무료 전자 책

배우기

Haskell Language

#haskell

145

1:

[Haskell](#) .

:

- **:** . ( ) . .
- : . , ( ) .
- **:** GHC .
- **:** . .
- **:** .
- **:** Haskell .

Haskell Haskell 2010. 2016 5 Haskell 2020 .

. , , , ,

2010	2012-07-10
98	2002-12-01

Examples

, !

"Hello, World!" Haskell :

```
main :: IO ()
main = putStrLn "Hello, World!"
```

`main` `IO ()` "" I / O , `()` ("") ( ). `main` .

`helloworld.hs` GHC :

```
ghc helloworld.hs
```

`"Hello, World!"` :

```
./helloworld
Hello, World!
```

`runhaskell` `runghc` .

```
runhaskell helloworld.hs
```

REPL . GHC `ghci` Haskell .

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

`load` (`:l`) ghci `load` .

```
ghci> :load helloworld
```

`:reload` (`:r`) ghci .

```
Prelude> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )

<some time later after some edits>

*Main> :r
Ok, modules loaded: Main.
```

:

`main` .

```
main :: IO ()
```

`IO ()` .

Hindley-Milner , : `main :: IO ()` . `main` . . .

- Haskell . . "" GHCi . .
- . . . .

.

```
main = putStrLn "Hello, World!"
```

, :

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

(Haskell ,) :

```
main = do
    putStrLn "Hello, World!"
    return ()
```

do (I / O) do (I / O).

do IO return .

main = putStrLn "Hello, World!" main = putStrLn "Hello, World!" putStrLn "Hello, World!" putStrLn "Hello, World!" IO () . "" putStrLn "Hello, World!" , putStrLn "Hello, World!" main .

putStrLn .

```
putStrLn :: String -> IO ()
-- thus,
putStrLn (v :: String) :: IO ()
```

putStrLn I / O-action ( ) . main putStrLn "Hello, World!" putStrLn "Hello, World!" .

"Hello World!". ( ) .

1

```
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

- Integral . Int Integer .
- (Integral a) => a
- fac :: a -> a fac a a a
- product .
- [1..n] enumFromTo 1 n 1 ≤ x ≤ n .

2

```
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

. 0 ( ) ( ) . fac .

---

fac GHC . .

,

Haskell .

.

```
f (0)  <-  0
f (1)  <-  1
f (n)  <-  f (n-1) + f (n-2)
```

. .

```
fibs !! n  <-  f (n)
```

```
                  ┌──────┐   ┌──────┐   ┌──────┐
                  │ f(0) │   │ f(1) │   │ f(2) │
fibs  ->  0 : 1 : │  +   │ : │  +   │ : │  +   │ :  .....
                  │ f(1) │   │ f(2) │   │ f(3) │
                  └──────┘   └──────┘   └──────┘

                  ┌────────────────────────────────────┐
                  │ f(0)   :   f(1)   :   f(2)   :  .....  │
                  └────────────────────────────────────┘
      ->  0 : 1 :               +
                  ┌────────────────────────────────────┐
                  │ f(1)   :   f(2)   :   f(3)   :  .....  │
                  └────────────────────────────────────┘
```

:

```
fibn n = fibs !! n
    where
    fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`zipWith` . `zipWith (+) [x1, x2, ...] [y1, y2, ...]` `[x1 + y1, x2 + y2, ...]` .

`fibs` [`scanl`] :

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`scanl` `foldl` ., `scanl f z0 [x1, x2, ...]` `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...` `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...`

., `fib` . n .

```
GHCi> let fib n = fibs !! n  -- (!!) being the list subscript operator
-- or in point-free style:
```

```
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

REPL

Haskell Haskell Haskell REPL (read-eval-print-loop) . REPL IO . `help` . .

GHC (i)

Glorious / Glasgow Haskell *GHCi* . *GHC* . *Cabal* *Stack* . Unix *Stack* .

```
curl -sSL https://get.haskellstack.org/ | sh
```

GHC . `stack` . Haskell Platform . .

1. *GHC* () *Cabal / Stack* ( )
2. , . .

( ).

- , , :

  ```
  sudo apt-get install haskell-platform
  ```

- :

  ```
  sudo dnf install haskell-platform
  ```

- :

  ```
  sudo yum install haskell-platform
  ```

- :

  ```
  sudo pacman -S ghc cabal-install haskell-haddock-api \
                 haskell-haddock-library happy alex
  ```

- :

  ```
  sudo layman -a haskell
  sudo emerge haskell-platform
  ```

- OSX :

```
brew cask install haskell-platform
```

- MacPorts OSX :

```
sudo port install haskell-platform
```

, *GHCi* `ghci` . .

```
me@notebook:~$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

`Prelude>` . , Haskell REPL REPL Haskell . `:q` `:quit` . *GHCi* `:?` .

. `.hs` `:l` `:load` REPL .

GHCi *GHC* . Haskell `.hs` . `.hs` `main` . `.test.hs` .

```
ghc test.hs
```

`main` .

1. .
   - *GHC* ( )
   -
   -
   -
   -
2. IHaskell IPython haskell markdown ( ) .

:

100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

Eratosthenes , data-ordlist :

```
import qualified Data.List.Ordered

ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))
```

```
_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

( )

```
ps = sieve [2..]
     where
     sieve (x:xs) = [x] ++ sieve [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

:

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

```
nubBy (((>1).).gcd) [2..]          -- i.e., nubBy (\a b -> gcd a b > 1) [2..]
```

`nubBy` `Data.List` `(\\)` .

REPL .

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> result
150
```

.

```
-- demo.hs

module Demo where
-- We declare the name of our module so
-- it can be imported by name in a project.

x = 5

y = 2 * 5 + x

result = y * 10
```

.

: https://riptutorial.com/ko/haskell/topic/251/--

2: Attoparsec

Attoparsec " / " .

Attoparsec .

API Parsec API .

`ByteString` , `Text` `Char8` . `OverloadedStrings` .

`Parser ia`	. `i` (: `ByteString` .
`IResult ir`	`Fail i [String] String` , `Partial (i -> IResult ir)` `Done ir` .

Examples

 .

:

 : 0800 1600.

"" : - ""- "0800"- "1600".

.

```
data Day = Day String

day :: Parser Day
day = do
  name <- takeWhile1 (/= ':')
  skipMany1 (char ':')
  skipSpace
  return $ Day name
```

.

```
data TimePortion = TimePortion String String

time = do
    start <- takeWhile1 isDigit
    skipSpace
    end <- takeWhile1 isDigit
    return $ TimePortion start end
```

. " " .

```
data WorkPeriod = WorkPeriod Day TimePortion
```

```
work = do
    d <- day
    t <- time
    return $ WorkPeriod d t
```

.

```
parseOnly work "Monday: 0800 1600"
```

-

Attoparsec . .

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import           Data.ByteString            (ByteString, readFile)
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                    hiding (readFile, take)

-- The DIB section from a bitmap header
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- The entire bitmap header
data Header = Header DIB Int Reserved Reserved Int
              deriving (Show)
```

. 4 .

, DIB 2

```
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

, .

```
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: Parser Int
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

.

```
bitmapHeader :: Parser Header
```

```
bitmapHeader = do
    dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

Attoparsec : https://riptutorial.com/ko/haskell/topic/9681/attoparsec

3: cofree comonads

Examples

~~

```
data Empty a
```

.

```
data Cofree Empty a
   --  = a :< ...  not possible!
```

Cofree (Const c) ~ Writer c

```
data Const c a = Const c
```

.

```
data Cofree (Const c) a
     = a :< Const c
```

```
data Writer c a = Writer c a
```

~~

```
data Identity a = Identity a
```

.

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

```
data Stream a = Stream a (Stream a)
```

Cofree Maybe ~ ~ NonEmpty

```
data Maybe a = Just a
             | Nothing
```

.

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
```

```
    | a :< Nothing
```

```
data NonEmpty a
    = NECons a (NonEmpty a)
    | NESingle a
```

Cofree (w) ~ WriterT w

```
data Writer w a = Writer w a
```

.

```
data Cofree (Writer w) a
    = a :< (w, Cofree (Writer w) a)
```

```
data Stream (w,a)
    = Stream (w,a) (Stream (w,a))
```

`WriterT w Stream` .

```
data WriterT w m a = WriterT (m (w,a))
```

Cofree (Either e) ~~ NonEmptyT (Writer e)

```
data Either e a = Left e
                | Right a
```

.

```
data Cofree (Either e) a
    = a :< Left e
    | a :< Right (Cofree (Either e) a)
```

```
data Hospitable e a
    = Sorry_AllIHaveIsThis_Here'sWhy a e
    | EatThis a (Hospitable e a)
```

`NonEmptyT (Writer e) a`

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

Cofree (Reader x) ~ x

```
data Reader x a = Reader (x -> a)
```

.

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

```
data Plant x a
     = Plant a (x -> Plant x a)
```

.

cofree comonads : https://riptutorial.com/ko/haskell/topic/8258/cofree-comonads---

4: Data.Aeson - JSON

Examples

Aeson JSON .

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

Person .

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

`Data.Aeson` `encode` `decode` `Person` `ToJSON` `FromJSON` `FromJSON` . `Generic` for `Person` `Generic` . `Generic` .

```
instance ToJSON Person
instance FromJSON Person
```

! `ToJSON` .

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

`encode` `Person` () Bytestring .

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

`decode` :

```
> encodeNewPerson "Hans" "Wurst" 30
"{\"lastName\":\"Wurst\",\"age\":30,\"firstName\":\"Hans\"}"


> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

Data.Aeson.Value

```
{-# LANGUAGE OverloadedStrings #-}
```

```
module Main where

import Data.Aeson

main :: IO ()
main = do
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer)
] :: Value
  print . encode $ example
```

JSON . ,

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

.

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

Data.Aeson - JSON : https://riptutorial.com/ko/haskell/topic/4525/data-aeson----json

5: Data.Text

Text Haskell String . String :

```
type String = [Char]
```

Text . .

Text String . String API , Text String .

OverloadedStrings .

Examples

OverloadedStrings Text Text .

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "\n\r\t   leading and trailing whitespace   \t\r\n"
```

strip Text .

```
ghci> T.strip myText
"leading and trailing whitespace"
```

stripStart .

```
ghci> T.stripStart myText
"leading and trailing whitespace   \t\r\n"
```

stripEnd .

```
ghci> T.stripEnd myText
"\n\r\t   leading and trailing whitespace"
```

filter .

```
ghci> T.filter /=' ' "spaces in the middle of a text string"
```

```
"spacesinthemiddleofatextstring"
```

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "mississippi"
```

`splitOn` `Text` `Texts` .

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

`splitOn` `intercalate` .

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

`split` `Text` .

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

`Data.Text.Encoding` .

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

`decodeUtf8` . UTF-8 `decodeUtf8With` `decodeUtf8With` .

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

`isInfixOf :: Text -> Text -> Bool` `Text` `Text` .

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

`isPrefixOf :: Text -> Text -> Bool` `Text` `Text` .

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

`isSuffixOf :: Text -> Text -> Bool` `Text` `Text` .

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "mississippi"
```

`index` .

```
ghci> T.index myText 2
's'
```

`findIndex` `(Char -> Bool)` Text Nothing .

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

`count` `Text` `Text` .

```
ghci> count ("miss"::T.Text) myText
1
```

Data.Text : https://riptutorial.com/ko/haskell/topic/3406/data-text

6: GHCi

GHCI GHC REPL.

Examples

GHCi

`ghci` GHCI .

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

GHCi

GHCI . .

```
Prelude Data.List Control.Monad> -- etc
```

`:set prompt` .

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

`:set prompt "foo> "` GHCi .

GHCi

GHCi `~/.ghci` . GHCi .

```
$ echo ":set prompt \"foo> \"" > ~/.ghci
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from ~/.ghci
foo>
```

`:l` `:load` - .

```
$ echo "f = putStrLn \"example\"" > example.hs
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
ghci> :l example.hs
[1 of 1] Compiling Main               ( example.hs, interpreted )
Ok, modules loaded: Main.
ghci> f
example
```

GHCi

`:q` GHCi `:q` `:quit`

```
ghci> :q
Leaving GHCi.

ghci> :quit
Leaving GHCi.
```

`CTRL` + `D` (OSX `Cmd` + `D`) `:q` .

GHCi (`:l filename.hs`) GHCi `:r` `:reload` `:reload` .

```
ghci> :r
OK, modules loaded: Main.

ghci> :reload
OK, modules loaded: Main.
```

GHCi

GHCi (`:loaded`) .

:

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

GHCi :

```
Prelude> :load mySum.hs
[1 of 1] Compiling Main             ( mySum.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

.

```
*Main> :break 2
Breakpoint 0 activated at mySum.hs:2:3-39
```

GHCi :

```
*Main> doSum 12
Stopped at mySum.hs:2:3-39
_result :: IO () = _
n :: Integer = 12
[mySum.hs:2:3-39] *Main>
```

`:list` `:list` :

```
[mySum.hs:2:3-39] *Main> :list
1  doSum n = do
2    putStrLn ("Counting to " ++ (show n))   -- GHCi will emphasise this line, as that's where
we've stopped
3    let v = sum [1..n]
```

.

```
[mySum.hs:2:3-39] *Main> n
12
:continue
Counting to 12
sum to 12 = 78
*Main>
```

`:{` `:}` . GHCi .

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

GHCi : https://riptutorial.com/ko/haskell/topic/3407/ghci-

7: GHCJS

GHCJS GHC API JavaScript   Haskell.

Examples

"Hello World!" Node.js

`ghcjs` `ghc`   .   Node.js SpiderMonkey jsshell   . :

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
Hello world!
```

GHCJS : https://riptutorial.com/ko/haskell/topic/9260/ghcjs

8: Google

Haskell `htprotoc` :

1. Github
2.

`$HOME/.local/bin/ hprotoc` .

Examples

.proto ,

`.proto` `person.proto`

```
package Protocol;

message Person {
    required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

.

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

.

```
Loading filepath: "/<path-to-project>/person.proto"
All proto files loaded
Haskell name mangling done
Recursive modules resolved
./Protocol/Person.hs
./Protocol.hs
Processing complete, have a nice day.
```

`hprotoc` haskell `Person.hs` `Protocol` :

```
import Protocol (Person)
```

Stack add

```
  protocol-buffers
, protocol-buffers-descriptor
```

`build-depends:`

```
Protocol
```

.cabal exposed-modules .

ByteString ByteString .

ByteString ("Person") Haskell , import messageGet .

```
import Text.ProtocolBuffers (messageGet)
```

Person .

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
    Left   _           -> Nothing
    Right  (person, _)  -> Just person
```

Google : https://riptutorial.com/ko/haskell/topic/5018/google--

9: Gtk3

- obj <- <widgetName> New - (: Windows, Buttons, Grids)
- <widget> [<attributes>] - Attr self (: buttonLabel)
- on <widget> <event> <IO action> - IO (: buttonActivated)

Haskell Gtk3 (: Ubuntu APT `libghc-gtk` . `stack` , Gtk3 `cabal` . `gtk2hs-buildtools` . `gtk` , `gtk3` Gtk `cabal install gtk2hs-buildtools` . `cabal install gtk2hs-buildtools build-depends` .

Examples

Gtk Hello World

Gtk3 "Hello World" . .

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  on window objectDestroy mainQuit
  set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
  set button [ buttonLabel := "Hello World" ]
  on button buttonActivated $ do
    putStrLn "A \"clicked\"-handler to say \"destroy\""
    widgetDestroy window
  set window [ containerChild := button ]
  widgetShowAll window
  mainGUI -- main loop
```

Gtk3 : https://riptutorial.com/ko/haskell/topic/7342/gtk3

10: IO

Examples

IO

`io-streams` IO . :

- `InputStream` :
- `OutputStream` :

`makeInputStream :: IO (Maybe a) -> IO (InputStream a)` . `read :: InputStream a -> IO (Maybe a)` . `Nothing` EOF .

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
  is <- S.makeInputStream $ randomInt  -- create an InputStream
  forever $ printStream =<< S.read is  -- forever read from that stream
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
  r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "Nada!"
printStream (Just a) = putStrLn $ show a
```

IO : https://riptutorial.com/ko/haskell/topic/4984/io-

11: RankNTypes

GHC `Rank2Types` `RankNTypes` .

- `Rank2Types` `RankNTypes` .
- `forall` .

Examples

RankNTypes

StackOverflow . .

RankNTypes : https://riptutorial.com/ko/haskell/topic/8984/rankntypes----

12: XML

XML

Examples

`xml`

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | Create XML from a Package
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

XML : https://riptutorial.com/ko/haskell/topic/9264/xml

13: zipWithM

`zipWithM` `zipWith` `mapM` `map` : .

`Control.Monad`

- zipWithM :: m => (a -> b -> mc) -> [a] -> [b] -> m [c]

Examples

.

5 . , .

. `Nothing` .

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

`zipWithM` .

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

$ 5 `Nothing` .

zipWithM : https://riptutorial.com/ko/haskell/topic/9685/zipwithm

14:

`Traversable` `mapM :: Monad m => (a -> mb) -> [a] -> m [b]` `Applicative` `Applicative` .

Examples

Traversable Functor Foldable

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

`Traversable` `Foldable` `Functor` USING `fmapDefault` `foldMapDefault` `Data.Traversable` .

```
instance Functor MyType where
    fmap = Traversable.fmapDefault

instance Foldable MyType where
    foldMap = Traversable.foldMapDefault
```

`fmapDefault` `Identity` applicative functor `traverse` .

```
newtype Identity a = Identity { runIdentity :: a }

instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity x = Identity (f x)

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

`foldMapDefault` `Const` applicative functor , monoidal .

```
newtype Const c a = Const { getConst :: c }

instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Traversable

`traverse` `Applicative` `fmap` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

```
instance Traversable Tree where
    traverse f Leaf = pure Leaf
    traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

`DeriveTraversable` GHC `Traversable` . `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving (Functor, Foldable, Traversable)  -- also using DeriveFunctor and
DeriveFoldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving (Functor, Foldable, Traversable)

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> traverse print (preorder myTree)
'b'
'a'
'c'
ghci> traverse print (postorder myTree)
'a'
'c'
```

```
'b'
```

.

. Backwards applicator .

```
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
    Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

Backwards "traverse" . traverse Backwards) .

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

Reverse newtype Data.Functor.Reverse .

Traversable

```
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

Traversable t "" a . f :: a -> fb traverse Applicative . sequenceA Traversable Applicative .

Traversable

mapAccum .

```
--                                                          A Traversable structure
--                                                                      |
--                                                        A seed value  |
```

```
--                                                              |     |
--                                                             |-|  |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
--                                       |------------------|              |--------|
--                                                |                             |
--                       A folding function which produces a new mapped        |
--                         element 'c' and a new accumulator value 'a'          |
--                                                                              |
--                                                             Final accumulator value
--                                                              and mapped structure
```

fmap . foldl / foldr .

, tails mapAccumR inits mapAccumL .

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
["", "a", "ab", "abc"]
```

mapAccumL State applicator .

