Pattern matching -> specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.
guards -> a way of testing whether some property of a value (or several of them) are true or false. (similar to if statements)
=> -> class constraint thing. BASICS: Ord (compare, (<), (<=), (>=), (>), max, min) , Eq(==, /=), Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen, enumFromTo, enumFromThenTo)
Higher order functions -> can take functions as parameters and return functions as return values.
Polymorphic functions -> functions that have type variables
typeclass -> sort of interface that defines some behavior. If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes.

LISTS / BASICS

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

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)

reverse :: [a] -> [a]
reverse = foldl(flip(:))[]
```

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

```
(!!) :: [a] ->Int-> a
xs !! n | n < 0 = error
[] !! _ = error
(x:_) !! 0 = x
(_:xs) !! n = xs!!(n-1)
```

```
foldl :: (a -> b -> a) -> a ->[b]->a
foldl f z [] = z
foldl f z (x:xs)= foldl f (f z x) xs
foldr :: (a -> b -> b)->b -> [a] ->b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs

concatMap :: (a->[b])->[a]->[b]
concatMap f = concat . map f
```

```
num :: Eq a => [a] -> [a]
>>removes duplicate elems

delete :: Eq a=>a->[a]->[a]
>> removes 1st occurrence

(\\) :: Eq =>[a]->[a]->[a]
>> list difference

Union :: Eq a=> [a]->[a]->[a]
>> returns union of list
```

```
take :: Int -> [a] -> [a]
take n _
  | n <= 0 = []
take _ [] = []
take n (x:xs) =
  x:take (n-1) xs

drop n xs
  | n < 1 = xs
drop n (_:xs) =
    drop (n-1) xs
```

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

insert(Ord a)=> a->[a]->[a]
insert x [] = [x]
insert x zs@(y:ys) =
  if x < y then x:zs
  else y:(insert x ys)  ..
filterAtPositions :: (Int
-> Bool) -> [a] -> [a]
filterAtPositions p xs = [x
| (x,i) <- (zip xs [0..]),
p i]
```

```
splitAt :: Int->[a]->[a]
splitAt n xs =
  (take n xs,drop n xs)

zip :: [a]->[b]->[(a,b)]
zip = zipWith (,)
split :: [a] -> ([a],[a])
split xs =
(filterAtPositions even
xs, filterAtPositions odd
xs)
```

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

unzip :: [(a,b)] -> ([a],[b])
unzip = foldr (\(x,y)~(xs,ys) ->
  (x:xs, y:ys)) ([], [])

Unzip3 :: [(a,b,c)] -> ([a], [b],[c])
Unzip3 = foldr(\(a,b,c)~(as,bs,cs) ->
  (a:as,b:bs,c:cs)) ([],[],[])
```

```
curry :: ((a,b)->c)->a->b->c
curry f x y = f (x,y)

uncurry :: (a->b->c)->((a,b)->c)
uncurry f p = f (fst p) (snd p)

fst  :: (a,b) -> a
snd  :: (a,b) -> b
```

```
maximum, minimum
:: (Ord a) => [a] -> a
maximum, minimum [] =
  error "empty list"
maximum xs = foldl1 max xs
minimum xs = foldl1 min xs
```

```
elem, notElem
:: (Eq a) => a->[a]->Bool
elem x = any (== x)
notElem x = all (/= x)
```

```
iterate :: (a ->a)->a-> [a]
iterate f x =
  x:iterate f (f x)
cycle  :: [a] -> [a]
cycle [] = error "empty"
cycle xs = xs'
  where xs' = xs ++ xs'
```

```
suffixes :: [a] -> [[a]]
suffixes [] = [[]]
suffixes xs =
  xs:(suffixes (tail xs))
```

```
prefixes :: [a] -> [[a]]
 prefixes xs =
  map(reverse(suffixes(reverse xs)))
```

@: Matches when the pattern matches. Additionally binds the name to the whole expression. Ex: **`z@(x:xs)`** binds `(x:xs)` to **`z`**

GENERAL STUFF

Algebraic data types

```
Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show)
Bool = False| True deriving (Eq,Ord,Enum,Read,Show,Bounded)
Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show,Eq)
```

Make:

```
data Name = Apple a b | Orange a b
using :: Name -> a
using (Apple x y)  = x * y
using (Orange x y) = x / y
```

Type classes (Ex.):

