HAYER ORDAR PHUNKSHUNS

λ (max 4 5

5

λ (max 4) 5

5

(max :: (Ord a) => a -> a -> a
max :: (Ord a) => a -> (a -> a) } equivalent

function

space - is function application operator

we can get a partially applied function

multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z

multThree 3 5 9

((multThree 3) 5) 9

multThree :: (Num a) => a -> (a -> (a -> a))

λ let multTwoWithNine = multThree 9

λ multTwoWithNine 2 3

54

λ let multWithEighteen = multTwoWithNine 2

λ multWithEighteen 10

180

compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100

divideByTen :: (Floating a) => a -> a
divideByTen = (/10)

divideByTen 200
200 / 10
(/10) 200

isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])

(-4)  X
(subtract 4)

λ multThree 3 4  X  can't print (show) function

applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

λ applyTwice (+3) 10
16
λ applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
λ applyTwice ("HAHA" ++) "HEY"
"HAHA HAHA HEY"
λ applyTwice (multThree 2 2) 9
144
```

λ applyTwice (3:) [1]
[3, 3, 1]

zipWith' :: (a → b → c) → [a] → [b] → [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

λ zipWith' (+) [4, 2, 5, 6] [2, 6, 2, 3]
[6, 8, 7, 9]
λ zipWith' max [6, 3, 2, 1] [7, 3, 1, 5]
[7, 3, 2, 5]
λ zipWith' (++) ["foo", "bar", "baz"] ["fighters", "hoppers", "aldrin"]
["foo fighters", "bar hoppers", "baz aldrin"]
λ zipWith' (*) (replicate 5 2) [1..]
[2, 4, 6, 8, 10]
λ zipWith' (zipWith' (*)) [[1,2,3], [3,5,6], [2,3,4]] [[3,2,2], [3,4,5], [5,4,3]]
[[3, 4, 6], [9, 20, 30], [10, 12, 12]]

flip' :: (a → b → c) → (b → a → c)
flip' f = g
  where g x y = f y x

flip' :: (a → b → c) → (b → a → c)
flip' f x y = f y x

λ flip' zip [1,2,3,4,5] "hello"
[('h',1), ('e',2), ('l',3), ('l',4), ('o',5)]
λ zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

λ map (+3) [1,5,3,1,6]
[4,8,6,4,9]
λ map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!", "BANG!", "POW!"]
λ map (replicate 3) [3..6]
[[3,3,3], [4,4,4], [5,5,5], [6,6,6]]
λ map (map (^2)) [[1,2], [3,4,5,6], [7,8]]
[[1,4], [9,16,25,36], [49,64]]
λ map fst [(1,2), (3,5), (6,3), (2,6), (2,5)]
[1,3,6,2,2]

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

λ filter (>3) [1,5,3,2,1,6,4,3,2,1]

[5,6,4]

λ filter (==3) [1,2,3,4,5]

[3]

λ filter even [1..10]

[2,4,6,8,10]

λ 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]]

λ filter ('elem' ['a'..'z'])

"u LaUgH aT mE BeCaUsE I aM diFfeRent"

"uagameasadifeent"

λ filter ('elem' ['A'..'Z']) "i laUgh At You BecAuse

u r aLL the same"

"GAYBALLS"

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

quicksort [] = []

quicksort (x:xs) =

let smallerSorted = quicksort (filter (<=x) xs)

largerSorted = quicksort (filter (>x) xs)

in smallerSorted ++ [x] ++ largerSorted

largestDivisible :: (Integral a) => a

largestDivisible = head (filter p [100000, 99999, ..])

wher p x = x `mod` 3829 == 0

λ sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650

λ sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])
166650

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

λ chain 10
[10, 5, 16, 8, 4, 2, 1]
λ chain 1
[1]
λ chain 30
[30 ... 1]

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

λ let listOfFuns = map (*) [0..]
λ (listOfFuns !! 4) 5
20

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