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
    pure x = State $ \s -> (s, x)
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)

mapAccumL f z t = runState (traverse (State . flip f) t) z
```

mapAccumR mapAccumL .

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

t Traversable , ta : "" "":

```
data Traversed t a = Traversed { shape :: t (), contents :: [a] }
```

"" Foldable "" .

, ta Traversed ta Functor Foldable .

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

traverse

.

```
import Control.Monad.State

-- invariant: state is non-empty
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

Traversable break . recombine break . recombine recombine . break . contents shape .

Traversed t Traversable . traverse , Traversable , .

```
instance Traversable (Traversed t) where
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

zip ,

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

transpose sequenceA ,

```
-- transpose exchanges the inner list with the outer list
--           +---+-->--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+-->--+---+

-- sequenceA exchanges the inner Applicative with the outer Traversable
--                                               +------>------+
--                                               |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                                  |       |
--                                                  +--->---+
```

[] Traversable and Applicative sequenceA *n-ary* *zip* .

[] " " Applicative - "" Applicative . Control.Applicative ZipList newtype .

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

ZipList Applicative transpose .

```
transpose :: [[a]] -> [[a]]
transpose = getZipList . traverse ZipList
```

```
ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

: https://riptutorial.com/ko/haskell/topic/754/-

15:

Higher Order Function .

Examples

.

Haskell .

.

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

. , .

:

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

(:) . , .

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

.

. ( ) λ . , .

.

```
\x -> let {y = ...x...} in y
```

.

.

```
f(x) = x^2
```

```
f    x =  x^2

f = \x -> x^2
```

f `\x -> x^2` .

`map` , `a -> b` `a -> b` . `map` , . ,

```
\x -> let {y = ...x...} in y
```

`x` `a` `a` , `...x...` `x` `y` `b` `b` . .

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

Haskell . , .

`div` .

```
div :: Int -> Int -> Int
```

6 2 3 .

```
Prelude> div 6 2
3
```

. `div 6` `Int -> Int` . 2 3.

, " `Int` " " `Int` `Int` " . . `div` .

```
div :: Int -> (Int -> Int)
```

. , " ( )."

: https://riptutorial.com/ko/haskell/topic/4539/-

16:

Examples

Cabal . Cabal , Gloss `cabal install gloss` .

Hackage GitHub .

1. `gloss/gloss-rendering/` `cabal install`
2. `gloss/gloss/` `cabal install`

Gloss `display` .

`import Graphics.Gloss` . .

```
main :: IO ()
main = display window background drawing
```

`window` `Display` .

```
-- Defines window as an actual window with a given name and size
window = InWindow name (width, height) (0,0)

-- Defines window as a fullscreen window
window = FullScreen
```

`InWindow` `(0,0)` .

1.11 : Gloss `FullScreen` . `FullScreen (1024,768)` `FullScreen (1024,768)` `FullScreen (1024,768)`

`background` `Color` . .

```
background = white
```

. . ([1]).) 80 .

```
drawing = Circle 80
```

Hackage .

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    where
      window = InWindow "Nice Window" (200, 200) (0, 0)
      background = white
```

```
    drawing = Circle 80
```

: https://riptutorial.com/ko/haskell/topic/5570/-

17:

`Profunctor` `Profunctor` `profunctors` `Data.Profunctor` .

"" .

- dimap :: Profunctor p => (a -> b) -> (c -> d) -> pbc -> pad
- lmap :: Profunctor p => (a -> b) -> pbc -> pac
- rmap :: Profunctor p => (b -> c) -> pab -> pac
- dimap id id = id
- lmap id = id
- rmap id = id
- dimap fg = lmap f. g
- lmap f = dimap f id
- rmap f = dimap id f

Profunctor Hackage " bifunctor."

? , bifunctor functor . `fmap` .

" (covariant)" profunctor . (`rmap`) `Profunctor p => (b -> c) -> pab -> pac` `Profunctor p => (b -> c) -> pab -> pac` . .

" " . (`lmap`) `Profunctor p => (a -> b) -> pbc -> pac` `Profunctor p => (a -> b) -> pbc -> pac` . . `a -> b` `a -> b` `a` .

: . `Functor` typeclass "" `Data.Functor.Contravariant` `Contravariant` `Data.Functor.Contravariant` ( ) `Cofunctor` `Data.Cofunctor` .

Examples

(->)

(->) profunctor . , .

```
instance Profunctor (->) where
    lmap f g = g . f
    rmap f g = g . g
```

: https://riptutorial.com/ko/haskell/topic/9694/

18:

(.) .

```
(.) :: (b -> c) -> (a -> b) ->  (a -> c)
(.)       f           g          x =  f (g x)     -- or, equivalently,

(.)       f           g     =   \x -> f (g x)
(.)       f     =    \g     ->  \x -> f (g x)
(.) =    \f     ->   \g     ->  \x -> f (g x)
(.) =    \f     ->  (\g     -> (\x -> f (g x) ) )
```

(b -> c) -> (a -> b) -> (a -> c) (b -> c) -> (a -> b) -> a -> c -> ""

```
 f g x y z ...    ==    (((f g) x) y) z ...
```

" " . x g f .

```
(.)       f           g          x =  r
                                      where r = f (g x)
-- g :: a -> b
-- f ::      b -> c
-- x :: a
-- r ::           c

(.)       f           g     =   q
                                where q = \x -> f (g x)
-- g :: a -> b
-- f ::      b -> c
-- q :: a      -> c

....
```

, .

```
(.) f g x  =  (f . g) x  =  (f .) g x  =  (. g) f x
```

" " " " .

```
(.) f g    =  (f . g)    =  (f .) g    =  (. g) f
--         1             2             3
```

x . η-shrinkment.

```
(f . g) x   =   f (g x)
```

"" x ; x (f . g) point-free .

Examples

`(.)` .

```
(f . g) x = f (g x)
```

, , .

```
((^2) . succ) 1        --    4
```

`(.)` `(<<<)` . ,

```
(+ 1) <<< sqrt $ 25    --    6
```

`Control.Category` `(>>>)` .

```
-- (>>>) :: Category cat => cat a b -> cat b c -> cat a c
-- (>>>) :: (->) a b -> (->) b c -> (->) a c
-- (>>>) :: (a -> b) -> (b -> c) -> (a -> c)
( f >>> g ) x = g (f x)
```

:

```
sqrt >>> (+ 1) $ 25    --    6.0
```

. ,

```
(f .: g) x y = f (g x y)           -- which is also
             = f ((g x) y)
             = (f . g x) y         -- by definition of (.)
             = (f .) (g x) y
             = ((f .) . g) x y
```

, η - `(f .: g) = ((f .) . g)`

```
(.:) f g    = ((f .) . g)
            = (.) (f .) g
            = (.) ((.) f) g
            = ((.) . (.)) f g
```

so `(.:) = ((.) . (.))` , .

:

```
(map (+1) .: filter) even [1..5]      --   [3,5]
(length   .: filter) even [1..5]      --   2
```

: https://riptutorial.com/ko/haskell/topic/4430/-

19:

1. `[] :: [a]`
2. `(:) :: a -> [a] -> [a]`
3. head - .

   `head :: [a] -> a`
4. - .

   `last :: [a] -> a`
5. tail - .

   `tail :: [a] -> [a]`
6. init - .

   `init :: [a] -> [a]`
7. xs !! i - xs i .

   `(!!) :: Int -> [a] -> a`
8. n xs - n . xs

   `take :: Int -> [a] -> [a]`
9. map :: (a -> b) -> [a] -> [b]
10. filter :: (a -> Bool) -> [a] -> [a]
11. (++) :: [a] -> [a]
12. concat :: [[a]] -> [a]

1. `[a]` `[]` `a` .
2. `[]` .
3. `[]` LHS `f [] = ...` , .
4. `x:xs` `x` `xs` `xs`
5. `f (x:xs) = ...` . `x` `xs` .
6. `f (a:b:cs) = ...` `f (a:(b:cs)) = ...` . `a` `b` `cs` .
7. `f ((a:as):bs) = ...` `f (a:(as:bs)) = ...` . `a` , `as` , `bs` .
8. `f (x:[]) = ...` `f [x] = ...` . .
9. `f (a:b:[]) = ...` `f [a,b] = ...` . .
10. `f [a:b] = ...` . `a` `b` .
11. `[a,b,c]` `(a:b:c:[])` . `(:)` `[]` .
12. `all@(x:y:ys)` `all` ( ) `(x:y:ys)` .

Examples

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

(++) xs     [] = xs
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

Haskell Prelude `[]` . `Int` , .

```
xs :: [Int]    -- or equivalently, but less conveniently,
xs :: [] Int
```

Haskell . .

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

.

- `[]` .
- `(:)` "cons" , . Consing `x` ( ) `a` `xs` (`a`) ( ) `x` ( ) `xs` .

.

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []     -- ((:) is right-associative)
-- or  = [12, 99, 37]          -- (syntactic sugar for lists)
```

`(++)` `(:)` `[]` .

.

```
listSum :: [Int] -> Int
```

```
listSum []          = 0
listSum (x:xs) = x + listSum xs
```

.

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

.

Haskell Prelude `map` , `filter` . .

n (0).

```
list = [1 .. 10]

firstElement = list !! 0           -- 1
```

`!!` .

```
list !! (-1)     -- *** Exception: Prelude.!!: negative index

list !! 1000     -- *** Exception: Prelude.!!: index too large
```

`Data.List.genericIndex` `!!` `Integral` .

```
import Data.List (genericIndex)

list `genericIndex` 4              -- 5
```

O (n) . `Data.Vector` .

1 10 .

```
[1..10]    -- [1,2,3,4,5,6,7,8,9,10]
```

start .

```
[1,3..10]  -- [1,3,5,7,9]
```

Haskell , .

```
[1,3,5..10] -- error
[1,3,9..20] -- error
```

.

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

. [1..] [1..] . .

```
take 5 [1..]   -- returns [1,2,3,4,5] even though [1..] is infinite
```

.

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5] , though 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002] , though 1.2000000000000002 > 1.2
```

`Enum` typeclass . `a` , `b` , `c` `Enum` .

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

`Bool`

```
 [False ..]      -- [False,True]
```

`False` (, `False..` . `False`).

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]

length [1 .. 10]   --    10

reverse [1 .. 10]  --    [10, 9 .. 1]

take 5 [1, 2 .. ]  --    [1, 2, 3, 4, 5]

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

. step `foldr` ( ) .

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs         -- = foldl f (acc `f` x) xs
```

`foldl` .:

```
foldl (+) 0 [1, 2, 3]     -- is equivalent to ((0 + 1) + 2) + 3
```

foldl (foldl).

```
foldl (+) 0 [1, 2, 3]                       --  foldl (+)    0    [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                  --  foldl (+)   (0 + 1)    [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]             --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []      --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                   --            (((0 + 1) + 2) + 3)
```

((0 + 1) + 2) + 3 . (fab) (a `f` b) ((+) 0 1) (0 + 1) .

.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)            -- = x `f` foldr f z xs
```

foldr .:

```
foldr (+) 0 [1, 2, 3]      -- is equivalent to 1 + (2 + (3 + 0))
```

foldr (foldr).

```
foldr (+) 0 [1, 2, 3]                       --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                  -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))             -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))      -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                     -- 1 + (2 + (3 +            0   ))
```

1 + (2 + (3 + 0)) , ((+) 3 0) (3 + 0) .

`map`

( ) . map map :

```
-- Simple add 1
map (+ 1) [1,2,3]
[2,3,4]

map odd [1,2,3]
[True,False,True]

data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

-- Extract just the age from a list of people
map (\(Person n g a) -> a) [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
[31,29]
```

`filter`

:

```
li = [1,2,3,4,5]
```

`filter :: (a -> Bool) -> [a] -> [a]` .

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- Something slightly more complicated
comfy i = notTooLarge && isEven
  where
     notTooLarge = (i + 1) < 5
     isEven = even i

filter comfy li        -- [2]
```

.

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
    isFemale (Person _ Female _) = True
    isFemale _ = False

onlyLadies [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
-- [Person "Ellie" Female 29]
```

zip .

```
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

> zip [1,3,5] [2,4,6]
> [(1,2),(3,4),(5,6)]
```

:

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

:

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

> unzip [(1,2),(3,4),(5,6)]
> ([1,3,5],[2,4,6])
```

: https://riptutorial.com/ko/haskell/topic/2281/

20:

- addDays :: -> ->
- diffDays :: Day -> Day -> Integer
- fromGregorian :: -> Int -> Int -> Day

  ```
  convert from proleptic Gregorian calendar. First argument is year, second month number
  (1-12), third day (1-31). Invalid values will be clipped to the correct range, month
  first, then day.
  ```

- getCurrentTime :: IO UTCTime

`time` `Data.Time` .

Examples

`getCurrentTime` .

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

`fromGregorian` .

```
fromGregorian 1984 11 17  -- yields a Day
```

,

`Day` .

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
addDays 1 (fromGregorian 2000 12 31)
-- 2001-01-01
```

:

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

.

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

.

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

: https://riptutorial.com/ko/haskell/topic/4950/-

21: (GHC)

Examples

@Viclib typeclass .

.

```
class ListIsomorphic l where
    toList    :: l a -> [a]
    fromList  :: [a] -> l a
```

`toList . fromList == id` `toList . fromList == id` . GHC ?

GHC . GHC (`fromList :: Seq a -> [a]` `Seq$fromList`).

`fromList` `fromList` - - `toList` .

```
{-# RULES
  "protect toList"   toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"  forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

(GHC) : https://riptutorial.com/ko/haskell/topic/4914/----ghc-

22:

Examples

PostgreSQL-simple PostgreSQL . DB / API .

.

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- execute with no parameter substitution
  print i
```

PostreSQL-Simple `query` .

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

`execute` insert / update SQL `execute` .

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

: https://riptutorial.com/ko/haskell/topic/4444/

23:

.

-
-

Examples

`forkIO`

`forkIO` `forkIO` .

`forkIO :: IO () -> IO ThreadId` `IO` `ThreadId` .

`ghci` .

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..100000000]
ThreadId 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- some time later....
Prelude Control.Concurrent> 50000005000000
```

!

`MVar '

`MVar a` `Control.Concurrent` .

- `newEmptyMVar :: IO (MVar a)` - `MVar a` . `MVar a`
- `newMVar :: a -> IO (MVar a)` - `MVar` .
- `takeMVar :: MVar a -> IO a` - `MVar` .
- `putMVar :: MVar a -> a -> IO ()` - `MVar` `MVar` .

1 1 .

```
import Control.Concurrent
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar will block 'til m is non-empty!
```

.

```
main2 = loop
  where
    loop = do
```

```
        m <- newEmptyMVar
        n <- getLine
        putStrLn "Calculating. Please wait"
        -- In another thread, parse the user input and sum
        forkIO $ putMVar m $ sum [1..(read n :: Int)]
        -- In another thread, wait 'til the sum's complete then print it
        forkIO $ print =<< takeMVar m
        loop
```

takeMVar MVar MVar . putMVar putMVar . , .

.

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

MVar .

```
concurrent ma mb     -- new thread 1
concurrent mb ma     -- new thread 2
```

.

1. 1 ma ma
2. 2 mb mb .

1 mb . 2 2 1 ma . !

(Software Transactional Memory), TVar a .

TVar a STM . MVar MVar STM .

atomically :: STM a -> IO a

STM .

readTVar :: TVar a -> STM a

TVar . :

```
value <- readTVar t
```

writeTVar :: TVar a -> a -> STM ()

TVar TVar .

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

:

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- Initialise a new TVar
  shared <- atomically $ newTVar 0
  -- Read the value
  before <- atomRead shared
  putStrLn $ "Before: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
  after <- atomRead shared
  putStrLn $ "After: " ++ show after
  where timesDo = replicateM_
        milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

: https://riptutorial.com/ko/haskell/topic/4426/

24:

Examples

sum algebraic `data` .

```
data StandardType = StandardType String Int Bool --standard way to create a sum type

data RecordType = RecordType { -- the same sum type with record syntax
    aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
  r :: RecordType
> :t aString
  aString :: RecordType -> String
> aString r
  "Foobar"
```

.

```
case r of
  RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

.

( )

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

`undefined` .

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
     Fields of RecordType not initialized: isTrue
> isTrue r
  Error 'undefined'
```

. .

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

.

.

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

:

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

`alex` `Person` .

```
anotherAlex = alex { age = 31 }
```

`alex` `anotherAlex` `anotherAlex` .

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

newtype

`newtype` . newtype . monads `run` monoid `get` , `un` .

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

newtype Product a = Product { getProduct :: a }

newtype Fancy = Fancy { unfancy :: String }
  -- a fancy string that wants to avoid concatenation with ordinary strings
```

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
-- > 756
```

RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}

data Client = Client { firstName     :: String
                     , lastName      :: String
                     , clientID      :: String
                     } deriving (Show)

printClientName :: Client -> IO ()
printClientName Client{..} = do
    putStrLn firstName
    putStrLn lastName
    putStrLn clientID
```

`Client{..}` `Client`

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

matcher .

```
Client { firstName = "Joe", .. }
```

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

.

```
data Person = Person { age :: Int, name :: String }
```

.

`age` `name` `age` `name` .

```
age :: Person -> Int
name :: Person -> String
```

. , .

.

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

RHS (`x`) .

NamedFieldPuns NamedFieldPuns

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

`NamedFieldPuns` . RHS `name` .

RecordWildcards RecordWildcards

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

`RecordWildCards` `RecordWildCards` . (`name` `age`)

`Person` .

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

.

: https://riptutorial.com/ko/haskell/topic/1950/-

25:

Lens Haskell , , , , , getter setter Java .

?

( ) . . `lens` . .

. . , `_1` .

. `Lens sa s a` ,

1. `a`
2. `a` .

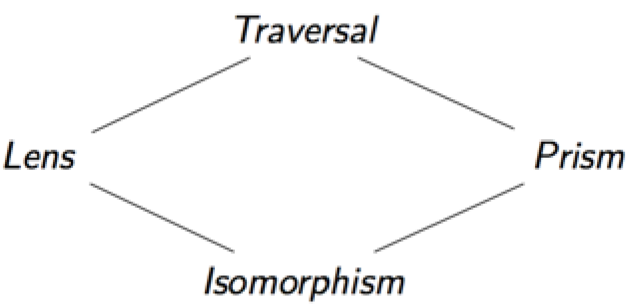
`get` `set` .

.

	...
0	
	...

. .

. , .



. () , . 0 1 . (affine traversal).

. . . ., .

Examples

```
("a", 1) ^. _1 -- returns "a"
("a", 1) ^. _2 -- returns 1
```

```
("a", 1) & _1 .~ "b" -- returns ("b", 1)
```

```
("a", 1) & _2 %~ (+1) -- returns ("a", 2)
```

both

```
(1, 2) & both *~ 2 -- returns (2, 4)
```

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

x y .

