Monaadilised interpretaatorid

- Lihtsa funktsionaalse keele avaldised

```
type Name  = String

data Expr  = Var Name
           | Con Int
           | Lam Name Expr
           | App Expr Expr
```

- Väärtused

```
data Value = Num Int
           | Fun (Value -> M Value)
```

Monaadilised interpretaatorid

- Keskonnad

```
type Env    = [(Name,Value)]

lookupE :: Name -> Env -> M Value
lookupE x []                        = fail ("Free variable: " ++ x)
lookupE x ((y,v):e) | x == y        = return v
                    | otherwise     = lookupE x e
```

- Interpretaator

```
eval :: Expr -> Env -> M Value
eval (Var x)   env = lookupE x env
eval (Con i)   env = return (Num i)
eval (Lam x e) env = return (Fun(\a -> eval e ((x,a):env)))
eval (App h e) env = do Fun f <- eval h env
                        a     <- eval e env
                        f a
```

Monaadilised interpretaatorid

- Primitiivsed operaatorid

```
initialE :: Env
initialE = [("+", mkop (+)), ("-", mkop (-)),
            ("*", mkop (*)), ("/", divop)]
  where mkop op = Fun (\(Num a) ->
                       return (Fun (\(Num b) ->
                       return (Num (a `op` b)))))
        divop   = Fun (\(Num a) ->
                       return (Fun (\(Num b) ->
                       if b == 0
                          then fail "Division by zero"
                          else return (Num (a `div` b)))))
```

Monaadilised interpretaatorid

- Standartne interpretaator

```
newtype Id a = Id a

instance Monad Id where
         (Id x) >>= f  = f x
         return x      = Id x

type M a = Id a
```

- Veatöötlusega interpretaator

```
type M a = Maybe a
```

Monaadilised interpretaatorid

- Olekumonaad

```
newtype S s a = S (s -> (a,s))

instance Monad (S s) where
       return x = S (\s -> (x,s))
       (S f) >>= k = S (\s -> case f s of
                                (x,s') -> case k x of
                                             S g -> g s')
```

Monaadilised interpretaatorid

- Olekumonaad (järg)

```
getS :: S s s
getS  = S (\s -> (s,s))

setS :: s -> S s ()
setS x = S (\s -> ((),x))

runS :: S s a -> s -> (a,s)
runS (S f) s = f s

tick :: S Int ()
tick = do s <- getS
          setS (s+1)
```

Monaadilised interpretaatorid

- Reduktsiooni samme lugev interpretaator

```
type M a = S Int a


...
eval (App h e) env = do Fun f <- eval h env
                        a     <- eval e env
                        tick
                        f a
```

- Reduktsiooni samme kasutav interpretaator

```
data Expr  = ...
           | Count
...
eval Count     env = do c <- getS
                        return (Num c)
```

Monaadilised interpretaatorid

- Väljundmonaad

```
newtype O a = O (String,a)

instance Monad O where
     return x = O ("",x)
     (O (s,x)) >>= k = let O (s',y) = k x in O (s++s', y)

outO :: Show a => a -> O ()
outO x = O (show x, ())
```

Monaadilised interpretaatorid

- Vahetulemusi väljastav interpretaator

```
type M a = O a

data Expr  = ...
           | Out Expr
...
eval (Out e) env = do v <- eval e env
                      outO v
                      return v
```

Monaadilised interpretaatorid

- call-by-name interpretaator

```
data Value = Num Int
           | Fun (M Value -> M Value)

type Env   = [(Name,M Value)]

lookupE :: Name -> Env -> M Value
lookupE x []                       = fail ("Free variable: " ++ x)
lookupE x ((y,v):e) | x == y       = v
                    | otherwise    = lookupE x e

...
eval (App e1 e2) env = do Fun f <- eval e1 env
                          f (eval e2 env)
```

Monaadilised interpretaatorid

- call-by-name interpretaator (järg)

```
initialE :: Env
initialE = [("+", mkop (+)), ("-", mkop (-)),
            ("*", mkop (*)), ("/", divop)]
  where mkop op = return (Fun (\x -> do {Num a <- x;
                  return (Fun (\y -> do {Num b <- y;
                  return (Num (a `op` b))})))})))
        divop   = return (Fun (\x -> do {Num a <- x;
                  return (Fun (\y -> do {Num b <- y;
                  if b == 0
                     then fail "Division by zero"
                     else return (Num (a `div` b))})))})))
```

Monaadid

- Monaadidega seotud funktsioone

```
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 ())

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)

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

Monaadid

- Monaadidega seotud klasse

```
class  Functor f  where
       fmap  :: (a -> b) -> f a -> f b

instance Functor [] where
       fmap = map

instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)
```

Monaadid

- Monaadidega seotud klasse

```
class  (Monad m) => MonadPlus m  where
    mzero :: m a
    mplus :: m a -> m a -> m a

instance  MonadPlus Maybe  where
    mzero                = Nothing
    Nothing `mplus` ys   = ys
    xs      `mplus` ys   = xs

instance  MonadPlus []  where
    mzero = []
    mplus = (++)
```

- Teegis `Monad`

Monaadid

- Monaadidega seotud funktsioone

```
liftM   :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f  =  \a -> do { a' <- a; return (f a') }

liftM2  :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f =  \a b -> do { a' <- a; b' <- b; return (f a' b') }

liftM3  :: (Monad m) => (a -> b -> c -> d) ->
                        (m a -> m b -> m c -> m d)
liftM3 f =  \a b c -> do { a' <- a; b' <- b; c' <- c;
                           return (f a' b' c') }
```

Monaadid

- Monaadidega seotud funktsioone

```
msum          :: MonadPlus m => [m a] -> m a
msum xs        =  foldr mplus mzero xs

join          :: (Monad m) => m (m a) -> m a
join x         =  x >>= id

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

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

Monaadid

- Monaadidega seotud funktsioone

```
guard    :: MonadPlus m => Bool -> m ()
guard p   =  if p then return () else mzero

foldM    :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a []      =  return a
foldM f a (x:xs) =  f a x >>= \ y -> foldM f y xs

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