```
:t foldr :: Foldable t => (a -> b-> b) -> b -> t a -> b
:t insert :: Ord a =>  a -> [a] -> [a]
:t foldr insert :: (Ord a, Foldable t) => [a] -> t a -> [a]
:t even :: Integral a => [a] -> a
:t type of partially applied functions
:t filter :: (a -> Bool) -> [a] -> [a]
:t filter even~~~  :: Integral a => [a] -> [a]
:t foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
:t (:) :: a -> [a] -> [a]
:t map (\x -> (x,x)) :: [t] -> [(t,t)]
:t map (\x -> x) :: [b] -> [b]
:t map :: (a -> b) -> [a] -> [b]
:t (\x -> x) :: r -> r
```

Lambda / List comprehensions:

`(\x -> f)` ~~ **Given x, apply to f, only happens once**
Ex. `(\x -> x + 4) 5 == 9`

`[f x | x <- xs, b]` ~~ apply f to elem x in list xs if b is true
Ex: `let xs = [1..3] in [x * 2 | x <- xs, even x] == [4]`

Kinds:

Think of kinds as the type for a data constructor.
--Primitives are usually * (Int, Float, String, Double)

```
:k Tree ~~~  :: * -> *   :k Maybe * -> *
:k Tree Char ~~~ :: *    :k Maybe Int/Char *
:k Either ~~~ :: * -> * -> *
:k State ~~~ :: * -> * -> *
:k Ord, Integral, Num, Enum ~~~ :: * -> GHC.Prim.Constraint
:k State [Int] ~~~ :: * -> *
```
:k Either Int Char :: *

Remember:

```
head :: [a] -> a     tail :: [a] -> [a]    length :: [a] -> Int
head (x:_) = x       tail (_:xs) = xs      length [] = 0
head [] = error      tail [] = error       length (_:y) =
last :: [a] -> a     init :: [a] -> [a]      1 + length y
last[x] = x          init [x] = []         null :: [a]-> Bool
last(_:xs)=last xs   init (x:xs) =         null [] =  True
last [] = error        x:init xs           null (_:_) = False
                     init [] = error
```

!!! DO NOT use guards for pattern matching of algebraic data types !!!

Sorting

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs)=quicksort left ++ [x] ++ quicksort right
  where left   = filter (\y -> y < x) xs
        right  = [y | y <- xs, y >= x]

insertsort :: (Ord a) => [a] -> [a]
insertsort = foldl insert []
insert :: (Ord a) => [a] -> a -> [a]
insert [] x = [x]
insert ys@(x:xs) z
     | z < x        = z : ys
     | otherwise    = x : (insert xs z)

mergesort :: (Ord a) => [a] -> [a]
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort ls) (mergesort rs)
  where ls = take (length xs `div` 2) xs
        rs = drop (length xs `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge as@(x:xs) bs@(y:ys) =
  if x < y then x:merge xs bs
           else y:merge as ys

bubbleSort :: (Ord a) => [a] -> [a]
bubbleSort [] = []
bubbleSort x = (iterate swap x) !! (length x) -1)
  where swap [x] = [x] swap (x:y:zs)
    | x > y     = y swap (x:zs)
    | otherwise = x : swap (y:zs)

selectionSort :: (Ord a) => [a] -> [a]
selectionSort [] = []
selectionSort xs = minx : (selectionSort zs)
  where minx = minimum xs
        zs   = delete minVal xs
newtype Store a b = Store(Map.Map a b)deriving (Show,Eq)

emptyStore :: Ord a => Store a b
emptyStore = (Store Map.empty)
insertStore :: Ord a => a -> b -> Store a b -> Store a b
insertStore k v (Store sto) = Store (Map.insert k v sto)
lookupStore :: Ord a => a -> Store a b -> Maybe b
lookupStore k (Store sto) = Map.lookup k sto
```

```
depth :: Tree a -> Integer
depth Nil = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
collapse :: Tree a -> [a]
collapse Nil = []
collapse (Node x t1 t2)=collapse t1 ++ [x] ++ collapse t2

sumTree :: Num a => Tree a -> a
sumTree Nil = 0
sumTree (Node n left right) =
  n + (sumTree left) + (sumTree right)
```

Trees

```
traverse :: Tree a -> [a]
traverse Nil = []
traverse (Node x left right) =
  [x] ++ (traverse left) ++ (traverse right)

occurs :: Eq a => a -> Tree a -> Bool
occurs _ Nil = False
occurs x (Node v left right) =
  (x == v) || occurs x left || occurs x right

insTree :: Ord a => a -> Tree a -> Tree a
insTree x Nil = (Node x Nil Nil)
insTree x (Node v lt rt)
    | x == v      = Node x lt rt
    | x > v       = Node v lt (insTree x rt)
    | otherwise   = Node v (insTree x lt) rt

successor :: Ord a => a -> Tree a -> Maybe a
successor _ Nil = Nothing
successor x (Node v l r)
   | x >= v    = successor x r
   | otherwise = Just (sccr v l)
   where sccr s Nil = s
         sccr s (Node v l r) =
           if x >= v then sccr s r
           else sccr v l
replace :: Eq a => a -> a -> Tree a -> Tree a
 replace _ _ Nil = Nil
 replace x y (Node z left right)
    | (z == x)  = Node y left' right'
    | otherwise = Node z left' right'
    where left'   = (replace x y left)
          right'  = (replace x y right)
delete val Nil = Nil
delete val (Node v t1 t2)
   | (val > v)  = Node v t1 (delete val t2)
   | (val < v)  = Node v (delete val t1) t2
   | t2 == Nil  = t1
   | t1 == Nil  = t2
   | otherwise  = join t1 t2
minTree :: Ord a => Tree a -> Maybe a
minTree Nil = Nothing
minTree (Node v t1 _)
   | t1 == Nil  = Just v
   | otherwise  = minTree t1
join :: (Ord a) => Tree a -> Tree a -> Tree a
join t1 t2 = Node mini t1 newt
  where (Just mini) = minTree t2
        newt        = delete mini t2
count :: Eq a => a -> [a] -> Int
count x = length . filter (x==)
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs                .. /with folds
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False
 ys
```

