Administrivia

- Student hours (or by appointment):
  - Listed on Canvas in Syllabus
  - MTWTh 10:30 – 11:30 a.m. via Zoom
- First assignment has been posted
  - due Friday 26 May by 10 pm
- Second assignment posted for those finished with HW1
  - due Friday 2 June by 10 pm

Recall: Recursive Functions – one more example

Calculate the lengths of the sublists in a list:

```
lengthofSublist :: [[a]] -> [Int]
lengthofSublist [] = []
lengthofSublist (x:xs) = (length x) : (lengthofSublist xs)

k = lengthofSublist [[1,2,3],[4,5],[6],[]] -- returns [3,2,1,0]
```

How can we implement using *tail recursion*?

Did you get it working?

Recall: Recursive Functions – one more example

Is it more efficient than standard recursion?

Calculate the lengths of the sublists in a list:

```
lengthofSublist :: [[a]] -> [Int]
lengthofSublist [] = []
lengthofSublist (x:xs) = (length x) : (lengthofSublist xs)

k = lengthofSublist [[1,2,3],[4,5],[6],[]] -- returns [3,2,1,0]
```

CptS 355- Programming Language Design

Functional Programming in Haskell
Higher Order Functions

Instructor: Jeremy E. Thompson

Su 2023

Washington State University
World Class. Face to Face.

Programming Aside

- Students may struggle with testing their function definitions
- Testing suite essentially only useful when all functions have been defined
- Useful approaches:
  - As mentioned previously, can stub out all functions, OR
  - define your own main function in main.hs for function testing

```
import HW1

main = do
    print (exists 1 [1,2,3])
    print (exists 'l' "hello")
    print (exists 'x' "hello")
```

Haskell: Higher-Order Functions

- A function is higher-order if:
  - it takes another function as an *argument*, or
  - it *returns* a function as its result
- Functional programs make extensive use of higher-order functions to make programs *smaller* and more *elegant*
- We use higher-order functions to encapsulate common *patterns* of computation
- Note: Higher order functions are *prohibited* for Assignment 1 and required for Assignment 2

Higher Order Functions: `map`

- Creating a new list with the *same number* of elements (by altering a given list) is a very common *pattern* that we do in programming

- Examples:

```
allSquares :: Num a => [a] -> [a]
allSquares [] = []
allSquares (x : xs) = x * x : allSquares xs
```

Is this higher order?

```
strToUpper :: String -> String
strToUpper [] = []
strToUpper (chr : xs) = (Data.toUpper chr) : (strToUpper xs)
```

- This type of computation is very common
- Haskell has a built-in function `map` which takes a function `op` and a `list`
  - constructs new list by applying function `op` to every element of the input list

```
map op [e1,e2,e3,e4]
⇓
[(op e1),(op e2),(op e3),(op e4)]
```

Higher Order Functions: map

Map function :

```
map' :: (a -> b) -> [a] -> [b]
map' op [] = []
map' op (x : xs) = (op x) : (map' op xs)
```

Is this higher order?

- We can redefine allSquares and strToUpper functions using map

```
allSquares :: Num a => [a] -> [a]
allSquares xL = map square xL
                where square x = x * x
```

```
allSquares :: Num a => [a] -> [a]
allSquares xL = map (^2) xL
```

Will this work?

```
import Data.Char as Data

strToUpper :: String -> String
strToUpper xS = map toUpper xS
```

Anonymous Functions in Haskell

- We can also define anonymous functions (i.e., functions without names):
  - Instead of:

```
functionName a1 a2 ⋯ an = body
```

  - We write:

```
\a1 a2 ⋯ an -> body
```

  - Examples:

```
\x -> x * x         -- anonymous function calculating the square.
sq = \x -> x * x    -- can bind the function value to a variable (e.g., sq)
(\x -> x * x) 5     -- can directly call the anonymous function ; this will return 25

-- can pass the anonymous function as argument to a higher order function
sqAll = map (\x -> x * x) [1,2,3,4,5]
```

```
(\x y -> x*y) 5 2 --anonymous function call with two arguments
```

