findall(Object,Goal,List).

produces a list List of all the objects Object that satisfy the goal Goal. Often Object is simply a variable, in which case the query can be read as: Give me a list containing all the instantiations of Object which satisfy Goal.

Predicate:
min_list(List_input, Out_element); max_list
sublist, subtract, is_list,
member(X, List) → With instantiated list, it will return member of that list

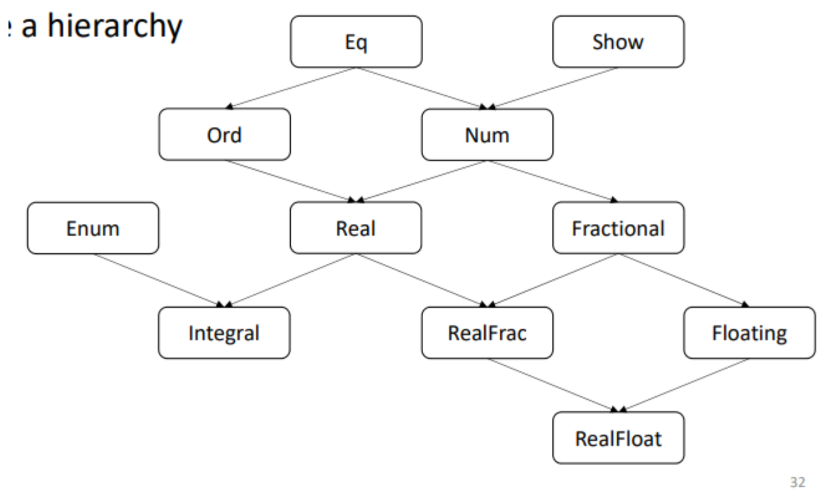
Prolog: CAPTAIL for Variables

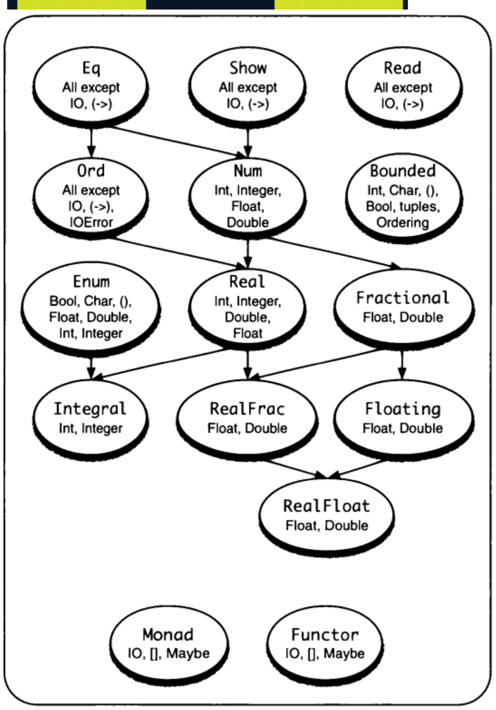
- Find the list of Activities that give the minimum result
solve(CurrentState, Action, Time)

OldStateState= state(OldLeft, OldRight);
Find all the list of time; Choose the minimum time; output the action match the time

bestAnswers(CurrentState, ans(BestRoute, BestTime)):-

- findall(Time, solve(CurrentState, _, Time), TimeList),
- min_list(TimeList, BestTime),
- findall(Action, solve(CurrentState, Action, BestTime), ActionList),
- member(BestRoute, ActionList).





c `elem` ['A'..'Z']
`mod`, `rem` ; length
if True then --- else
succ, pred

sortBy, foldl, foldr (with starting condition)
foldl1; foldr1 get first element as the starting
head, tail, init, last
zip, unzip, zipWith, words, unwords

Ordering (GT, LT or EQ)

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]

mapWhile f p []      = []
mapWhile f p (x:xs)
  | p x              = f x : mapWhile f p xs
  | otherwise        = []
```

```
mapWhile (2+) (>7) [8,12,7,13,16]
⇝  2+8 : mapWhile (2+) (>7) [12,7,13,16]
⇝  10 : 2+12 : mapWhile (2+) (>7) [7,13,16]
⇝  10 : 14 : []
⇝  [10,14]
```

Class- Classtype

```
class HasLength a where
	len :: a -> Int
instance HasLength [a] where
	len x = length x

instance HasLength (a, b) where
	len _ = 2

instance HasLength (a, b, c) where
	len _ = 3
```

class (HasLength a) => CanBeEmpty a where -- <u>CanBeEmpty is child from HasLength</u>

```
	isEmpty :: a -> Bool

instance CanBeEmpty [a] where
	isEmpty [] = True
	isEmpty _ = False

instance CanBeEmpty (a, b) where
	isEmpty _ = False

instance CanBeEmpty (a, b, c) where
	isEmpty _ = False

status :: CanBeEmpty a => a -> String
status x
  | isEmpty x = "Empty"
  | otherwise = "Contains " ++ (show (len x)) ++ " elements."

class Eq a where
```

(==) :: a → a→ Bool

Define functions

```
oneLookupFirst  :: Eq a => [ (a,b) ] -> a -> b
oneLookupSecond :: Eq b => [ (a,b) ] -> b -> a
```

`oneLookupFirst` takes a list of pairs and an item, and returns the second part of the first pair whose first part equals the item. You should explain what your function does if there is no such pair. `oneLookupSecond` returns the first pair with the roles of first and second reversed.