```
let p = Point 5.0 6.0
p ^. x     -- returns 5.0
set x 10 p -- returns Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- returns Point { _x = 6.0, _y = 6.0 }
```

```
data Person = Person { _personName :: String }
makeFields ''Person
```

HasName , Person name Person HasName HasName . .

```
data Entity = Entity { _entityName :: String }
makeFields ''Entity
```

makeFields Template Haskell . , .

. ~ = .

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

: Lens' Simple Lens' .

&

- .

```
change :: A -> A
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

& . .

```
change a = flip execState a $ do
    lensA %= operationA
    lensB %= operationB
    lensC %= operationC
```

lensX id lensX modify .

.

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses ''Point
makeLenses ''Entity
makeLenses ''World
```

.

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- move the first entity
    entities . ix 0 . position . x += 1

    -- do some operation on all of them
    entities . traversed . position %= \p -> p `pointAdd` ...

    -- or only on a subset
    entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

Template Haskell .

Template Haskell demystify ,

```
data Example a = Example { _foo :: Int, _bar :: a }
```

```
makeLenses 'Example
```

()

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

. .

```
foo :: Lens' (Example a) Int
--  :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; expand the alias
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
--  :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; expand the alias
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

, `wrap` "" "" "".

`Lens' sa` `a` `s` . `Prism' sa` `s` . `a`

(tuple) `_1 :: Lens' (a, b) a` . `_Just :: Prism' (Maybe a) a` `Maybe a` `a` `Just` `Nothing` .

- `view :: Lens' sa -> (s -> a)` " `s` " `a` .
- `set :: Lens' sa -> (a -> s -> s)` `s` `a` "" `a`
- `review :: Prism' sa -> (a -> s)` `a` `s` ""
- `preview :: Prism' sa -> (s -> Maybe a)` "" `s` . `a`

`Lens' sa` `s` `r` `(r, a)` . , `Prism' sa` `s` `Either ra` `r` . .

```
-- `Lens' s a` is no longer supplied, instead we just *know* that `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- `Prism' s a` is no longer supplied, instead we just *know* that `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

`Traversal' sa` `s` 0 . `a`

```
toListOf :: Traversal' s a -> (s -> [a])
```

`Traversable` `t` `traverse :: Traversal (ta) a` .

Traversal a

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

f :: Lens' sa a s . g :: Prism' ab 0 1 b a . f . g Traversal' sb f g 0 1 b S s .

f :: Lens' ab a g :: Lens' bc f . g f f . g Lens' ac g g . :

- ( )
- Lens view " " . . , . OO .

Lens Lens , (.) " Lens -like" . lens . x . y x y .

makeLenses Lens ES Control.Lens.TH makeClassy . makeClassy makeLenses makeLenses .
.

```
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

.

```
class HasFoo t where
   foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

makeFields

(StackOverflow)

, capacity . makeFields .

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
           , TemplateHaskell
  #-}

module Foo
where

import Control.Lens

data Foo
  = Foo { fooCapacity :: Int }
  deriving (Eq, Show)
$(makeFields ''Foo)
```

```
data Bar
  = Bar { barCapacity :: Double }
  deriving (Eq, Show)
$(makeFields ''Bar)
```

ghci :

```
*Foo
λ let f = Foo 3
|     b = Bar 7
|
b :: Bar
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
λ barCapacity b
7.0
it :: Double

*Foo
λ f ^. capacity
3
it :: Int

*Foo
λ b ^. capacity
7.0
it :: Double

λ :info HasCapacity
class HasCapacity s a | s -> a where
  capacity :: Lens' s a
    -- Defined at Foo.hs:14:3
instance HasCapacity Foo Int -- Defined at Foo.hs:14:3
instance HasCapacity Bar Double -- Defined at Foo.hs:19:3
```

`HasCapacity sa` , `Lens' s` (`a a`). () "" . . . makeFieldsWith lensRules .

, ghci -ddump-splices . Foo.hs :

```
[1 of 1] Compiling Foo              ( Foo.hs, interpreted )
Foo.hs:14:3-18: Splicing declarations
    makeFields ''Foo
  ======>
    class HasCapacity s a | s -> a where
      capacity :: Lens' s a
    instance HasCapacity Foo Int where
      {-# INLINE capacity #-}
      capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: Splicing declarations
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
```

```
    {-# INLINE capacity #-}
    capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
Ok, modules loaded: Foo.
```

`HasCapcity` `HasCapcity` Foo . Bar .

`HasCapcity` . `makeFields` . .

: https://riptutorial.com/ko/haskell/topic/891/

26: /

Reader . http://adit.io/posts/2013-06-10-three-useful-monads.html .

Examples

`ask` (https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask) . .
:

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
  x <- ask
  return $ "The value is: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
  x <- reader id
  lift $ print "Hello from within"
  liftAnnotated $ print "Hello from within..."
  return $ "The value is: " ++ show x
```

:

```
"The value is: 100"
"Hello from within"
"Hello from within..."
"The value is: 200"
```

/ : https://riptutorial.com/ko/haskell/topic/9320/---

27:

Examples

SmallCheck, QuickCheck HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "String length <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "String length <= 2" $  -- should fail
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "String length <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "String length <= 4" $  -- should fail
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "String comparison 1" $
      assertEqual "description" "OK" "OK"

  , testCase "String comparison 2" $  -- should fail
      assertEqual "description" "fail" "fail!"
  ]
```

:

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

Cabal   :

```
cabal exec runhaskell test.hs
```

: https://riptutorial.com/ko/haskell/topic/3816/-

28:

. Monad . IO . IO a a " a ".

m (: [] Maybe) instance Monad m . " a " ma .

Examples

Maybe - null . .

.

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

halve Int .

halve ?

```
takeOneEighth :: Int -> Maybe Int              -- (after you read the 'do' sub-section:)
takeOneEighth x =
  case halve x of                                 --  do {
    Nothing -> Nothing
    Just oneHalf ->                               --      oneHalf    <- halve x
      case halve oneHalf of
        Nothing -> Nothing
        Just oneQuarter ->                        --      oneQuarter <- halve oneHalf
          case halve oneQuarter of
            Nothing -> Nothing                    --      oneEighth  <- halve oneQuarter
            Just oneEighth ->
              Just oneEighth                      --      return oneEighth }
```

- takeOneEighth halve .
- halve , takeOneEighth .
- halve .

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                            -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                         -- also, f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

.

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve             -- or,
    -- return x >>= halve >>= halve >>= halve             -- which is parsed as
```

```
    -- (((return x) >>= halve) >>= halve) >>= halve        -- which can also be written as
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x     -- or, equivalently, as
    --  halve <=<     halve <=<     halve      $        x
```

`<=<` `(g <=< f) x = g =<< fx` `(f >=> g) x = fx >>= g` .

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve               -- infixr 1 <=<
        -- or, equivalently,
        --      halve >=> halve >=> halve               -- infixr 1 >=>
```

`Monad` typeclass .

```
1.  return x >>= f  =  f x
2.    m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

`m` , `f` `a -> mb` `g` `b -> mc` .

, `>=>` Kleisli .

```
1.    return >=> g  =  g                        -- do { y <- return x ; g y } == g x
2.    f >=> return  =  f                        -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h  =  f >=> (g >=> h)       -- do { z <- do { y <- f x; g y } ; h z }
                                                --  == do { y <- f x ; do { z <- g y; h z } }
```

. .

`Maybe` .

1. - `return x >>= f = fx`

```
return z >>= f
= (Just z) >>= f
= f z
```

2. - `m >>= return = m`

- `Just`

```
Just z >>= return
= return z
= Just z
```

- `Nothing`

```
Nothing >>= return
= Nothing
```

3. - `(m >>= f) >>= g = m >>= (\x -> fx >>= g)`

- Just

```
-- Left-hand side
((Just z) >>= f) >>= g
= f z >>= g

-- Right-hand side
(Just z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- Nothing

```
-- Left-hand side
(Nothing >>= f) >>= g
= Nothing >>= g
= Nothing

-- Right-hand side
Nothing >>= (\x -> f x >>= g)
= Nothing
```

IO

`IO a` `a` . `IO` .

`IO a` , `a` . , `getLine :: IO String` `getLine` - `getLine` .

, `main :: IO ()` Haskell / .

`IO` `IO a` :

- `(>>)` , .

    ```
    -- print the lines "Hello" then "World" to stdout
    putStrLn "Hello" >> putStrLn "World"
    ```

- . . `>>=` . `IO` , `(>>=) :: IO a -> (a -> IO b) -> IO b` .

    ```
    -- get a line from stdin and print it back out
    getLine >>= putStrLn
    ```

- . `do` notation .

    ```
    -- make an action that just returns 5
    return 5
    ```

IO .

. :

```
instance Monad [] where
```

```
  return x = [x]
  xs >>= f = concat (map f xs)
```

. xs >>= f , f :: a -> [b] xs f xs , . **do notation** . . .

```
sumnd xs ys = do
  x <- xs
  y <- ys
  return (x + y)
```

, Control.Monad liftM2 .

```
sumnd = liftM2 (+)
```

:

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

GHC 7.10, Applicative Monad (, Monad Applicative). Applicative (pure , <*>) Monad (return , >>=) .

pure return pure = return . <*> .

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
       -- = mf >>= (\f -> mx >>= (\x -> return (f x)))
       -- = [r   | f <- mf, x <- mx, r <- return (f x)]   -- with MonadComprehensions
       -- = [f x | f <- mf, x <- mx]
```

ap .

Monad "" Applicative

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

.

▪

.

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

Monad Monad m => ma -> a .

, (: "unwrap"). .

```
extract :: Maybe a -> a
extract (Just x) = x          -- Sure, this works, but...
extract Nothing  = undefined  -- We can’t extract a value from failure.
```

, `IO a -> a` . .

`do` -notation . .

```
do x <- mx                                    do x <- mx
   y <- my       is equivalent to                do y <- my
   ...                                              ...
```

```
do let a = b                                  let a = b in
   ...           is equivalent to               do ...
```

```
do m                                          m >> (
   e             is equivalent to               e)
```

```
do x <- m                                     m >>= (\x ->
   e             is equivalent to               e)
```

```
do m             is equivalent to             m
```

, .

```
example :: IO Integer
example =
  putStrLn "What's your name?" >> (
    getLine >>= (\name ->
      putStrLn ("Hello, " ++ name ++ ".") >> (
        putStrLn "What should we return?" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

```
example :: IO Integer
example = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ".")
  putStrLn "What should we return?"
  line <- getLine
  let n = (read line :: Integer)
  return (n + n)
```

Monad

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

>>= :

```
(>>=) :: m a -> (a -> m b) -> m b
```

- `ma` " ". *a*
- `a -> mb` " *a* *b* " .

`>>=` .

`Monad` .

```
return :: a -> m a
```

. `return` `return` .

`return x` **x** . (:)

```
return x >>= f        ≡  f x      --  “left identity” monad law
       x >>= return   ≡  x        -- “right identity” monad law
```

: https://riptutorial.com/ko/haskell/topic/2968/

29:

Examples

, . .

.

.

```
newtype Counter = MkCounter {cValue :: Int}
  deriving (Show)

-- | 'inc c n' increments the counter by 'n' units.
inc :: Counter -> Int -> Counter
inc (MkCounter c) n = MkCounter (c + n)
```

.

- 0 .
- 3
- 3 .
- 5
- 2 .

. .

```
-- | CounterS is a monad.
type CounterS = State Counter

-- | Increment the counter by 'n' units.
incS :: Int-> CounterS ()
incS n = modify (\c -> inc c n)
```

:

```
-- | The computation we want to run, with the state monad.
mComputationS :: CounterS ()
mComputationS = do
  incS 3
  incS 3
  incS 3
  incS 5
  incS 5
```

. .

. OO *(dependency injection*) .

.

- (, , `r`)
- ( ). `a`

  ra

. `ReaderT` .

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

`ReaderT` .

```
type CounterRS = ReaderT Int CounterS
```

`incR` (`ask`), `CounterS` `lift` ( ).

```
-- | Increment the counter by the amount of units specified by the environment.
incR :: CounterRS ()
incR = ask >>= lift . incS
```

.

```
-- | The computation we want to run, using reader and state monads.
mComputationRS :: CounterRS ()
mComputationRS = do
  local (const 3) $ do
    incR
    incR
    incR
    local (const 5) $ do
      incR
      incR
```

: !

.

. , .

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

`w` ( ) , `m` . `a`

, .

```
type CounterWRS = WriterT [Int] CounterRS
```

lift .

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

.

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
  local (const 3) $ do
    incW
    incW
    incW
    local (const 5) $ do
      incW
      incW
```

. (,) .

:

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

. .

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

inc . '

ghci REPL ghci .

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

: https://riptutorial.com/ko/haskell/topic/7752/-

30:

Examples

Monoid

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

`Monoid` :

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- prepending an empty list is a no-op

x `mappend` mempty = x   <->   xs ++ [] = xs  -- appending an empty list is a no-op

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs           -- appending lists is associative
```

Monoids

`mconcat :: [a] -> a` `Monoid` typeclass :

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

`mconcat = foldr mappend mempty` .

Monoids

monoidal : 1 0 . . `newtypes` , `Sum` `Product` .

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

`newtype` .

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

Monoid for ()

`()` `Monoid` . type `()` `mempty` `mappend` .

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

: https://riptutorial.com/ko/haskell/topic/2211/-

31:

- module Name - .
- module Name (functionOne, Type (..)) - functionOne, Type Type .
- import Module -
- MN -
- import (justThisFunction) -
- import (functionName, Type) - functionName Type

Haskell :

- .
- " "

.

-
-

haskell.org .

Examples

`Business.hs` , `import Business` :

```
module Business (
    Person (..), -- ^ Export the Person type and all its constructors and field names
    employees   -- ^ Export the employees function
) where
-- begin types, function definitions, etc
```

. Hierarchical .

.

```
module X (Person (..)) where
```

`People.hs` .

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

:

```
module People (Person (..)) where
```

`Person` `Friend` `Foe` .

module .

```
module People where
```

`Person` , , `isFoe` export .

.

```
import qualified Data.Stream (map) as D
```

`Data.Stream` `map` `Data.Stream` `D.` .

```
D.map odd [1..]
```

`Prelude` `map` .

Prelude . .

`Data.Stream` `map` , `head` `tail` . Prelude . `hiding` :

```
import Data.Stream -- everything from Data.Stream
import Prelude hiding (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- etc
```

, Prelude `Data.Stream` `qualified` .

. ( ) `qualified` .

```
import qualified Data.Stream as D
```

`Prelude` `Data.Stream` `map` .

```
map (== 1) [1,2,3] -- will use Prelude.map
D.map (odd) (fromList [1..]) -- will use Data.Stream.map
```

`import Data.Text as T` , `Text` `T.Text`

. .

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
```

```
Bar.hs
```

.

```
-- file Foo.hs
module Foo where

-- file Bar.hs
module Bar where

-- file Foo/Bar.hs
module Foo.Bar where

-- file Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

:

- .
- .

: https://riptutorial.com/ko/haskell/topic/5234/

32:

Examples

Haskell comprehension . set comprehensions . .

```
[ x | x <- someList ]
```

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

x .

```
[ f x | x <- someList ]
```

.

```
map f someList
```

:

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

x . .

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

x . x .

, .

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

list comprehension guard. .

```
[x    | p x]   ===   if p x then [x] else []
```

. ,

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` must be defined in outer scope
```

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- or,

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)
```

```
-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(`>>=` `infixl 1` ,). :

```
[ x        | x <- [1..4], even x]            -- [2,4]

[ x^2 + 1 | x <- [1..100], even x ]          -- map (\x -> x^2 + 1) (filter even [1..100])
```

. . ,

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

Parallel List Comprehensions ,

```
[(x,y) | x <- xs | y <- ys]
```

~ .

```
zip xs ys
```

:

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

List comprehensions .

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]     -- [(1,2),(3,10)]
```

.

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]    -- [(1,2),(3,10)]
```

`let` , . .

```
[x | x <- [1..4], x <- [x*x+1], even x]        -- [2,10]
```

`do` .

```
[f x | x <- xs]                f  <$> xs          do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]       fs <*> xs          do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]      f  =<< xs          do { x <- xs ; y <- f x ; return y }
```

`Control.Monad.guard` :

```
[x   | x <- xs, even x]                    do { x <- xs ; guard (even x) ; return x }
```

: https://riptutorial.com/ko/haskell/topic/4970/-

33:

Examples

.

, :

"" Functor

```
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

`Free` "Free Monad over `TeletypeF` " .

```
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

`f` `Functor` `Free f` `Monad` `Monad` (`do` notation) `Teletype` .

```
import Control.Monad -- we can use the standard combinators

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

`Teletype a` `Teletype a` "" `IO a` `IO a`

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
  run (PrintLine str x) = putStrLn *> return x
  run (ReadLine f) = fmap f getLine
```

`Teletype a` `IO` `Teletype a` "" .

```
> interpretTeletype mockingbird
hello
```

```
hello
goodbye
goodbye
this will go on forever
this will go on forever
```

.

Free Fix .

```
data Free f a = Return a
              | Free (f (Free f a))

newtype Fix f = Fix { unFix :: f (Fix f) }
```

, Free Fix . Free Fix , Free Return a .

foldFree iterM ?

Free m : iterM :: (Functor f, Monad m) => (f (ma) -> ma) -> (Free fa -> ma) foldFree :: Monad m => (forall x. fx -> mx) -> (Free fa -> ma) . ?

Teletype a IO . Free fa .

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

Pure .

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

Free Teletype ? teletypeF :: TeletypeF (Teletype a) IO a ., IO runIO :: TeletypeF a -> IO a a .

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
runIO (ReadLine k) = fmap k getLine
```

runIO interpretTeletype . teletypeF :: TeletypeF (Teletype a) Free TeletypeF . runIO (runIO teletypeF :: IO (Teletype a)). IO >>= combinator Teletype a .

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

foldFree interpretTeletype , runIO . foldFree .

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

foldFree rank-2 . eta . foldFree Monad m => (f (Free fa) -> m (Free fa)) -> Free fa -> ma ,
eta f Free eta . foldFree eta .

iterM . () f . iterM A *paramorphism* foldFree *catamorphism* .

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

Freer (Prompt, Operational) . Functor .

Freer i :: * -> * . . , State .

```
data StateI s a where
    Get :: StateI s s  -- the Get instruction returns a value of type 's'
    Put :: s -> StateI s ()  -- the Put instruction contains an 's' as an argument and returns
()
```

:>>= . :>>= a ., a , a b , :>>= b .

```
data Freer i a where
    Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

a :>>= . GADT i a .

: - Freer Free . CoYoneda functor CoYoneda .

```
data CoYoneda i b where
  CoYoneda :: i a -> (a -> b) -> CoYoneda i b
```

Freer i Free (CoYoneda i) . Free f ~ CoYoneda i .