```
λ zipWith (\a b -> (a * 30 + 3) / b ) [...] [...]
[...]
```

```
λ map (\(a,b) -> a+b) [(1,2), (3,5), (6,3), (3,6), (2,5)]
[3, 8, 9, 8, 7]
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

} gimmick to illustrate currying

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = \x y -> f y x
```

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

```
λ sum' [3, 5, 2, 1]
11
```

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc)
                   False ys

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

map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

foldl1 , foldr1 (take element from list itself)

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

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

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

```
reverse' = foldl (flip (:)) []
```

λ scanl (+) 0 [3,5,2,1]
[0, 3, 8, 10, 11]
λ scanr (+) 0 [3,5,2,1]
[11, 8, 3, 1, 0]
λ scanl1 (\acc x → if x > acc then x else acc) [...]
[...]
λ scanl (flip (:)) [] [3,2,1]
[[], [3], [2,3], [1,2,3]]

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

λ sqrtSums
131
λ sum (map sqrt [1..131])
1005. ...
λ sum (map sqrt [1..130])
993, ...

```
($) :: (a → b) → a → b
f $ x = f x
```
right associative.

```
f a b c
= ((f a) b) c
```
} default = left associative

```
sum (map sqrt [1..130])
sum $ map sqrt [1..130]

sqrt (3+4+9)
sqrt $ 3 + 4 + 9

sum (filter (>10) (map (*2) [2..10]))
sum $ filter (>10) $ map (*2) [2..10]

λ map ($3) [(4+), (10*), (^2), sqrt]
[7.0, 20.0, 9.0, 1.732...]

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
        right associative

λ map (\x -> negate (abs x)) [...]
[...]

λ map (negate . abs) [5,-3,-6,7,...]
[-5,-3,-6,-7,...]

λ map (\xs -> negate (sum (tail xs))) [[1..5], [3..6], [1..7]]
[-14, -15, -27]

λ map (negate . sum . tail) [[1..5],[3..6], [1..7]]
[-14, -15, -27]
```

```
sum (replicate 5 (max 6.7 8.9))
(sum . replicate 5 . max 6.7 ) 8.9
sum . replicate 5 . max 6.7 $ 8.9

replicate 100 (product (map (*3) (zipWith max [--] [--])))
replicate 100 . product . map (*3) . zipWith max [--] $ [7.

