MonadReader Class (12A)

Copyright (c) 2016 - 2018 Young W. Lim.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
8/21/18

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

MonadReader Class

Installing **mtl**

sudo apt-get install cabal-install

cabal update
cabal install mtl

ghci -package such-and-such
ghc-pkg list | grep such-and-such.
ghci -hide-package <package> flag on the command line
ghc-pkg hide <package> to hide the package by default
ghc-pkg --user hide <package> home directory packages

https://stackoverflow.com/questions/50321045/could-not-find-module-control-monad-state-after-updating-mtl
https://stackoverflow.com/questions/3102164/how-do-i-get-ghci-to-see-packages-i-installed-from-cabal

Auto-lifting in **mtl** MonadReader

Each **monad** in the **mtl** is defined in terms of a type class.

Reader is an instance of **MonadReader**,
ReaderT is also an instance of **MonadReader**

anything that wraps a **MonadReader** is
also set up to be a **MonadReader**

asks and **local** functions will work without any (manual) lifting.
Other **mtl monads** behave in a similar way.

https://wiki.haskell.org/Monad_Transformers_Explained

MonadReader Class Definition

```
class Monad m => MonadReader r m | m -> r where

(ask | reader), local

ask :: m r
ask = reader id

local :: (r -> r) -> m a  -> m a

reader :: (r -> a) -> m a
reader f = do
    r <- ask
    return (f r)

asks :: MonadReader r m => (r -> a)  -> m a
asks = reader
```

See examples in **Control.Monad.Reader**.

Note, the partially applied function type **(->) r** is a simple **reader** monad.

cf)
instance (Monad m) => Monad (ReaderT r m) where

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

MonadReader instance ReaderT

```
class Monad m => MonadReader r m | m -> r where

ask :: m r
local :: (r -> r) -> m a  -> m a

reader :: (r -> a) -> m a


instance Monad m => MonadReader r (ReaderT r m) where
   ask      :: ReaderT r m r
   local    :: (r -> r) -> ReaderT r m a -> ReaderT r m a
   reader   :: (r -> a) -> ReaderT r m a
```

m

ReaderT r m

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

MonadReader Class Methods

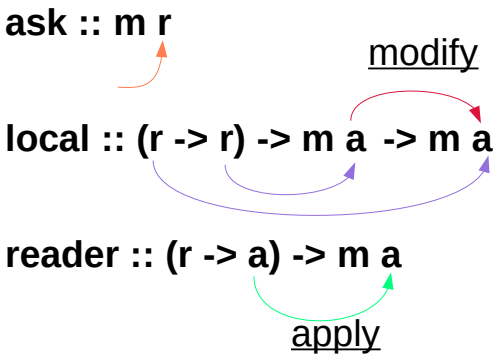
```
class Monad m => MonadReader r m | m -> r where


(ask | reader), local

ask :: m r          -- retrieves the monad environment.

local :: (r -> r)   -- the selector function to modify the environment.
      -> m a        -- reader to run in the modified environment.
      -> m a        -- executes a computation in a modified environment.

reader :: (r -> a)  -- the selector function to apply to the environment.
       -> m a       -- retrieves a function of the current environment.
```



http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

ReaderT Monad Example

```
import Control.Monad.Reader

liftReaderT :: m a -> ReaderT r m a
liftReaderT m = ReaderT (const m)

eg2 :: ReaderT Int IO String
eg2 = do
        e <- ask :: ReaderT Int IO Int
        liftReaderT $ print $ "in eg2 the env  is: " ++ (show e)
        return $ "returned value: " ++ show e


*Main> runReaderT eg2 100
"in eg2 the env is: 100"
"returned value: 100"
```

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

MonadReader – **ask**, **asks** methods

```
class Monad m => MonadReader r m | m -> r where

ask :: m r
ask = reader id

local :: (r -> r) -> m a  -> m a

reader :: (r -> a) -> m a
reader f = do
    r <- ask
    return (f r)

asks :: MonadReader r m => (r -> a)  -> m a
asks = reader
```

```
class Monad m =>  …

ask :: m r
```

retrieves the monad environment.

```
asks:: MonadReader r m =>
        (r -> a) -> m a
```

retrieves a function applied result of the current environment.

