Computer Lab Exercises Week 2  
Some preliminary code:

```haskell
module LabExerc2 where

import List

Datatype for agents:

data Agent = A | B | C | D | E deriving (Eq,Ord,Enum)

a, alice, b, bob, c, carol, d, dave, e, ernie :: Agent
a = A; alice = A
b = B; bob = B
c = C; carol = C
d = D; dave = D
e = E; ernie = E

instance Show Agent where
    show A = "a"; show B = "b"; show C = "c"; show D = "d" ; show E = "e"

Datatype for basic propositions:

data Prop = P Int | Q Int | R Int deriving (Eq,Ord)

instance Show Prop where
    show (P 0) = "p"; show (P i) = "p" ++ show i
    show (Q 0) = "q"; show (Q i) = "q" ++ show i
    show (R 0) = "r"; show (R i) = "r" ++ show i

Datatype for epistemic models with sets of designates states (candidates for “the real world”):
```
data EpistM state = Mo
  [state]
  [Agent]
  [(state,[Prop])]
  [(Agent,state,state)]
  [state]
deriving Eq

An example:

s5example :: EpistM

s5example = (Mo [0..3]
  [a..c]
  [(0,[]),(1,[P 0]),(2,[Q 0]),(3,[P 0, Q 0])]
  ([ (a,x,x) | x <- [0..3] ]++)
  [ (b,x,x) | x <- [0..3] ]++
  [ (c,x,y) | x <- [0..3], y <- [0..3] ])

1. Define a function

   powerlist :: [a] -> [[a]]

   that gives the powerlist of a given list. E.g., powerlist [1..3] should yield the list of

   [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]

   (possibly in a different order).

2. Define a function

   sortL :: Ord a => [[a]] -> [[a]]

   that sorts lists in order of increasing length, and lists of the same length by the usual

   lexicographical order. It is allowed to use auxiliary functions from the List module.

3. Define a function

   apply :: (Eq a) => [(a,b)] -> a -> b

   such that apply pairs x gives the value of x in pairs, when the list pairs is

   considered as a function. In other words, look up the pair (x,y), if it occurs, and

   return the value y. Otherwise give an error message.

   Note: This function can be used for applying the val component of an epistemic

   model to a state in the model.
4. Write a function for converting an epistemic model of type EpistM \texttt{state} to a model of type EpistM \texttt{Integer}, with a state list \([0\ldots]\).

Hint: you can use \texttt{apply (zip states [0\ldots])} for the conversion, where \texttt{states} is the list of states of the input model and \texttt{apply} is the function from the previous exercise.

5. The epistemic alternatives for agent \(b\) in state \(s\) are the states reachable through \(R_b\) from \(s\). Implement a function for extracting the epistemic alternatives for a given agent from a list of type \([(\texttt{Agent, state, state})]\). The type declaration is

\[
\text{alternatives} :: \text{Eq state} =>
\[(\texttt{Agent, state, state})\] \to \text{Agent} \to \text{state} \to [\text{state}]
\]

6. (Homework) Define a function

\[
\text{isTrueAt} :: \text{Ord state} \to \text{EpistM state} \to \text{state} \to \text{Form} \to \text{Bool}
\]

for evaluation of formulas in epistemic models. Use the following definition of formulas:

\[
data \text{Form} = \text{Top} \\
| \text{Prop Prop} \\
| \text{Neg Form} \\
| \text{Conj [Form]} \\
| \text{Disj [Form]} \\
| \text{K Agent Form} \\
| \text{EK [Agent]} Form \\
| \text{CK [Agent]} Form \\
deriving (\text{Eq,Ord})
\]

\[
\text{instance Show Form where}
\text{show Top} = "T" \\
\text{show (Prop p)} = \text{show p} \\
\text{show (Neg f)} = '-':(\text{show f}) \\
\text{show (Conj fs)} = '&': \text{show fs} \\
\text{show (Disj fs)} = 'v': \text{show fs} \\
\text{show (K agent f)} = '[':\text{show agent}++":"]++\text{show f} \\
\text{show (EK agents f)} = 'E': \text{show agents ++ show f} \\
\text{show (CK agents f)} = 'C': \text{show agents ++ show f}
\]

Hint: use code from the course slides whenever possible (see below).
For your convenience, the relevant code from the course slides is collected here:

```haskell
type Rel a = [(a,a)]

infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
    nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x = x
         | otherwise = lfp f (f x)

rtc :: Ord a => [a] -> Rel a -> Rel a
rtc xs r = lfp (\ s -> (sort.nub) (s ++ (r@@s))) i
    where i = [(x,x) | x <- xs ]

genK :: Ord state => [(Agent,state,state)] -> [Agent] -> Rel state
    genK r ags = [ (x,y) | (a,x,y) <- r, a 'elem' ags ]

eqS :: Ord a => Rel a -> a -> [a]
eqS r x = (sort.nub) [ z | (y,z) <- r, x == y ]

genAlts :: Ord state => [(Agent,state,state)] -> [Agent] -> state -> [state]
genAlts r ags s = eqS (genK r ags) s

comonK :: Ord state => [(Agent,state,state)] -> [Agent] -> [state] -> Rel state
comonK r ags xs = rtc xs (genK r ags)

comonAlts :: Ord state => [(Agent,state,state)] -> [Agent] -> [state] -> state -> [state]
comonAlts r ags xs s = eqS (comonK r ags xs) s

rel :: Agent -> EpistM a -> Rel a
rel a (Mo states agents val rels actual) =
    [ (x,y) | (agent,x,y) <- rels, a == agent ]
```