fn = ceiling . negate . tan . cos . max 50

oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000)
    - filter odd . map (^2) $ [1..]
```

classmate

Date 11.9.2019
Page 1

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length.nub

λ :m + Data.List

λ :m + Data.List Data.Map Data.Set

import Data.List (nub, sort)

import Data.List hiding (nub)

import qualified Data.Map

import qualified Data.Map as M

λ intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
λ intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]

λ intercalate " " ["hey", "there", "guys"]
"hey there guys"
λ intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

λ transpose [[1,2,3], [4,5,6], [7,8,9]]
[[1,4,7], [2,5,8], [3,6,9]]
λ transpose ["hey", "there", "guys"]
["htg", "ehu", "yey", "rs", "e"]

λ map sum $ transpose [[0,3,5 9], [10,0,0,9], [8,5,1,-1]]
[18, 8, 6, 17]

adding polynomials together

foldl', foldl1' —— non lazy versions (no stack overflow)

λ concat ["foo", "bar", "car"]
"foo bar car"
λ concat [[3,4,5], [2,3,4], [2,1,1]]
[3,4,5,2,3,4,2,1,1]

λ concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]

λ and $ map (>4) [5,6,7,8]
True
λ and $ map (==4) [4,4,4,3,4]
False

λ or $ map (==4) [2,3,4,5,6,1]
True
λ or $ map (>4) [1,2,3]
False

λ any (==4) [2,3,5,6,1,4]
True
λ all (>4) [6,9,10]
True
λ all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
λ any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True

λ take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
λ take 3 $ iterate (++"haha") "haha"
["haha","hahahaha","hahahahahaha"]

λ splitAt 3 "heyman"
("hey","man")
λ splitAt 100 "heyman"
("heyman","")
λ splitAt (-3) "heyman"
("","heyman")
λ let (a,b) = splitAt 3 "foobar"
in b ++ a
"barfoo"

λ takeWhile (>3) [6,5,4,3,2,1,...]
[6,5,4]
λ takeWhile (/=' ') "This is a sentence"
"This"

λ sum $ takeWhile (<10000) $ map (^3) [1..]
53361

λ dropWhile (/=' ') "This is a sentence"
" is a sentence"
λ dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]

λ let stock = [(994.4, 2008, 9, 1), (995.2, 2008, 9, 2),
(999.2, 2008, 9, 3), (1001.4, 2008, 9, 4), (998.3, 2008, 9, 5)]
λ head (dropWhile (\(val, y, m, d) -> val < 1000) stock)
(1001.4, 2008, 9, 4)

λ let (fw, rest) = span (/=' ') "This is a sentence"
in "First word: " ++ fw ++ ", the rest:" ++ rest
"First word: This, the rest: is a sentence."

λ break (==4) [1,2,3,4,5,6,7]
([1,2,3], [4,5,6,7])
λ span (/=4) [1,2,3,4,5,6,7]
([1,2,3], [4,5,6,7])

λ sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
λ sort "This will be sorted soon"
" Tbbdeehiillnooorssstw"

λ group [1,1,1,2,2,3,3,3,3,2,2,2,5,6,7]
[[1,1,1],[2,2],[3,3,3,3],[2,2,2],[5],[6],[7]]

λ map (\l@(x:xs) → (x, length l)) . group . sort $ [1,1,1,...]
[(1,4), (2,7), ...]

λ inits "woot"
["", "w", "wo", "woo", "woot"]
λ tails "woot"
["woot", "oot", "ot", "t", ""]
λ let w = "woot" in zip (inits w) (tails w)
[("", "woot"), ("w", "oot"), ...]

search :: (Eq a) ⇒ [a] → [a] → Bool
search needle haystack =
  let nlen = length needle
  in foldl (\acc x → if take nlen x == needle then True else acc)
           False (tails haystack)

λ "cat" `isInfixOf` "im a cat burglar"
True
λ "Cat" `isInfixOf` "..."
False
λ "cats" `isInfixOf` "..."
False

• isPrefixOf , isSuffixOf

λ partition (`elem` ['A'...'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN", "sidneyeddy")
λ partition (>3) [1, 3, 5, 6, 3, 2, 1, 0, 3, 7]
([5, 6, 7], [1, 3, 3, 2, 1, 0, 3])

λ find (>4) [1, 2, 3, 4, 5, 6]
Just 4
λ find (>9) [...]
Nothing
λ :t find
find :: (a -> Bool) -> [a] -> Maybe a

λ :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
λ 4 `elemIndex` [...]
Just 3
λ 10 `elemIndex` [...]
Nothing

λ ' ' `elemIndices` "Where are the spaces?"
[5, 9, 13]

λ findIndex (==4) [5,3,2,1,6,4]
Just 5
λ findIndex (==7) [...]
Nothing
λ findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0, 6, 10, 14]

λ zipWith3 (\x y z → x+y+z) [1,2,3] [4,5,2,2] [2,2,3]
[7, 9, 8]
λ zip4 [3,3,3] [2,2,2] [5,5,3] [3,2,2]
[(3, 2, 5, 2), (3, 2, 5, 2), (3, 2, 3, 2)]

λ lines "first line \n second line \n third line"
["first line", "second line", "third line"]

λ unlines [...]
"..."

words, unwords
nub → weed out duplicate elements.

λ nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
λ nubs "Lots of words and stuff"
"Lots fwrdanu"

λ delete 'h' "hey there ghang!"
"ey there ghang!"
λ delete 'h' . delete 'n' $ "hey there ghang!"
"ey tere ghang!"

\\ - list difference function
λ [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
λ "Im a big baby" \\ "big"
"Im a baby"

} like doing delete a . delete b

union
λ "hey man" `union` "man what's up"
"hey man wt'sup"

λ [1..7] `intersect` [5..10]
[5,6,7]

λ insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]

} for insertion sort

genericLength
genericTake
genericDrop
genericSplitAt
genericIndex
genericReplicate

} Num versions (not Int)

nubBy
deleteBy
unionBy
intersectBy
groupBy

} use custom equality fn. instead of ==

Data.Function `on` :: (b → b → c) → (a → b) → a → a → c
f `on` g = \x y → f (g x) (g y)

λ groupBy ((==) `on` (>0)) [-4,-2,-1,0,2,5,-2,-14,2,2]
[[-4,-2,-1,0],[2,5],[-2,-14],[2,2]]

sortBy
insertBy
maximumBy
minimumBy

} custom compare fn.

λ let xs = [[5,4,5,4,4], [1,2,3], [3,5,4,3], [], [2], [2,2]]
λ sortBy (compare `on` length) xs
[[], [2], [2,2], [1,2,3], [3,5,4,3], [5,4,5,4,4]]

Data.Char...
isControl
isSpace
isLower
isUpper
isAlpha
isAlphaNum
isPrint
isDigit
isOctDigit
isHexDigit
isLetter
isMark
isNumber
isPunctuation
isSymbol
isSeparator
isAscii
isLatin1
isAsciiUpper
isAsciiLower

λ all isAlphaNum "bobby283"
True
λ all isAlphaNum "eddy the fish!"
False

λ groupBy (==) `on` isSpace "hey guys its me"
["hey", " ", "guys", " ", "its", " ", "me"]
λ words "hey guys its me"
["hey", "guys", "its", "me"]
λ (filter (not . any isSpace) . groupBy ((==) `on` isSpace)
    $ "...")
[...]

generalCategory :: Char → GeneralCategory
λ generalCategory ' '
Space
λgeneralCategory 'a'
LowercaseLetter

toUpper
toLower
toTitle
digitToInt
} on Char

λ map digitToInt "FF85AB"
[15,15,8,5,10,11]

intToDigit
ord, chr

