Haskell is 1) pure w/ no side effects 2) lazy evaluation 3) statically typed, no type declarations, supports polymorphic types

You can declare a type by: **y = 1 :: Int**

**++** concatenates, not increments

Currying: function application happens one argument at a time // it happens with stuff like fold // its parenthesis notation tbh

functions: show :: Show a => a -> String: convert the given value into a string.

functions: (==), (/=) :: Eq a => a -> a -> Bool: equality and inequality.

functions: (), (<=), (>=) :: Ord a => a -> a-> Bool: less than, greater than, less or equal, greater or equal

functions: (+), (-), (\*) :: Num a => a -> a -> a: arithmetic operations.

functions: div, mod :: Integral a => a -> a -> a: division.

functions: (/) :: Fractional a => a -> a -> a: division.

Returning tuple

strPair :: Integer -> (Integer, String)

strPair x = (x, show x)

strPair 5 -- will return (5,"5")

Extract the element at a specific index position out of a list

(!!) :: [a] -> Int -> a

["zero","one","two","three","four","five","six"] !! 5 ⇒ "five"

"CptS355“ !! 4 ⇒ '3'

y:\_ = ['c','a','t'] -- y will be assigned to 'c’

\_:xs = ['c','a','t'] -- xs will be assigned to ['a','t'] or 'at'

extractDigits :: String -> String

extractDigits [] = []

extractDigits (chr : xs) | isDigit chr = chr : (extractDigits xs)

| otherwise = extractDigits xs

let ... in introduces a variable/function before it can be used:

reverse' :: [a] -> [a]

reverse' [] = []

reverse' (x:xs) = let

snoc x xs = xs ++ [x]

in x `snoc` (reverse' xs)

A variable bound with **let** has a so called scope. That is, it is only “visible” after the in in the context of a computational block. A variable bound with **where** is “visible” anywhere in the body of a function preceding the declaration

Tail Recursive AddUp

addup2 :: Num p => p-> [p] -> p

addup2 accum [] = accum

addup2 accum (x:xs) = (addup2 (accum + x) xs) // we use the parameter instead of the call stack to keep track of the sum

Tail Recursion : Pop the caller before the call, allowing the callee to reuse the same stack space

Use a helper function to add the additional parameter, while the parent function keeps the original pattern

Anonymous function: sqAll = map (\x -> x \* x) [1,2,3,4,5]

foldR op base [e1,e2,e3,e4] ⇒ op e1 (op e2 (op e3 (op e4 base)))

foldL op acc [e1,e2,e3,e4] ⇒ (op (op (op (op acc e1) e2) e3) e4) // Tail Recursive

copyList :: [a] -> [a] copyList xL = **foldr** (:) [] xL

copyList2 xL = **reverse** (**foldl** (\xs x -> x:xs) [] xL)

**tailmap** :: (a -> b) -> [a] -> [b]

tailmap op xL = reverse (aux\_map op xL [])

where aux\_map f [] acc = acc

aux\_map f (x:xs) acc = aux\_map f xs ((f x) : acc)

**tailfilter** :: (a -> Bool) -> [a] -> [a]

tailfilter op xL = reverse (aux\_filter op xL [])

where aux\_filter f [] acc = acc aux\_

filter f (x:xs) acc | (f x) = (aux\_filter f xs (x : acc))

| otherwise = (aux\_filter f xs acc)

**eliminateDuplicates** :: Eq a => [a] -> [a]

eliminateDuplicates [] = [] -- base case, if the list is empty

eliminateDuplicates [x] = [x] -- base case, if the list onlly has one element

eliminateDuplicates (x:xs)

|x `elem` xs = eliminateDuplicates xs

|otherwise = x : eliminateDuplicates xs -- if its unique then we add it to the list so it will take the last occurance of the duplicate

**commons\_tail** :: Eq a => [a] -> [a] -> [a]

commons\_tail [] [] = [] -- base case, empty lists return empty lists

commons\_tail [] l2 = [] -- base case, if one is empty then the elemts can't appear in both

commons\_tail l1 [] = []

commons\_tail l1 l2 = commons\_tail\_helper l1 l2 []

where

commons\_tail\_helper [] [] buf = reverse buf

commons\_tail\_helper l1 [] buf = reverse buf

commons\_tail\_helper [] l2 buf = reverse buf

commons\_tail\_helper (x:xs) l2 buf = if (x `elem` l2) && (x `notElem` buf) then commons\_tail\_helper xs l2 (x:buf) else commons\_tail\_helper xs l2 buf

**find\_languages**::(Eq a1, Eq a2) => [(a2, [a1])] -> [a2] -> [a1]

find\_languages [] [] = []

find\_languages l1 [] = []

find\_languages [] l2 = []

find\_languages l1 l2 = commons\_all (map snd (filter ((`elem` l2).fst) l1))

**nested\_max** :: [[Int]] -> Int

nested\_max [] = minBound

nested\_max l1 = foldr max minBound (map (foldr (max) minBound ) l1)

-- Binary tree with data only in leaves

data Tree a = LEAF a | NODE (Tree a) (Tree a) deriving (Show, Eq)

-- example Tree Int

tree4 = NODE (NODE (LEAF "one") (LEAF "two")) (NODE (LEAF "three") (LEAF "four"))

-- count the number of leaves in the tree

nLeaves :: Num p => Tree a -> p

nLeaves (LEAF \_) = 1

nLeaves (NODE t1 t2) = (nLeaves t1) + (nLeaves t2)

-- call nLeaves

l = nLeaves tree4 --returns 4

-- make a copy of the tree

copyTree :: Tree a -> Tree a

copyTree (LEAF x) = LEAF x

copyTree (NODE t1 t2) = NODE (copyTree t1) (copyTree t2)

-- call copyTree

tree4\_copy = copyTree tree4

--tree map

treeMap :: (t -> a) -> Tree t -> Tree a

treeMap op (LEAF x) = LEAF (op x)

treeMap op (NODE t1 t2) = NODE (treeMap op t1) (treeMap op t2)

-- call treeMap

strUpper s = map toUpper s --should import Data.Char

tree4\_uppercase = treeMap strUpper tree4

-- Pre-Order traversal; returns a list of the leaf values

preOrder :: Tree a -> [a]

preOrder (LEAF x) = [x]

preOrder (NODE t1 t2) = (preOrder t1) ++ (preOrder t2)

--call PreOrder

pList = preOrder tree4