19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Higher Order Functions

HIGHER ORDER FUNCTIONS

- Functions that can take functions as parameters and return functions as return values are called higher order functions.
- Curried Functions:
  - Every function in Haskell officially only takes one parameter.
  - All the functions that accepted several parameters so far have been curried functions.
  - Example: max 4 5

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

So how is it possible that we defined and used several functions that take more than one parameter so far?

- Doing max 4 5 first creates a function that takes a parameter and returns either 4 or that parameter, depending on which is bigger.
- Then, 5 is applied to that function and that function produces our desired result.

ANALYZING MAX FUNCTION

- max 4 5 – putting space between 4 and 5 is just function application
- space is sort of like an operator and it has the highest precedence
- Let's examine the type of max.

  ghci> :t max

  max :: Ord a => a -> a -> a
- This can also be written as max :: (Ord a) => a -> (a -> a).
  - That could be read as: max takes an a and returns (that's the ->) a function that takes an a and returns an a.
  - That's why the return type and the parameters of functions are all simply separated with arrows.
  - If we call a function with too few parameters, we get back a partially applied function, meaning a function that takes as many parameters as we left out.

EXAMPLE:

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

- What really happens when we do multThree 3 5 9 or ((multThree 3) 5) 9?
- First, 3 is applied to multThree, because they're separated by a space. That creates a function that takes one parameter and returns a function.
- Then 5 is applied to that, which creates a function that will take a parameter and multiply it by 15.
- 9 is applied to that function and the result is 135.
- Remember that this function's type could also be written as multThree :: (Num a) => a -> (a ->(a -> a)) . The thing before the -> is the parameter that a function takes and the thing after it is what it returns.
- So our function takes an a and returns a function of type (Num a) => a -> (a ->
- a).
- Similarly, this function takes an a and returns a function of type (Num a) => a -> a. And this function, finally, just takes an a and returns an a.

EXECUTING THE MULTTHREE EXAMPLE:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

- By calling functions with too few parameters, so to speak, we're creating new functions on the fly.

MULTTHREE 3 4 AND GHCI

- What happens if we try to just do multThree 3 4 in GHCI instead of binding it to a name with a let or passing it to another function?

```
ghci> multThree 3 4
<interactive>:68:1: error:
    * No instance for (Show (Integer -> Integer))
        arising from a use of `print'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
```

- GHCI is telling us that the expression produced a function of type a -> a but it doesn't know how to print it to the screen.
- Functions aren't instances of the Show typeclass, so we can't get a neat string representation of a function.
- When we do, say, 1 + 1 at the GHCI prompt, it first calculates that to 2 and then calls show on 2 to get a textual representation of that number.
- And the textual representation of 2 is just the string "2", which then gets printed to our screen.

ANOTHER EXAMPLE

- What if we wanted to create a function that takes a number and compares it to 100?

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

compareWithHundred x = compare 100 x

ghci> compareWithHundred 99

GT
```

- Notice that the x is on the right hand side on both sides of the equation. Now let's think about what compare 100 returns.
- It returns a function that takes a number and compares it with 100.

COMPAREWITHHUNDRED REWRITTEN

- We can rewrite this as:

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

- The type declaration stays the same, because compare 100 returns a function. Compare has a
- type of (Ord a) => a -> (a -> Ordering) and calling it with 100 returns a (Num a, Ord a) => a -> Ordering.
- The additional class constraint sneaks up there because 100 is also part of the Num typeclass.

MAKING INFIX FUNCTIONS TO BE PARTIALLY APPLIED USING SECTIONS

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
ghci> divideByTen 200
20.0
```

To section an infix function, simply surround it with parentheses and only supply a parameter on one side. That creates a function that takes one parameter and then applies it to the side that's missing an operand.

Calling, say, divideByTen 200 is equivalent to doing 200 / 10, as is doing (/10) 200.

A FUNCTION THAT CHECKS IF A CHARACTER SUPPLIED TO IT IS AN UPPERCASE LETTER:

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

SECTIONS AND USING –

- From the definition of sections, (-4) would result in a function that takes a number and subtracts 4 from it.
- However, for convenience, (-4) means minus four.
- So if you want to make a function that subtracts 4 from the number it gets as a parameter, partially apply the subtract function like so: (subtract 4).
- Example:

```
subtract4 :: (Num a) => a -> a

subtract4 = (subtract 4)
```

FUNCTIONS AS ARGUMENTS - EXAMPLE

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

- first parameter is a function (of type a -> a) that takes something and returns that same thing.
- second parameter is something of that type (a) also and the return value is also of the same type (a)
- The function can also be Int -> Int or String -> String or whatever.
- But then, the second parameter to also has to be of that type.