Higher Order Functions: `filter`

- Filter function takes a "predicate" function and a list; and returns a list consisting of the elements of the original list for which the predicate function returns True
  - predicate function: a function that returns a Bool value

Example predicate function:

```
isNeg :: (Ord a, Num a) => a -> Bool
isNeg x = if x<0 then True else False
```

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

  - Filter examples:

```
negatives :: (Ord a, Num a) => [a] -> [a]
negatives xL = filter isNeg xL
negatives [-3,-2,-1,0,1,2,3] -- returns [-3,-2,-1]

import Data.Char
extractDigits :: String -> String
extractDigits strings = filter isDigit strings
extractDigits "CptS355" -- returns 355
```

Higher Order Functions: `filter`

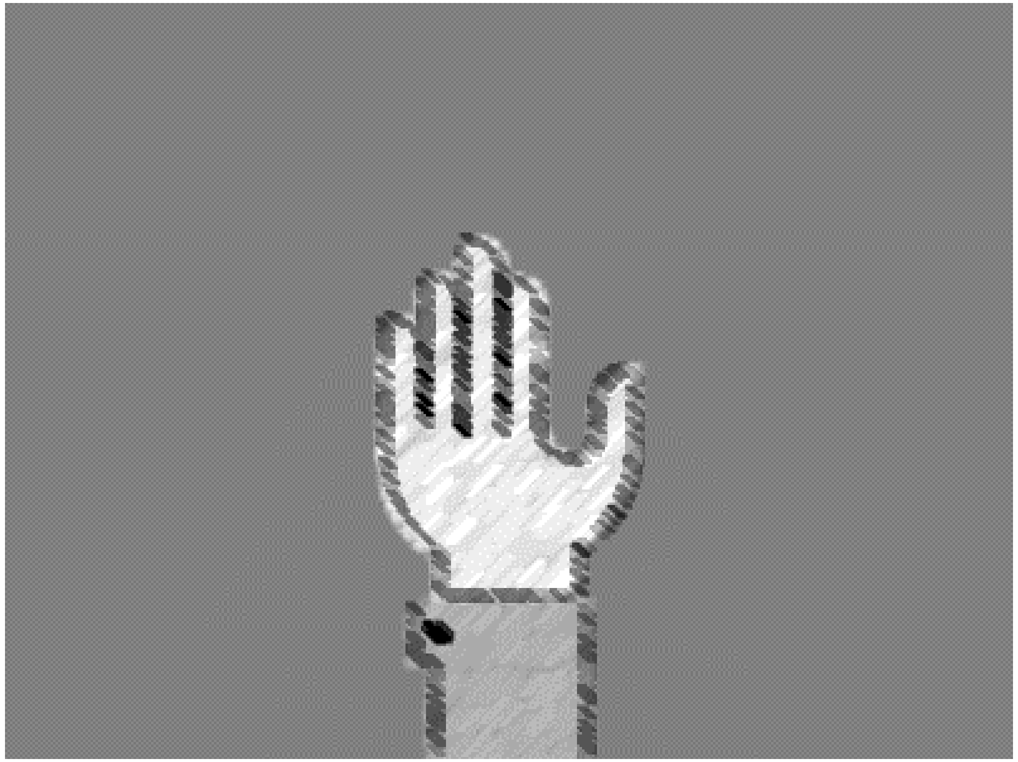
- `filterSmaller` – revisited

```
filterSmaller [] v = []
filterSmaller (x:xs) v | (x >= v) = x:(filterSmaller xs v)
                       | otherwise = (filterSmaller xs v)
```

- How can we re-write `filterSmaller` using `filter`? Write **two** ways

Breakout!

Questions?



