- Garbage collection
    - Why garbage collect (keep in mind different paradigms)
        - Javascript/java makes free heap space for new objects
        - C++ memory leak or a dangling reference
        - Declaeative language means what you want computer to do
    - Pros + cons
        - Automatic garbage collection
        - Pros:
            - Prevent difficult bugs
            - Worry about what's important
            - Essential in functional language; data are immutable;
        - Cons:
            - Not suitable in real-time system
            - No pointer arithmetic
        - Prerequire
            - Typed pointers
            - Known(computable) locations
    - Reference counting
        - How does it work? -> it counts how many references to the heap objects, when the reference count drop to zero, the object is unreachable and can be collected.
        - The good -> immediately available space; no long pauses; simplicity;
        - The bad -> must handle cycle, additional work every assignment, more space needed; cannot clean all memory;
    - Mark & sweep
        - How does it work?-> go through heap, mark everything useless, starting with all pointers outside heap recursively mark useful; reclaim useless blocks in heap
        - The good: deal with cycle; accuracy to mark useful or useless object; faster than count reference
        - The bad -> garbage collection run when low on memory space; garbage collector program takes up space; take a long time to traverse; heap can be big; memory fragmentation;
    - Generational
        - How does it work? Distinguish young generation(most frequency or newest or often) and tenured generation ; when an object survives a GC it is promoted to the next generation.
        - What is it based on? Weak generational hypothesis
        - The good: reduce time of GC and order objects
        - The bad: extra cost when young -> tenured
- +Lambas :are anonymous functions that are used because we need some functions only once (\); a function without a name;
    - similar: make code clear;

  - How to create them. When to use them. JS equivalent (lambda (=>)) lambda function in javascript: let boo = () =>2; cannot know this, argument; used in pure function, such as reduce, map, filter.. callback function.
- Given function, give type.
  Examples:
  - f g h x = ((h g) x) ➔ h(g,x)=f :: a -> (a->b->c)->b->c (left association)
  - f g h x = h (g x) ➔ f:: (a->b)->(b->c)->a->c
  - (.) ➔ (.) :: (b -> c) -> (a -> b) -> a -> c
  - not . fst ➔ (Bool, b) -> Bool fst :: (a,b)->a not:: Bool -> Bool
  - f x y = x == y ➔ f :: (Eq a)=>a->a->Bool
  - f x y = (x==y, x > y) ➔ f:: (Ord a)=>a->a->(Bool,Bool)
  - f x = x + 1 ➔ f:: (Num a)=> a -> a
  - f x = x ++ x➔ f:: [a] -> [a]
  - Haskell HW 1
- Given type, give function (body)
  Examples:
  - f :: (a,[b]) -> [(b,a)] ➔ f(x,ys) = [(head ys,x)] / = zip ys repeat x repeat :: a -> [a]
  - f :: (a, (a->b)) -> [b] ➔ f(x,g) = [g x];
- Lists
  - What is [4] short for? A element 4 of list
  - What can be put in lists? A list is sequence of values of the same type; no length is required
  - What is the value of (tail [‘a’] == tail [1])? Error cannot compare different types
  - Elem at index operator. [1,2,3] !! 0 = 1
  - How do I get away with defining infinite lists in Haskell? Map or take function, zip
    - take [1,2…]
    - map
- Type synonyms
  - String [Char];
    - They are just about giving some types different names so that they make more sense to someone reading our code and documentation
  - Why are they useful? Special type of lists, list operators can work on them. Write is more clear than a type of list
    - Type declarations can be nested, cannot be recursive;
    - Easy to read and understand;
    - type String = [Char]
- List comprehensions
  - How to use/construct them. Similar function in imperative langs?
  - For, while loop == list comprehensons
  - [desired results|prediction] prediction is Bool type
  - Prediction go after the binding parts and are separated from them by a comma
- Sections
  - (+2), (2/), (/2), subtract 4,(4-)