```
Pure :: a -> Free (CoYoneda i) a
Free :: CoYoneda i (Free (CoYoneda i) b) -> Free (CoYonda i) b ~
        i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
```

Freer i Freer i ~ Free (CoYoneda i) .

CoYoneda i A Functor i , Freer A Monad i , i Functor .

```
instance Monad (Freer i) where
    return = Return
    Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- using `(->) r`'s instance of Functor, so fmap
= (.)
```

`Freer` .

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

`State s` `Freer (StateI s)` .

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI s a -> State s a
          toState Get = State.get
          toState (Put x) = State.put x
```

: https://riptutorial.com/ko/haskell/topic/1290/-

34:

Examples

~~

```
data Empty a
```

.

```
data Free Empty a
     = Pure a
-- the Free constructor is impossible!
```

```
data Identity a
     = Identity a
```

~~ (Nat,) ~ Writer Nat

```
data Identity a = Identity a
```

.

```
data Free Identity a
     = Pure a
     | Free (Identity (Free Identity a))
```

```
data Deferred a
     = Now a
     | Later (Deferred a)
```

(fst) `(Nat, a)` , `Writer Nat a`

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
```

~~ MaybeT (Nat)

```
data Maybe a = Just a
             | Nothing
```

.

```
data Free Maybe a
     = Pure a
     | Free (Just (Free Maybe a))
```

```
     | Free Nothing
```

```
data Hopes a
     = Confirmed a
     | Possible (Hopes a)
     | Failed
```

(fst) (Nat, Maybe a) , MaybeT (Writer Nat) a

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

(w) ~~ [w]

```
data Writer w a = Writer w a
```

.

```
data Free (Writer w) a
     = Pure a
     | Free (Writer w (Free (Writer w) a))
```

```
data ProgLog w a
     = Done a
     | After w (ProgLog w a)
```

( ) Writer [w] a .

(Const c) ~~ c

```
data Const c a = Const c
```

.

```
data Free (Const c) a
     = Pure a
     | Free (Const c)
```

```
data Either c a
     = Right a
     | Left c
```

(x) ~~ (x)

```
data Reader x a = Reader (x -> a)
```

.

```
data Free (Reader x) a
      = Pure a
      | Free (x -> Free (Reader x) a)
```

```
data Demand x a
      = Satisfied a
      | Hungry (x -> Demand x a)
```

`Stream x -> a`

```
data Stream x = Stream x (Stream x)
```

: https://riptutorial.com/ko/haskell/topic/8256/---

35: -

Examples

reactive-banana-wx GUI . `IO` FRP .

`Control.Event.Handler` `AddHandler a` `a -> IO ()` `addHandler` . `Event a` .

```
import Data.Char (toUpper)

import Control.Event.Handler
import Reactive.Banana

main = do
    (inputHandler, inputFire) <- newAddHandler
```

`a` `String` .

FRP `EventNetwork` . `compile` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
```

`fromAddHandler` `AddHandler a` `Event a` . .

" " .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
    forever $ do
        input <- getLine
        inputFire input
```

`Event` . `Event` . `Event` `Functor` typeclass . `Event` . `Event` [] `IO` .

`Event` s `fmap` .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
```

`Event` . `a -> IO ()` `fmap` `reactimate` .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
        let inputEventReaction = fmap putStrLn inputEvent' -- this has type `Event (IO ())
        reactimate inputEventReaction
```

inputFire "something" "SOMETHING" .

Behavior a . Event Behavior Applicative n Behavior (<$> <*>) n Behavior .

Event a Behavior a accumE .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

accumE Behavior Event .

Event fmap Behavior (<*>) .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

Behavior changes :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
        inputChanged <- changes inputBehavior
```

changes Event a Event (Future a) . reactimate' reactimate reactimate . .

EventNetwork compile EventNetwork .

```
main = do
    (inputHandler, inputFire) <- newAddHandler

    eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
```

```
        reactimate inputEventReaction

    inputFire "This will NOT be printed to the console!"
    actuate eventNetwork
    inputFire "This WILL be printed to the console!"
```

-   : https://riptutorial.com/ko/haskell/topic/4186/---

36:

IO "IO " .

: putStrLn (print), hslogger Debug.Trace .

Examples

hslogger

hslogger Python logging API , stdout stderr .

WARNING .

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                   updateGlobalLogger, warningM)

main = do
  debugM "MyProgram.main" "This won't be seen"
  infoM "MyProgram.main" "This won't be seen either"
  warningM "MyProgram.main" "This will be seen"
```

updateGlobalLogger .

```
  updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "This will now be seen"
```

MyProgram MyParent.Module .

: https://riptutorial.com/ko/haskell/topic/9628/-

37:

Examples

. . Haskell .

`Functor` typeclass Type `F` `Functor` (`Functor F`)

```
fmap :: (a -> b) -> (F a -> F b)
```

`F` "" . ( ) `F a` `a` `fmap` . `Maybe`

```
instance Functor Maybe where
  fmap f Nothing = Nothing     -- if there are no values contained, do nothing
  fmap f (Just a) = Just (f a) -- else, apply our transformation
```

, " `Mappable` `Functor` `Mappable` ?".

Functor `Functor` "Functor" .

. `a -> b` . . `F` `F a -> F b` .

Functor Haskell `a -> b` `F` - `F a -> F b` `F` . `fmap`

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

`Functor` " " `Functor` .

`C` .

- `Obj(C)` ;
- (`Hom(C)` `Hom(C)` . `a` `b` `Obj(C)` morphism `f` `Hom(C)` `f : a -> b` , morphism `a` `b` `hom(a,b)` ;
- (*identity* morphism) - `a : Obj(C)` . `id : a -> a` ;
- (. morphisms) `f : a -> b` , `g : b -> c` morphism `a -> c`

.

```
For all f : a -> x, g : x -> b, then id . f = f and g . id = g
```

```
For all f : a -> b, g : b -> c and h : c -> d, then h . (g . f) = (h . g) . f
```

, ( ) .

Haskell, `Category` Control.category typeclass .

```
-- | A class for categories.
--   id and (.) must form a monoid.
class Category cat where
    -- | the identity morphism
    id :: cat a a

    -- | morphism composition
    (.) :: cat b c -> cat a b -> cat a c
```

, `cat :: k -> k -> *` morphism - `cat ab` (,) `cat ab` . `a` , `b` `c` `Obj(C)` . `Obj(C)` `k` . `k ~ *` .

:

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

`Monad` `Kleisli` `Category` .

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli f . Kleisli g = Kleisli (f >=> g)
```

.

(†) . () `a` () (`id :: a -> a`). (`(.) :: (b -> c) -> (a -> b) -> a -> c`)

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

Hask .

. , . **Hask** `f` , `g` :

```
 f . g == id == g . f
```

, .

`((),a)` `a` `a` . :

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

`f . g == id == g . f` .

, . , **Hask** , **Hask** **endofunctors Hask,** , . endofunctors :

```
F :: * -> *
```

( ) Haskell .

```
fmap (f . g) = (fmap f) . (fmap g)
fmap id = id
```

`[]` , `Maybe` `(-> r)` **Hask** .

. `F :: * -> *` (`forall a . F a -> G a` `forall a . F a -> G a`).

monoid monoidal , morphisms :

```
zero :: () -> M
mappend :: (M,M) -> M
```

Hask endofunctors :

```
return :: a -> m a
join :: m (m a) -> m a
```

.

, `undefined` . **Hask** ( ) . .

Hask

, **X** , **Y** **Z** : **π1 : Z → X** **π2 : Z → Y** ; . , **f1 : W → X** **f2 : W → Y** , **π1 → g = f1** **π2 → g = f2** **g : W → Z** .

Haskell **Hask** . `Z` `A` , `B` .

```
-- if there are two functions
f1 :: W -> A
f2 :: W -> B
-- we can construct a unique function
g  :: W -> Z
-- and we have two projections
p1 :: Z -> A
p2 :: Z -> B
-- such that the other two functions decompose using g
p1 . g == f1
p2 . g == f2
```

`A` , `B` `(A,B)` `fst` `snd` . `f1 :: W -> A` `f2 :: W -> B` . .

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
```

```
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

:

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

A B (A,B) . .

```
data Pair a b = Pair a b
```

, (B,A) (B,A,()) , .

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

. A B . (A,B) (B,A,()) .

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())

iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

. . , (A,(B,Bool)) fst fst . snd fst . snd A B fst . snd :

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

:

```
fst          . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

.

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

, , (A,(B,Bool)) **Hask** A B

Coproduct

A B A B . Either ab . Either a (b,Bool) .

.

. . **X, Y Z i_1 : X → Z I_2 : Y → Z; X Y** . , **f1 : X → W f2 : Y → W** , **g → i1 =**

f1 , g → i2 = f2

Hask .

```
-- if there are two functions
f1 :: A -> W
f2 :: B -> W
-- and we have a coproduct with two inclusions
i1 :: A -> Z
i2 :: B -> Z
-- we can construct a unique function
g  :: Z -> W
-- such that the other two functions decompose using g
g . i1 == f1
g . i2 == f2
```

Hask `A` `B` coproduct `Either ab` .

```
-- Coproduct
-- The two inclusions are Left and Right
data Either a b = Left a | Right b

-- If we have those functions, we can decompose them through the coproduct
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

`Functor` `a` (**Hask**) `F a` `a -> b` (**Hask** morphism) `F a -> F b` . .

.

- (Hask)
- ()

.

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

`Applicative` `Monoidal` .

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
    f <*> fa = (\(f, a) -> f a) <$> (mcat f fa)
```

: https://riptutorial.com/ko/haskell/topic/2261/-

38:

[Data.Vector] . , () . C Storable . Int .

Haskell Wiki .

:

- Data.Vector.Unboxed .
- Data.Vector .
- C Data.Vector.Storable .

;

- . Data.Vector.Generic

Examples

Data.Vector

Data.Vector .

`Data.Vector` `Vector` .

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

.

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]

Prelude Data.Vector> :t x
x :: Vector (Vector Integer)
```

:

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

(`map`) (`fold`)

`map` 'D `fold'd,` `'d and` zip`'d :

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

.

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

.

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

: https://riptutorial.com/ko/haskell/topic/4738/

39:

/	
`data Eval a`	Eval Monad.
`type Strategy a = a -> Eval a`	. , .
`rpar :: Strategy a`	( )
`rseq :: Strategy a`	.
`force :: NFData a => a -> a`	. Control.DeepSeq .

(Simon Marlow) , . Haskell . PDF .

Simon Marlow .

(:) . .

. "". , . . .

. WHNF .

Examples

Monad

Haskell `rpar` `rseq` `Control.Parallel.Strategies` `Eval` Monad .

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  return (a,b)
```

`main` "return", `a` `b` `rpar` .

: `-threaded` .

Rpar

`rpar :: Strategy a` (: `type Strategy a = a -> Eval a`).

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  where
    loop = do
      putStrLn "Enter a number"
      n <- getLine

      let lim = read n :: Int
          hf  = quot lim 2
          result = runEval $ do
            -- we split the computation in half, so we can concurrently calculate primes
            as <- rpar (force (primesBtwn 2 hf))
            bs <- rpar (force (primesBtwn (hf + 1) lim))
            return (as ++ bs)

      forkIO $ putStrLn ("\nPrimes are: " ++ (show result) ++ " for " ++ n ++ "\n")
      loop

-- Compute primes between two integers
-- Deliberately inefficient for demonstration purposes
primesBtwn n m = eratos [n..m]
  where
    eratos []     = []
    eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

.

```
Enter a number
12
Enter a number

Primes are: [2,3,5,7,8,9,10,11,12] for 12

100
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,7
for 100

200000000
Enter a number
-- waiting for 200000000
200
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107,108
for 200

-- still waiting for 200000000
```

rseq

`rseq :: Strategy a` Weak Head Normal Form .

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  rseq a
  return (a,b)
```

`rpar` . , `a` WHNF .

: https://riptutorial.com/ko/haskell/topic/6887/

40:

.

.

```
max :: (Ord a) => a -> a -> a
max m n
  | m >= n = m
  | otherwise = n
```

max :: (Ord a) => a -> a -> a a a () a a . max a a a -> a a -> a . a a a .

max max :: (Ord a) => a -> (a -> a) .

max .

```
Prelude> :t max
max :: Ord a => a -> a -> a

Prelude> :t (max 75)
(max 75) :: (Num a, Ord a) => a -> a

Prelude> :t (max "Fury Road")
(max "Fury Road") :: [Char] -> [Char]

Prelude> :t (max "Fury Road" "Furiosa")
(max "Fury Road" "Furiosa") :: [Char]
```

max 75 max "Fury Road" .

. Haskell a a -> a .

Examples

"". .

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

addOne Int .

```
> addOne 5
6
> map addOne [1,2,3]
[2,3,4]
```

.

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

(+ x) . add .

`add 5 2` 7.

Sectioning .

, "ing" .

```
> (++ "ing") "laugh"
"laughing"
```

. .

.

```
> ("re" ++) "do"
"redo"
```

.

```
> ((++) "re") "do"
"redo"
```

.

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

`-1` `-1` `- 1` . `subtract` .

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

: https://riptutorial.com/ko/haskell/topic/1954/-

41:

Examples

Bool .

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

. QuickCheck 100 True .

prop_ .

quickCheck 100 .

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK, passed 100 tests.
```

quickCheck .

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- plainly not true for all xs

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** Failed! Falsifiable (after 2 tests):
[()]
```

quickCheckAll prop_ quickCheckAll Template Haskell .

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

idempotent :: Eq a => (a -> a) -> a -> Bool
idempotent f x = f (f x) == f x

prop_sortIdempotent = idempotent sort

-- does not begin with prop_, will not be picked up by the test runner
sortDoesNotChangeLength xs = length (sort xs) == length xs


return []
main = $quickCheckAll
```

return [] . Template Haskell .

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent from tree.hs:7 ===
+++ OK, passed 100 tests.
```

`Arbitrary` QuickCheck .

`Arbitrary` `Arbitrary` `Gen` `arbitrary` .

`Arbitrary` .

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- randomly select one of the cases from the list
        End <$> arbitrary,  -- call a's instance of Arbitrary
        Cons <$>
            arbitrary <*>  -- call a's instance of Arbitrary
            arbitrary  -- recursively call NonEmpty's instance of Arbitrary
        ]
```

(==>) .

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

`==>` . QuickCheck .

```
prop_overlySpecific x y = x == 0 ==> x * y == 0

ghci> quickCheck prop_overlySpecific
*** Gave up! Passed only 31 tests.
```

quickcheck . .

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
    longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

`quickCheckWith` `stdArgs` (10) , , , 10 .

: https://riptutorial.com/ko/haskell/topic/1156/-

42:

Examples

data .

```
data Foo = Bar | Biz
```

data = . After = | . . :

Foo . Bar Biz .

```
let x = Bar
```

. Foo x . .

```
:t x
```

```
x :: Foo
```

. .

```
data Foo = Bar String Int | Biz String
```

Bar .

```
:t Bar
```

```
Bar :: String -> Int -> Foo
```

Bar .

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

:

```
data Foo a b = Bar a b | Biz a b
```

Haskell . . . a b .

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

, , , , Person .

.

```
data Person = Person String String Int Int String String String
```

, .

```
getPhone :: Person -> Int
getPhone (Person _ _ _ phone _ _ _) = phone
```

, . .

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
                       , age           :: Int
                       , phone         :: Int
                       , street        :: String
                       , code          :: String
                       , town          :: String }
```

phone .

```
:t phone
phone :: Person' -> Int
```

. :

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

.

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

RecordWildCards

: https://riptutorial.com/ko/haskell/topic/4057/----

43:

( ) . . " " .

`Num` typeclass . (Prelude `Data.Complex`).

```
λ> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- Defined in 'GHC.Num'
instance RealFloat a => Num (Complex a) -- Defined in 'Data.Complex'
instance Num Word -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int -- Defined in 'GHC.Num'
instance Num Float -- Defined in 'GHC.Float'
instance Num Double -- Defined in 'GHC.Float'
```

`Num` "" `(/)` `Fractional` .

```
λ> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- Defined in 'GHC.Real'
instance RealFloat a => Fractional (Complex a) -- Defined in 'Data.Complex'
instance Fractional Float -- Defined in 'GHC.Float'
instance Fractional Double -- Defined in 'GHC.Float'
```

`Real` . `Num` `Ord` , . , ( ).

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- Defined in 'GHC.Real'
instance Real Word -- Defined in 'GHC.Real'
instance Real Integer -- Defined in 'GHC.Real'
instance Real Int -- Defined in 'GHC.Real'
instance Real Float -- Defined in 'GHC.Float'
instance Real Double -- Defined in 'GHC.Float'
```

`RealFrac` .

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- Defined in ‘GHC.Real’
instance RealFrac Float -- Defined in ‘GHC.Float’
instance RealFrac Double -- Defined in ‘GHC.Float’
```

`Floating` (`Fractional`) .

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
  GHC.Float.expm1 :: a -> a
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- Defined in ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- Defined in ‘Data.Complex’
instance Floating Float -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
```

: `sqrt . negate :: Floating a => a -> a` `sqrt . negate :: Floating a => a -> a` `NaN` ("not-a-number") . ( ) .

Examples

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

. Haskell `Num` ( ), `pi` 0 .

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

GHC . `list0 :: Num a => [a]` (: `Num` `Num` `Double`).

`Fractional Int ... '

. .

. :

```
averageOfList ll = sum ll / length ll
```

`(/)` . `(/) :: Fractional a => a -> a -> a` (`length :: Foldable t => ta -> Int`) `Int` (`Int Fractional`) , .

`fromIntegral :: (Num b, Integral a) => a -> b` . `Integral` `Num` :

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

`(+)` ?

```
λ> :t (+)
(+) :: Num a => a -> a -> a
```

`sqrt` ?

```
λ> :t sqrt
sqrt :: Floating a => a -> a
```

`sqrt . fromIntegral` ? `sqrt . fromIntegral` ?

```
sqrt . fromIntegral :: (Integral a, Floating c) => a -> c
```

: https://riptutorial.com/ko/haskell/topic/8616/

44:

Examples

OS X

:

```
brew install haskell-stack
```

helloworld :

```
stack new helloworld simple
```

`helloworld` .

.

```
➜ helloworld ls
LICENSE          Setup.hs          helloworld.cabal src            stack.yaml
```

`src` `Main.hs` `Main.hs` . `helloworld` "". `Main.hs` "Hello, World!" .

Main.hs

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

`helloworld` .

```
stack build # Compile the program
stack exec helloworld # Run the program
# prints "hello world"
```

LTS ()

Stackage Haskell . .

.

`stack.yaml` `stack.yaml` . `stack.yaml` .

```
resolver: lts-6.8
```

Stackage `lts` . `lts-6.8` . .

```
https://www.stackage.org/lts-6.8 # if a different version is used, change 6.8 to the correct
resolver number.
```

Lens-4.13 .

`helloworld.cabal` .

