Monaadidest üldisemalt

Haskellis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Intuitsioon: Tüüp `m a` on nagu konteiner, kuhu saab `a` tüüpi väärtust hoida. (Aga igal `m a` ei pruugi sisaldada `a` tüüpi väärtust.)

Näiteks:

- `Maybe a`
- `[a]`
- `IO a`
- ...

Monaadidest üldisemalt (järg.)

Teoorias peaks funktsioonid rahuldama järgnevaid võrdusi:

- Functor

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

- Applicative

```
                pure id <*> v == v
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
          pure f <*> pure x == pure (f x)
               u <*> pure y == pure ($ y) <*> u
```

- Monad

```
            return a >>= k  ==  k a
              m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Maybe monaad

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

- intuitsioon: `Nothing` – viga, nurjumine
- definitsioon

```
return x  =  Just x

(Just a) >>= f  =  f a
Nothing  >>= f  =  Nothing
```

- Näide (pseudokood):

```
getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

Listi monaad

```
-- data [a] = [] | a : [a]
```

- intuitsioon: mitmesus
- definitsioon

```
return x   =  [x]

xs  >>= f  =  concat (map f xs)
```

- Näide (pseudokood):

```
sõpradePalgad :: Person -> [Int]
sõpradePalgad isik = do
  sõber <- getFriends isik
  töö   <- getEmployers sõber
  return (getPay töö sõber)
```

- … see on sama mis listikomprehensioon

Parsimise monaad

```
type Parser a = String -> [(a,String)]

return x = \ s -> [(x,s)]

p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

```
expr =     do n <- term
              keyw "+"
              m <- expr
              return (n+m)
       <|> term

term =     do n <- atom
              keyw "*"
              m <- term
              return (n*m)
       <|> atom

atom =     do keyw "("
              e <- expr
              keyw ")"
       <|> parse_int
```

Seisundi monaad

```
State s a   -- s on seisundi tüüp; a väärtuse tüüp
```

- definitsioon

```
get :: State s s
put :: s -> State s ()
runState :: State a b -> a -> (a, b)
```

- Tavaliselt tehakse tüübisünonüüm iga konkreetse alamprogrammi jaoks.

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a
```

...

Seisundi monaad II

- Näide:

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a

applyIsik :: String -> (Double -> Double) -> PangaArvutus ()
applyIsik nimi f = do
  s <- get
  put (map g s)
    where g (n,s) | n==nimi   = (n, f s)
                  | otherwise = (n, s)

rahaVälja :: String -> Float -> PangaArvutus Bool
rahaVälja nimi summa = do
  s <- get
  case lookup nimi s of
    Nothing -> return False
    Just r  -> do
      applyIsik nimi (\ x -> x-summa)
      return True
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)

get :: State s s
get s = (s, s)

put :: s -> State s a
put s _ = (s, ())

runState :: State s a -> s -> (s, a)
runState f = f

return :: a -> State s a
return x s = (s, x)

(>>=) :: State s a -> (a -> State s b) -> State s b
(f >>= g) s = g x s'
    where (s',x) = f s
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)

get :: s -> (s, s)
get x = (x, x)

put :: s -> s -> (s, ())
put s _ = (s, ())

runState :: (s -> (s, a)) -> s -> (s, a)
runState f = f

return :: a -> s -> (s, a)
return x s = (s, x)

(>>=) :: (s -> (s, a)) -> (a -> s -> (s, b)) -> s -> (s, b)
(f >>= g) s = g x s'
    where (s',x) = f s
```

IO modeleerimine seisundimonaadiga

```
type IO a = State 🌍 a
```

ehk

```
type IO a = 🌍 -> (🌍, a)
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

```
Main> sequence_ [print 1, print 'a']
1
'a
```

```
Main> it
()
```

```
Main> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
```

```
Main> sequence [Just 1, Nothing, Just 3]
Nothing
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `map`

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Näited:

```
Main>mapM print [1..5]
1
2
3
4
5
[(), (), (), (), ()]
```

```
Main>mapM print (Just 6)
6
Just ()
```

Monaadilised funktsioonid standardprelüüdis

- "for"-tsükkel

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM x y = mapM x y
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ x y = mapM_ x y
```

Näited:

```
main = do
  forM_ [1..10] $ \ i -> do
    let n = fact i
    putStrLn $ show i ++ "! = " ++ show n
```

Monaadilised funktsioonid standardprelüüdis

- Tingimuslik täitmine

```
when :: Monad m => Bool -> m () -> m ()
when p s = if p then s else return ()
unless :: Monad m => Bool -> m () -> m ()
unless p s = when (not p) s
```

Näited:

```
import System.Environment (getArgs)
import System.Directory (doesDirectoryExist)

main = do names <- getArgs
          forM_ names $ \ dir -> do
                b <- doesDirectoryExist dir
                when b $ putStrLn dir
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `filter`

```
filterM :: Monad m -> (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x : xs) = do
  b  <- p x
  ys <- filterM p xs
  return (if b then x : ys else ys)
```

Näited:

```
main = do names <- getArgs
          dirs  <- filterM doesDirectoryExist names
          mapM_ putStrLn dirs
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (1)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do x <- m
               return (f x)
```

Näited:

```
countLines :: FilePath -> IO Int
countLines = liftM (length . lines) . readFile
```

Enamasti kasutatakse funktsiooni:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (2)

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do x1 <- m1
                    x2 <- m2
                    return (f x1 x2)
```

Näited:

```
Main> liftM2 (+) (Just 1) (Just 2)
Just 3
Main> liftM2 (+) (Just 1) Nothing
Nothing
Main> liftM2 (+) [0, 3] [5, 6, 7]
[5,6,7,8,9,10]
```

Analoogiliselt on defineeritud **`liftM3`**, **`liftM4`** ja **`liftM5`** .

Monaadilised funktsioonid standardprelüüdis

- Monaadiline applikatsioon

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Näited:

```
Main> [(+2)] `ap` [1, 2, 3]
[3, 4, 5]
Main> [(+2), ( 2)] `ap` [1, 2, 3]
[3,4,5,2,4,6]
```

```
liftMn f x1 x2 ... xn = return f `ap` x1 `ap` x2 `ap` ... `ap` xn
```

Aplikatsioon on võimalik ka osade mitte-monaadide puhul! (<*>)

```
return f `ap` x1 `ap` x2 `ap` ... `ap` xn = f <$> x1 <*> x2 <*> ... <*> xn
```

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega…
- … aga kunagi tuleb aeg neid ühendada terviklikuks programmiks.
- Monaadide kombineerimiseks on monaaditeisendajad. Selle asemel, et kasutada monaadi `Maybe` kasutame mõnedel juhtudel monaadi `MaybeT m`, kus `m` on mingi teine monaad.
- Haskelli nõrkus — raske on ette näha, mis informatisiooni on vaja läbi programmi kaasas kanda.