Monády 3

Monady sú použiteľný nástroj pre programátora poskytujúci:

- modularitu – skladať zložitejšie výpočty z jednoduchších (no side-effects),
- flexibilitu – výsledný kód je ľahšie adaptovateľný na zmeny,
- izoluje side-effect operácie (napr. IO) od čisto funkcionálneho zvyšku.

Štruktúra prednášok:

- Monády - prvý dotyk
  - Functor
  - Applicative
  - Monády – princípy a zákony
- Najbežnejšie monády
  - Maybe/Error monad
  - List monad
  - IO monad
  - State monad
  - Reader/Writer monad
  - Continuation monad
- Transformátory monád
- Monády v praxi

Control.Monad

```haskell
sequence :: (Monad m) => [m a] -> m [a]
mapM     :: (Monad m) => (a -> m b) -> [a] -> m [b]

forM     :: (Monad m) => [a] -> (a -> m b) -> m [b]

mapM f as = sequence (map f as)
forM = flip mapM

zipWithM :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys = sequence (zipWith f xs ys)

replicateM :: (Monad m) => Int -> m a -> m [a]
replicateM n x = sequence (replicate n x)

filterM  :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
foldM    :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a

guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = zero
```

```haskell
sequence_ :: (Monad m) => [m a] -> m ()
mapM_     :: (Monad m) => (a -> m b) -> [a] -> m ()

forM_     :: (Monad m) => [a] -> (a -> m b) -> m ()

mapM_ f as = sequence_ (map f as)
forM_ = flip mapM_

zipWithM_ :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys = sequence_ (zipWith f xs ys)

replicateM_ :: (Monad m) => Int -> m a -> m ()
replicateM_ n x = sequence_ (replicate n x)

foldM_ :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m ()
```

https://hackage.haskell.org/package/base-4.6.0.1/docs/src/Control-Monad.html

mapM, forM
(Control.Monad)

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

forM       :: (Monad m) => [a] -> (a -> m b) -> m [b] -- len zámena args.
forM       = flip mapM
```

```
> mapM (\x->[x,11*x]) [1,2,3]
[[1,2,3],[1,2,33],[1,22,3],[1,22,33],[11,2,3],[11,2,33],[11,22,3],[11,22,33]]

> mapM (\x -> [True, False]) [1,2,3]
[[True,True,True],[True,True,False],[True,False,True],[True,False,False],
 [False,True,True],[False,True,False],[False,False,True],[False,False,False]]

> forM [1,2,3] (\x->[x,11*x])
[[1,2,3],[1,2,33],[1,22,3],[1,22,33],[11,2,3],[11,2,33],[11,22,3],[11,22,33]]

> mapM print [1,2,3]
1
2
3
[(),(),()]
```

```
> mapM_ print [1,2,3]
1
2
3
```

```
mapM_ (putStrLn.show)
[1,2,3]
1
2
3
```

filterM
(Control.Monad)

```
filterM     :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

```
> filterM (\x->[True, False]) [1,2,3]              -- potenčná množina, powerset
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

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

foldM
(Control.Monad)

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

foldM f a_1 [x_1, …, x_n] =
    do {
        a_2 <- f a_1 x_1;
        a_3 <- f a_2 x_2;
        …
        a_n <- f a_{n-1} x_{n-1};
        return f a_n x_n }

```
foldM f a (x:xs) = do y<-f a x
                      foldM f y xs
```

```
> foldM (\y -> \x ->
      do { print (show x++"..."++ show y);
           return (x*y)})
    1 [1..10]
???
```

```
> foldM (\y -> \x ->  do print (show x++"..."++ show y); return (x+y))  0 [1..10]
???
```

Error monad

```
newtype Either a b = Right a | Left b
instance (Error e) => Monad (Either e) where
        return x = Right x
        Right x >>= f = f x
        Left err >>= f = Left err
        fail msg = Left (strMsg msg)
```

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)

eval          :: Term -> Either String Int
eval(Con a)   = return a
eval(Div t u) = do
                valT <- eval t
                valU <- eval u
                if valU == 0 then
                    fail "div by zero"          -- throwError "div by zero"
                else
                    return (valT `div` valU)
> eval (Div (Con 1972) (Con 23))
Right 85
> eval (Div (Con 1972) (Con 0))
*** Exception: div by zero
```

Reader monad
(Control.Monad.Reader)

```
main :: IO ()
main = do params <- loadParams
          let result = func1 params
          print result
data Params = Params { p1 :: String, p2 :: String, p3 :: String }    deriving (Show)
loadParams :: IO Params
loadParams = do p1 <- lookupEnv "JAVA_HOME"
                p2 <- lookupEnv "OS"
                p3 <- lookupEnv "HOMEDRIVE"
                return $ Params (fromMaybe "no java" p1)
                                (fromMaybe "unknown" p2)
                                (fromMaybe "no drive" p3)
func1 :: Params -> String
func1 params = "Result: " ++ (show (func2 params))

func2 :: Params -> Int
func2 params = 2 + floor (func3 params)

func3 :: Params -> Float
func3 params = (fromIntegral $ length $ p1 params ++ p2 params ++ p3 params)*3.14
```

Reader.hs

Reader monad
(Control.Monad.Reader)

Reader monáda sa používa, ak máme **nemenné** prostredie, ktoré zdieľa viac výpočtov

newtype Reader r a = Reader (r -> a)   -- r je typ čítaného prostredia, a je typ výsledku

data Reader r a = Reader { runReader :: (r -> a) }

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

func :: Reader Params a
runReader func params :: a   -- runReader :: Reader r a -> r -> a
-- runReader func :: r -> a

-- získa prostredie
ask :: Params

func :: Reader Params String

func = do params <- **ask**

...

Reader monad
(Control.Monad.Reader)

```
main' :: IO ()
main' = do params <- loadParams
           let result = runReader func1' params
           putStrLn result

func1' :: Reader Params String
func1' = do params <- ask
            result <- func2'
            return $ "Result: " ++ (show result)

func2' :: Reader Params Int
func2' = do params <- ask
            result <- func3'
            return $ 2+floor(result)

func3' :: Reader Params Float
func3'  = do params <- ask
             let result = (fromIntegral $ length $ p1 params++p2 params++p3 params)*3.14
             return result
```

```
loadParams :: IO Params
params :: Params
func1' :: Reader Params String
runReader :: Reader r a -> r -> a
result :: String
```

```
ask :: m r, Reader Params String
params :: Params
```

Reader.hs

Writer monad

(Control.Monad.Writer)

```
newtype Writer w a = Writer { runWriter :: (a, w) }
instance (Monoid w) => Monad (Writer w) where
      return x = Writer (x, mempty)
      (Writer (x,v)) >>= f =
          let (Writer (y, v')) = f x
          in Writer (y, v `mappend` v')
```

Writer monáda sa používa,ak máme výpočet **produkujúci stream dát**,ktoré akumulujeme

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)

line        :: Term -> Int -> String
line t a    = "eval (" ++ show t ++ ") <=" ++ show a ++ "\n"

eval           :: Term -> Writer String Int
eval x@(Con a)   =
         do tell (line x a)
              return a
eval x@(Div t u) =
         do  valT <- eval t
             valU <- eval u
             tell (line x (valT `div` valU))
             return (valT `div` valU)
```

```
eval           :: Term -> Writer String Int
eval x@(Con a)    = writer (a, line x a)


eval x@(Div t u) =
         do  valT <- eval t
               valU <- eval u
               let result = (valT `div` valU)
               writer (result, (line x result))
```

Writer.hs

Writer monad

(Control.Monad.Writer)

```
newtype Writer w a = Writer { runWriter :: (a, w) }
instance (Monoid w) => Monad (Writer w) where
      return x = Writer (x, mempty)
      (Writer (x,v)) >>= f =
          let (Writer (y, v')) = f x
          in Writer (y, v `mappend` v')
```

Writer String Int

Writer w a

w = typ akumulátora

a = typ výsledku

-- vráti dvojicu, hodnotu a akumulátor

runWriter :: Writer w a -> (a,w)

-- vráti len akumulátor

execWriter :: :: Writer w a -> w

-- pripíše hodnotu do akumulátora, žiaden výsledok

tell :: w -> m ()

-- pripíše hodnotu do akumulátora, vráti výsledok

writer :: (a,w) -> m a

```
out :: Int -> Writer [String] Int
out x = writer (x, ["number: " ++ show x])
mult :: Writer [String] Int
mult = do {a <- out 3; b <- out 5; return (a*b) }
```

```
t :: Term
t = (Div (Div (Con 1972) (Con 2)) (Con 23))

> eval t
WriterT (Identity (42,
"eval (Con 1972) <=1972\neval (Con 2) <=2\neval (Div (Con 1972) (Con 2)) <=986\neval (Con 23) <=23\neval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=42\n"))
> runWriter (eval t)
(42,"eval (Con 1972) <=1972\neval (Con 2) <=2\neval (Div (Con 1972) (Con 2)) <=986\neval (Con 23) <=23\neval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=42\n")
> execWriter (eval t)
"eval (Con 1972) <=1972\neval (Con 2) <=2\neval (Div (Con 1972) (Con 2)) <=986\neval (Con 23) <=23\neval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=42\n"

> putStr $ execWriter (eval t)
eval (Con 1972) <=1972
eval (Con 2) <=2
eval (Div (Con 1972) (Con 2)) <=986
eval (Con 23) <=23
eval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=42

> runWriter mult
(15,["number: 3","number: 5"])
> execWriter mult
["number: 3","number: 5"]
> mapM_ putStrLn $ execWriter (mult)
number: 3
number: 5
```

Writer.hs

Writer monad

(Control.Monad.Writer)

```
gcd’ :: Int -> Int -> Writer [String] Int
gcd’ a b  | b == 0 = do
                          tell ["result " ++ show a]
                          return a
          | otherwise = do
                          tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
                          gcd’ b (a `mod` b)
```

```
gcd' :: Int -> Int -> Writer [String] Int
gcd' a b  | b == 0 = writer (a, ["result " ++ show a])
          | otherwise = do let modulo = (a `mod` b)
                           result <- gcd' b modulo
                           writer (result, [show a ++ " mod " ++ show b ++ " = " ++ show modulo])
```

```
> mapM_ putStrLn (execWriter $ gcd' 2024 64)
result 8
16 mod 8 = 0
24 mod 16 = 8
40 mod 24 = 16
64 mod 40 = 24
2024 mod 64 = 40
```

Gcd.hs

Euclid’s Game

hra pre dvoch hráčov

začínajú s dvomi prirodzenými číslami na tabuli

Jediné pravidlo:

- odčítajte menšie od väčšieho a napíšte na tabuľu, ale také, aké tam nie je

Ten kto napíše posledné číslo vyhráva, prehráva ten, čo už nevie ťahať

Aká je víťazná stratégia ?

13 6
7
1
5
2
11
9
4
3
8
10
12 modrý vyhral

18 12
6 modrý vyhral

16 6
10
4
2
14
12
8 červený vyhral

21 9
12
3
6
18
15 modrý vyhral

23 8
15
7
1
14
22
21
20
19
18
17
16
13
12
11
10
9
6
5
4
3
2 modrý vyhral

State monad

(Control.Monad.State)

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

instance Monad (State s) where
   return a            = State \s -> (a,s)
   (State x) >>= f = State \s ->
                                    let (v,s') = x s in runState (f v) s'

class (Monad m) => MonadState s m | m -> s where
   get :: m s                                   -- get vráti stav z monády
   put :: s -> m ()                             -- put prepíše stav v monáde

modify :: (MonadState s m) => (s -> s) -> m ()
modify f = do      s <- get
                   put (f s)
```

Čo je newtype vs. data vs. type

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

State s a má rovnakú reprezentáciu ako (s -> (a,s)), ale nie je to

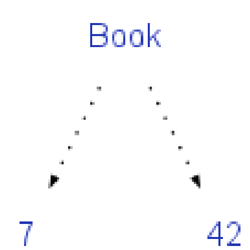
type State s a = s -> (a,s)

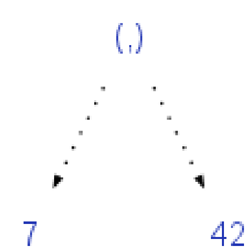
data State s a = State { runState :: (s -> (a,s)) }

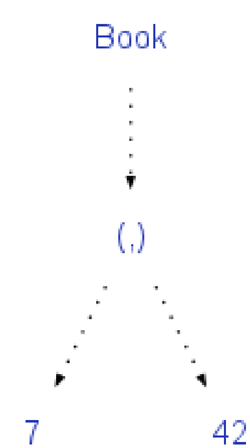
State s a je reprezentovaná krabicou State s pointrom na (s -> (a,s))

Príklad:

data Book = Book Int Int **newtype** Book = Book (Int, Int) **data** Book = Book (Int, Int)







http://stackoverflow.com/questions/5889696/difference-between-data-and-newtype-in-haskell newtype State s a = State { runState :: (s -> (a,s)) }

State s a
(basics-1)

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

- runState :: State s a -> (s -> (a, s)) -- vráti funkciu state monády
- evalState :: State s a -> s -> a -- vráti výsledok state monády pre stav s
- execState :: State s a -> s -> s -- vráti výsledný stav state monády pre vstupný stav s

:t runState ((return "hello") :: State Int String)
runState ((return "hello") :: State Int String) :: **Int -> (String, Int)**

runState ((return "hello") :: State Int String) 77 = ("hello",77)
evalState ((return "hello") :: State Int String) 77 = "hello"
execState ((return "hello") :: State Int String) 77 = 77

state.hs

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

State s a
(basics-2)

```
return :: a -> State s a        -- monáda s výsledkom x::a, stavom s
return x s = (x,s)                                 -- return x = \s -> (x,s)

get :: State s s                -- stav state monády je jej výsledkom
get s = (s,s)                                      -- get = \s -> (s,s)

runState get 1 = (1,1)

put :: s -> State s ()          -- prepíše stav monády x, výsledok je nezaujímavý
put x s = ((),x)                                   -- put x = \s -> ((),x)

runState (put 5) 1 = ((),5)
runState (do { put 5; return 'X' }) 1 = ('X',5)

modify :: (s -> s) -> State s ()
modify f = do { x <- get; put (f x) }

runState (modify (+3)) 1 = ((),4)
runState (do { modify (+3); return "hello"}) 1 = ("hello",4)
```

state.hs

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

State s a

(basics-3)

let increment = do { x <- get; put (x+1); return x } in runState increment 77
= (77,78)

```
gets :: (s -> b) -> State s b          -- aplikuje funkciu na stav monády
gets f = do { x <- get; return (f x) }
```

runState (gets (+1)) 77 = (78,77)

evalState (gets (+1)) 77 = 78 -- vráti výsledok state monády pre vstupný stav s, po aplikovaný funkcie

execState (gets (+1)) 77 = 77 -- vráti výsledný stav state monády pre vstupný stav s, a ten sa nezmenil

runState (modify (+1)) 77 = ((),78)

state.hs

Eval s vlastnou State Monad

(bolo minule)

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)
type State = Int
data SM a       =  SM (State-> (a, State))
instance Functor SM where …
instance Applicative SM where …
instance Monad SM where …
incState       :: SM ()
incState       = SM (\x -> ((),x+1))

evalSM'          :: Term -> SM Int
evalSM'(Con a)    = return a
evalSM'(Div t u)  = do valT<-evalSM' t
                        valU<-evalSM' u
                        incState
                        return(valT `div` valU)
goSM            :: Term -> State
goSM t          = let SM p = evalSM t, (result,state) = p 0 in state
```

```
> goSM' t
2
```

eval.hs

Eval

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)
type Stav = Int
```

stav výsledok

```
evalSM            :: Term -> State Stav Int
evalSM (Con a)    = return a
evalSM (Div t u) = do valT<-evalSM t
                      valU<-evalSM u
                      modify (+1)
                      return(valT `div` valU)
```

```
> runState (evalSM t) 0
(42,2)
> execState (evalSM t) 0
2
> evalState (evalSM t) 0
42
```

stateWithState.hs

```
pop :: State Stack Int
pop = state(\(x:xs) -> (x,xs))

push :: Int -> State Stack ()
push a = state(\xs -> ((),a:xs))
```