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

Reader Monad Example – **ask**, **asks**

```
import Control.Monad.Reader

stuff :: Reader Int String
stuff = do
  s <- ask
  return (show s ++ " green bottles")

stuff2 :: Reader Int String
stuff2 = asks $ \s -> (show s ++ " green bottles")

type IntRead = Reader Int

stuff3 :: IntRead String
stuff3 = asks show

stuff4 :: IntRead String
stuff4 = asks $ \s -> (show s ++ " green bottles")
```

```
*Main> print $ runReader stuff 99
"99 green bottles"

*Main> print $ runReader stuff2 99
"99 green bottles"

*Main> print $ runReader stuff3 99
"99"

*Main> print $ runReader stuff4 99
"99 green bottles"
```

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

Reader Monad Example

The purpose of Reader, instead of passing
parameters to a function

Reader is used as a **global state**, for "**constants**" etc
to avoid polluting every **function call** with **params**
(a function might only pass these **params** to other functions,
not even using them)

Modification of all these functions to use **Reader** is still necessary.

can use '**asks**' to avoid all the **do-block** boilerplate
can create an **alias** for the reader if it's used in lots of places

do-block boilerplate

```
stuff = do
  s <- ask
  return (show s ++ " green bottles")
```

alias for the reader

```
type IntRead = Reader Int

stuff3 :: IntRead String
stuff3 = asks show
```

-- See http://stackoverflow.com/questions/14178889/reader-monad-purpose

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

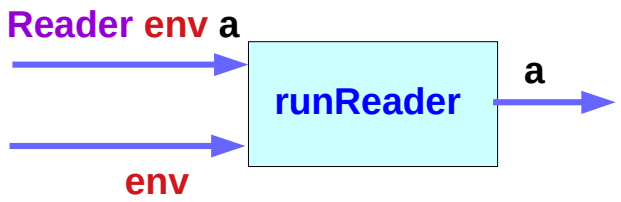
Purpose of Reader Monad

```
data Reader env a = …

instance Monad (Reader env)                    -- Reader is a monad

ask :: Reader env env                          -- get its environment
        copy

runReader :: Reader env a -> env -> a          -- to run the monad
```



https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Purpose of Reader Monad – configuration information

```
data Reader env a = …
instance Monad (Reader env)          -- Reader is a monad
ask :: Reader env env                -- get its environment
runReader :: Reader env a -> env -> a -- to run the monad
```

the **reader monad** is useful in passing (implicit) **configuration information** through a **computation** (a monadic value).

a "**constant**" in a **computation** is accessed at various points
In order to perform the same **computation** with different **values**,
use a **reader monad**

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Purpose of Reader Monad – ask, local

```
newtype Reader env a = Reader {runReader :: env -> a}


instance Functor (Reader env) where
  fmap f (Reader g) = Reader $ f . g


instance Monad (Reader env) where
  return x = Reader (\_ -> x)
  (Reader f) >>= g = Reader $ \x -> runReader (g (f x)) x


ask = Reader $ \x -> x

local f (Reader g) = Reader $ \x -> runReader g (f x)
```

Reader is just a fancy name
for **functions**!

runReader extracts
the inside function

every **Monad** is also a **Functor**:

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Purpose of Reader Monad – reader Monad

Okay, so the **reader** monad is just a function.
Why have **Reader** at all? Good question. Actually, you don't need it!

```
instance Functor ((->) env) where
  fmap = (.)

instance Monad ((->) env) where
  return = const
  f >>= g = \x -> g (f x) x
```

These are even simpler.
What is more, **ask** is just **id** and
local is just function composition in the other order!

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Purpose of Reader Monad –

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
    return a = Reader $ \_ -> a
    m >>= k = Reader $ \r -> runReader (k $ runReader m r) r

asks :: (r -> a) -> Reader r a
asks f = Reader f

ask :: Reader a a
ask = Reader id
```

https://passy.svbtle.com/dont-fear-the-reader

Purpose of Reader Monad

Expression = a **Reader**
Free variables = uses of **ask**
Evaluation environment = **Reader** execution environment.
Binding constructs = **local**

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Reader Monad – example

