Double is a real floating-point number with double the precision. Double use twice as many bits to represent numbers(cost of memory).

- Bool is a Booleaan type(True, False).

- Tuples are types, their definition depends on their length and the types of their components. Tuples have atmost 62 elements. Empty tuple () is also type, which can have only a single value: ().
- :t head

  head :: [a] -> a – Here, a is type variable(it can be of any type). More like generics.
- Functions using type variables are called polymorphic functions.

Type classes :

- Interface that defines some behavior.
- :t (==)

  (==) :: (Eq a) => a -> a -> Bool --> Everything before '=>' is called a class constraint. All standard Haskell types (except for input/output types and functions) are instances of Eq.
- Eq for types that support equality testing. Functions its instances implement are == and /=.
- Ord for types whose values can be put in some order. Functions such as >,<,>=,<=.
- 'compare' function takes two values whose type is an Ord instance and returns an Ordering. Ordering is a type that can be GT, LT, or EQ, which represent greater than, lesser than, or equal, respectively.

  Eg: 5 `compare` 3 --> GT
- Show type class : show 3 --> "3". Values whose types are instances of the Show type class can be represented as strings.
- Read type class : Opposite to show. Takes string and returns a value whose type is an instance of this type class. (read :: (Read a) => String -> a)

  Eg: read "[1,2,3,4]" ++ [3] gives [1,2,3,4,3]
- read "4" -> gives error. As the GHCi doesnot know what has to be returned. Type annotations (explicitly tell Haskell what the type of expression should be) helps you.

  Eg: read "5" :: Int --> 5

  read "5.2" :: Int --> error
- [read "True", False, True, False] --> [True, False, True, False] :: learns the type read has to return from other elements of the list.
- Enum Type class : sequentially ordered types. Enum type can be used to its values in list ranges, the succ and pred functions, Ordering.

  Eg : [LT .. GT] --> [LT,EQ,GT]

  [3 .. 5] --> [3,4,5]

- Bounded Type class : have an upper bound and lower bound(minBound and maxBound functions).

  Eg: minBound :: Type --> the value. (Type : Int, Float.....)

  minBound and maxBound have a type of : Bounded a => a. Hence, they are polymorphic constants.

- Num Type class : Its instances act like numbers (all numbers, including real number integers). Type : (Num t) => t

  Eg : 20 :: Int --> 20.

  (5 :: Int) * (6 :: Integer) will result in a type error, while 5 * (6::Integer) will work just fine. As 5 can act like either an Integer or an Int , but not both at the same time.

- Floating Type class : Float and Double to store floating-point numbers.(sqrt, sin, cos)
- Integral Type class : Only integral/whole numbers(Int, Integer)

  Eg : fromIntegral :: (Num b, Integral a) => a -> b --> takes an integral number and turns it into a more general number.

- Being an instance of Eq is a prerequisite for being an instance of Ord.

Pattern Matching(better do it in a file and then load to GHCi):

- Pattern matching is used to specify patterns to which some data should conform and to deconstruct the data according to those patterns.
- When the passed argument conforms to a specified pattern, the corresponding function body will be used(checks the passed argument from top to bottom).
- Recursive functions : factorial :: Int -> Int

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- (Only for lists) A pattern like x:xs will bind the head of the list to x and the rest of it to xs.

  Eg: head' :: [a] -> a

  ```
  head' [] = error "Can't call head on an empty list, dummy!"
  head' (x:_) = x
  ```

  --error function takes a string as an argument and generates a runtime error using that string. Better avoid since crashes the program sometimes. (-- indicates comments)

- Can't use the ++ operator in pattern matches. It might not be able to tell what would be in which of the lists given.

- As-pattern : precede a regular pattern with a name and an @ character. Can use the name instead of writing all the thing after the name.

  Eg: firstLetter :: String -> String

  ```
  firstLetter "" = "Empty string, whoops!"
  firstLetter all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
  ```

- Guards (|): When we want our function to check if some property of those passed values is true or false.

  Eg:
  ```
  bmiTell :: Double -> String
  bmiTell bmi
    | bmi <= 18.5 = "underweight"
    | bmi <= 25.0 = "normal"
    | bmi <= 30.0 = "fat"
    | otherwise = "obese"
  ```
  --No ‘=’ after function name before first ‘ | ’

- Not only can we call functions as infix with backticks, we can also define them using backticks.