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY“
ghci> applyTwice (3:) [1]
[3,3,1]
```

ZIPWITH REVISITED

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

- If the type declaration of a function says it accepts an a -> b -> c function as a parameter, it will also accept an a -> a -> a function, but not the other way around!
- Remember that when you're making functions, especially higher order ones, and you're unsure of the type, you can just try omitting the type declaration and then checking what Haskell infers it to be by using :t.

- The first parameter is a function that takes two things and produces a third thing.
- They don't have to be of the same type, but they can.
- The second and third parameter are lists.
- The result is also a list.
- The first has to be a list of a's, because the joining function takes a's as its first argument.
- The second has to be a list of b's, because the second parameter of the joining function is of type b.
- The result is a list of c's.

EXECUTING ZIPWITH’

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ",
["foo fighters","bar hoppers","baz
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> 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]]
```

- A single higher order function can be used in very versatile ways.
- Imperative programming usually uses constructs like for loops, while loops, setting something to a variable, checking its state, etc. to achieve some behaviour and then wrap it around an interface, like a function.
- Functional programming uses higher order functions to abstract away common patterns, like examining two lists in pairs and doing something with those pairs or getting a set of solutions and eliminating the ones you don't need.

IMPLEMENTING FLIP

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

flip2' :: (a -> b -> c) -> b -> a -> c
flip2' f y x = f x y

ghci> flip2' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip2' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
ghci> zipWith flip2' div [2,2..] [10,8,6,4,2]
```

flip' takes a function that take a and b as arguments and returns a function that takes b and a as arguments.

MAPS ARE HIGHER ORDER FUNCTIONS

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

Map takes a function that takes an a and returns b, a list of a's and returns a list of b's.

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

- You've probably noticed that each of these could be achieved with a list comprehension.
- Map (+3) [1,5,3,1,6] is the same as writing [x+3 | x <- [1,5,3,1,6]].
- However, using map is much more readable for cases where you only apply some function to the elements of a list, especially once you're dealing with maps of maps and then the whole thing with a lot of brackets can get a bit messy.

FILTERS

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

Filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so in this case, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate.

```
ghci>  filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
```

- All of this could also be achieved with list comprehensions by the use of predicates.
- There's no set rule for when to use map and filter versus using list comprehension, you just have to decide depending on the code and the context !
- The filter equivalent of applying 50 several predicates in a list comprehension is either filtering something several times or joining the predicates with the logical && function.

A CLEANER QUICK SORT USING FILTER INSTEAD OF COMPREHENSIONS

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
let smallerSorted = quicksort (filter (<=x) xs)
biggerSorted = quicksort (filter (>x) xs)
in smallerSorted ++ [x] ++ biggerSorted
```

FIND THE LARGEST NUMBER UNDER 100,000 THAT'S DIVISIBLE BY 3829

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

- First make a list of all numbers lower than 100,000, descending.
- Then filter it by the predicate
- Because the numbers are sorted in a descending manner, the largest number that satisfies the predicate is the first element of the filtered list.
- We didn't even need to use a finite list for our starting set.
- That's laziness in action again.
- Because we only end up using the head of the filtered list, it doesn't matter if the filtered list is finite or infinite.
- The evaluation stops when the first adequate solution is found.

SUM OF ALL ODD SQUARES THAT ARE SMALLER THAN 10,000

```
ghci> takeWhile (/=' ') "elephants know how to party"
"elephants“
```

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

- First, begin by mapping the (^2) function to the infinite list [1..]
- Then filter them to get only the odd ones.
- And then, take elements from that list while they are smaller than 10,000.
- Finally, get the sum of that list.
- There’s no need to even define a function for that and can be done in one line in GHCI:

- Takes a predicate and a list and then goes from the beginning of the list and returns its elements while the predicate holds true.
- Once an element is found for which the predicate doesn't hold, it stops.

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

We could have also written this using list comprehensions too !

COLLATZ SEQUENCES

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

ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

- Take a natural number.
- If that number is even, divide it by two.
- If it's odd, we multiply it by 3 and then add 1 to that. Take the resulting number and apply the same thing to it, which produces a new number and so on.
- In essence, we get a chain of numbers.
- It is thought that for all starting numbers, the chains finish at the number 1.
- So if we take the starting number as13, we get the sequence: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
- 13*3 + 1 equals 40. 40 divided by 2 is 20, etc.
- We see that the chain has 10 terms.