```
import Control.Monad.Reader

example :: String
example = runReader computation "Hello"

computation :: Reader String String
computation = do
        g <- ask
        return $ g ++ ", Haskell"

main = putStrLn example

--    Hello, Haskell
```

https://passy.svbtle.com/dont-fear-the-reader

Reader Monad – example1

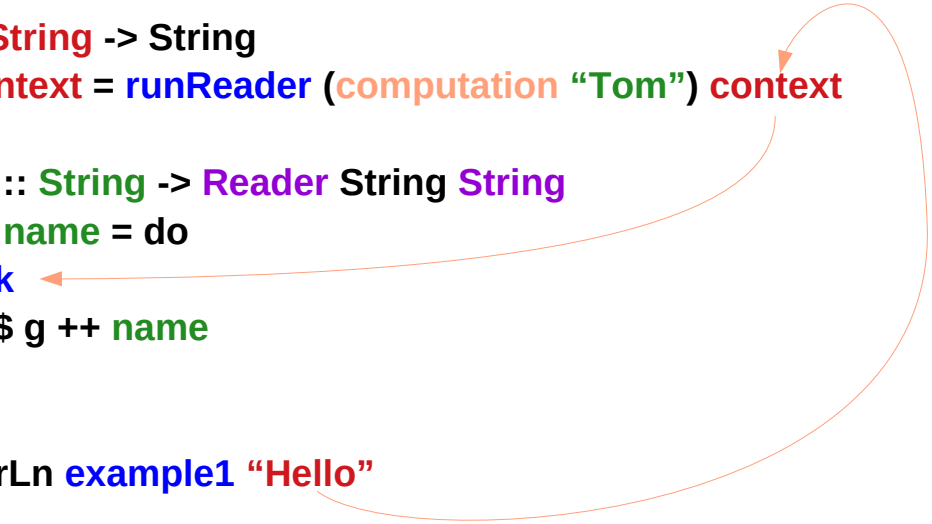
```
import Control.Monad.Reader

example1 :: String -> String
example1 context = runReader (computation "Tom") context

computation :: String -> Reader String String
computation name = do
        g <- ask
        return $ g ++ name

main :: IO ()
main = putStrLn example1 "Hello"

--    Hello, Tom
```



"Tom" : argument to computation
name : parameter of computation

computation : a function returns
    Reader String String
    monad value

g <- ask :
    access the configuration info
    with side effects

"Hello" : argument to example1
context : parameter of example1

example1 : a function that runs
    computation (a mondic value)
    with the context info

https://passy.svbtle.com/dont-fear-the-reader

Reader Monad – example2

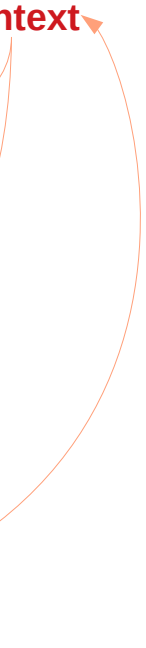
```
import Control.Monad.Reader

example2 :: String -> String
example2 context = runReader (greet "James" >>= end) context

greet :: String -> Reader String String
greet name = do
        g <- ask
        return $ g ++ ", " ++ name

end :: String -> Reader String String
end input = do
        isHello <- asks (== "Hello")
        return $ input ++ if isHello then "!" else "."

main :: IO ()
main = putStrLn $ example2 "Hello"
```



```
--    Hello, James!
```

greet "James" returns
a monadic value of the type
Reader String String

whose result has the type of
String and is passed into end
via >>= operator
the result string is taken as an
argument of end

asks (== "Hello")
asks :: (r -> a) -> m a
(== "Hello") :: (r -> a)

https://passy.svbtle.com/dont-fear-the-reader

Reader Monad – example

pricing an asset can do without any monads.
But to deal with multiple currencies,
on the fly conversion between currencies is needed.

```
type CurrencyDict = Map CurrencyName Dollars
currencyDict :: CurrencyDict
```

You can then call this dictionary in your code....but that won't work!
The currency dictionary is immutable and so has to be the same
not only for the life of your program, but from the time it gets compiled!