Prolog	programs are a collection of **Facts** and **Rules** that we can **Query.** Focuses on describing facts and relationships about problems instead of solving a problem
Query	Query = "?-" when using a Prolog terminal you may give a command within the current working Prolog file. If the given query cannot be matched to anything, an error will be returned.
Facts	Fact = police(wee, woo), where wee and woo are **atoms** (constants, can't start with _, first letter lowercase), police is a **predicate**.
Rules	**:-** ~~ if right is true, then left is true EX: cow(moo) :- isSound(moo) /// used when you want to say that a fact depends on a group of facts
Database	File where Facts and Rules are stored (AKA Knowledge Base) ---- consult('file.pl'). or [file]. // if you plan to modify database, mark as dynamic -- dynamic(wombo/2) for wombo(w, m)
	Clauses: Facts and Rules /// **Variables:** object we can't name at time of execution, begins with uppercase letter or _ // same name =/= same variable used in 2 different questions
Structure	An object made up from many other **objects (**components), Allow us to add context about what an object is to avoid confusion. Have functor followed by list of arguments
Trace	Use this to see how prolog evaluates queries one at a time /// **notrace** turns off trace
recursion	Cycles through possible results until related returns a true // USE RECURSION TO LOOP
lists	Store atoms, complex terms, variables, numbers, and other lists. Used to store data that has an unknown number of elements. Add items with **| (list constructor)**

Backtracking examples

In database:

```
eats(fred,pears).
eats(fred,tbone_steak).
eats(fred,apples).

?- eats(fred,FoodItem).
FoodItem = pears; FoodItem = tbone_steak; FoodItem = apples
```

In Rules:

```
host(X):-
  birthday(X),
  happy(X).

birthday(tom).
birthday(fred).
birthday(helen).
happy(mary).
happy(jane).
happy(helen).

?- host(Who).
birthday(tom) YES, happy(tom) NO.
birthday(fred) YES, happy(fred) NO.
birthday(helen) YES, happy(helen) YES.
Who = helen.
```

Using cuts: ! is a cut

Use them for efficiency and avoidance of alternative answers

Ex 1 (without them)

```
max(X,Y,Y):-  X  =<  Y.
 max(X,Y,X):-  X>Y.
```

What's the problem? There is a potential inefficiency, explores unnecessary rules. Suppose this definition is used as part of a larger program, and somewhere along the way max(3,4,Y) is called. The program will correctly set Y=4 . But now consider what happens if at some stage backtracking is forced. The program will try to re-satisfy max(3,4,Y) using the second clause. This is completely pointless: the maximum of 3 and 4 is 4 and that's that.

Ex 2 (with them)

```
max(X,Y,Y)  :-  X  =<  Y,!.
max(X,Y,X)  :-  X > Y.
```

Note how this works. Prolog will reach the cut if max(X,Y,Y) is called and X =< Y succeeds. In this case, the second argument is the maximum, and that's that, and the cut commits us to this choice. On the other hand, if X =< Y fails, then Prolog goes onto the second clause instead.

Note that this cut does not change the meaning of the program. Our new code gives exactly the same answers as the old one, but it's more efficient.

Ex 3

```
add(Element, List, Result) :-
  member(Element, List), !,
  Result = List.

add(Element, List, [Element | List]).
```

```
interleave :: ([a],[a]) -> [a]
interleave (xs,[]) = xs
interleave ([],ys) = ys
interleave ((x:xs),(y:ys)) = x:y:(interleave (xs,ys))
occursNum :: Eq a => Tree a -> a -> Int
occursNum Nil _ = 0
occursNum (Node x left right) y =
  (if (x == y) then 1 else 0) + (occursNum left y) +
(occursNum right y)
```

General examples

```
permutation([], []).
permutation(List, [Element|Permutation]) :-
  select(Element, List, Rest),      %finds values
  permutation(Rest, Permutation).   %acts on these choices
```

