

---

Wild Haskell

---

# Contents

<b>1</b>	<b>Install Haskell and your first program Guess Number</b>	<b>1</b>
1.1	Build Tools . . . . .	1
<b>2</b>	<b>From State to Monad Transformers</b>	<b>3</b>
<b>3</b>	<b>Learning Lens</b>	<b>7</b>



# Chapter 1

## Install Haskell and your first program Guess Number

### 1.1 Build Tools

Nix Stack

[<https://docs.haskellstack.org/en/stable/README/>](<https://docs.haskellstack.org/en/stable/README/>)

make sure ‘stack –version‘ outputs latest stack version, currently ‘Version

1.6.5‘.

‘stack upgrade‘

Editor Integration

VScode with haskell-ide-engine

Haskell Syntax Highlight

[haskell-ide-engine](<https://github.com/haskell/haskell-ide-engine>)

Install on MacOS, you need to install ‘icu4c‘ on your machine. ““ git clone

<https://github.com/haskell/haskell-ide-engine> cd haskell-ide-engine make ““

““bash stack install hoogle

hoogle generate stack install ““

Stack

““bash stack version

stack upgrade ““

Your first haskell program

stack new guessNumber simple

**module Main where**

```
import Control.Monad
import Data.Ord      (compare)
import System.IO     (readLn)
```

```
import           System.Random (randomRIO)

guess :: Int -> IO ()
guess secretNumber = do
    print "guess a number"
    guessNumber <- readLn :: IO Int
    case compare guessNumber secretNumber of
        LT -> (print "Too Small!") >> (guess secretNumber)
        EQ -> print "You Win!"
        GT -> (print "Too big!") >> (guess secretNumber)

main :: IO ()
main = do
    secretNumber <- randomRIO (0, 100) :: IO Int
    guess secretNumber
```

## Chapter 2

# From State to Monad Transformers

This post aims to cover `State` course which is skipped in videos.

Basics

First, let's look what is `State`.

```
newtype State s a = State runState :: s -> (a, s)
```

In case you are not familiar with [`newtype`](<https://wiki.haskell.org/Newtype>).

It is just like 'data' but it can only \*has exactly one constructor with exactly one field in it.\* In this case 'State' is the constructor and 'runState' is the single field. The type of 'State' constructor is '(s -> a, s) -> State s a', since 'State' uses record syntax, we have a field accessor 'runState' and its type is `State s a -> s -> (a, s)`. `A State is a function from a state value s to (a produced value a, and a resulting state s).` An importance takeaway which might not so oblivious to Haskell beginner is that: 'State' is merely a function with type '`s -> (a,s)`', (a function wrapped inside constructor 'State', to be exact, and we can unwrap it using 'runState').

Secondly, 'State' function takes a 's' representing a state, and produces a tuple contains value 'a' and new state 's'.

exec

The first exercise is to implement 'exec' function, it takes a 'State' function and initial 'state' value, returns the new state.

It is pretty straightforward, we just need to apply 'State' function with 'state' value, and takes the second element from the tuple.

A simple solution can be

```
exec :: State s a -> s -> s
exec f initial = (\ (_, y) -> y) (runState f initial)
```

Since the state value 'initial' appears on both sides of '=', we rewrite it a little more point-free. 'runState f' is a partial applied function takes 's' returns '(a,s)'

```
exec f = (\(_, y) -> y) . runState f
```

```
exec f = P.snd . runState f
```