```
  build-depends: base >= 4.7 && < 5
```

:

```
  build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

LTS ( ), resolver number . :

```
resolver: lts-6.9
```

`stack build` LTS 6.9 .

`stack new helloworld simple` "helloworld".

`stack build` .

```
stack exec helloworld-exe
```

```
stack install
```

.

```
/Users/<yourusername>/.local/bin/
```

`stack` . `--profile` .

```
stack build --profile
```

GHC (`-prof`) . `stack` . `+RTS` .

```
stack exec -- my-bin +RTS -p
```

.

```
stack list-dependencies
```

.

. .

```
stack dot --external | grep template-haskell
```

`stack dot` . .

```
stack dot --external | dot -Tpng -o my-project.png
```

.

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

: https://riptutorial.com/ko/haskell/topic/2970/

45:

. State sa s a , " " . mtl transformers .

State . , State ? .

- .
- State .

action :: State sa .

- action .
- s action s s ;
- runState :: State sa -> s -> (a, s) stateful "" .

State . s runState " " . (. .

Examples

.

```
data Tree a = Tree a [Tree a] deriving Show
```

.

```
tag :: Tree a -> Tree (a, Int)
```

State .

```
import Control.Monad.State

-- Function that numbers the nodes of a `Tree`.
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep is where the action happens.  This just gets the ball
    -- rolling, with `0` as the initial counter value.
    evalState (tagStep tree) 0

-- This is one monadic "step" of the calculation.  It assumes that
-- it has access to the current counter value implicitly.
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a subtrees) = do
    -- The `get :: State s s` action accesses the implicit state
    -- parameter of the State monad.  Here we bind that value to
    -- the variable `counter`.
    counter <- get

    -- The `put :: s -> State s ()` sets the implicit state parameter
    -- of the `State` monad.  The next `get` that we execute will see
```

```
    -- the value of `counter + 1` (assuming no other puts in between).
    put (counter + 1)

    -- Recurse into the subtrees.  `mapM` is a utility function
    -- for executing a monadic actions (like `tagStep`) on a list of
    -- elements, and producing the list of results.  Each execution of
    -- `tagStep` will be executed with the counter value that resulted
    -- from the previous list element's execution.
    subtrees' <- mapM tagStep subtrees

    return $ Tree (a, counter) subtrees'
```

postIncrement .

get put + 1 postIncrement C :

```
postIncrement :: Enum s => State s s
postIncrement = do
    result <- get
    modify succ
    return result
```

.

.

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
    a' <- action a
    subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

postIncrement tagStep :

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

Traversable

mapTreeM Traversable .

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

Monad

`Applicative` (`<*>`) .

`tag` .

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

`Tree` `Traversable` .

`Traversable`

GHC `DeriveTraversable` .

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}

data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

: https://riptutorial.com/ko/haskell/topic/5740/-

46:

Examples

(!) .

```
foo (!x, y) !z = [x, y, z]
```

x z weak head normal form . :

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

Bang Haskell 2010 BangPatterns .

. .

(,) . ( ) (: - \x -> ..). .

, ( ) . , .

.

(: let x = 1:x in x) . [1,1, ...] . let y = 1+y in y undefined .

RNF . (WHNF) . e e Con e1 e2 .. en Con . \x -> e1 ; f e1 e2 .. en , f n (, e). , e1..en e1..en . undefined .

Haskell WHNF . - e , WHNF , .

seq WHNF . seq xy y (seq xy y). y WHNF x WHNF . -XBangPatterns bang WHNF .

```
f !x y = ...
```

f x WHNF y () . .

```
data X = Con A !B C .. N
```

Con B , (, 2) B WHNF .

(~pat) . , . .

.

```
f1 :: Either e Int -> Int
```

```
f1 ~(Right 1) = 42
```

```
λ» f1 (Right 1)
42
λ» f1 (Right 2)
42
λ» f1 (Left "foo")
42
λ» f1 (error "oops!")
42
λ» f1 "oops!"
*** type mismatch ***
```

`Left` .

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
3
λ» f2 (Right (error "oops!"))
*** Exception: oops!
λ» f2 (Left "foo")
*** Exception: lazypat.hs:5:1-21: Irrefutable pattern failed for pattern (Right x)
λ» f2 (error "oops!")
*** Exception: oops!
```

`let` , :

```
act1 :: IO ()
act1 = do
    ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
    ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

`act1` , `act2` `putStrLn s1` `[s1, s2]` `s1` .

```
λ» act1
> ["foo"]
Done
λ» act2
> ["foo"]
*** readIO: no parse ***
```

`data` (`!`) . .

.

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

: https://riptutorial.com/ko/haskell/topic/3798/

47:

TypeFamilies . GADT . GeneralizedNewTypeDeriving . Haskell .

SafeNewtypeDeriving .

Examples

Haskell Wiki :

```
type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool
```

x Inspect . x .

, x . RoleAnnotations .

```
type role Inspect nominal
```

.

```
data List a = Nil | Cons a (List a)

type family DoNotInspect x
type instance DoNotInspect x = List x
```

DoNotInspect x x .

, X . RoleAnnotations .

```
type role DoNotInspect representational
```

. .

: https://riptutorial.com/ko/haskell/topic/8753/

48:

, 0 , 1 , 42 , ...

Num fromInteger Num a => a . , Num

.

: 0.0 , -0.1111 , ...

fromRational Fractional a => a - Fractional

GHC OverloadedStrings String fromString . Data.String.IsString Data.String.IsString Data.String.IsString .

String Text ByteString .

[1, 2, 3] . GHC 7.8 OverloadedLists .

[] .

```
> :t []
[] :: [t]
```

OverloadedLists .

```
[] :: GHC.Exts.IsList l => l
```

Examples

```
Prelude> :t 1
1 :: Num a => a
```

Num .

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
it :: Double
Prelude> 1 :: Word
1
it :: Word
```

.

Prelude> 1 :: String

```
<interactive>:
    No instance for (Num String) arising from the literal `1'
    In the expression: 1 :: String
    In an equation for `it': it = 1 :: String
```

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

. Fractional .

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

.

```
Prelude> 1.0 :: Int
<interactive>:
    No instance for (Fractional Int) arising from the literal `1.0'
    In the expression: 1.0 :: Int
    In an equation for `it': it = 1.0 :: Int
```

(,) . .

```
Prelude> :t "foo"
"foo" :: [Char]
```

OverloadedStrings .

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

. OverloadedStrings fromString String Text .

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"

-- The following two examples are only allowed with OverloadedStrings

withText :: Text
```

```
withText = "Hello Text"      -- instead of: withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- instead of: withBS = Data.ByteString.pack "Hello ByteString"
```

`pack` `String` (`[Char]`) `Text` `ByteString` .

`OverloadedStrings` .

GHC OverloadedLists .

Data.Map .

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

(M.fromList) :

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

: https://riptutorial.com/ko/haskell/topic/369/--

49:

- foreign import ccall unsafe "foo"hFoo :: Int32 -> IO Int32 {- `foo` Haskell `hFoo` . -}

Cabal C C ++ . , `ao` `bo` ( ) `C-sources: ac, bc` cabal . # 12152 . Cabal `C-sources C-sources: bc, ac` `C-sources` . `C-sources: bc, ac` . `C-sources` .

(.h) C ++ `#ifdef __cplusplus` guard `#ifdef __cplusplus` . GHC C ++ . C ++ .

`ccall` . `ccall` `stdcall` ( ) . `unsafe` . , 1 .

Examples

C

C C . C .

foo.c :

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs :

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

`unsafe` 'safe' , C Haskell . `foo` C Haskell `unsafe` .

C cabal .

foo.cabal :

```
name:                 foo
version:              0.0.0.1
build-type:           Simple
extra-source-files:   *.c
cabal-version:        >= 1.10

executable foo
  default-language: Haskell2010
  main-is:        Foo.hs
  C-sources:      foo.c
  build-depends:  base
```

.

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

Haskell C .

C . GUI . C .

C :

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

Haskell .

```
foreign import ccall "header.h event_callback_add"
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

`Object_Event_Cb` C `Object_Event_Cb` Haskell `Callback` .

```
type Callback = Ptr () -> Ptr () -> IO ()
```

, `Callback` . `FunPtr Callback` :

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

C .

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- callback code
    return ()
callbackAdd cpPtr
```

`FunPtr` .

```
freeHaskellFunPtr cbPtr
```

: https://riptutorial.com/ko/haskell/topic/7256/--

50:

Examples

Servant API .

- ( )
- (haskell) .
- ,
-
- ...

Servant API . API .

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int
} deriving (Eq, Show, Generic)

instance ToJSON User  -- automatically convert User to JSON
```

API .

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
```

`/users` `SortBy` `sortby` `/users` `GET` `User` JSON .

.

```
-- This is where we'd return our user data, or e.g. do a database lookup
server :: Server UserAPI
server = return [User "Alex" 31]

userAPI :: Proxy UserAPI
userAPI = Proxy

app1 :: Application
app1 = serve userAPI server
```

`8081` API .

```
main :: IO ()
main = run 8081 app1
```

Stack Servant API . .

Yesod

Yesod `stack new` .

- `yesod-minimal` . Yesod .
- `yesod-mongo` . MongoDB DB .
- `yesod-mysql` . MySQL DB .
- `yesod-postgres` . PostgreSQL DB .
- `yesod-postgres-fay` . PostgreSQL DB . Fay .
- `yesod-simple` . .
- `yesod-sqlite` . SQlite DB .

`yesod-bin` `yesod` . `yesod` .

`Application.hs` . .

`Foundation.hs` `App` . `HandlerT` `getYesod` `getYesod` .

`Import.hs` `Import.hs` .

`Model.hs` DB Template Haskell . DB .

`config/models` DB . `Model.hs` .

`config/routes` URI . HTTP `{method}{RouteR}` .

`static/` . `Settings/StaticFiles.hs` .

`templates/` .

, `Handler/` .

IO `HandlerT` . , , `runDB` DB , IO , . HTML `defaultLayout` .

: https://riptutorial.com/ko/haskell/topic/4721/-

51:

Examples

. 3 .

-

- : :

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
    Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

-

typeclass . .

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- Other instances might appear in other modules, but two instances cannot overlap
-- in a way that would produce different results.
```

. . `VectorSpace` .

```
class VectorSpace v where
  type Scalar v :: *
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

Scalar . . , .

, . , typecheck :

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

, bar Num Literal, . Bar "inverse direction" . injective † invertible (Bar a = String).

---

† .

.

```
{-# LANGUAGE TypeFamilies #-}
data family List a
data instance List Char = Nil | Cons Char (List Char)
data instance List () = UnitList Int
```

Nil :: List Char UnitList :: Int -> List ()

---

typeclasses . typeclass " " . , .

```
class Container f where
  data Location f
  get :: Location f -> f a -> Maybe a

instance Container [] where
  data Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | otherwise      = Nothing

instance Container Tree where
  data Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
  get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

. . , servant Server a a . Proxy . , servant serve ... Proxy a -> Server a -> ... a Proxy a Proxy data .

: https://riptutorial.com/ko/haskell/topic/2955/-

52:

Examples

. .

. :

```
type Color = Red | Yellow | Green
```

3 . `Bool` **2** .

```
type Bool = True | False
```

. . `f` `g` .

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

. .

, 2 .

```
type Bit  = I    | O
type Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
booleanBit False = O
```

`bitValue . booleanBit == id == booleanBit . bitValue` `bitValue . booleanBit == id == booleanBit . bitValue`

1 0

1 . Haskell, unit `()` type `()` . `()` .

0 . Haskell **Void** , `Data.Void` `Data.Void` . .

```
data Void
```

.   .

```
data Sum a b = A a | B b
data Prod a b = Prod a b
```

.

,   `Either` `(,)`   .   .

```
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

. , 1 + 2, 2 + 1 3   . 1 + 2 = 3 = 2 + 1.

```
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())     = Red
f (Right True)  = Green
f (Right False) = Blue

g :: Color -> Sum () Bool
g Red   = Left ()
g Green = Right True
g Blue  = Right False

f' :: Sum Bool () -> Color
f' (Right ())   = Red
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

commutativity, associativity distributivity   .

```
-- Commutativity
Sum a b           <=> Sum b a
Prod a b          <=> Prod b a
-- Associativity
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- Distributivity
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

.

```
data List a = Nil | Cons a (List a)
```

,

(a) = 1 + a * (a)

List (a) .

(a) = 1 + a + a * a + a * a * a + a * a * a * a + ...

`[]` . `[x]` `a` ; `[x,y]` `a` ; . List .

```
-- Not working Haskell code!
data List a = Nil
            | One a
            | Two a a
            | Three a a a
            ...
```

. :

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

.

(a) = 1 + a * (a) * (a)

.

(a) = 1 + a + 2 (a * a) +5 (a * a * a) + 14 (a * a * a * a) + ...

, n n .

. .

`(a,a,a)` .

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

.

d / da (a * a * a) = 3 * a * a

.

. , `a` n `b` m , `a -> b` n m .

, `Bool -> Bool` `(Bool,Bool)` , $2 * 2 = 2^2$.

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
```

```
iso2 (x,y) = (\p -> if p then x else y)
```

: https://riptutorial.com/ko/haskell/topic/4905/-

53:

TypeApplications .

TypeApplications .

{-# LANGUAGE TypeApplications #-} .

Examples

. .

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

. a Num , Show . , , a Int

```
main = print (x :: Int)
```

.

```
main = print @Int x
```

print .

```
print :: Show a => a -> IO ()
```

a a . .

. ? ! TypeApplications TypeApplications . .

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

.

Java, C # C ++ generics / templates .

C # .

```
public static T DoNothing<T>(T in) { return in; }
```

float DoNothing(5.0f) DoNothing<float>(5.0f) . .

Haskell .

```
doNothing :: a -> a
doNothing x = x
```

`ScopedTypeVariables` , `Rank2Types` `RankNTypes` .

```
doNothing :: forall a. a -> a
doNothing x = x
```

`doNothing 5.0` `doNothing @Float 5.0` .

. ?

```
const :: a -> b -> a
```

`const @Int` `a` `Int` , `b` ? `const :: forall a b. a -> b -> a` `forall` `const :: forall a b. a -> b -> a` . `a` , `b` .

. , .

? .

```
const @_ @Int
```

.

```
const @_ @Int :: a -> Int -> a
```

.

```
class SizeOf a where
    sizeOf :: a -> Int
```

. `sizeOf` `a` `sizeOf` , .

`Proxy` .

```
data Proxy a = Proxy
```

. .

```
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

. ? `sizeOf :: Int` , `sizeOf :: SizeOf a => Int` `sizeOf :: forall a. SizeOf a => Int` `sizeOf :: forall a. SizeOf a => Int` .

. `sizeOf` , `Int` . `a` . `{-# LANGUAGE AllowAmbiguousTypes #-}` . .

, ! `sizeOf @Int` . `a` `Int` `a` . !

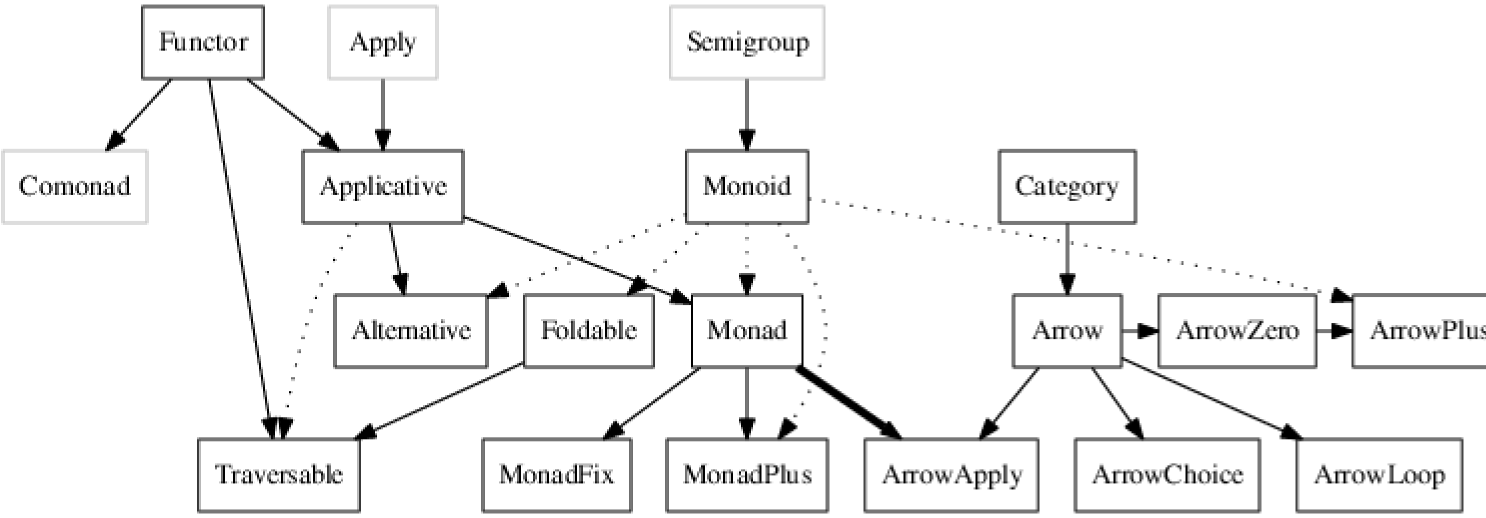
: https://riptutorial.com/ko/haskell/topic/10767/-

54:

. , , .

Haskell `base` . .

Typeclassopedia .



Examples

Haskell . `Maybe` .

`Maybe` . , `Maybe` . `Maybe` .

.

```
Maybe a = Just a | Nothing
```

`Maybe` `Just` `Nothing` . `Just` `Maybe` , `Nothing` . `Just "foo"` `Maybe String` . `Maybe` . `Nothing` `Maybe a` `a` .

`Maybe` . `Either` , `IO` List . . .

. , .

Haskell `Functor` typeclass . .

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

`fmap` `fmap` . , , , a b . ( ) a . b ( ) .

, `fmap` functor . `Functor` , . `Applicative` `Monad` .

Type : Ord

Haskell . , `Ord` `Eq` `Ordering` `compare` . `Ord` `min` `max` .

`=>` `Ord` `Eq` `a` .

```
data Ordering = EQ | LT | GT

class Eq a => Ord a where
    compare :: Ord a => a -> a -> Ordering
    (<)     :: Ord a => a -> a -> Bool
    (<=)    :: Ord a => a -> a -> Bool
    (>)     :: Ord a => a -> a -> Bool
    (>=)    :: Ord a => a -> a -> Bool
    min     :: Ord a => a -> a -> a
    max     :: Ord a => a -> a -> a
```

`compare` .

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- Note the use of (==) inherited from Eq
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
               LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
               LT -> y
               GT -> x
```

`Ord` `compare` `(<=)` , .

Eq

Prelude `IO` (`Int` , `String` , `Eq a => [a]`) `Eq` . `Eq` .