Recall: Haskell: Higher-Order Functions

- A function is higher-order if:
  - it takes another function as an *argument*, or
  - it *returns* a function as its result

- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant

- We use higher-order functions to encapsulate common patterns of computation

Recall: Higher Order Functions: `filter`

- `filterSmaller` – revisited

```
filterSmaller [] v = []
filterSmaller (x:xs) v | (x >= v) = x:(filterSmaller xs v)
                       | otherwise = (filterSmaller xs v)
```

- How can we re-write `filterSmaller` using `filter`? Write **two** ways

```
filterSmaller [] v = []
filterSmaller iL v = filter (isGreater v) iL
  where
    isGreater v x = (x >= v)  --using where clause
```

```
fS [] v = []
fS iL v = filter ((\v x -> x>=v) v) iL  --using anonymous function
```

```
fS [] v = []
fS iL v = filter ((<=) v) iL  --using function currying
```

Higher Order Functions: `foldr`

- Remember the following functions:

```haskell
addup :: Num p => [p] -> p
addup []      = 0
addup (x:xs) = x + (addup xs)
```

```haskell
minList :: [Int] -> Int
minList []      = maxBound
minList (x:xs) = x `min` minList xs
```

```haskell
concatStr :: [String] -> String
concatStr [] = ""
concatStr (x:xs) = x ++ (concatStr xs)
```

- These 3 functions follow the same *pattern*
- There are only small differences, which are:
  1. What we did to combine the elements in the list (addition vs comparison vs concatenation)
  2. What we used as the base case

Higher Order Functions: `foldr`

- Now we will look into another higher-order function that is an abstraction of this pattern and it is called the "`foldr`" function

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = x `op` (foldr' op base xs)
```

OR

Does `foldr` expect a *prefix* or an *infix* function?

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = op x (foldr' op base xs)
```

- `foldr` folds a list together by successively applying the function `op` to the elements of the input list

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

Is this *tail* recursion?

Note: Not Haskell syntax

Higher Order Functions: foldr

- Examples:

```haskell
addup :: Num a => [a] -> a
addup xL = foldr (+) 0 xL
```

```haskell
addup :: Num p => [p] -> p
addup []      = 0
addup (x:xs) = x + (addup xs)
```

```haskell
minList :: [Int] -> Int
minList xL = foldr min maxBound xL
```

```haskell
minList :: [Int] -> Int
minList []      = maxBound
minList (x:xs) = x `min` minList xs
```

```haskell
concatStr :: [String] -> String
concatStr xL = foldr (++) "" xL
```

```haskell
concatStr :: [String] -> String
concatStr [] = ""
concatStr (x:xs) = x ++ (concatStr xs)
```

```haskell
reverse' :: [a] -> [a]
reverse' iL= foldr (\x xs -> xs ++ [x]) [] iL
```

```haskell
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
              where snoc x xs = xs ++ [x]
```

```haskell
allEven :: [Int] -> Bool
allEven iL = foldr (\x b -> even x && b) True iL
```

```haskell
allEven :: [Int] -> Bool
allEven [] = True
allEven (x:xs) = x `allE` (allEven xs)
            where allE x b = (even x) && b
```

foldr

Examples:

- What will this `mystery` function do?

```
cons :: a -> [a] -> [a]
cons x xs = x:xs

mystery xL = foldr cons [] xL

mystery [1,2,3,4,5]
```

Higher Order Functions: foldr - cont.

- How does `foldr` work?
  - It traverses the list from right to left and applies the combining function
- For example:

```
addup xL = foldr (+) 0 xL
addup [1,2,3]
```

```
addup 1 (foldr addup 0 [2,3])
addup 1 (addup 2 (foldr addup 0 [3]))
addup 1 (addup 2 (addup 3 (foldr addup 0 [])))
addup 1 (addup 2 (addup 3  0))
addup 1 (addup 2 3)
addup 1 5
6
```

- There is a variation of the fold function called "`foldl`" which traverses the list from left to right. i.e.,

```
(addup (addup (addup 0 1) 2) 3)
```

Tail recursive `foldl`

- "`foldl`" iterates over the elements from left to right

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' op acc [] = acc
foldl' op acc (x:xs) = foldl' op (acc `op` x) xs
```

Tail-recursive

```
foldl op acc [e1,e2,e3,e4]
    ⇒ (op (op (op (op acc e1) e2) e3) e4)
```

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = x `op` (foldr' op base xs)
```

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

Tail recursive `foldl`

```
copyList :: [a] -> [a]
copyList xL = foldr (\x xs -> x:xs) [] xL
```

- How should we re-write copyList using `foldl` ?