We could also write ‘exec’

```
‘exec (State f) = P.snd . f’
```

Basic Usage of State

```
module Dice where
```

```
import Control.Applicative
```

```
import Control.Monad.Trans.State
```

```
import System.Random
```

```
rollDiceIO :: IO (Int, Int)
```

```
rollDiceIO = liftA2 (,) (randomRIO (1, 6)) (randomRIO (1,6))
```

```
rollNDiceIO :: Int -> IO [Int]
```

```
rollNDiceIO 0 = pure []
```

```
rollNDiceIO count = liftA2 (:) (randomRIO (1, 6)) (rollNDiceIO (count - 1))
```

```
clumsyRollDice :: (Int, Int)
```

```
clumsyRollDice = (n, m)
```

```
  where
```

```
    (n, g) = randomR (1, 6) (mkStdGen 0)
```

```
    (m, _) = randomR (1, 6) g
```

```
-- rollDice :: StdGen -> ((Int, Int), StdGen)
```

```
-- rollDice g = ((n, m), g'')
```

```
--   where
```

```
--       (n, g') = randomR (1, 6) g
```

```
--       (m, g'') = randomR (1, 6) g'
```

```
-- use state to construct
```

```
rollDie :: State StdGen Int
```

```
rollDie = state $ randomR (1, 6)
```

```
-- use State as Monad
```

```
rollDieM :: State StdGen Int
```

```
rollDieM = do generator <- get
```

```
      let (value, generator') = randomR (1, 6) generator
```

```
      put generator'
```

```
      return value
```

```
rollDice :: State StdGen (Int, Int)
```



```
rollDice = liftA2 (,) rollDieM rollDieM
```

```
rollNDice :: Int -> State StdGen [Int]
rollNDice 0 = state (\s -> ([], s))
rollNDice count = liftA2 (:) rollDieM (rollNDice (count - 1))
```

How about draw card from a deck

```
module Deck where
```

```
import Control.Applicative
```

```
import Control.Monad.Trans.State
```

```
import Data.List
```

```
import System.Random
```

```
data Rank = One | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten | Jack | Queue | King deriving (Show, Eq, Ord)
```

```
data Suit = Diamonds | Clubs | Hearts | Spades deriving (Bounded, Enum, Show, Eq, Ord)
```

```
data Card = Card Suit Rank deriving (Show, Eq, Ord)
```

```
type Deck = [Card]
```

```
fullDeck :: Deck
```

```
fullDeck = [Card suit rank | suit <- enumFrom minBound,
                             rank <- enumFrom minBound]
```

```
removeCard :: Deck -> Int -> Deck
```

```
removeCard [] _ = []
```

```
removeCard deck index = deck' ++ deck''
```

```
    where (deck', remain) = splitAt (index + 1) deck
          deck''        = drop 1 remain
```

```
drawCard :: State (StdGen, Deck) Card
```

```
drawCard = do (generator, deck) <- get
    let (index, generator') = randomR (0, length deck) generator
    put (generator', removeCard deck index)
    return $ deck !! index
```

```
drawNCard :: Int -> State (StdGen, Deck) [Card]
```

```
drawNCard 0 = state (\s -> ([], s))
```

```
drawNCard count = liftA2 (:) drawCard (drawNCard $ count - 1)
```

How about folding a list using 'State' <https://github.com/yuanw/applied-haskell/blob/2018/monad->

transformers.mdhow-about-state

```
foldState :: (b -> a -> b) -> b -> [a] -> b
foldState f accum0 list0 =
    execState (mapM_ go list0) accum0
  where
    go x = modify' (\accum -> f accum x)
```

Why we cannot compose any two monad

```
import Control.Applicative
```

```
newtype Compose f g a = Compose (f (g a))
```

```
instance (Functor f, Functor g) => Functor (Compose f g) where
    fmap f (Compose h) = Compose ((fmap . fmap) f h)
```

```
instance (Applicative f, Applicative g) => Applicative (Compose f g) where
    pure = Compose . pure . pure
    (Compose h) <*> (Compose j) = Compose $ pure (<*>) <*> h <*> j
```

```
instance (Monad f, Monad g) => Monad (Compose f g) where
    (Compose fga) >>= m = Compose $ fga >>= \ ga -> let gfgb = ga >>= (return . m) in un
```

<https://stackoverflow.com/questions/7040844/applicatives-compose-monads-dont>

Build a composed monad by hand

```
newtype StateEither s e a = StateEither
  { runStateEither :: s -> (s, Either e a)
  }
```

Let's implement the functor instance of this typeP

References [https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State) <https://haskell.fpcomplete.com/library/rio>

# Chapter 3

## Learning Lens

Lenses, Folds, and Traversals What is a Lens ? Lenses address some part of a “structure” that always exists, either look that part, or set that part. “structure” can be a computation result, for example, the hour in time. Functional setter and getter. Data.Lens

```
data Lens s a = Lens { set  :: s -> a -> a
                      , view :: s -> a
                      }
view :: Lens s a -> s -> a
set  :: Lens s a -> s -> a -> s
```

view looks up an attribute a from s view is the getter, and set is the setter. Laws  
1. set l (view l s) s = s 2. view l (set l s a) = a 3. set l (set l s a) b = set l s b Law 1 indicates lens has no other effects since set and view in Lens both start with s ->, so we can fuse the two functions into a single function ‘s -> (a -> s, a)’.