```
> 3 == 2
False
> 3 == 3
True
```

- `(==) :: Eq a => a -> a -> Boolean` `(/=) :: Eq a => a -> a -> Boolean` ( )

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

---

* Ord

Ord

`Ord` `Int` , `String` `[a]` (: `Ord a` `Ord a`) . `Ord` "" . , "" `Ord` .

`Ord` `(<=)` , `(<)` , `(>)` , `(>=)`

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

* `compare :: Ord a => a -> a -> Ordering` `(<=) :: Ord a => a -> a -> Boolean` (`compare` `(<=)`)

* `compare :: Ord a => a -> a -> Ordering`
* `(<=) :: Ord a => a -> a -> Boolean`
* `(<) :: Ord a => a -> a -> Boolean`
* `(>=) :: Ord a => a -> a -> Boolean`
* `(>) :: Ord a => a -> a -> Boolean`
* `min :: Ord a => a -> a -> a`
* `max :: Ord a => a -> a -> a`

* Eq

Monoid `Monoid` , . `Monoid` (`mappend` `(<>)`) , "" (`mempty`) , :

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

`Monoid` " ". `Monoid` . , `Monoid` `Monoid` (`Foldable` `Traversable`).

* `mempty :: Monoid m => m`
* `mappend :: Monoid m => m -> m -> m`

. , , .

`Integer` (`Int` , `Integer` , `Word32`) (`Double` , `Rational` ,) . , , † .

. .

* `fromInteger :: Num a => Integer -> a` . ( ). Haskell `Integer` `Int` `Complex` `Double` 5 .

- `(+) :: Num a => a -> a -> a` . , associative commutative , ,

```
a + (b + c) ≡ (a + b) + c
a + b ≡ b + a
```

- `(-) :: Num a => a -> a -> a` . ( ) :

```
(a - b) + b ≡ (a + b) - b ≡ a
```

- `(*) :: Num a => a -> a -> a` . (multiplication), :

```
a * (b * c) ≡ (a * b) * c
a * (b + c) ≡ a * b + a * c
```

.

- `negate :: Num a => a -> a` . . `-1` `negate 1` .

```
-a ≡ negate a ≡ 0 - a
```

- `abs :: Num a => a -> a` . .

```
abs (-a) ≡ abs a
abs (abs a) ≡ abs a
```

`abs a ≡ 0` `a ≡ 0` .

. `abs a >= 0` . , `abs` (,) ‡ .

- `signum :: Num a => a -> a` . sign `-1` `1` . 0 . `signum` .

```
abs (signum a) ≡ 1   -- unless a≡0
signum a * abs a ≡ a -- This is required to be true for all Num instances
```

Haskell 2010 6.4.4 `Num` .

, hmatrix `Num` . . `+` `-` , `*` . , `*` ?
- . `VectorSpace` .

---

† , "" `(-4 :: Word32) == 4294967292` : `(-4 :: Word32) == 4294967292` .

‡ : . `Num` `abs` `signum` . .

: https://riptutorial.com/ko/haskell/topic/1879/-

55:

- cabal <command> <command> .
- **[]**
    -
        -
    -
        -
    -
        -
    -
        -
    -
        -
    -
        -
    -
        - Cabal .
- **[]**
    -
        - / ()
    -
        - .cabal ( )
    -
        - .
    -
        - /
    -
        -
    -
        - .
    - repl
        -
    -
        - /
    -
        - /
    -
        - .
    - sdist
        - (.tar.gz)
    -
        - Hackage .
    -
        -
    -
        -
    -
        -
    -
        - Haddock HTML
    -
        - HsColour HTML .
    -
        - .
    -
        - .
- **[ ]**
    -
        - / /
            - cabal [FLAGS]
            - [FLAGS]
            - Cabal [FLAGS] PATHS
            - [FLAGS] PATHS
            - cabal [FLAGS]
            - cabal hc-pkg [] [-] [-] [ARGS]
    -
        -
    - repl
        -

Examples

: aeson :

```
cabal install aeson
```

Haskell . (: Haskell).

Haskell :

```
cabal sandbox init
```

`cabal install` .

:

```
cabal sandbox hc-pkg list
```

:

```
cabal sandbox delete
```

:

```
cabal sandbox add-source /path/to/dependency
```

: https://riptutorial.com/ko/haskell/topic/4740/

56:

Applicative f :: * -> * .

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

f Functor . pure Applicative . <*> ("") Applicative fmap .

Applicative .

```
pure id <*> a = a                              -- identity
pure (.) <*> a <*> b <*> c = a <*> (b <*> c)   -- composition
pure f <*> pure a = pure (f a)                 -- homomorphism
a <*> pure b = pure ($ b) <*> a                -- interchange
```

Examples

Applicative Functor Functor fmap . Applicative .

```
class Functor f => PairingFunctor f where
  funit :: f ()                  -- create a context, carrying nothing of import
  fpair :: (f a,f b) -> f (a,b)  -- collapse a pair of contexts into a pair-carrying context
```

Applicative .

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

,

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

Maybe (functor) .

```
instance Applicative Maybe where
    pure = Just

    Just f <*> Just x = Just $ f x
    _ <*> _ = Nothing
```

pure Just Maybe Maybe . (<*>) Maybe A Maybe . (Just) . Nothing .

`<*> :: [a -> b] -> [a] -> [b]` Cartesian :

```
fs <*> xs = [f x | f <- fs, x <- xs]
         -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

, , . 2 2 :

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

"" `Applicative` . .

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

`Stream` `Applicative` . `pure` - .

```
instance Applicative Stream where
    pure x = let s = Stream x s in s
    Stream f fs <*> Stream x xs = Stream (f x) (fs <*> xs)
```

`ZipList` newtype "zippy" `Applicative` .

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

`zip` `Applicative` `pure` .

```
    pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

:

```
ghci> getZipList $ ZipList [(+1),(+2)] <*> ZipList [3,30,300]
[4,32]
```

. `nx 1` (`nx 1`) 1 (`1 xm`) `nxm` (,); 1 1 ( ).

`(->) r` `pure` `<*>` `K` `S` .

```
pure :: a -> (r -> a)
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

`pure` `const` `<*>` .

```
instance Applicative ((->) r) where
    pure = const
    f <*> g = \x -> f x (g x)
```

"" . , `(->) Nat` , isomorphic ...

```
-- | Index into a stream
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | List all the return values of the function in order
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

`Applicative` .

: https://riptutorial.com/ko/haskell/topic/8162/-

57:

- bimap :: (a -> b) -> (c -> d) -> pac -> pbd
- :: (a -> b) -> pac -> pbc
- : (b -> c) -> pab -> pac

`Functor` . , `f` A `Functor` , `fa` `a -> b` `fb` (`fmap`).

`Bifunctor` . `f` `Bifunctor` , `fab` , `a -> c` `b -> d` `bimap` `fcd` .

`first` int `fmap` , 1 `second` int `fmap` , `bimap` 1 2 covariantly .

Examples

Bifunctor

2

`(,)` `Bifunctor` .

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

`bimap` .

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

`Either`

`Either` `Bifunctor` `Left` `Right` .

```
instance Bifunctor Either where
    bimap f g (Left x)  = Left (f x)
    bimap f g (Right y) = Right (g y)
```

`first` `second` (`bimap`).

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

,

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

Bifunctor

`Bifunctor` (`f :: * -> * -> *`) , .

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

`bimap` `fmap` .

`f` `Bifunctor` *functor* *bifunctor* .

```
bimap id id = id  -- identity
bimap (f . g) (h . i) = bimap f h . bimap g i  -- composition
```

`Bifunctor` `Data.Bifunctor` . GHC 7.10 . `bifunctors` .

: https://riptutorial.com/ko/haskell/topic/8020/-

58: GHC

Glasgow Haskell (GHC) Haskell 2010 . `module` `LANGUAGE` programa . GCH 7 .

`LANGUAGE` `{-# LANGUAGE ExtensionOne, ExtensionTwo ... #-}` . `{-# LANGUAGE` `#-}` . `LANGUAGE` .

Examples

MultiParamTypeClasses

. MPTC .

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
    convert i = fromIntegral i
```

.

MPTC .

FlexibleInstances

.

```
All instance types must be of the form (T a1 ... an)
where a1 ... an are *distinct type variables*,
and each type variable appears at most once in the instance head.
```

, `[a]` `[Int]` . `FlexibleInstances` .

```
class C a where

-- works out of the box
instance C [a] where

-- requires FlexibleInstances
instance C [Int] where
```

Haskell `String` . `String` `[Char]` . `Text` `ByteString` .

`OverloadedStrings`

```
"test" :: Data.String.IsString a => a
```

. `[Char]` Data.Text Data.ByteString .

SQL SQL Postgresql-simple `OverloadedStrings` .

`IsString` `fromString` . † :

```
data Foo = A | B | Other String deriving Show

instance IsString Foo where
  fromString "A" = A
  fromString "B" = B
  fromString xs  = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

---

† Lyndon Maydwell (GitHub `sordina`) .

TupleSections

() .

```
(a,b) == (,) a b

-- With TupleSections
(a,b) == (,) a b == (a,) b == (,b) a
```

N-

2 (arity)

```
(,2,) 1 3 == (1,2,3)
```

:

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

.

```
map (\a -> (a, "tag")) [1,2,3]
```

UnicodeSyntax

.

ASCII		
`::`	∷	

ASCII		
->	→	, , case
=>	⇒	
forall	∀	
<-	←	do
*	★	( ) (: Int :: ★)
>-	⤚	Arrows proc
-<	⤙	Arrows proc
>>-	⤜	Arrows proc
-<<	⤛	Arrows proc

:

```
runST :: (forall s. ST s a) -> a
```

```
runST :: (∀ s. ST s a) → a
```

---

* ★ . * ★ * (*) . :

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

() 16 (0x 0X) (0o 0O). BinaryLiterals (0b 0B) .

```
0b1111 == 15     -- evaluates to: True
```

ExistentialQuantification

† , , , .

abstract-base-class : contains .

```
data S = forall a. Show a => S a
```

GADT .

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

: , `S` , `show` , .

```
instance Show S where
    show (S a) = show a    -- we rely on (Show a) from the above
```

:

```
ss = [S 5, S "test", S 3.0]
```

(polymorphic) .

```
mapM_ print ss
```

Existentials , Haskell . `Show` . `S` . . .

, . , "". `S` (`Show`) " ". . `Couldn't match type 'a0' with '()' 'a0' is untouchable` `Couldn't match type 'a0' with '()' 'a0' is untouchable` .

---

† . ( ).

Existential Rank-N . . `show` ( ) . . , . . `{-# LANGUAGE Rank2Types #-}` .

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

`\arg -> case arg of` `\arg -> case arg of` `\case` .

.

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

.

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
```

```
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

LambdaCase .

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

RankNTypes

.

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

String , int . ! .

, ! GHC `a` . , :

```
putStrLn (show' string)
```

GHC `show' :: String -> String` , `string` A `String` . `show' int` `show' int` .

`RankNTypes` `show'` .

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

`show'` , . `a`

`RankNTypes` `forall ...` . , N .

GHC 7.8 .

OverloadedLists, OverloadedStrings desugared :

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

Set , Vector Map .

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

IsList GHC.Exts .

IsList Item , fromList :: [Item l] -> l , toList :: l -> [Item l] fromListN :: Int -> [Item l] -> l . fromListN . .

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
  toList   = Set.toList
```

- GHC .

a, b, c x x a, b c .

```
class SomeClass a b c x | a b c -> x where ...
```

. abc x .

.

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

MTL .

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

MonadReader a ((->) Foo) => a a ~ Foo . .

SomeClass x abc . .

GADT

. , ADT

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

, IntLit :: Int -> Expr a : universially , a Expr a . a ~ Bool IntLit :: Int -> Expr Bool If (IntLit 1) e1 e2 . If If .

Generalized Algebraic Data Types . Expr GADT .

```
data Expr a where
  IntLit :: Int -> Expr Int
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

IntLit IntLit Int -> Expr Int IntLit 1 :: Expr Bool .

GADT . , Expr a Expr a .

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- Here we can use `(+)` because x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- Here we can use `not` because b :: Bool
   not b
crazyEval (If b thn els) =
  -- Because b :: Expr Bool, we can use `crazyEval b :: Bool`.
  -- Also, because thn :: Expr a and els :: Expr a, we can pass either to
  -- the recursive call to `crazyEval` and get an a back
  crazyEval $ if crazyEval b then thn else els
```

, IntLit x a ~ Int (a ~ Bool not if_then_else_) (+) .

ScopedTypeVariables

ScopedTypeVariables . :

```
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

a , b c (where a . ScopedTypeVariables .

PatternSynonyms

.

Data.Sequence . Seq , O (1) (un) consing (un) snocing .

. Seq .

```
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
```

```
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

.

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
    x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

.

```
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

`PatternSynonyms` snoc-list .

```
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

`uncons` :

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

RecordWildCards

RecordWildCards .

GHC : https://riptutorial.com/ko/haskell/topic/1274/-ghc--

59:

Examples

GADTs .

```
data DataType a where
    Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
    Constr3 :: DataType Int
```

GADT . N-ary (nullary) .

DataType Constr1 , Constr2 Constr3 .

Constr1 . data DataType a = Constr1 Int a (Foo a) | ...

Constr2 a Show , . , a Show .

```
foo :: DataType a -> String
foo val = case val of
    Constr2 x -> show x
    ...
```

Show a , -> .

Constr3 DataType Int , DataType a A Constr3 , a ~ Int . .

: https://riptutorial.com/ko/haskell/topic/2971/----

60:

Examples

.

```
main = do
    input <- getContents
    putStr input
```

:

```
This is an example sentence.
And this one is, too!
```

:

```
This is an example sentence.
And this one is, too!
```

: . , 50MiB `getContents` Haskell ( ) . 50MiB .

```
main = do
    line <- getLine
    putStrLn line
```

:

```
This is an example.
```

:

```
This is an example.
```

```
readFloat :: IO Float
readFloat =
    fmap read getLine


main :: IO ()
main = do
    putStr "Type the first number: "
    first <- readFloat

    putStr "Type the second number: "
    second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

:

```
Type the first number: 9.5
Type the second number: -2.02
```

:

```
9.5 + -2.02 = 7.48
```

I / O

`Handle` `System.IO` .. , `getLine :: IO String` `hGetLine :: Handle -> IO String` .

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- show file contents line by line
    end <- hIsEOF handle
    when ( not end ) $ do
        line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
    hPutStr stderr "Type a filename: "
    filename <- getLine
    handle <- openFile filename ReadMode
    dumpFile handle filename 1
    hClose handle
```

`example.txt` :

```
This is an example.
Hello, world!
This is another example.
```

:

```
Type a filename: example.txt
```

:

```
example.txt:1: This is an example.
example.txt:2: Hello, world!
example.txt:3: This is another example
```

(end-of-file)

I / O `isEOF` Haskell `isEOF` EOF . .

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
    end <- isEOF
    if end then
        putStrLn $ "End-of-file reached at line " ++ show line ++ "."
    else do
        getLine
        eofTest $ line + 1


main :: IO ()
main =
    eofTest 1
```

:

```
Line #1.
Line #2.
Line #3.
```

:

```
End-of-file reached at line 4.
```

, † / . IO .

```
-- | The interesting part of the program, which actually processes data
--   but doesn't do any IO!
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | A simple wrapper that only fetches the data from disk, lets
--   'reverseWords' do its job, and puts the result to stdout.
main :: IO ()
main = do
   content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

`loremipsum.txt`

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

.

```
elit
adipiscing
consectetur
amet,
```

```
sit
dolor
ipsum
Lorem
```

mapM_ putStrLn .

---

† , . , Haskell . . . readFile Data.Text Data.Text .

IO `main`

Haskell IO () main .

```
main :: IO ()
main = putStrLn "Hello world!"
```

IO . Hello world! .

main IO a IO a .

```
other :: IO ()
other = putStrLn "I won't get printed"

main :: IO ()
main = putStrLn "Hello world!"
```

. other .

other main . main IO . IO do -notation .

```
other :: IO ()
other = putStrLn "I will get printed... but only at the point where I'm composed into main"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

,

```
Hello world!
I will get printed... but only at the point where I'm composed into main
```

other main .

IO

. (:).

Haskell ( ) IO . , IO . , IO Int Int I / O . IO , IO (, sensical) IO IO .

x IO x main Haskell .

IO

IO . IO .

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

( ) IO IO , IO IO .

```
return :: a -> IO a
```

(,) IO IO . , I / O .

. , (>>) :: IO a -> IO b -> IO b (>>=) .

.

```
main :: IO ()
main =
  putStrLn "What is your name?" >>
  getLine >>= \name ->
  putStrLn ("Hello " ++ name ++ "!")
```

putStrLn :: String -> IO () getLine :: IO String .

: (>>= , >> return) .

IO

IO . , s1 ; s2 s1 , s2 , s1 >> s2 .

IO . return . IO . , return () >> putStrLn "boom" "boom" .

IO .

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

ID, ID . .

```
(>=>) :: (a -> IO b) -> (b -> IO c) -> a -> IO c
(f >=> g) x = (f x) >>= g
```

```
return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
(f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

I / O . , . . putStrLn "X" >> putStrLn "Y" "XY" . lazily I / O . , I / O . getContents readFile . Lazy I / O Haskell .

IO do

Haskell IO IO . do notation * >>= , >> return .

do notation . .

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")


main = do {
  putStrLn "What is your name?" ;
  name <- getLine ;
  putStrLn ("Hello " ++ name ++ "!")
  }
```

.

---

* do .

'a' "IO" " "

" IO a , ? : " a (:) ?

. ! IO a " (a) " a main (b) . , " ". a

, , IO , . a IO , do -notation :

```
-- assuming
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "My computation resulted in: " ++ show int

newComputation :: IO ()
newComputation = do
  int <- myComputation       -- we "bind" the result of myComputation to a name, 'int'
  putStrLn $ getMessage int  -- 'int' holds a value of type Int
```

(getMessage) Int String , () IO myComputation do notation . IO , newComputation . .

stdout

Haskell 2010 Prelude IO .

`putChar :: Char -> IO ()` - char stdout .

```
Prelude> putChar 'a'
aPrelude>  -- Note, no new line
```

`putStr :: String -> IO ()` - String stdout .

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- Note, no new line
```

`putStrLn :: String -> IO ()` - stdout String .

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

`print :: Show a => a -> IO ()` - Show stdout a

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe is an instance of the `Show` type class
Just 'a'
Prelude> print Nothing
Nothing
```

`deriving` `Show` .

```
-- In ex.hs
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- Person is an instance of `Show`, thanks to `deriving`
```

GHCi :

```
Prelude> :load ex.hs
[1 of 1] Compiling ex             ( ex.hs, interpreted )
Ok, modules loaded: ex.
*Main> main  -- from ex.hs
Person {name = "Alex"}
*Main>
```

`stdin`

Haskell 2010 Prelude IO .

getChar :: IO Char - stdin Char

```
-- MyChar.hs
main = do
  myChar <- getChar
  print myChar

-- In your shell

runhaskell MyChar.hs
a -- you enter a and press enter
'a'  -- the program prints 'a'
```

getLine :: IO String - stdin String . .