```
computePrice :: Reader CurrencyDict Dollars
computePrice
   = do currencyDict <- ask
      -- insert computation here
```

```
Reader env a
Reader r a
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Reader Monad – example

```
type CurrencyDict = Map CurrencyName Dollars
currencyDict :: CurrencyDict
currencyDict :: Map CurrencyName Dollars          -- Map k e

computePrice :: Reader CurrencyDict Dollars
computePrice
   = do currencyDict <- ask
      -- insert computation here

(Ord k, Read k, Read e) => Read (Map k e)

computePrice :: Reader CurrencyDict Dollars
computePrice :: Reader Map CurrencyName Dollars Dollars
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

Data Map (dictionary) Example

```
import Data.Map (Map, (!))
import qualified Data.Map as Map

main = do
   let m0  = Map.empty
   let m1  = Map.insert "k1" 7 m0
   let m   = Map.insert "k2" 13 m1
   putStrLn $ "map: " ++ show m

   let v1  = m ! "k1"
   putStrLn $ "v1: " ++ show v1
   putStrLn $ "len: " ++ show (Map.size m)
   let m'  = Map.delete "k2" m
   putStrLn $ "map: " ++ show m'
   let prs = Map.lookup "k2" m'
   putStrLn $ "prs: " ++ show prs
   let n   = Map.fromList [("foo", 1), ("bar", 2)]
   putStrLn $ "map: " ++ show n
```

```
$ runhaskell maps.hs
map: fromList [("k1",7),("k2",13)]
v1: 7
len: 2
map: fromList [("k1",7)]
prs: Nothing
map: fromList [("bar",2),("foo",1)]
```

https://lotz84.github.io/haskellbyexample/ex/maps

Data Map (dictionary) Example

```
import Prelude hiding (lookup)
import Data.Map

employeeDept       = fromList([("John","Sales"), ("Bob","IT")])
deptCountry        = fromList([("IT","USA"), ("Sales","France")])
countryCurrency    = fromList([("USA", "Dollar"), ("France", "Euro")])

employeeCurrency :: String -> Maybe String
employeeCurrency name = do
   dept <- lookup name employeeDept
   country <- lookup dept deptCountry
   lookup country countryCurrency

main = do
   putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
   putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

```
John's currency: Just "Euro"
Pete's currency: Nothing
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Simple Reader Usage (1)

```
import Data.Map (Map, (!))
import qualified Data.Map as Map
import Control.Monad.Reader

type Bindings = Map String Int;

-- Returns True if the "count" variable contains correct bindings size.
isCountCorrect :: Bindings -> Bool
isCountCorrect bindings = runReader calc_isCountCorrect bindings

-- The Reader monad, which implements this complicated check.
calc_isCountCorrect :: Reader Bindings Bool
calc_isCountCorrect = do
    count <- asks (lookupVar "count")
    bindings <- ask
    return (count == (Map.size bindings))
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

Simple Reader Usage (2)

```
-- The selector function to  use with 'asks'.
-- Returns value of the variable with specified name.
lookupVar :: String -> Bindings -> Int
lookupVar name bindings = maybe 0 id (Map.lookup name bindings)

sampleBindings = Map.fromList [("count",3), ("1",1), ("b",2)]

main = do
   putStr $ "Count is correct for bindings " ++ (show sampleBindings) ++ ": ";
   putStrLn $ show (isCountCorrect sampleBindings);


-- Count is correct for bindings fromList [("1",1),("b",2),("count",3)]: True

calc_isCountCorrect :: Reader Bindings Bool
calc_isCountCorrect = do
   count <- asks (lookupVar "count")
```

```
lookupVar name(=”count”) bindings(=Bindings)
maybe 0 id (Map.lookup ”count” Bindings)
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

Data.Maybe

data Maybe a

The Maybe type encapsulates an optional value.
A value of type Maybe a
either contains a value of type a (represented as Just a),
or it is empty (represented as Nothing).
Using Maybe is a good way to deal with errors or exceptional cases
without resorting to drastic measures such as error.

The Maybe type is also a monad.
It is a simple kind of error monad,
where all errors are represented by Nothing.
A richer error monad can be built using the Either type.

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data.Maybe – maybe method (1)

maybe :: b -> (a -> b) -> Maybe a -> b

The **maybe function** takes
- a default **value** (**b**),
- a **function** (**a->b**), and
- a **Maybe value** (**Maybe a**).

If the Maybe value is **Nothing**, the **function** returns the default **value**.
Otherwise, it applies the **function** to the **value** inside the **Just**
and returns the **result**.