    - Partially applied function, meaning a function that takes as many parameters as we left out.
    - Must have parentheses infix function
- Overloaded vs polymorphic functions
    - What does that mean? What’s the difference?
        - Polymorphic: type contains one or more type variables
        - A polymorphic function is called overloaded if its type contains one or more class constraints
    - How can you tell the difference between a polymorphic and an overloaded function in Haskell?
        - The difference is that polymorphism basically means that there is one algorithm for different types of operands whereas overloading means that for one symbol there are different implementation.
- Pattern matching
    - General how to use.
    - Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.
    - The patterns will be checked from top to bottom and when it conforms to a pattern.
- Function application
    - Precedence?
        - Highest precedence putting a space between two things
        - Lowest precedence $
    - How to do function application in Haskell
        -
- Haskell Garbage Collection
    - Is it needed?
        - Yes produce temporary data in immutablility
    - What does it use? (generational)
    - What makes generational so easy in Haskell
        - Immutable data has no pointer
- What is the type of
    - map:: (a->b) ->[a]->[b]
    - foldr :: (a->b->b)->b->[a]->b
    - foldr f v [] = v
    - foldr f v (x:xs) = f x (foldr f v xs)
    - Foldl :: (b->a->b)->b->[a]->b
    - What is different about foldr vs foldr’ and foldl vs foldl’
        - Foldr’ is more strict than foldr so that it prevents overflow
- Upper and lower case letters. Where can they be used?
    - Lower case ➔ argument, function names; type variables
    - Upper case➔ Type and constructor names
- Currying
    - Functions that take their one argument at a time
- Practice: (include type WITH class constraints if necessary)

- Write zip
  - zip :: [a] -> [b] -> [(a,b)]
  - zip [] _ = []
  - zip _ [] = []
  - zip (x:xs) (y:ys) = (x,y):zip xs ys
- Length
  - length :: [a] -> Int // Num p => [a] -> p
  - length [] = 0
  - length (_:xs) = 1 + length xs
  - OR length xs = sum [1|_<-xs]
  - OR length xs = foldr (_ n -> 1+n) 0 xs
- elemIndex' -- define a function elemIndex' which return `Just index` where index is the index of the first occurance or `Nothing` if it is not in the list.
  - elemIndex' :: a->[a]-> Maybe Int
  - elemIndex x ys = if index == length ys then Nothing else Just Index
    - where index = foldl(\y acc-> if x == y then 0 else acc+1) 0 ys

  filter --takes a function that maps elements to a Bool, a list of elements and return a list that is all the elements which satisfy the function
  - filter:: (a->Bool)->[a]->[a]
  - filter _ [] = []
  - filter p (x:xs)

        | p x = x:filter p xs
        |otherwise = filter p xs

    filter f xs= [x|x<-xs, f x]
- map --takes a function that maps elements from some domain to some range, a list of elements in the domain and returns each of those elements mapped to the range.
  - map :: (a->b)->[a]->[b]
  - map _ [] = []
  - map f (x:xs) = f x : map f xs
  - map f xs = [f x | x<-xs]
- Reverse --take a list and reverse the order.
  - reverse [] = []
  - reverse [x] = [x]
  - reverse xs = (last xs): reverse (init xs)
  - Or reverse (x:xs) = reverse xs ++ [x]

Using foldr to define functions like sum, and, etc

sum' = foldr (+) 0

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

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

```
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)
```

- Referential Transparency
    - Definition
        - given the same function and the same input value, you will always receive the same output
        - if a function is called twice with the same parameters, it's guaranteed to return the same result
    -
    - Then how is randomness achieved?
        - Different seed
- Data types
    - How do you create your own? (syntax)
        - data Name = Firstname | Lastname; The type of name has two values called firstname and lastname;
        - date Type = Value constructors;
        - data Shape = Circle Float Float Float | Rectangle Float Float Float Float
            - Float means parameter of Circle
    - How do you pattern match on it?

        ```
        surface :: Shape -> Float
        surface (Circle _ _ r) = pi * r ^ 2
        surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) *
        ```
        -

Data Bool = True|False