```
Prelude> getLine
Hello there!  -- user enters some text and presses enter
"Hello there!"
```

read :: Read a => String -> a -

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

.

- `getContents :: IO String` - . .
- `interact :: (String -> String) -> IO ()` - String -> String . .

: https://riptutorial.com/ko/haskell/topic/1904/

61:

(: data-fix recursion-schemes ( )). Hayoo .

Examples

Fix "" .

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Fix f f . f Fix f Fix f . f Fix f .

. r .

```
{-# LANGUAGE DeriveFunctor #-}

-- natural numbers
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: Nat
zero = Fix Zero_
suc :: Nat -> Nat
suc n = Fix (Suc_ n)


-- lists: note the additional type parameter a
-- data List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)


-- binary trees: note two recursive occurrences
-- data Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

Catamorphisms , . cata (*)* fixpoint . cata f Functor .

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix
```

```
-- list example
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    where alg Nil_ = z
          alg (Cons_ x acc) = f x acc
```

(anamorphisms), , . `ana` ( ) *coalgebra* fixpoint . `ana` `f` `Functor` .

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- list example
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                         Nothing -> Nil_
                         Just (x, y) -> Cons_ x y
```

`ana` `cata` . .

,

. . .

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- no mention of Fix!
```

:

```
hylo f g = cata g . ana f
         = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- definition of cata and ana
         = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
         = g . fmap (cata g . ana f) . f  -- Functor law
         = g . fmap (hylo f g) . f  -- definition of hylo
```

Paramorphisms . .

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

Prelude `tails` paramorphism .

```
tails :: List a -> List (List a)
tails = para alg
    where alg Nil_ = cons nil nil  -- [[]]
          alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

Apomorphisms corecursion. .

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

`apo` `para` *dual* . . `para` `apo` `Either` , .

: https://riptutorial.com/ko/haskell/topic/2984/-

62:

Foldable t :: * -> * . fold .

t Foldable ta a "" ta . foldMap :: Monoid m => (a -> m) -> (ta -> m) : foldMap :: Monoid m => (a -> m) -> (ta -> m) . Monoid Monoid ( ).

Examples

Foldable

length a ta .

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' may contain zero or one 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' always contains exactly one 'a'
```

length .

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

Int length . fromIntegral .

Dual . Dual .

```
newtype Dual a = Dual { getDual :: a }

instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

foldMap foldMap Dual . Reverse Data.Functor.Reverse .

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

reverse :

```
reverse :: [a] -> [a]
```

```
reverse = toList . Reverse
```

Foldable

Foldable foldMap foldr .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Foldable Tree where
    foldMap f Leaf = mempty
    foldMap f (Node l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f acc Leaf = acc
    foldr f acc (Node l x r) = foldr f (f x (foldr f acc r)) l
```

.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

DeriveFoldable GHC Foldable . Node .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving Foldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving Foldable

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving Foldable

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x
```

```
ghci> toList (inorder myTree)
"abc"
ghci> toList (preorder myTree)
"bac"
ghci> toList (postorder myTree)
"acb"
```

toList Foldable ta S. a

```
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

toList .

```
class Foldable t where
    -- ...
    toList :: t a -> [a]
    toList = foldr (:) []
```

Foldable

traverse_ Foldable Applicative . . Traversable .

```
-- using the Writer applicative functor (and the Sum monoid)
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- using the IO applicative functor
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- nothing printed
```

for_ traverse_ . foreach .

```
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

sequenceA_ Applicative Foldable .

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
```

```
two
```

traverse_ .

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

sequenceA_ .

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

Foldable Functor traverse_ sequenceA_ .

```
traverse_ f = sequenceA_ . fmap f
```

Foldable Monoid

foldMap (A) Monoid .

foldMap foldr . , Foldable .

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

Product :

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

Foldable

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- and a number of optional methods
```

( ) Foldable a a . foldMap Monoid Monoid .

Foldable

null foldable ta a True , False . null True length 0.

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
False
ghci> null ('x', 3)
False
```

`null` .

```
class Foldable t where
    -- ...
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

: https://riptutorial.com/ko/haskell/topic/753/-

63:

Examples

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

:

```
> isort [5,4,3,2,1]
```

:

```
[1,2,3,4,5]
```

:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

:

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))

firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

.

:

```
> msort [3,1,4,5,2]
```

:

```
[1,2,3,4,5]
```

:

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
    go [a] = a
    go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                      ++ [x] ++
               qsort [b | b <- xs, b >= x]
```

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
               t | t == s    -> t
                 | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2    = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

bogosort .

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

( ).

.

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

: https://riptutorial.com/ko/haskell/topic/2300/-

64:

( ) . (+) , ().

Examples

`&&` AND, `||` OR.

`==` , `/=` , `<` / `<=` `>` / `>=` .

`+` , `-` `/` . (- `quot` `div`). .

- `^` . ( ) . :

  ```
  4^5  ≡  (4*4)*(4*4)*4
  ```

- `^^` . :

  ```
  3^^(-2)  ≡  1 / (2*2)
  ```

  `^` (: `4^^5 :: Int` `4^5 :: Int` `4^^5 :: Rational`).

- `**` . `^` `^^` .

  ```
  2**pi  ≡  exp (pi * log 2)
  ```

.

- `:` (cons cons pron) . (" ") .

- `++` .

  ```
  [1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
  ```

`!!` .

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

($O(1)$ N $O()$ $O(N)$); .

- `$` .

```
f $ x  ≡  f x
       ≡  f(x)  -- disapproved style
```

. $! .

- . .

```
(f . g) x  ≡  f (g x)  ≡  f $ g x
```

- `>>` . : `writeFile "foo.txt" "bla" >> putStrLn "Done."` .
- `>>=` . `readLn >>= \x -> print (x^2)` .

Haskell . , .

```
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

.

```
infixr 5 >+<
```

(`>+<` `++` `:` do) .

Haskell . .

- Hayoo Hoogle . .
- GHCi IHaskell `:i` `:t` (**i** nfo **t** ype) . ,

```
Prelude> :i +
class Num a where
  (+) :: a -> a -> a
  ...
        -- Defined in 'GHC.Num'
infixl 6 +
Prelude> :i ^^
(^^) :: (Fractional a, Integral b) => a -> b -> a
        -- Defined in 'GHC.Real'
infixr 8 ^^
```

`^^` `+` , , `^^` .

`:t` .

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

: https://riptutorial.com/ko/haskell/topic/6792/-

65:

Examples

GHC .

`-prof -fprof-auto` .

.

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

:

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

.

```
./Main +RTS -p
```

`main.prof` , :

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                      individual     inherited
COST CENTRE MODULE                  no.      entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

GHC (`-fprot-auto`) `{-# SCC "name" #-} <expression>` Haskell . "name" `<expression>` .

```
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
```

```
        print $ {-# SCC "print_list" #-} (length l)
```

-fprof +RTS -p ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p Main.prof .

: https://riptutorial.com/ko/haskell/topic/4342/

66: - .

Examples

Map .

```
Map.fromList [("Alex", 31), ("Bob", 22)]
```

Map .

```
> Map.singleton "Alex" 31
fromList [("Alex",31)]
```

`empty` .

```
empty :: Map k a
```

Data.Map `union` , `difference` `intersection` `intersection` .

`null` Map .

```
> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)]
False

> Map.null $ Map.empty
True
```

.

`member :: Ord k => k -> Map ka -> Bool` `k` `Map ka` `True` .

```
> Map.member "Alex" $ Map.singleton "Alex" 31
True
> Map.member "Jenny" $ Map.empty
False
```

`notMember` .

```
> Map.notMember "Alex" $ Map.singleton "Alex" 31
False
> Map.notMember "Jenny" $ Map.empty
True
```

key `findWithDefault :: Ord k => a -> k -> Map ka -> a` .

```
Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x'
Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'
```

.

```
> let m = Map.singleton "Alex" 31
fromList [("Alex",31)]

> Map.insert "Bob" 99 m
fromList [("Alex",31),("Bob",99)]
```

```
> let m = Map.fromList [("Alex", 31), ("Bob", 99)]
fromList [("Alex",31),("Bob",99)]

> Map.delete "Bob" m
fromList [("Alex",31)]
```

`containers` `Data.Map` `Map` .

`Data.Map` Prelude .

```
import qualified Data.Map as Map
```

`Map` `Map` `Map.` , .

```
Map.empty -- give me an empty Map
```

Monoid

`Map kv` Monotype :

- `mempty` `Map` , `Map.empty` `Map.empty`
- `m1 <> m2` `m1` `m2` , , `m1` `m2` , `m1` `m1 <> m2` . `Monoid` `Map.union` .

- . : https://riptutorial.com/ko/haskell/topic/4591/-----

67:

. . (,) " . GHC 7.8 . . .

Haskell .

GHC .

Examples

(_) . .

. . (GHC trac).

> GHC .
>
> `-fdefer-type-errors` `-fdefer-typed-holes` .
>
> `-fwarn-typed-holes` . `-fdefer-type-errors` `-fdefer-typed-holes` `-fdefer-typed-holes` . ( ) `-fno-warn-typed-holes` . .

`undefined` `undefined` , . ( ) , ( ) . :

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: Warning:
    Found hole `_var' with type: Int
    Relevant bindings include
      x :: [a] (bound at <interactive>:19:2)
      it :: [a] -> Int (bound at <interactive>:19:1)
    In the first argument of `(+)', namely `_var'
    In the expression: _var + length (drop 1 x)
    In the expression: \ x -> _var + length (drop 1 x)
```

GHCi repl `it` (`[a] -> Int`).

.

`Foo` `Bar` (`Foo` `Bar`) . `Foo` , , . !

.

```
instance Foo Bar where
```

.

```
Bar.hs:13:10: Warning:
No explicit implementation for
  'foom' and 'quun'
```

```
In the instance declaration for ‘Foo Bar’
```

, `foom` `foom` `Bar` . ? . :

```
instance Foo Bar where
  foom = _
```

" " .

```
Bar.hs:14:10:
    Found hole ‘_’ with type: Bar -> Gronk Bar
    Relevant bindings include
      foom :: Bar -> Gronk Bar (bound at Foo.hs:4:28)
    In the expression: _
    In an equation for ‘foom’: foom = _
    In the instance declaration for ‘Foo Bar’
```

`Bar` . . .

`Gronk` `Gronk` ? [Hayoo](https://hayoo.fh-wedel.de/) . : , . , `Gronk a` .

```
instance Foo Bar where
  foom bar = _ Gronk
```

, `Gronk` ,

```
    Found hole ‘_’
      with type: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
    Where: ‘b0’ is an ambiguous type variable
```

. . `Gronk` .

```
instance Foo Bar where
  foom bar = Gronk _ _
```

.

```
    Found hole ‘_’ with type: [(Int, Bar)]
    Relevant bindings include
      bar :: Bar (bound at Bar.hs:14:29)
      foom :: Bar -> Gronk Bar (bound at Foo.hs:15:24)
    In the second argument of ‘Gronk’, namely ‘_’
    In the expression: Gronk _ _
    In an equation for ‘foom’: foom bar = Gronk _ _
```

`bar` (`Relevant bindings`). . . .

(: haskell-mode Emacs). IDE .

: https://riptutorial.com/ko/haskell/topic/4913/--

68: HTML

Examples

div ID .

Taggy-lens HTML .

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package
taggy-lens

{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "\
    \<!doctype html><html><body>\
    \<div>first div</div>\
    \<div id=\"thediv\">second div</div>\
    \<div id=\"not-thediv\">third div</div>"

main :: IO ()
main = do
    T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

`id="article"` div .

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package
taggy-lens --package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer

somehtmlSmall :: Text
somehtmlSmall =
    "<!doctype html><html><body>\
    \<div id=\"article\"><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>\
    \</body></html>"
```

```
renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
            allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                     (children %~
                      filter (\n -> n ^? element . name /= Just "script")))
                mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- outputs:
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

@ duplode SO

HTML : https://riptutorial.com/ko/haskell/topic/6962/----html--

69: &

?

GHC . .

? (?)

. Template Haskell . "" 0 1 .

0 1 . ( ) .

( ) 0 1. .

1 `Q Exp` , `Q Type` 0 . . `Q Exp` (0) 1 ,

1 . . *n* *m*> *n* . 1 .

```
>:t [| \x -> $x |]

<interactive>:1:10: error:
    * Stage error: `x' is bound at stage 2 but used at stage 1
    * In the untyped splice: $x
      In the Template Haskell quotation [| \ x -> $x |]
```

Template Haskell ?

, Haskell . , . Template Haskell . TH .

Examples

Q

`Language.Haskell.TH.Syntax` `Q :: * -> *` constructor, abstract . `Q` TH ( ). `x` `QX` .

.

- ,
    -
    -
    - (,)
- (, , ,)
- (GHC 7.10)
- GHC (GHC 8.0)

`Q` `newName :: String -> Q Name` . .

`Q` `Functor,Monad,Applicative` `Language.Haskell.TH.Lib` `Q` . TH ast .

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

`ExpQ` , `TypeQ` , `DecsQ` `PatQ` `Q` AST .

TH `runQ :: Quasi m => Q a -> ma` `Quasi IO` `Q` `IO` . `runQ :: Q a -> IO a` `IO` . `Q` . `IO` .

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

. .

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

2 (:) 20 (20).

Template Haskell `n` `curryN` .

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

`curryN` (Haskell AST) `curryN` .

```
curryN n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
```

. .

```
  let args = map VarP (f:xs)
```

`args` `f x1 x2 .. xn` . . , let LHS ( ) .

```
      ntup = TupE (map VarE xs)
```

. . (`VarP`) (`VarE`) .

```
  return $ LamE args (AppE (VarE f) ntup)
```

AST `\f x1 x2 .. xn -> f (x1, x2, .. , xn)` .

'' .

```
...
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
        [| $(varE f) $(tupE (map varE xs)) |]
```

`[| \ $(map varP (f:xs)) -> .. |]` Haskell 'list' `[| \ $(map varP (f:xs)) -> .. |]` . `\ var -> ..` `\ var -> ..` `PatQ` , .

GHCi TH .

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

.

Template Haskell Quasiquotes

`-XTemplateHaskell` GHC . . Template Haskell .

- `$(...)` Template Haskell . `(...)` .
- `$` . Template Haskell `$` . , `f$g` `($)` `fg` . Template Haskell .
- `$` . .
- splice Haskell AST AST Haskell .
- , , . `Q Exp` , `Q Pat` , `Q Type` , `Q [Decl]` . , .

(: QuasiQuotation)

- .
  - `[e|..|]` `[|..|]` - `..` `Q Exp` .
  - `[p|..|]` - `..` `Q Pat` .

- [t|..|] - .. Q Type .
- [d|..|] - .. Q [Dec] .

- AST .
- (: `\x -> [| x |]`) `\x -> [| $(lift x) |]` , `lift :: Lift t => t -> Q Exp` .

```
class Lift t where
  lift :: t -> Q Exp
  default lift :: Data t => t -> Q Exp
```

- ( ) `$$(..)` `$$(..)` `(..)` .
- `e` `Q (TExp a)` `$$e` `a` `Q (TExp a)` .
- `[||..||]` `[||..||]` `..` `a` . `Q (TExp a)` .
- : `unType :: TExp a -> Exp` .

- QuasiQuotes - (`e,p,t,d`) QuasiQuotes . .
- `[iden|...|]` , `iden` `Language.Haskell.TH.Quote.QuasiQuoter` .
- `QuasiQuoter` .

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

- `Language.Haskell.TH.Syntax.Name` . Template Haskell Haskell .
- `'e` `'T` . `e` `T` ( ).

& : https://riptutorial.com/ko/haskell/topic/5216/---amp---

70: (, ...)

- Haskell .
- (written `()`) .
- . . .

Examples

.

. .

```
(1, 2)
```

.

```
(1, 2, 3)

(1, 2, 3, 4)
```

unsugared .

```
(,) 1 2     -- equivalent to (1,2)
(,,) 1 2 3  -- equivalent to (1,2,3)
```

.

```
("answer", 42, '?')
```

.

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

. .

```
(Int, Int)
```

.

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

.

```
(String, Int, Char)
```

.

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

. (,) .

```
myFunction1 (a, b) = ...
```

.

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

.

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

`Prelude` `Data.Tuple` `fst` `snd` .

```
fst (1, 2) -- evaluates to 1

snd (1, 2) -- evaluates to 2
```

.

```
case (1, 2) of (result, _) => result -- evaluates to 1

case (1, 2) of (_, result) => result -- evaluates to 2
```

.

```
case (1, 2, 3) of (result, _, _) => result -- evaluates to 1

case (1, 2, 3) of (_, result, _) => result -- evaluates to 2

case (1, 2, 3) of (_, _, result) => result -- evaluates to 3
```

Haskell `fst` `snd` . Hackage `tuple` `Data.Tuple.Select` .

(uncurrying).

uncurry (Prelude Data.Tuple) .

```
uncurry (+) (1, 2) -- computes 3

uncurry map (negate, [1, 2, 3]) -- computes [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- computes 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- computes [3, 7, 11]

uncurry (curry f) -- computes the same as f
```

(currying)

Prelude Data.Tuple curry .

```
curry fst 1 2 -- computes 1

curry snd 1 2 -- computes 2

curry (uncurry f) -- computes the same as f

import Data.Tuple (swap)
curry swap 1 2 -- computes (2, 1)
```

swap (Data.Tuple) .

```
import Data.Tuple (swap)
swap (1, 2) -- evaluates to (2, 1)
```

.

```
case (1, 2) of (x, y) => (y, x) -- evaluates to (2, 1)
```

(p1, p2) . , (Data.Function.fix) .

```
fix $ \(x, y) -> (1, 2)
```

(x, y) . (1, 2) .

```
fix $ \ ~(x, y) -> (1, 2)
```

(, ...) : https://riptutorial.com/ko/haskell/topic/5342/---------

71:

hackage :

.

" " .

await :: Monad m => Consumer' ama

. a .

yield :: Monad m => a -> Producer' am ()

. a .

Pipes Producer , Consumer Effect Pipes.Tutorial .

Examples

Producer yield :

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

:

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each is a utility function exported by Pipes
```

Producer :

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

Consumer await await .

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

:

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
  liftIO $ putStrLn ("I received: " ++ numStr)
```

await yield await .

```
type Pipe a b = Proxy () a () b
```

Int String :

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

runEffect

runEffect Pipe :

```
main :: IO ()
main = do
  runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

runEffect Proxy Effect .

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

(X Void).

>-> Producer , Consumer Pipe Pipe .

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Producer , Consumer , Pipe Effect Proxy . >-> . .