TRY IT YOURSELF !

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

TRY IT YOURSELF

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

LAMBDAS

- Basically anonymous functions
- Used when some functions are needed only once
- Normally lambdas are made with the sole purpose of passing it to a higher-order function
- To make a lambda, write a \ (backslash) followed by parameters, separated by spaces
- After that comes a -> and then the function body
- Lambdas are usually surrounded by parentheses, as they extend all the way to the right.
- Example:

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

- Lambdas are expressions, that's why we can just pass them like that.
- The expression (\xs -> length xs > 15) returns a function that tells us whether the length of the list passed to it is greater than 15.

LAMBDA – FEATURES

Partial application vs lambda

- The expressions map (+3) [1,6,3,2] and map (\x -> x + 3)[1,6,3,2] are equivalent
- Both (+3) and (\x -> x + 3) are functions that take a number and add 3 to it.
- Making a lambda in this case is inefficient since using partial application is much more readable.

Lambdas can take any number of parameters

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

PATTERN MATCHING USING LAMBDAS

- Like normal functions, lambdas can be used for pattern matching.
- The only difference is that several patterns cannot be defined for one parameter, like making a [] and a (x:xs) pattern for the same parameter and then having values fall through.
- If a pattern matching fails in a lambda, a runtime error occurs, so care must be taken!
- Example:

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

Lambdas are normally surrounded by parentheses unless we mean for them to extend all the way to the right.

FOLDS

- A fold takes a binary function, a starting value (accumulator) and a list to fold up.
- The binary function itself takes two parameters.
- The binary function is called with the accumulator and the first (or last) element and produces a new accumulator.
- Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on.
- Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

FOLDL FUNCTION (THE LEFT FOLD)

- Folds the list up from the left side
- The binary function is applied between the starting value and the head of the list.
- This produces a new accumulator value and the binary function is called with that value and the next element, etc.

```
--mysum.hs
mysum :: (Num a) => [a] -> a
mysum [] = 0
mysum (x:xs) = x + sum xs
```

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

- \acc x -> acc + x is the binary function.
- 0 is the starting value and xs is the list to be folded up.

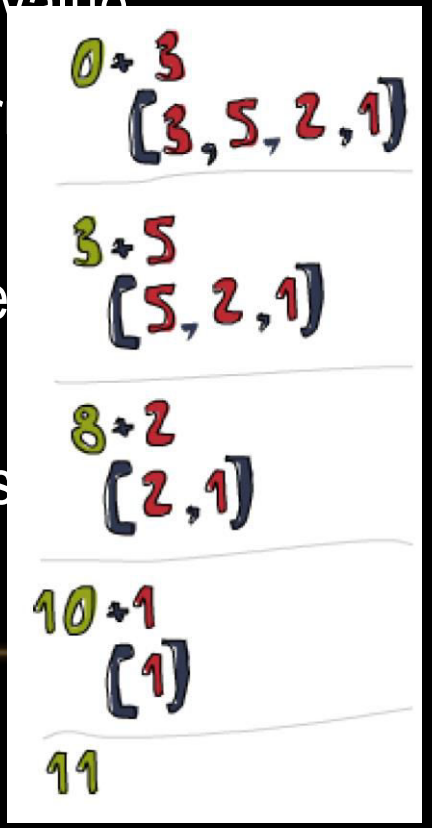
SUM IN ACTION

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

```
ghci> sum' [3,5,2,1]
11
```

- First, 0 is used as the acc parameter to the binary function and 3 is used as the x (or the current element) parameter.
- 0 + 3 produces a 3 and it becomes the new accumulator value.
- Next 3 is used as the accumulator value and 5 as the cur
  and 8 becomes the new accumulator value.
- Moving forward, 8 is the accumulator value, 2 is the curre
  new accumulator value is 10.
- Finally, that 10 is used as the accumulator value and 1 as
  element, producing an11.

The greenish brown number is the accumulator value.



FOLDR FUNCTION (THE RIGHT FOLD)

- foldr works in a similar way to the left fold except that the accumulator eats up the values from the right
- Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so \acc x -> .)
- the right fold's binary function has the current value as the first parameter and the accumulator as the second one (so \x acc -> ...)
- right fold has the accumulator on the right, because it folds from the right side.
- The accumulator value (and hence, the result) of a fold can be of any type. It can be a number,a boolean or even a new list.

