Solution to the exercises

Exercise 1:

In file `elevator.cry`:

```
type State = {  cb : [4][12],  // call buttons - 1/floor, 4 total
               cbl : [4][12],  // call button lights - 1/floor, 4 total
                eb : [4][12],  // elevator buttons - 1/floor, 4 total
               ebl : [4][12],  // elevator button lights - 1/button, 4 total
               ids : [12],     // elevator door sensor
               ods : [4][12],  // outside door sensor - 1/floor, 4 total
               mtr : [12],     // motor direction, in motion
                fs : [4][12],  // floor sensor - 1/floor, 4 total
                pe : [12] }    // person(s) in elevator
```

Exercise 2:

In file `elevator.cry`:

```
initState : State
initState = {  cb = [0,0,0,0], // no call buttons pressed
              cbl = [0,0,0,0], // all call button lights are off
               eb = [0,0,0,0], // no elevator buttons pressed
              ebl = [0,0,0,0], // elevator button lights are off
              ids = 0,         // elevator door is open
              ods = [0,1,1,1], // outside floor door is open on 1st floor only
              mtr = 0,         // elevator is stationary
               fs = [1,0,0,0], // elevator is on 1st floor
               pe = 0 }        // no one is in the elevator
```

Running in Cryptol:

```
Main> :l elevator.cry
Loading module Cryptol
Loading module Main
Main> :s base=10
Main> initState.ods
[0, 1, 1, 1]
Main> initState.ods@0
0
Main> initState.ods!0
1
```

Exercise 3:

Example of BAD state from `elevator.cry`:

```
badState : State
badState = { cb = [0,0,0,0], // no call buttons pressed
            cbl = [0,0,0,0], // all call button lights are off
             eb = [0,0,1,0], // 3rd floor elevator button is pressed
            ebl = [0,0,0,0], // elevator button lights are off
            ids = 1,         // elevator door is closed
            ods = [1,1,1,1], // outside floor doors are all closed
            mtr = 0,         // elevator is stationary
             fs = [1,0,0,0], // elevator is on 1st floor
             pe = 0 }        // no one is in the elevator
```

Example of bad transition although both states are legal:

```
initState = {  cb = [0,0,0,0],   nextState = {  cb = [0,0,0,0],
              cbl = [0,0,0,0],                 cbl = [0,0,0,0],
               eb = [0,0,0,0],                  eb = [0,0,0,1],
              ebl = [0,0,0,0],                 ebl = [0,0,0,1],
              ids = 0,                         ids = 1,
              ods = [0,1,1,1],                 ods = [1,1,1,1],
              mtr = 0,                         mtr = 1
               fs = [1,0,0,0],                  fs = [0,0,1,0],
               pe = 0 }                         pe = 1 }
```

Illegal transition because elevator suddenly jumped to floor 3 from floor 1.

Exercise 4:

Example: Create function

```
legalNextState : State -> State -> Bit
legalNextState s1 s2 = ...
```

that returns `True` if and only if `s1` and `s2` are legal `States` and the transition from `s1` to `s2` is legal. Also create function

```
legal : State -> Bit
legal s = ...
```

that returns `True` if and only if `s` is a legal `State`. Then a property to check is this:

```
elevatorCannotReverseInMotion : State -> State -> Bit
property elevatorCannotReverseInMotion s1 s2 =
   if legal s1 /\ legal s2 /\ legalNextState s1 s2 then
   ~((s1.mtr == 1 /\ s2.mtr == 2) \/ (s1.mtr == 2 /\ s2.mtr == 1)) else True
```

Legal next `States` always have `mtr` not changing or changing from 1 to 0 or from 2 to 0 or from 0 to 1 or from 0 to 2. In that case the property will prove.

Exercise 5:

The solution to the fox, chicken, corn, farmer puzzle from Lesson 2.3 is shown here as an example of what needs to be done, although this problem is much more involved. The solution is annotated to help see how the solution is crafted.

```
// Definition of state in this puzzle
type OneBank = [4]
type BankState = {left : OneBank, right: OneBank }

// Start and end states defined
startState : BankState
startState = { left = 0b1111, right = 0b0000 }
endState : BankState
endState = { left = 0b0000, right = 0b1111 }

// A final elevator simulation should have all these properties except from
// uniqueness and there is no end state unless one is specifically asked for.
// Also, in this case, the run is only for 8 states whereas the elevator
// problem will have at least 100.
solution : [8]BankState -> Bit
property solution states =
   (neighborsConsistent states) /\
   (allStatesSafeAndValid states) /\
   (allStatesUnique states) /\
   (states ! 0 == endState) /\
   (states @ 0 == startState)
```

```
// make sure, in sequence states, that the next state in sequence can be a next
// state of the current state in sequence.  This is reasonable for the elevator
// problem except here there are only 3 possible next states – that number is
// higher for the elevator problem.
neighborsConsistent states = z ! 0
  where
    z = [True]#[ ((y == nextState x 0) \/
                  (y == nextState x 1) \/
                  (y == nextState x 2)) /\ p
               | x <- states | y <- drop `{1} states | p <- z]

// if s.left is even then s.right is odd and represents the move to opposite
// bank.  Then out.left will be s.left OR s.right masked with 0bXXX1
// where 0bXXX is 1 << idx and out.right will be s.right masked with 0xXXX0
// if s.right is even then s.left is odd and represents the move to opposite
// bank.  Then out.right will be s.right OR s.left masked with 0bXXX1
// where 0bXXX is 1 << idx and out.left will be s.left masked with 0xXXX0
//
// This function returns a next state given state s.  In this puzzle there are
// only a few possibilities and all those are handled by idx.  The corresponding
// function in the elevator problem will be more complicated.
nextState : BankState -> [4] -> BankState
nextState s idx =
   if s.right % 2 == 1 then
     { left = s.left || s.right && (0b0001 || (1 << (idx+1))),
       right = s.right && 0b1110 && (~ (1 << (idx+1):[4])) }
   else
     { left = s.left && 0b1110 && (~ (1 << (idx+1):[4])),
       right = s.right || s.left && (0b0001 || (1 << (idx+1))) }

// True iff all states in sequence ‘states’ are safe states and are valid.
// This will be needed in the elevator problem as well as queries should be
// on the entire history of elevator operations which are presented as a
// sequence of States.  This should not need to be modified as long as
// allStatesSafeAndValid is written correctly
allStatesSafeAndValid : {n} (fin n) => [n]BankState -> Bit
allStatesSafeAndValid states = z ! 0
  where
    z = [True]#[ safeAndValidState y /\ x | y <- states | x <- z ]

// a state is safe and valid if no bank has a number 6 or 10 and the xor of
// both banks is 0b1111.  The puzzle has a simple way to check if a state is
// legal: an animal or corn or farmer has to be on one side of the river only.
// The elevator problem would have the same function but with a lot of logic
// checks.
safeAndValidState : BankState -> Bit
safeAndValidState s =
   ~(s.left == 10 \/ s.left == 6 \/ s.right == 10 \/ s.right == 6) /\
     s.left ^ s.right == 0xF

// returns True iff there are two or more states in sequence ‘states’ that
// are identical.  This function is not necessary for the elevator problem
allStatesUnique states = ~(z ! 0)
  where
    z = [False]#[ (states@i == states@j /\ ~(i == j)) \/ k
                | k <- z | i <- [0..7] , j <- [0..7] ]
```