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

Quiz:

- What will the following code return?

```
foldr (-) 0 [3,2,1]
```

a) 0
b) 2
c) -6
d) error

- What will the following code return?

```
foldl (-) 0 [3,2,1]
```

a) 0
b) 2
c) -6
d) error

Tail recursive map

- `map`

```
map' :: (a -> b) -> [a] -> [b]
map' op [] = []
map' op (x : xs) = (op x) : (map' op xs)
```

Base case

- Tail recursive map: `tailmap`

```
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)
```

acc init

Tail recursive `filter`

- `filter`

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

- Tail recursive `filter`: `tailfilter`

```
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)
```

Examples: `map`, `fold`, `filter`

```
gt :: Ord a => a -> a -> a
gt x y = if x < y then y else x
```

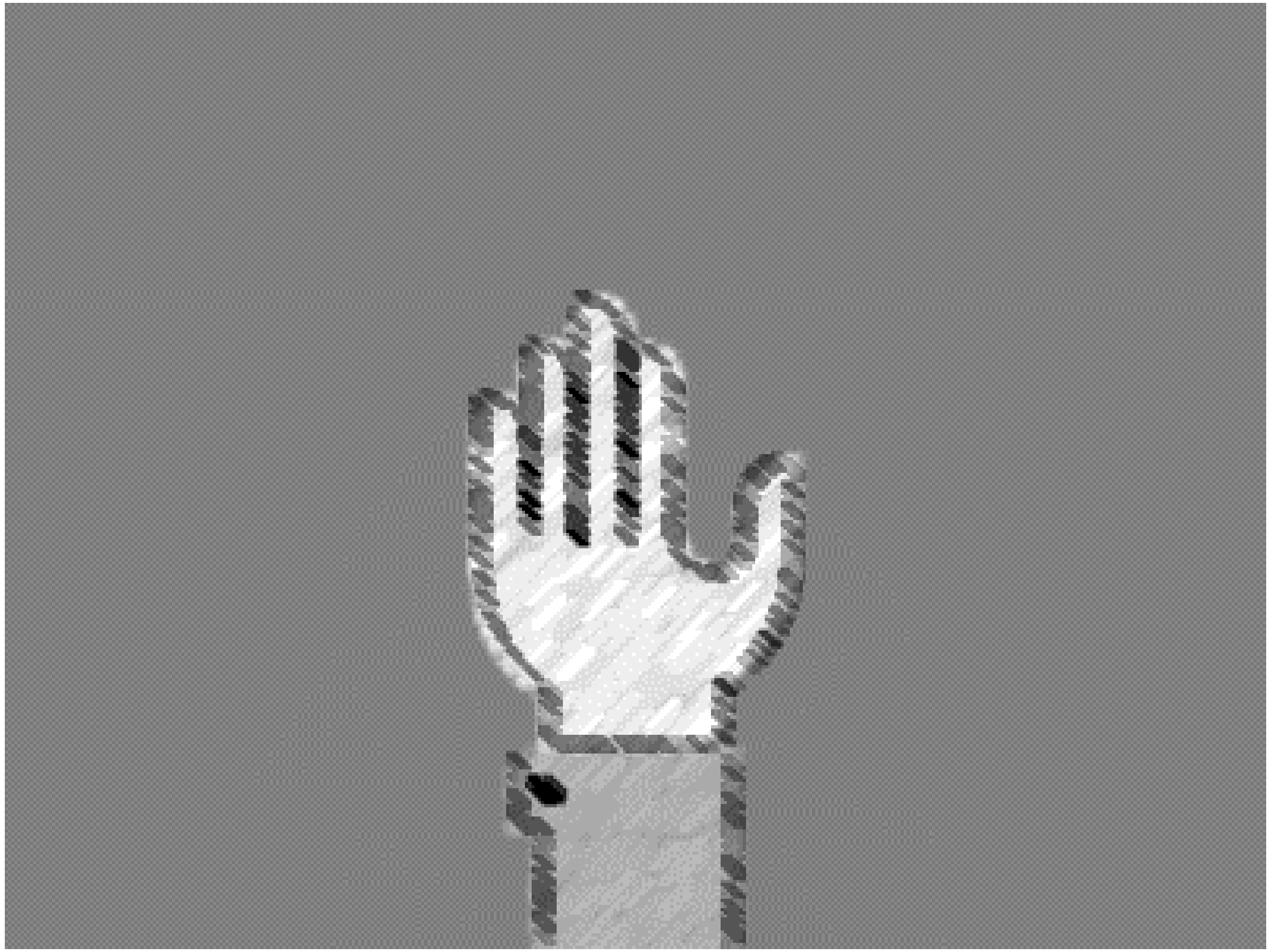
- How can we use "`foldr`" and "`gt`" to find the maximum value in a nested list of integers?