```
>>> maybe False odd (Just 3)
True

>>> maybe False odd Nothing
False
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data.Maybe – maybe method (2)

maybe :: b -> (a -> b) -> Maybe a -> b

Read an integer from a string using **readMaybe**.
If we succeed, return twice the integer; that is, apply **(*2)** to it.
If instead we fail to parse an integer, return **0** by default:

```
>>> import Text.Read ( readMaybe )

>>> maybe 0 (*2) (readMaybe "5")
10
>>> maybe 0 (*2) (readMaybe "")
0
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data.Maybe – maybe method (3)

maybe :: b -> (a -> b) -> Maybe a -> b

Apply **show** to a **Maybe Int**.
If we have **Just n**, we want to show the underlying **Int n**.
But if we have **Nothing**, we return the **empty string**
instead of (for example) "Nothing":

```
>>> maybe "" show (Just 5)
"5"
>>> maybe "" show Nothing
""
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

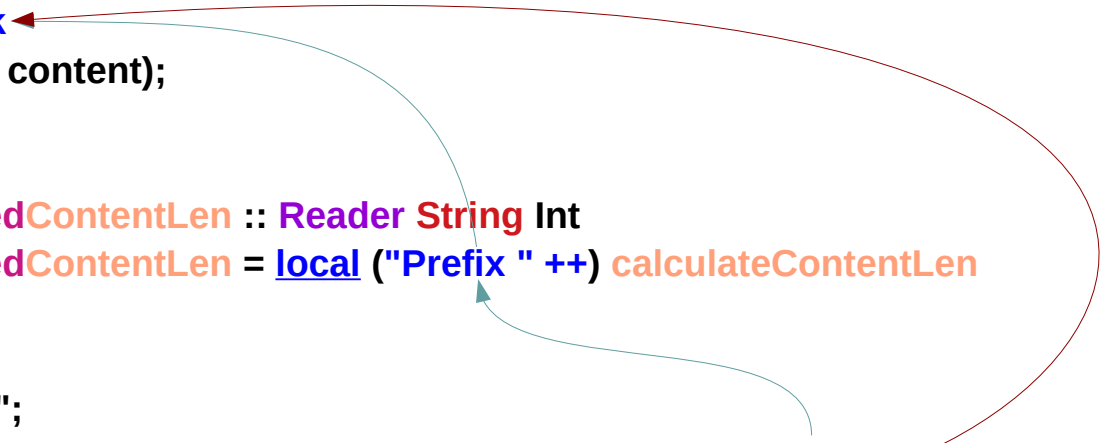
Modifying Reader Content with local

```
import Control.Monad.Reader

calculateContentLen :: Reader String Int
calculateContentLen = do
   content <- ask
   return (length content);


calculateModifiedContentLen :: Reader String Int
calculateModifiedContentLen = local ("Prefix " ++) calculateContentLen

main = do
   let s = "12345";
   let modifiedLen = runReader calculateModifiedContentLen s
   let len         = runReader calculateContentLen s
   putStrLn $ "Modified 's' length: " ++ (show modifiedLen)
   putStrLn $ "Original 's' length: " ++ (show len)
```



-- Calls calculateContentLen after adding a prefix to the Reader content.

```
-- Modified 's' length: 12
-- Original 's' length: 5
```

”Prefix 12345”

“12345”

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

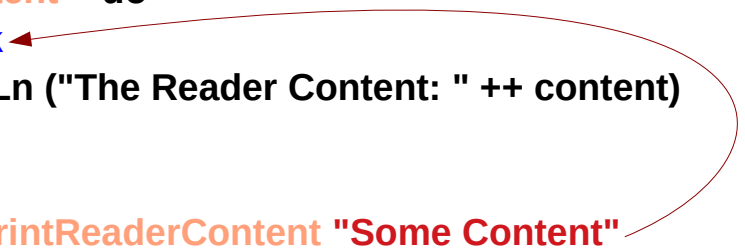
ReaderT Monad Transformer

```
import Control.Monad.Reader

-- The Reader/IO combined monad, where Reader stores a string.
printReaderContent :: ReaderT String IO ()
printReaderContent = do
   content <- ask
   liftIO $ putStrLn ("The Reader Content: " ++ content)

main = do
   runReaderT printReaderContent "Some Content"


-- The Reader Content: Some Content
```



```
Content :: String
ask
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

Source (1)

