Exercise 21

```haskell
       -- add random
v2 = do
    num <- randTarget
    lim <- readNumber "Guess limit"
    runGame2 num lim 1

runGame2 num lim count = do
    guess <- readNumber "Guess"
    let v = verdict num guess
    case v of
        Right m -> do
            putStrLn m
        Left m -> do
            putStrLn m
            if count < lim
            then runGame2 num lim (count+1)
            else putStrLn "Game over"


readTarget = readNumber "Target number"

randTarget = do
    putStrLn "Guess the number ????"
    g <- newStdGen
    return . fst $ randomR (1,100) g

readNumber msg = do
    putStr $ msg ++ ": "
    line <- getLine
    case readEither line :: Either String Int of
        Left e -> do
            putStrLn e
            readNumber msg
        Right n -> return n

to_rand = getStdGen >>= \g -> return $ fst $ (randomR (1,100) g :: (Int, StdGen))

verdict target guess = do
    case compare guess target of
        EQ -> Right "You win!"
        LT -> Left "Too low"
        GT -> Left "Too high"
```

```haskell
-- check the impossible number
v3 = do
    g <- newStdGen
    range <- getRange
    lim <- readNumber "Guess limit"
    let (num,_) = randomR range g
    runGameRg num (Nothing , Nothing) (<lim) 1

runGame2 num lim count = do
    guess <- readNumber "Guess"
    let v = verdict num guess
    case v of
        Right m -> do
            putStrLn m
        Left m -> do
            putStrLn m
            if count < lim
            then runGame2 num lim (count+1)
            else putStrLn "Game over"


readTarget = readNumber "Target number"

readGuess range = do
    guess <- readNumber "Guess"
    if inRange range guess
    then return guess
    else do
        putStrLn "Impossible answer"
        readGuess range

inRange (lo, hi) guess =
    maybe True (<guess) lo
    && maybe True (>guess) hi

getRange = do
    lo <- readNumber "Lower bound"
    hi <- readNumber "Upper bound"
    if lo > hi
    then do
        putStrLn "Invalid range"
        getRange
    else return (lo, hi)

verdict' target guess (lo, hi) = do
    case compare guess target of
        EQ -> Right "You win!"
        LT -> Left ("Too low", (Just guess, hi))
        GT -> Left ("Too high", (lo, Just guess))

runGameRg num range cont count = do
    guess <- readGuess range
    let v = verdict' num guess range
    case v of
        Right m -> do
            putStrLn m
        Left (m, range') -> do
            putStrLn m
            if cont count
            then runGameRg
                num range' cont (count+1)
            else putStrLn "Game over"


randTarget = do
    putStrLn "Guess the number ????"
    g <- newStdGen
    return . fst $ randomR (1,100) g

readNumber msg = do
    putStr $ msg ++ ": "
    line <- getLine
    case readEither line :: Either String Int of
        Left e -> do
            putStrLn e
            readNumber msg
        Right n -> return n

to_rand = getStdGen >>= \g -> return $ fst $ (randomR (1,100) g :: (Int, StdGen))

verdict target guess = do
    case compare guess target of
        EQ -> Right "You win!"
        LT -> Left "Too low"
        GT -> Left "Too high"
```

```haskell
--Function nRandomR
nRandomRs ::
  (RandomGen g, Random a, Integral n)
    => (a, a) -> n -> g -> ([a], g)
nRandomRs _ 0 gen = ([], gen)
nRandomRs range n gen =
  let (val, gen') = randomR range gen
      (rest, gen'') = nRandomRs range (n-1) gen'
  in (val:rest, gen'')
```