  e.g.,

  `maxLL [[6,4,2],[-1,7],[1,3],[]]  => 7`

```
maxLL xs = let
 maxL xL = foldr gt (minBound::Int) xL
           in maxL (map maxL xs)
```

Questions?



Breakout Examples: `map, fold, filter`

```
cons0 :: Num a => [a] -> [a]
cons0 xs = 0:xs
```

- How can we use "`map`" and "`cons0`" to prepend `0` to each sublist in a given list?

  `e.g., cons0L [[1,2],[3],[4,5],[]]`

  `[[1,2],[3],[4,5],[]]   => [[0,1,2],[0,3],[0,4,5],[0]]`

```
consX :: a -> [a] -> [a]
consX x xs = x:xs
```

- How can we use "`map`" and "`consX`" to prepend an *item* to each sublist in a given list?

  `e.g., consXL "z" [["1"],["2","3"],[]]`

  `[["1"],["2","3"],[]] => [["z","1"],["z","2","3"],["z"]]`

Combining Multiple Recursive Patterns

- How can we use "`map`", and "`filter`" to find the sum of sqrt of elements in a list of integers? -- list *may* contain negative integers…

```
sumOfSquareRoots :: (Ord a, Floating a) => [a] -> a
sumOfSquareRoots xs = sum (map sqrt (filter (\x -> x>0 ) xs))
```

- How can we find the sum of sqrt of elements in a nested list of integers?

  e.g. `[[25,16,-9],[0,9,-5],[]] => 12.0`

```
sumOfSqrtNested :: (Ord a, Floating a) => [[a]] -> a
sumOfSqrtNested xs = sum (map sumOfSquareRoots xs)
     where sumOfSquareRoots xL = sum (map sqrt (filter (\x -> x>0 ) xL))
```

Function application with lower precedence

- Parameterized functions, such as map, filter, and foldr/foldl, are often called *combinators*
  - We call the one-line definition of sumOfSquareRoots combinator-based
  - A combinator-based expression tends to involve *many parentheses*
  - To avoid this, Haskell's Prelude provides some more combinators
  - For example:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

- `$` is right associative and has *precedence level* `0` - which is the weakest level of precedence in Haskell

```
sqrt (average 60 30)
```

```
sqrt $ average 60 30
```

- first evaluate the application of average to `60` and `30`, and then, apply `sqrt` to the result

```
sumOfSquareRoots xs = sum (map sqrt (filter (\x -> x>0 ) xs))
```

```
sumOfSquareRoots xs = sum $ map sqrt $ filter (\x -> x>0) xs
```

Function composition

```
sumOfSquareRoots :: (Ord a, Floating a) => [a] -> a
sumOfSquareRoots xs = sum (map sqrt (filter (\x -> x>0 ) xs))
```

```
sumOfSquareRoots xs = sum $ map sqrt $ filter (\x -> x>0) xs
```

- We would like to drop the `xs` parameter in `sumOfSquareRoots` and *create a* *__curried function__*

```
sumOfSquareRoots = sum $ map sqrt $ filter (\x -> x>0)
```

This wont work (will give a compiler error). `filter`, `map`, and `sum` are nested function calls

- Function composition allows us to apply `filter`, `map`, and `sum` as a pipeline

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

The composition `f . g` of two functions `f` and `g` produces a new function that given an argument `x` first applies `g` to `x`, and then, applies `f` to the result of that first application

```
sumOfSquareRoots = sum . map sqrt . filter (\x -> x>0)
sumOfSquareRoots [-1,4,-4,-3,25,16,-9]   -- returns 11.0
```

`sumOfSquareRoots` as a partial function.

Questions?