- ‘where’ keyword (after the guards and then use it to define one or more variables or functions aligned in a single column): written at the end and visible to all guards

  Eg :
  ```
  bmiTell :: Double -> Double  -> String
  bmiTell w h
    | bmi <= 18.5 = "underweight"
    | bmi <= 25.0 = "normal"
    | bmi <= 30.0 = "fat"
    | otherwise = "obese"
        where bmi = w/h ∧ 2
  ```

- let : allows you to bind to variables anywhere and are expressions themselves. Local and don’t span across the guards. The variables that you define with let are visible within the entire let expression (‘let’ expressions are expressions whereas ‘where’ bindings are not).

  Eg:
  ```
  cylinder :: Double -> Double -> Double
  cylinder r h =
      let sideArea = 2 * pi * r * h
          topArea = pi * r ∧ 2
  ```

in sideArea + 2 * topArea

- Seperated by ‘;’ to bind several variables inline and can’t align them in columns.
- Pattern matching with let expressions : (let (a, b, c) = (1, 2, 3) in a+b+c) * 100 --> 600
- In list comprehensions : calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi > 25.0]
- Omitting the ‘in’ part in let expression makes the GHCi remember it.
- ‘case’ : to execute blocks of code for specific values of a particular variable(can be used anywhere).

  Eg: head' xs = case xs of [] -> error "No head for empty lists!"

  (x:_) -> x

Recursion:

- maximum' :: (Ord a) => [a] -> a

  maximum' [] = error "maximum of empty list!"

  maximum' [x] = x

  maximum' (x:xs) = max x (maximum' xs)

- Quicksort : quicksort :: (Ord a) => [a] -> [a]

  quicksort [] = []

  quicksort (x:xs) =

  let smallerOrEqual = [a | a <- xs, a <= x]

  larger = [a | a <- xs, a > x]

  in quicksort smallerOrEqual ++ [x] ++ quicksort larger

Higher-order functions:

- Functions that take functions as parameters or return functions as return values.
- Curried functions take only one parameter(instead of many), when it is called it returns a function that takes another parameter.
- Infix functions can also be partially applied by using sections. To section an infix function, simply surround it with parentheses and supply a parameter on only one side. That creates a function that takes one parameter and then applies it to the side that’s missing an operand.

  Eg : divideByTen :: (Floating a) => a -> a

  divideByTen = (/10)

```
divideByTen 200
```

(OR) `(/10) 200`

- For subtraction : (subtract 4) or (-4), careful when you really want to use the negative 4, i.e the negative number.
- Functions are not the instances of the Show type class. So we need a print function doing the printing job.
- zipWith library : Takes a joining function and 2 lists, then applies the joining function on each of the elements of the two lists.

```
Eg :    zipWith' :: (a ->b -> c) -> [a] -> [b] -> [c]
        zipWith' _ [] _ = []
        zipWith' _ _ [] = []
        zipWith'  f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

- flip library : Takes a function and returns a function that is like our original function, but with the first two arguments flipped.

```
Eg :    flip' :: (a -> b -> c) -> (b -> a -> c)
        flip' f = g
                where g x y = f y x
(OR)  flip' :: (a -> b -> c) -> b -> a -> c
        flip' f y x =  f x y
```

- Map function : `map :: (a->b) -> [a] -> [b]`

```
map _  [] = []
map f (x:xs) = f x : map f xs
```

- Filter function :

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x          = x : filter p xs
    | otherwise = filter p xs
```

- Eg : `let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]`
  `--> [[1,2,3],[3,4,5],[2,2]]`

  (There's no set rule for when to use map and filter versus using list comprehensions. You just need to decide what's more readable depending on the code and the context.)

- Quicksort : quicksort :: (Ord a) => [a] -> [a]

```
quicksort [] = []
quicksort (x:xs) =
let smallerOrEqual = filter (<= x) xs
larger = filter (> x) xs
    in quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

- //Largest number divisible by 3829 and less than 100000

```
largestDivisible :: Integer
largestDivisible = head (filter p [100000,99999..])
                    where p x = x `mod` 3829 == 0
```

- takeWhile function : Takes a predicate and a list. Starting at the beginning of the list, it returns the list's elements as long as the predicate holds true.

  Eg : takeWhile (/= ' ') "elephants know how to party" --> "elephants"

- Sum of all odd squares that are 10000 :

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

  (OR) sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])

- Collatz sequences :
  - Start with any natural number.
  - If the number is 1, stop.
  - If the number is even, divide it by 2.
  - If the number is odd, multiply it by 3 and add 1.
  - Repeat the algorithm with the resulting number.

  (For all starting numbers, the chain will finish at the number 1)

```
chain :: Integer -> [Integer]
chain [1] = 1
chain n
    | even n = n:chain (n `div` 2)
    | odd n  = n:chain (n*3 +1)
```

  Number of such chains of numbers from 1 to 100 and length > 15,

```
numLongChains :: Int
```

```
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

- Map function with the mapping function having more than one parameter.