Sorting:

```
insert(X, [], [X]).
insert(X, [H | T], [X, H | T]) :-
  X =< H.
insert(X, [H | T1], [H | T2]) :-
  X > H,
  insert(X, T1, T2).

insertSort([], []).
insertSort([H | T1], L) :-
  insertSort(T1, L1),
  insert(H, L1, L).
```

Accumulators:

```
rev(L, R) :- acc_rev(L, [], R).
acc_rev([], A, A).
acc_rev([H|T], A, R)  :-
   acc_rev(T, [H|A], R).
```

member.pl

```
mem(X, [X | _ ]).
mem(X, [_ | Tail]) :- (X, Tail).
```

duplicates.pl

```
remove_duplicates([],[]).
remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail), !,
  remove_duplicates(Tail, Result).
remove_duplicates([Head | Tail], [Head | Result]) :-
  remove_duplicates(Tail, Result).
```

Transitive Closure:

```
bigger(elephant, horse).
bigger(horse, donkey).
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).

concat_lists([], L, L).
concat_lists([H|T1], L2, [H|T3]) :- concat_lists(T1,L2, T3)
```

```
inter :: Ord a => Set a -> Set a -> Set a
inter (Set xs) (Set ys) = Set (int xs ys)
int :: Ord a => [a] -> [a] -> [a]
int [] ys = []
int xs [] = []
int (x:xs) (y:ys)
  | x <  y    = int xs (y:ys)
  | x == y    = x : int xs ys
  | otherwise = int (x:xs) ys
powerSet (Set y) = Set (empty : map Set (foldr
  (\x ps -> [x] : map (x:) ps ++ ps) [] y))
filterFirst :: (a -> Bool) -> [a] -> [a]
filterFirst _ [] = []
filterFirst p (x:xs)
  | not (p x)   = xs
  | otherwise   = x:filterFirst p xs
filterLast :: (a -> Bool) -> [a] -> [a]
filterLast p = reverse . (filterFirst p) . reverse
```

Extras

Haskell Data Structure Example:

```
data Shape =Circle Float | Rectangle Float Float | Square Float

instance Show Shape where
  show (Circle r) = "Circle radius:" ++ (show r)
  show (Rectangle h w) =
    "Rectangle height:"++ (show h) ++ ",width:" ++ (show w)
  show (Square s) = "Square side:" ++ (show s)

area :: Shape -> Float
area (Circle r)      = pi * r * r
area (Rectangle h w) = h * w
area (Square s)      = s * s

filter (even . (\x -> x + 1)) [2,3,4,5] == [3,5]
map (\(x,y) -> x + y) [(1,2),(3,4),(5,6)] == [3,7,11]
map odd [x | x <- [1..8], even x] == [False,False,False,False]
map (\(a,b) -> (+) a b) $ zip [4,3,2,1] [1..] == [5,5,5,5]
zipWith (\x y -> 2 * x + y) [1..4] [5..8] == [7,10,13,16]
foldr min 0 [2,9,3,8,7,5,6] == 0, max == 9
foldl (flip (:)) [] [1,2,3,4,5] == [5,4,3,2,1]
foldr (\x -> (\y -> y)) 6 [1,2,3,4,5] == 6
take 3 $ filter (even . (\n -> 2*n)) [1..] == [1,2,3]
map (\x -> 2*x) [2,4,8,16] == [4,8,16,32]
foldr (:) [] [1,2,3,4,5] == [1,2,3,4,5]
```

These can easily be converted to maximum

```
minimum []     = error "empty list"
minimum [x]    = x
minimum (x:xs) = min x (minimum xs)
minimum_tail :: Ord a => [a] -> a
minimum_tail []     = error "empty list"
minimum_tail (x:xs) = minimum_iterator x xs
minimum_iterator :: Ord a => a -> [a] -> a
minimum_iterator m []     = m
minimum_iterator m (x:xs) = minimum_iterator (min m x) xs
minimum_foldl :: Ord a => [a] -> a
minimum_foldl []     = error "empty list"
minimum_foldl (x:xs) = foldl min x xs
filterSet :: (a -> Bool) -> Set a -> Set a
filterSet p (Set xs) = Set (filter p xs)
subSet :: Ord a => Set a -> Set a -> Bool
subSet (Set xs) (Set ys) = subS xs ys
subS :: Ord a => [a] -> [a] -> Bool
subS [] ys = True
subS xs [] = False
subS (x:xs) (y:ys)
  | x <  y    = False
  | x == y    = subS xs ys
  | otherwise = subS (x:xs) ys
union :: Ord a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set (uni xs ys)
uni :: Ord a => [a] -> [a] -> [a]
uni [] ys = ys
uni xs [] = xs
uni (x:xs) (y:ys)
  | x <  y    = x : uni xs (y:ys)
  | x == y    = x : uni xs ys
  | otherwise = y : uni (x:xs) ys
```