```
data Shape = Circle Float |
          Rectangle Float Float |
          Triangle Float Float Float deriving Show


area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h
area (Triangle a b c) = sqrt(p * (p - a) * (p - b) * (p - c))
  where p = (a + b + c) / 2

instance Eq Shape where
  s1 == s2 = abs((area s1) - (area s2)) < 0.001
  -- s1 == s2 = (area s1) == (area s2)  -- this is because floating number

instance Show Shape where
  show (Circle _) = "Circle"
  show (Rectangle w h)
    | w == h    = "Square"
    | otherwise = "Rectangle"
  show (Triangle a b c)
    | a == b && b == c          = "equilateral triangle"
    | a /= b && b /= c && a /= c = ""
    | otherwise                 = "triangle"
```

```
data BinaryTree = NilT | Node Int BinaryTree BinaryTree deriving Show

height :: BinaryTree -> Int
height NilT = 0
height (Node _ left right) = 1 + max (height left) (height right)

inTree :: BinaryTree -> Int -> Bool
inTree NilT _ = False
inTree (Node x left right) target
  | x == target = True
  | otherwise   = inTree left target || inTree right target
```

```haskell
addNode :: BinaryTree -> Int -> BinaryTree
addNode NilT n = Node n NilT NilT
addNode (Node x left right) n
  | n <= x    = Node x (addNode left n) right
  | otherwise = Node x left (addNode right n)

listToSearchTree :: [Int] -> BinaryTree -> BinaryTree
listToSearchTree [] t = t
listToSearchTree (x : xs) t = listToSearchTree xs (addNode t x)

list2tree :: [Int] -> BinaryTree -> BinaryTree
list2tree values t = foldl addNode t values

treeMap :: (Int -> Int) -> BinaryTree -> BinaryTree
treeMap _ NilT = NilT
treeMap f (Node x left right) = Node (f x) (treeMap f left) (treeMap f right)

data BST a = BSTNil | BSTNode a (BST a) (BST a)

bstAddNode :: (Ord a) => BST a -> a -> BST a
bstAddNode BSTNil n = BSTNode n BSTNil BSTNil
bstAddNode (BSTNode x left right) n
  | n <= x   = BSTNode x (bstAddNode left n) right
  | otherwise = BSTNode x left (bstAddNode right n)


inOrder :: (Show a) => BST a -> String
inOrder BSTNil = ""
inOrder (BSTNode x left right) = (inOrder left) ++ show x ++ (inOrder right)
```

```haskell
data BinaryTree = NilT | Node Int BinaryTree BinaryTree deriving Show

height :: BinaryTree -> Int
height NilT = 0
height (Node _ left right) = 1 + max (height left) (height right)

inTree :: BinaryTree -> Int -> Bool
inTree NilT _ = False
inTree (Node x left right) target
  | x == target = True
  | otherwise   = inTree left target || inTree right target

addNode :: BinaryTree -> Int -> BinaryTree
addNode NilT n = Node n NilT NilT
addNode (Node x left right) n
  | n <= x    = Node x (addNode left n) right
  | otherwise = Node x left (addNode right n)

listToSearchTree :: [Int] -> BinaryTree -> BinaryTree
listToSearchTree [] t = t
listToSearchTree (x : xs) t = listToSearchTree xs (addNode t x)

list2tree :: [Int] -> BinaryTree -> BinaryTree
```

```
list2tree values t = foldl addNode t values

treeMap :: (Int -> Int) -> BinaryTree -> BinaryTree
treeMap _ NilT = NilT
treeMap f (Node x left right) = Node (f x) (treeMap f left) (treeMap f right)

data (Ord a) => BST a = BSTNil | BSTNode a (BST a) (BST a)

bstAddNode :: (Ord a) => BST a -> a -> BST a
bstAddNode BSTNil n = BSTNode n BSTNil BSTNil
bstAddNode (BSTNode x left right) n
  | n <= x   = BSTNode x (bstAddNode left n) right
  | otherwise = BSTNode x left (bstAddNode right n)


inOrder :: (Ord a, Show a) => BST a -> String
inOrder BSTNil = ""
inOrder (BSTNode x left right) = (inOrder left) ++ show x ++ (inOrder right)
```

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
```

This declaration can be read: "Any type `a` that belongs to the `Eq` type class has two defined operators, `==` and `/=`, both of which return a `Bool` when applied to two values of type `a`." Intuitively, these two operations should define equality and inequality, respectively, but nothing in the language enforces this intuition.

As another example, the `poly` function from the previous section would use the `Num` type class, whose declaration looks like the following:

```
class Num a where
   (*) :: a -> a -> a
   (+) :: a -> a -> a
   negate :: a -> a
   ... <other numeric operations> ...
```

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/07-type-classes.php

http://www.cs.tufts.edu/comp/150PLD/Notes/TypeClasses.pdf
http://andrew.gibiansky.com/blog/haskell/haskell-typeclasses/