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

pipes Proxy . Pipe , Producer , Consumer Proxy .

Proxy Pipe await yield , m .

.

:

1. FirstMessage .
2. DoSomething 0 .
3. DoNothing DoNothing
4. 2 3 .

:

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
           | DoNothing
           | DoSomething Int
           deriving (Show,Generic)

instance Binary Command
```

.

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.map
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answers the same command that we have receveid

-- impure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "received message = " ++ (show c)
  return $ C.DoSomething 0
  -- whatever incoming command `c` from the client, answer DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                 putStrLn $ "Remote connection from ip = " ++ (show remoteAddress)
                 _ <- runEffect $ do
                   let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                   let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                   commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                   -- if we want to use the pureHandler
                   --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat
PipesBinary.Encode >-> PNT.toSocket connectionSocket
                 return ()
```

.

```
module Client where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
```

```
import qualified Pipes.Prelude as PipesPrelude
import qualified Pipes.Parse as PP
import qualified Command as C

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answer the same command received from the server

-- inpure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- whatever is received from server, answer DoNothing

main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "Connected to distant server ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
    return ()

sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

: https://riptutorial.com/ko/haskell/topic/6768/

72:

Examples

:

.

. . 5.32€ + 2.94$? .

.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

data USD
data EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

`GeneralisedNewtypeDeriving` `Amount` `Num` . GHC `Double` `Num` .

`(5.0 :: Amount EUR)` . `(1.13 :: Amount EUR) + (5.30 :: Amount USD)` .

haskell .

: https://riptutorial.com/ko/haskell/topic/5227/-

73:

`Functor` covariantly `f :: * -> *` . .

Functor . `Maybe a` `[a]` . Typeclassopedia Functors .

Functor 2 .

```
fmap id == id
```

```
fmap (f . g) = (fmap f) . (fmap g)
```

Examples

Functor

`Maybe` `Functor` .

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

`Maybe` `Functor` `Just` . (, `Maybe` `Nothing`) `fmap` .

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
Just 9
> fmap sqrt Nothing
Nothing
```

. (identity law)

```
fmap id Nothing
Nothing  -- definition of fmap
id Nothing  -- definition of id

fmap id (Just x)
Just (id x)  -- definition of fmap
Just x  -- definition of id
id (Just x)  -- definition of id
```

,

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- definition of (.)
fmap f Nothing  -- definition of fmap
```

```
Nothing  -- definition of fmap
fmap (f . g) Nothing  -- because Nothing = fmap f Nothing, for all f

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- definition of (.)
fmap f (Just (g x))  -- definition of fmap
Just (f (g x))  -- definition of fmap
Just ((f . g) x)  -- definition of (.)
fmap (f . g) (Just x)  -- definition of fmap
```

Functor .

```
instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

list comprehension : fmap f xs = [fx | x <- xs] .

fmap generalises map . map fmap Functor .

:

```
-- base case
fmap id []
[]  -- definition of fmap
id []  -- definition of id

-- inductive step
fmap id (x:xs)
id x : fmap id xs  -- definition of fmap
x : fmap id xs  -- definition of id
x : id xs  -- by the inductive hypothesis
x : xs  -- definition of id
id (x : xs)  -- definition of id
```

, :

```
-- base case
(fmap f . fmap g) []
fmap f (fmap g [])  -- definition of (.)
fmap f []  -- definition of fmap
[]  -- definition of fmap
fmap (f . g) []  -- because [] = fmap f [], for all f

-- inductive step
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- definition of (.)
fmap f (g x : fmap g xs)  -- definition of fmap
f (g x) : fmap f (fmap g xs)  -- definition of fmap
(f . g) x : fmap f (fmap g xs)  -- definition of (.)
(f . g) x : fmap (f . g) xs  -- by the inductive hypothesis
fmap (f . g) xs  -- definition of fmap
```

Functor . Functor .

```
instance Functor ((->) r) where
    fmap f g = \x -> f (g x)
```

fmap = (.) . fmap .

.

```
fmap id g
\x -> id (g x)  -- definition of fmap
\x -> g x  -- definition of id
g  -- eta-reduction
id g  -- definition of id
```

:

```
(fmap f . fmap g) h
fmap f (fmap g h)  -- definition of (.)
fmap f (\x -> g (h x))  -- definition of fmap
\y -> f ((\x -> g (h x)) y)  -- definition of fmap
\y -> f (g (h y))  -- beta-reduction
\y -> (f . g) (h y)  -- definition of (.)
fmap (f . g) h  -- definition of fmap
```

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

fmap f .

Functor , :

```
fmap id = id                    -- identity
fmap f . fmap g = fmap (f . g)  -- composition
```

<$> fmap fmap .

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Functor

Data.Functor , <$ $> .

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: Functor f => f a -> b -> f b
```

```
($>) = flip (<$)
```

void .

```
void :: Functor f => f a -> f ()
void = (() <$)
```

Functor Functor . Functor , (functor) generic .

ID . SKI I .

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

I , Identity Data.Functor.Identity .

. const SKI K .

```
newtype K c a = K c
```

K ca -values; a K () Proxy . K fmap !

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

K Data.Functor.Const Const .

.

Functor Functor . , (,) :: * -> * -> * types * , (:*:) :: (* -> *) -> (* -> *) -> (* -> *) functors * -> * .

```
infixl 7 :*:
data (f :*: g) a = f a :*: g a

instance (Functor f, Functor g) => Functor (f :*: g) where
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

Data.Functor.Product Product Product .

:*: (,) :+: Either Functor .

```
infixl 6 :+:
data (f :+: g) a = InL (f a) | InR (g a)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

:+: Data.Functor.Sum Sum .

, :.: (.) .

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

Compose Data.Functor.Compose .

I , K , :*: , :+: and :.: . . Functor Functor . I Fix .

	data Pair a = Pair aa	type Pair = I :*: I
2x2	type Grid a = Pair (Pair a)	type Grid = Pair :.: Pair
	data Nat = Zero | Succ Nat	type Nat = Fix (K () :+: I)
	data List a = Nil | Cons a (List a)	type List a = Fix (K () :+: K a :*: I)
	data Tree a = Leaf | Node (Tree a) a (Tree a)	type Tree a = Fix (K () :+: I :*: K a :*: I)
	data Rose a = Rose a (List (Rose a))	type Rose a = Fix (K a :*: List :.: I)

"" generics-sop . .

```
class Generic a where
    type Rep a  -- a generic representation built using a kit
    to :: a -> Rep a
    from :: Rep a -> a
```

Functor

Functor (") . () () .

(Hask) . **Hask Hask** .

Functor . f :: * -> * fmap :: (a -> b) -> (fa -> fb) . ,

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` (morphism) `:: a -> b` `:: fa -> fb` . `Functor` **Hask** ( ) .

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

`fmap` `:: a -> b` **Hask** .

`Functor` **Hask** *endo* . . .

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

Functor **Hask** . ,

```
instance Category (->) where        -- Hask
    id    = \x -> x
    f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

`DeriveFunctor` GHC `Functor` .

```
{-# LANGUAGE DeriveFunctor #-}

data List a = Nil | Cons a (List a) deriving Functor

-- instance Functor List where            -- automatically defined
--   fmap f Nil = Nil
--   fmap f (Cons x xs) = Cons (f x) (fmap f xs)

map :: (a -> b) -> List a -> List b
map = fmap
```

: https://riptutorial.com/ko/haskell/topic/3800/

74:

Examples

`Data.Proxy` `Proxy :: k -> *` (:) . .

```
{-# LANGUAGE PolyKinds #-}

data Proxy a = Proxy
```

`Proxy` `ScopedTypeVariables` `a` .

,

```
showread :: String -> String
showread = show . read
```

elaborator `Show` `Read` . `Proxy` :

```
{-# LANGUAGE ScopedTypeVariables #-}

import Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

`Proxy` `a` .

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- attempt to parse a char literal as a Bool
"*** Exception: Prelude.read: no parse
```

" "

`Proxy` `Proxy` . `Proxy` .

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

, `f` `fa` `Proxy :: Proxy a f` `Proxy :: Proxy a` .

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

() .

`Proxy` `f` `fa -> Proxy a` .

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

`()` .

```
unit :: a -> ()
unit _ = ()
```

, `Proxy` Functor . `()` .

: https://riptutorial.com/ko/haskell/topic/8025/

75:

Examples

.

.

```
absolute :: Int -> Int  -- definition restricted to Ints for simplicity
absolute n = if (n < 0) then (-n) else n
```

.

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

`otherwise` `True` .

Haskell `case` .

case Haskell .

:

```
longName :: String -> String
longName name = case name of
                   "Alex"  -> "Alexander"
                   "Jenny" -> "Jennifer"
                   _       -> "Unknown"  -- the "default" case, if you like
```

`case` .

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

`Maybe` .

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe can only take `Just a` or `Nothing`, so this wildcard suffices
```

.

```
isEmptyList :: [a] -> Bool
```

```
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

. : : ,

:

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
  | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "Poor child, try to get another job"
  | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
  | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
```

`where` ., `where`:

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "Poor child, try to get another job"
  | annualSalary <= mediumSalary = "Money, Money, Money!"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
  where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

, `where` (`hourlyRate * (weekHoursOfWork * 52)`) `where` .

`let` `let` `where` .

: https://riptutorial.com/ko/haskell/topic/3799/-

76:

C Haskell . C .

C Haskell , . C C .

, () .
.

Examples

C

```
plus(a, b); // Parentheses surrounding only the arguments, comma separated
```

```
(plus a b) -- Parentheses surrounding the function and the arguments, no commas
```

Haskell . .

```
plus a b -- no parentheses are needed here!
```

C

. , .
C :

```
plus(a, take(b, c));
```

Haskell :

```
(plus a (take b c))
-- or equivalently, omitting the outermost parentheses
plus a (take b c)
```

.

```
plus a take b c -- Not what we want!
```

`take` `b` `c` `plus` .
Haskell . `take` .

C .

```
plus(a, take, b, c); // Not what we want!
```

- 1

Haskell . . .

```
(((plus) 1) 2)
```

`(plus) 1 ((plus) 1)` , `2 (((plus) 1) 2)` . `plus 1 2` . .

`plus` .

`plus 1 1` .

`plus 1 2 1 2` , `3` .

- 2

, `map` .

```
map :: (a -> b) -> [a] -> [b]
```

. `map`

```
addOne x = plus 1 x
map addOne [1,2,3]
```

`addOne addOne` .

```
(addOne) x = ((plus) 1) x
```

`addOne x plus 1 x plus 1` . `addOne plus 1` , `addOne plus 1 plus 1`

```
map (plus 1) [1,2,3]
```

: https://riptutorial.com/ko/haskell/topic/9615/--

77:

Examples

`Arrow` , " " . , `Arrow (->)` . , :

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 where (greater, smaller) = partition (>x) ys
```

.

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

.

```
                        ──── minimum ────
                    ╱           *           ╲
──── partition (>x) >>>         *         >>>  uncurry (-) ───
                    ╲           *           ╱
                        ──── maximum ────
```

,

- `>>>` . composition (`<<<`). .
- ╱╲ "" . Haskell , :

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  `[Double]` ,

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  `Double` .
- `***` † . `maximum` `minimum` . .

  ```
  (***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
  ```

---

† **Hask** (, `Arrow (->)`) `f***g` `f` `g` . .

: https://riptutorial.com/ko/haskell/topic/4912/

78:

1. []
2. infixl []
3. []

`infixr`	.
`infixl`	.
`infix`	.
	(0 ~ 9, 9)
`op1, ... , opn`	

.

1. .
2. .
3. .

2 . .

, `1 + negate 5 * 2 - 3 * 4 ^ 2 ^ 1` `1 + negate 5 * 2 - 3 * 4 ^ 2 ^ 1` .

```
infixl 6 +
infixl 6 -
infixl 7 *
infixr 8 ^
```

1. `1 + (negate 5) * 2 - 3 * 4 ^ 2 ^ 1`
2. `1 + ((negate 5) * 2) - (3 * (4 ^ 2 ^ 1))`
3. `(1 + ((negate 5) * 2)) - (3 * (4 ^ (2 ^ 1)))`

Haskell 98 4.4.2 .

Examples

`infixl` `infixr` `infix` . , (base)

```
infixl 6 -
infixr 5 :
infix  4 ==
```

`infixl` `-` `1 - 2 - 3 - 4`

```
((1 - 2) - 3) - 4
```

`infixr` : `1 : 2 : 3 : []` ()

```
1 : (2 : (3 : []))
```

`infix ==` ., `True == False == True` . `True == (False == True)` `(True == False) == True` .

`infixl 9` .

. `0` `9` . . , (base)

```
infixl 6 +
infixl 7 *
```

`*` `+` `1 * 2 + 3` .

```
(1 * 2) + 3
```

, "".

- `fx `op` gy` (`fx `op` gy` `(fx)` op `(gy)` `` `op` `` .
- (: `infixl *!?`) `9` .

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- `infixl 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR``
- `infix 4 ==, /=, <, <=, >=, >`
- `infix ??`

: https://riptutorial.com/ko/haskell/topic/4691/-

S. No		Contributors
1		4444, Adam Wagner, alejosocorro, Amitay Stern, arseniiv, baxbaxwalanuksiwe, Benjamin Hodgson, Benjamin Kovach, Burkhard, Carsten, colinro, Community, Daniel Jour, Echo Nolan, erisco, gdziadkiewicz, HBu, J Atkin, Jan Hrcek, Jules, Kwarrtz, leftaroundabout, M. Barbieri, mb21, mnoronha, Mr Tsjolder, ocharles, pouya, R B, Ryan Hilbert, Sebastian Graf, Shoe, Stephen Leppik, Steve Trout, Tim E. Lord, Turion, user239558, Will Ness, zbw, λlεx
2	Attoparsec	λlεx
3	cofree comonads	leftaroundabout
4	Data.Aeson - JSON	Chris Stryczynski, Janos Potecki, Matthew Pickering, rob, xuh
5	Data.Text	Benjamin Hodgson, dkasak, Janos Potecki, jkeuhlen, mnoronha , unhammer
6	GHCi	Benjamin Hodgson, James, Janos Potecki, mnoronha, RamenChef, wizzup, λlεx
7	GHCJS	Mikkel
8	Google	Janos Potecki, λlεx
9	Gtk3	bleakgadfly
10	IO	λlεx
11	RankNTypes	ocharles
12	XML	Mikkel
13	zipWithM	zbw
14		Benjamin Hodgson, ErikR, J. Abrahamson
15		Community, Doruk, Matthew Pickering, mnoronha, Will Ness, λlεx
16		Wysaard, Zoey Hewll
17		zbw
18		arseniiv, Will Ness

19		Benjamin Hodgson, erisco, Firas Moalla, Janos Potecki, Kwarrtz, Lynn, Matthew Pickering, Matthew Walton, Mirzhan Irkegulov, mnoronha, Tim E. Lord, user2407038, Will Ness, Yosh, λlεx
20		mnoronha, Will Ness, λlεx
21	(GHC)	Cactus
22		λlεx
23		Janos Potecki, λlεx
24		Cactus, Janos Potecki, John F. Miller, Mario, Matthew Pickering, Will Ness, λlεx
25		Bartek Banachewicz, bennofs, chamini2, dfordivam, dsign, Hjulle, J. Abrahamson, John F. Miller, Kwarrtz, Matthew Pickering, λlεx
26	/	Chris Stryczynski
27		tlo
28		Alec, Benjamin Hodgson, Cactus, fgb, Kapol, Kwarrtz, Lynn, Mario Román, Matthew Pickering, Will Ness, λlεx
29		Damian Nadales
30		Benjamin Hodgson, Kwarrtz, mnoronha, Will Ness
31		Benjamin Hodgson, Kapol, Will Ness, λlεx
32		Cactus, Kwarrtz, mnoronha, Will Ness
33		Benjamin Hodgson, Cactus, J. Abrahamson, pyon, sid-kap
34		leftaroundabout
35	-	arrowd, Dair, Undreren
36		λlεx
37		arrowd, Benjamin Hodgson, Benjamin Kovach, J. Abrahamson, Mario Román, mmlab, user2407038
38		Benjamin Hodgson, λlεx
39		λlεx
40		arseniiv, Benjamin Hodgson, CiscoIPPhone, Dair, Matthew

		Pickering, Will Ness
41		Benjamin Hodgson, gdziadkiewicz, Matthew Pickering, Steve Trout
42		Janos Potecki, Kapol
43		ocramz
44		4444, curbyourdogma, Dair, Janos Potecki
45		Apoorv Ingle, Kritzefitz, Luis Casillas
46		Benjamin Hodgson, Benjamin Kovach, Cactus, user2407038, Will Ness
47		xuh
48		Adam Wagner, Carsten, Kapol, leftaroundabout, pdexter
49		arrowd, bleakgadfly, crockeea
50		arrowd, λlεx
51		leftaroundabout, mniip, xuh
52		Mario Román
53		Luka Horvat, λlεx
54		arjanen, Benjamin Hodgson, Billy Brown, Cactus, Dair, gdziadkiewicz, J. Abrahamson, Kwarrtz, leftaroundabout, mnoronha, RamenChef, Will Ness, zeronone, λlεx
55		tlo
56		Benjamin Hodgson, Kritzefitz, mnoronha, Will Ness, λlεx
57		Benjamin Hodgson, liminalisht
58	GHC	Antal Spector-Zabusky, Bartek Banachewicz, Benjamin Hodgson, Benjamin Kovach, Cactus, charlag, Doomjunky, Janos Potecki, John F. Miller, K48, Kwarrtz, leftaroundabout, Mads Buch, Matthew Pickering, mkovacs, mniip, phadej, Yosh, λlεx
59		mniip
60		3442, Benjamin Hodgson, David Grayson, J. Abrahamson, Jan Hrcek, John F. Miller, leftaroundabout, mnoronha, Sarah,

		user2407038, Will Ness, λlεx
61		arseniiv, Benjamin Hodgson
62		Benjamin Hodgson, Cactus, Dair, David Grayson, J. Abrahamson, Jan Hrcek, mnoronha
63		Brian Min, pouya, Romildo, Shoe, Vektorweg, Will Ness
64		leftaroundabout, mnoronha
65		λlεx
66	- .	Cactus, Itbot, Will Ness, λlεx
67		Cactus, leftaroundabout, user2407038, Will Ness
68	HTML	Kostiantyn Rybnikov, λlεx
69	&	user2407038
70	(, ...)	Cactus, mnoronha, Toxaris, λlεx
71		4444, Benjamin Hodgson, Stephane Rolland, λlεx
72		Benjamin Hodgson, Christof Schramm
73		Benjamin Hodgson, Delapouite, Janos Potecki, liminalisht, mathk, mnoronha, Will Ness, λlεx
74		Benjamin Hodgson
75		Delapouite, James, Janos Potecki, Will Ness, λlεx
76		Zoey Hewll
77		artcorpse, leftaroundabout
78		Alec, Will Ness