```
encode :: Int -> String -> String
encode shift msg =
  map (chr . (+ shift) . ord) msg
```

} Caesar cipher

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
phoneBook = [
  ("betty", "555-2938"),
  ("bonnie", "452-2928"),
  ...
]
```

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> k == key) $ xs

findKey key = foldr (\(k,v) acc ->
    if k == key then Just v else acc) Nothing
```

like lookup from Data.List

```
import qualified Data.Map as Map
λ Map.fromList phoneBook
fromList ...
λ Map.fromList [(1,2), (3,4) ...]
fromList [(1,2), (3,4), ...]
```

```
Map.fromList :: (Ord k) => [(k,v)] -> Map.Map k v

λ Map.empty
fromList []
λ Map.insert 3, 100 Map.empty
fromList [(3, 100)]
λ Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100), (4, 200), (5, 600)]

fromList' :: (Ord k) => [(k, v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty

λ Map.null Map.empty
True
λ Map.size Map.empty
0
λ Map.singleton 5 9
fromList [(5, 9)]

lookup  -> Maybe v
map, filter  return Map but operate on values
toList  -> return List associations
fromListWith -> doesnt discard duplicates, takes a function
                to decide how to deal with duplicates
insertWith ->
```

```
import qualified Data.Set as Set

Set.fromList [...]
Set.intersection set1 set2
Set.difference set1 set2
Set.union set1 set2

null
size
member
empty
singleton
insert
delete

map
filter

λ let setNub = Set.toList . Set.fromList
λ setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
λ nub "HEY WHATS CRACKALACKIN"
"HEY WHATSCRKLIN"

module Geometry.Sphere (
  ( volume,
    area
) where
  ... the definitions
```