```
import Control.Monad.Reader

example :: String
example = runReader computation "Hello"

computation :: Reader String String
computation = do
   g <- ask
   return $ g ++ ", Haskell"

main = putStrLn example

--    Hello, Haskell
```

https://passy.svbtle.com/dont-fear-the-reader

Source (2)

```
import Control.Monad.Reader

example1 :: String -> String
example1 context = runReader (computation "Tom") context

computation :: String -> Reader String String
computation name = do
   g <- ask
   return $ g ++ name

main :: IO ()
main = putStrLn $ example1 "Hello, "

--    Hello, Tom
```

https://passy.svbtle.com/dont-fear-the-reader

Source (3)

```
import Control.Monad.Reader

example2 :: String -> String
example2 context = runReader (greet "James" >>= end) context

greet :: String -> Reader String String
greet name = do
  g <- ask
  return $ g ++ ", " ++ name

end :: String -> Reader String String
end input = do
  isHello <- asks (== "Hello")
  return $ input ++ if isHello then "!" else "."

main :: IO ()
main = putStrLn $ example2 "Hello"
--    Hello, James!
```

https://passy.svbtle.com/dont-fear-the-reader

Source (4)

```
import Data.Map (Map, (!))
import qualified Data.Map as Map

main = do
   let m0  = Map.empty
   let m1  = Map.insert "k1" 7 m0
   let m   = Map.insert "k2" 13 m1
   putStrLn $ "map: " ++ show m

   let v1  = m ! "k1"
   putStrLn $ "v1: " ++ show v1
   putStrLn $ "len: " ++ show (Map.size m)
   let m'  = Map.delete "k2" m
   putStrLn $ "map: " ++ show m'
   let prs     = Map.lookup "k2" m'
   putStrLn $ "prs: " ++ show prs
   let n   = Map.fromList [("foo", 1), ("bar", 2)]
   putStrLn $ "map: " ++ show n
```

https://lotz84.github.io/haskellbyexample/ex/maps

Source (5)

```
import Prelude hiding (lookup)
import Data.Map

employeeDept    = fromList([("John","Sales"), ("Bob","IT")])
deptCountry     = fromList([("IT","USA"), ("Sales","France")])
countryCurrency = fromList([("USA", "Dollar"), ("France", "Euro")])

employeeCurrency :: String -> Maybe String
employeeCurrency name = do
    dept <- lookup name employeeDept
    country <- lookup dept deptCountry
    lookup country countryCurrency

main = do
    putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
    putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Source (6)

```
import Data.Map (Map, (!))
import qualified Data.Map as Map
import Control.Monad.Reader

type Bindings = Map String Int;

isCountCorrect :: Bindings -> Bool
isCountCorrect bindings = runReader calc_isCountCorrect bindings

calc_isCountCorrect :: Reader Bindings Bool
calc_isCountCorrect = do
   count <- asks (lookupVar "count")
   bindings <- ask
   return (count == (Map.size bindings))

lookupVar :: String -> Bindings -> Int
lookupVar name bindings = maybe 0 id (Map.lookup name bindings)

sampleBindings = Map.fromList [("count",3), ("1",1), ("b",2)]

main = do
   putStr $ "Count is correct for bindings " ++ (show sampleBindings) ++ ": ";
   putStrLn $ show (isCountCorrect sampleBindings);
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

Source (7)

```
import Control.Monad.Reader

calculateContentLen :: Reader String Int
calculateContentLen = do
   content <- ask
   return (length content);

-- Calls calculateContentLen after adding a prefix to the Reader content.
calculateModifiedContentLen :: Reader String Int
calculateModifiedContentLen = local ("Prefix " ++) calculateContentLen

main = do
   let s = "12345";
   let modifiedLen = runReader calculateModifiedContentLen s
   let len = runReader calculateContentLen s
   putStrLn $ "Modified 's' length: " ++ (show modifiedLen)
   putStrLn $ "Original 's' length: " ++ (show len)
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

Source (8)

```
import Control.Monad.Reader

-- The Reader/IO combined monad, where Reader stores a string.
printReaderContent :: ReaderT String IO ()
printReaderContent = do
   content <- ask
   liftIO $ putStrLn ("The Reader Content: " ++ content)

main = do
   runReaderT printReaderContent "Some Content"
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf