Monaadid

- Puu lehtede nummerdamine

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

labelTree           = fst . lab 0
lab c (Leaf x)      = (Leaf c, c+1)
lab c (Branch t1 t2) = let (t1',c1) = lab c  t1
                           (t2',c2) = lab c1 t2
                       in (Branch t1' t2', c2)
```

- “Imperatiivne versioon”

```
labTree t           = {count := 0;     lab t}

lab (Leaf x)        = {i := count;     count += 1;
                       return (Leaf i)}
lab (Branch t1 t2)  = {t1' := lab t1;  t2' := lab t2;
                       return (Branch t1' t2')}
```

Monaadid

- Olekuteisendajad

```
type IntSt a = Int -> (a,Int)

unit :: a -> IntSt a
unit x     = \s -> (x,s)

bind :: IntSt a -> (a -> IntSt b) -> IntSt b
m `bind` f = \s -> let (x1,s1) = m s
                       (x2,s2) = f x1 s1
                   in (x2,s2)
```

Monaadid

- Olekuteisendajad (järg)

```
getInt :: IntSt Int
getInt   = \s -> (s,s)

putInt :: Int -> IntSt ()
putInt x = \s -> ((),x)

runIntSt :: IntSt a -> Int -> a
runIntSt f s = fst (f s)

inc :: IntSt Int
inc = getInt       `bind` \i ->
      putInt (i+1) `bind` \_ ->
      unit i
```

Monaadid

- Puu lehtede nummerdamine (ver. 2)

```
labelTree t  = runIntSt (lab t) 0

lab (Leaf x)        = inc `bind` \i ->
                      unit (Leaf i)
lab (Branch t1 t2) = lab t1 `bind` \t1' ->
                      lab t2 `bind` \t2' ->
                      unit (Branch t1' t2')
```

Monaadid

- Monaadide klass

```
class  Monad m  where
    (>>=)     :: m a -> (a -> m b) -> m b
    (>>)      :: m a -> m b -> m b
    return    :: a -> m a
    fail      :: String -> m a
    m >> k    =  m >>= \_ -> k
    fail s    = error s
```

- Näide: listid

```
instance  Monad []  where
    m >>= k       =  concat (map k m)
    return x      =  [x]
    fail s        =  []
```

Monaadid

- Olekuteisendajad (järg)

```
newtype IntSt a = ISt (Int -> (a,Int))

instance Monad IntSt where
    return x = ISt (\s -> (x,s))
    k >>= f  = ISt (\s -> let ISt k' = k
                              (x,s') = k' s
                              ISt f' = f x
                          in f' s')
getInt      :: IntSt Int
getInt       = ISt (\s -> (s,s))

putInt      :: Int -> IntSt ()
putInt x     = ISt (\s -> ((),x))
```

Monaadid

- Olekuteisendajad (järg)

```
runIntSt     :: IntSt a -> Int -> a
runIntSt f s = let ISt f' = f
               in fst (f' s)
inc          :: IntSt Int
inc           = getInt       >>= \i ->
               putInt (i+1) >>
               return i
```

- Puu lehtede nummerdamine (ver. 3)

```
labelTree t          = runIntSt (lab t) 0
lab (Leaf x)         = inc >>= \i ->
                       return (Leaf i)
lab (Branch t1 t2)   = lab t1 >>= \t1' ->
                       lab t2 >>= \t2' ->
                       return (Branch t1' t2')
```

Monaadid

- `do`-süntaks

$$
\begin{array}{rcl}
\mathit{expr} & = & \texttt{do}\{\mathit{stmt};\ldots;\mathit{stmt}\} \\
\mathit{stmt} & = & \mathit{pat} \leftarrow \mathit{expr} \\
 & \mid & \mathit{expr} \\
 & \mid & \texttt{let}\ \mathit{decls}
\end{array}
$$

- Transleerimisreeglid

```
do {e}                    = e
do {e; stmts}             = e >> do {stmts}
do {v <- e; stmts}        = e >>= \v -> do {stmts}
do {let decls; stmts}     = let decls in do {stmts}
```

Monaadid

- Puu lehtede nummerdamine (ver. 4)

```
labelTree t          = runIntSt (lab t) 0
lab (Leaf x)         = do i <- inc
                          return (Leaf i)
lab (Branch t1 t2) = do t1' <- lab t1
                          t2' <- lab t2
                          return (Branch t1' t2')
```

- Monaadide seadused

$$
\begin{array}{lcl}
\mathit{return}\; x \gg= f & = & f\,x \\
m \gg= \mathit{return} & = & m \\
(m \gg= \lambda x \rightarrow k) \gg= f & = & m \gg= \lambda x \rightarrow (k \gg= f)
\end{array}
$$

Monaadilised interpretaatorid

- Aritmeetiliste avaldiste interpretaator

```
data Expr = Num Int
          | Expr `Plus`  Expr
          | Expr `Minus` Expr

eval :: Expr -> Int
eval (Num i)          = i
eval (e1 `Plus`  e2) = (eval e1) + (eval e2)
eval (e1 `Minus` e2) = (eval e1) - (eval e2)
```

Monaadilised interpretaatorid

- Uute operaatorite lisamine

```
data Expr = ...
          | Expr `Mul` Expr
          | Expr `Div` Expr

eval :: Expr -> Int
    ...
eval (e1 `Mul` e2) = (eval e1)   *   (eval e2)
eval (e1 `Div` e2) = (eval e1) `div` (eval e2)
```

- Veatöötlus puudub!

```
eval (Num 13 `Div` Num 0)  ==> ....
```

Monaadilised interpretaatorid

- Veatöötlusega interpretaator

```
data Maybe a = Nothing | Just a

eval :: Expr -> Maybe Int
eval (Num i)          = Just i
eval (e1 `Plus`  e2) = case eval e1 of
                         Nothing -> Nothing
                         Just v1 -> case eval e2 of
                                      Nothing -> Nothing
                                      Just v2 -> Just (v1+v2)
eval (e1 `Minus` e2) = ...
eval (e1 `Mul`   e2) = ...
```

Monaadilised interpretaatorid

- Veatöötlusega interpretaator (järg)

```
eval (e1 `Div` e2)
    = case eval e1 of
        Nothing -> Nothing
        Just v1 -> case eval e2 of
                     Nothing -> Nothing
                     Just v2 -> if v2 == 0
                                then Nothing
                                else Just (v1 `div` v2)
```

Monaadilised interpretaatorid

- Erandite monaad

```
instance Monad Maybe where
    Just x  >>= k = k x
    Nothing >>= k = Nothing
    return        = Just
    fail s        = Nothing
```

Monaadilised interpretaatorid

- Veatöötlusega interpretaator (v. 2)

```
eval :: Expr -> Maybe Int
eval (Num i)          = return i
eval (e1 `Plus`  e2) = do v1 <- eval e1
                          v2 <- eval e2
                          return (v1+v2)
eval (e1 `Minus` e2) = ...
eval (e1 `Mul`   e2) = ...
eval (e1 `Div`   e2) = do v1 <- eval e1
                          v2 <- eval e2
                          if v2 == 0
                             then fail "Division by zero"
                             else return (v1 `div` v2)
```

Monaadilised interpretaatorid

- "Keskkonnad" (enviroments)

```
data Expr = ... | Var String

type Env  = [(String,Int)]

lookup :: String -> Env -> Maybe Int
lookup x = \ env -> case [v | (y,v) <- env, y == x] of
                         []  -> fail ("Free variable: " ++ x)
                         v:_ -> return v
```

Monaadilised interpretaatorid

- “Keskkonnaga” interpretaator

```
eval :: Expr -> Env -> Maybe Int
eval (Num i)          = \env -> return i
eval (e1 `Plus`  e2) = \env -> do v1 <- eval e1
                                  v2 <- eval e2
                                  return (v1+v2)
eval (e1 `Minus` e2) = ...
eval (e1 `Mul`   e2) = ...
eval (e1 `Div`   e2) = ...
eval (Var x)          = \env -> lookup x env
```

Monaadilised interpretaatorid

- “Keskkondade” monaad

```
newtype Reader r a = R (r -> Maybe a)

bindR   :: Reader r a -> (a -> Reader r b) -> Reader r b
(R ma) `bindR` f = R (\ r -> ma r >>= \ a ->
                             case f a of R f' -> f' r)

runReader :: Reader r a -> r -> Maybe a
runReader (R r) e = r e

instance Monad (Reader r) where
         return a = R (\ _ -> return a)
         (>>=)    = bindR
         fail s   = R (\ _ -> fail s)
```

Monaadilised interpretaatorid

- "Keskkonnaga" interpretaator (v. 2)

```
eval :: Expr -> Reader Env Int
eval (Num i)          = return i
eval (e1 `Plus`  e2) = do v1 <- eval e1
                          v2 <- eval e2
                          return (v1+v2)
eval (e1 `Minus` e2) = ...
eval (e1 `Mul`   e2) = ...
eval (e1 `Div`   e2) = ...
eval (Var x)          = lookup x
```