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

SUM IN ACTION

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

- If we're mapping (+3) to [1,2,3] , we approach the list from the right side
- Take the last element, which is 3 and apply the function to it, which ends up being 6
- Then, prepend it to the accumulator, which is was[]. 6:[] is [6] and that's now the accumulator
- Apply (+3) to 2, that's 5 and we prepend (:) it to the accumulator
- The accumulator is now [5,6]
- Apply (+3) to 1 and prepend that to the accumulator and so the end value is [4,5,6]
- Left fold version:

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

- the ++ function is much more expensive than :, so we usually use right folds when we're building up new lists from a list.

DIFFERENCE BETWEEN LEFT AND RIGHT FOLDS

- Right folds work on infinite lists, whereas left ones don't!
- If you take an infinite list at some point and you fold it up from the right, you'll eventually reach the beginning of the list.
- However, if you take an infinite list at a point and you try to fold it up from the left, you'll never reach an end!

FOLDL1 AND FOLDR1

- The foldl1 and foldr1 functions work much like foldl and foldr, only you don't need to provide them with an explicit starting value.
- They assume the first (or last) element of the list to be the starting value and then start the fold with the element next to it.
- For Example: the sum function can be implemented as: `sum = foldl1 (+) .`
- Because they depend on the lists they fold up having at least one element, they cause runtime errors if called with empty lists.
- foldl and foldr, on the other hand, work fine with empty lists.
- When making a fold, think about how it acts on an empty list.
- If the function doesn't make sense when given an empty list, you can probably use a foldl1 or foldr1 to implement it.

SOME STANDARD LIBRARY FUNCTIONS USING FOLDS

```
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)
```

FUNCTION APPLICATION WITH $

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

- A 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)) whereas function application with $ is right-associative.
- Example: sqrt 3 + 4 + 9 is evaluated as (((sqrt 3) + 4) + 9)
- To get sqrt of (3+4+9 = 16) we must use as sqrt (3+4+9)
- Alternate: sqrt $ 3 + 4 + 9 [$ has lowest precedence of all operators]
- When a $ is encountered, the expression on its right is applied as the parameter to the function on its left.

EXAMPLE

- sum (filter (> 10) (map (*2) [2..10])) can be written using $ as

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

- Because $ is right-associative, f (g (z x)) is equal to f $ g $ z x

FUNCTION COMPOSITION

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

- Mind the type declaration. f must take as its parameter a value that has the same type as g's return value.
- Example: negate .(* 3) returns a function that takes a number, multiplies it by 3 and then negates it
- One of the uses for function composition is making functions on the fly to pass to other functions.
- Function composition is clearer and more concise than lambda

EXAMPLES

Using Lambda:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Using Function Composition:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Function composition is right-associative => f (g (z x)) is equivalent to (f . g . z) x

EXAMPLES

Using Lambda:

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

Using Function Composition:

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

FUNCTIONS WITH SEVERAL PARAMETERS

- Partially apply the functions just so much that each function takes just one parameter
- Example: sum (replicate 5 (max 6.7 8.9)) can be rewritten as (sum . replicate 5 . max 6.7) 8.9 or as sum . replicate 5 . max 6.7 $ 8.9
  - A function that takes what max 6.7 takes and applies replicate 5 to it is created.
  - Then, a function that takes the result of that and does a sum of it is created.
  - Finally, that function is called with 8.9.
- But normally, you just read that as: apply 8.9 to max 6.7, then apply replicate 5 to that and then apply sum to that.

FUNCTIONS WITH SEVERAL PARAMETERS

- replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))

  can be rewritten as

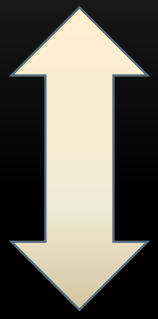
  replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]

- Start by putting the last parameter of the innermost function after a $ and then just composing all the other function calls, writing them without their last parameter and putting dots between them.

- If the expression ends with three parentheses, chances are that if you translate it into function composition, it'll have three composition operators.

POINT FREE STYLE

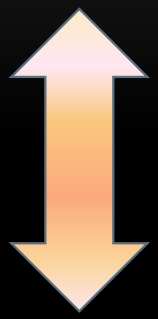
```
fn x = ceiling (negate (tan (cos (max 50 x))))
```



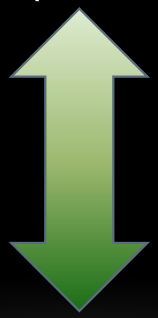
```
fn = ceiling . negate . tan . cos . max 50
```

TAKEWHILE EXAMPLE REVISITED USING COMPOSITION

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



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



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

THANK YOU