  Eg : let listOfFuns = map (*) [0..]

  (listOfFuns !! 4) 5 --> 20

  Lambdas(are just expressions):

- Anonymous functions that we use when we need a function only once.
- '\' : lambda function declarations, followed by parameters and then the '->' that result in function body.

  Eg : numLongChains :: Int

  numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))

- Pattern matching in lambdas : map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
- Only difference is you cannot define several patterns for one parameter like making [] and (x:xs) pattern for the same parameter and then having values fall through.
- If a pattern match fails in a lambda, a runtime error occurs.
- foldl (left fold) : The binary function is applied between the starting accumulator and the head of the list. Then the binary function is called with that value and the next element, and so on.

  Eg : sum' :: (Num a) => [a] -> a

  sum' xs = foldl (\acc x -> acc + x) 0 xs

  (OR) sum' = foldl (+) 0

- foldr (right fold) : The accumulator eats up the values from the right. The current list value is the first parameter, and the accumulator is the second in the binary function.

  Eg : map' :: (a -> b) -> [a] -> [b]

  map' f xs = foldr (\x acc -> f x : acc) [] xs

  (OR) map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs // '++' is much slower than ':'

- Right folds work on infinite lists, whereas left ones don't.
- Boolean accumulator :

  elem' :: (Eq a) => a -> [a] -> Bool

  elem' y ys = foldr (\x acc -> if x == y then True else acc) False ys

- foldl1 and foldr1 functions donot need the accumulator to be specified explicitly.

  Eg : maximum' :: (Ord a) => [a] -> a

  maximum' = foldl1 max --(First element as accumulator).

  As they depend on the lists they're called with having at least one element, these functions cause runtime errors if called with empty lists. Foldl and foldr, on the other hand, work fine with empty lists.

- Filter function :

  filter' :: (a -> Bool) -> [a] -> [a]

  filter' p = foldr (\x acc -> if p x then x : acc else acc) []

- The '&&' function works in such a way that if its first parameter is False , it disregards its second parameter, because the && function returns True only if both of its parameters are True.

  Eg : (&&) :: Bool -> Bool -> Bool

  True && x = x

  False && _ = False

- scanl and scanr functions are like foldl and foldr but they report all the intermediate accumulator states in the form of a list . scanl1 and scanr1 are analogous to foldl1 and foldr1.

  Eg : scanl (+) 0 [3,5,2,1] --> [0,3,8,10,11]

  scanr (+) 0 [3,5,2,1] --> [11,8,3,1,0]

  scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1] --> [3,4,5,5,7,9,9,9]

  scanl (flip (:)) [] [3,2,1] --> [[],[3],[2,3],[1,2,3]]

- When using a scanl, the final result will be in the last element of the resulting list. scanr will place the result in the head of the list.

- How many elements does it take for the sum of the square roots of all natural numbers to exceed 1,000?

  sqrtSums :: Int

  sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1

- '$' : Function application operator.

  Eg : ($) :: (a -> b) -> a -> b

  f $ x = f x

- Normal function application (putting a space between two things) has a really high precedence, the $ function has the lowest precedence. Function application with a space is left-associative (so f a b c is the same as ((f a) b) c), while function application with $ is right-associative, i.e, when a $ is encountered, the expression on its right is applied as the parameter to the function on its left.
- sum (map sqrt [1..130]) is same as sum $ map sqrt [1..130].
- Function composition with the '.' function, right associative, we can compose many functions at a time.

  Eg : (.) :: (b -> c) -> (a -> b) -> a -> c

  f . g = \x -> f (g x)
- Using lambdas : map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]

  Using function composition : map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
- f (g (z x)) is equivalent to (f . g . z) x
- For functions which takes several parameters, if we want to use them in function composition, we usually must partially apply them so that each function takes just one parameter.

  Eg : sum . replicate 5 $ max 6.7 8.9

  replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
- Function in point-free style :

  (i) sum' :: (Num a) => [a] -> a

  sum' = foldl (+) 0 instead of sum' xs = foldl (+) 0 xs

  (ii) fn = ceiling . negate . tan . cos . max 50
- If a function is too complex, writing it in point-free style can actually be less readable. For this reason, making long chains of function composition is discouraged. The preferred style is to use let bindings to give labels to intermediary results or to split the problem into subproblems that are easier for someone reading the code to understand.

Reference : Learn you Haskell for great good! A beginner's guide

Author : Miran Lipovača