So we could define Lens as “haskell data Lens s a = Lens (s -> (a -> s), a)  
data Store s a = Store (s -> a) s data Lens s a = Lens (s -> Store a s) “

Side bar Store Comonad url<https://stackoverflow.com/questions/8428554/what-is-the-comonad-typeclass-in-haskell> hurl<https://stackoverflow.com/questions/8766246/what-is-the-store-comonad>

Lens can form a category

Semantic Editor Combinator

The Power is in the dot

```
(.)          :: (b -> c) -> (a -> c) -> (a -> c)
(.).(.)      :: (b -> c) -> (a1 -> a2 -> b) -> a1 -> a2 -> c
(.).(.).(.)  :: (b -> c) -> (a1 -> a2 -> a3 -> b) -> a1 -> a2 -> a3 -> c
```

Detail Explantation

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



```
““ class (Functor f, Foldable f) => Traversable f where traverse :: Applicative
m => (a -> m b) -> f a -> m (f b) ““
```

```
““haskell fmapDefault :: forall t a b. Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f) ““
```

build ‘fmap’ out from traverse we can change ‘fmapDefault’

```
over l f = runIdentity . l ( Identity . f)
over traverse f = runIdentity . traverse ( Identity . f)
                  = fmapDefault f
                  = fmap f
```

type of ‘over :: ((a -> Identity b) -> s -> Identity t) -> (a -> b) -> s -> t’ type  
 Setter s t a b = (a -> Identity b) -> s -> Identity t so we could rewrite ‘over’ as over  
 :: Setter s t a b -> (a -> b) -> s -> t let’s apply setter

```
mapped :: Functor f => Setter (f a) (f b) a b
mapped f = Identity . fmap (runIdentity . f)
over mapped f = runIdentity . mapped (Identity . f)
                = runIdentity . Identity . fmap (runIdentity . Identity . f)
                = fmap f
```

### Examples

```
over mapped (+1) [1,2,3] ==> [2,3,4]
over (mapped . mapped) (+1) [[1,2], [3]] ==> [[2,3], [4]]
chars :: (Char -> Identity Char) -> Text -> Identity Text
chars f = fmap pack . mapped f . unpack
```

Laws for setters Functor Laws: 1. ‘fmap’ id = id 2. fmap f . fmap g = fmap (f . g)

Setter Laws for a legal Setter l. 1. over l id = id 2. over l f . over l g = over l (f . g)

Practices Simplest lens (1,2,3) ^ . \_2 ==> 2 view \_2 (1, 2, 3)

References SPJ Lenses: compositional data access and manipulation Edward Kmett’s NYC Haskell Meetup talk <http://comonad.com/haskell/Lenses-Folds-and-Traversals-NYC.pdf> Edward Kmett’s NYC Haskell Meetup talk slide <http://hackage.haskell.org/package/lens-tutorial-1.0.3/docs/Control-Lens-Tutorial.html> <https://www.youtube.com/watch?v=H01dw-BMmlE> <https://www.youtube.com/watch?v=QZy4Yml3LTY> <https://www.youtube.com/watch?v=T88TDS7L5DY> <http://lens.github.io/tutorial.html> [https://www.reddit.com/r/haskell/comments/9ded97/is\\_learning\\_how\\_to\\_use\\_the\\_lens\\_library\\_worth\\_it/e5hf9ai/](https://www.reddit.com/r/haskell/comments/9ded97/is_learning_how_to_use_the_lens_library_worth_it/e5hf9ai/) <https://blog.jle.im/entry/lenses-products-prisms-sums.html>