State Stack

stav

výsledok

```
type Stack = [Int]

pushAll :: Int -> State Stack String
pushAll 0    = return ""
pushAll n    = do {
               push n;
               str <- pushAll (n-1);
               nn <- pop;
               return (show nn ++ str)}
```

evalState vráti výslednú hodnotu

> **evalState** (pushAll 10) []

"10987654321"

execState vráti výsledný stav

> **execState** (pushAll 10) []

[]

```
type Stack = [Int]

pushAll' :: Int -> State Stack String
pushAll' 0    = return ""
pushAll' n    = do
               stack <- get  -- push n
               put (n:stack)
               str <- pushAll (n-1)
               (nn:stack') <- get  -- nn <- pop
               put stack'
               return (show nn ++ str)
```

> **evalState** (pushAll’ 10) []

"10987654321"

> **execState** (pushAll’ 10) []

[]

Stack.hs

Parkovací automat

Parkovaci automat sa zapína/vypína na signal '.', na začiatku je vypnutý. Keď je zapnutý, tak počíta počet automobilov, ktoré vošli '(' a počet automobilov, ktoré odišli ‚)'. Stav automatu je ich rozdiel, výsledok výpočtu je počet automobilov, ktoré na parkovisku zostali, počas období, keď bol zapnutý

```
type Vysledok = Int
type Stav = (Bool, Int)
loop :: String -> State Stav Vysledok
loop []     = do (_, result) <- get
                 return result
loop (x:xs) = do (on, diff) <- get
                 case x of
                      '(' | on             -> put (on, diff + 1)
                      ')' | on && diff >0  -> put (on, diff - 1)
                      '.'                  -> put (not on, diff)
                      _                    -> put (on, diff)
                 loop xs
```

```
main = mapM_ print [ evalState (loop input) (False, 0) |
                input <- [
                       "(()(.()()))((((.)(()",
                       "(()(.()())).)(()(.(.",
                       "(()(.()())).)(().))(.(())",
                       "(()(.()())).)(().))(.(()).)((.("]
        ]
```

Automat.hs

Dobré zátvorky ???

```
type Stav = (Bool, Int, Int)     -- .., počet (, počet [
loop :: String -> State Stav Bool
loop []     = do (ok, parents, brackets) <- get
                 return $ ok && parents == 0 && brackets == 0

loop (x:xs) = do (ok,parents, brackets) <- get
                 case x of
                   '('     -> put (ok,parents+1, brackets)
                   '['     -> put (ok,parents, brackets+1)
                   ')' | parents > 0    -> put (ok,parents-1, brackets)
                   ')' | parents <= 0 -> put (False,0,0)
                   ']'     -> put $ if (brackets>0) then (ok,parents, brackets-1)
                                                     else (False,0,0)
                 loop xs
main = mapM_ print [ evalState (loop input) (True, 0, 0) | input <- [
                     "()", "(())", "(()())", "(()()(()))", ")(", "([)]", "[[]]", "][", "()[]"] ]
```

```
> main
"() -> True"
"(()) -> True"
"(()()) -> True"
"(()()(())) -> True"
")( -> False"
"([)] -> True"
"[[]] -> True"
"][ -> False"
"()[] -> True"
```

Zatvorky.hs

Preorder so stavom
(Control.Monad.State)

```
data Tree a =      Nil |
                   Node a (Tree a) (Tree a)   deriving (Show, Eq)
```

stav

```
preorder :: Tree a -> State [a] ()              -- stav a výstupná hodnota
preorder Nil                       = return ()
preorder (Node value left right)   =
                                     do {                str :: [a]
                                        str<-get; -- get state=preorderlist
                                        put (value:str);  -- modify (value:)
                                        preorder left;
                                        preorder right;
                                        return () }
```

```
e :: Tree String
e = Node "c" (Node "a" Nil Nil) (Node "b" Nil Nil)

> execState (preorder e) []  -- stav
["b","a","c"]

> evalState (preorder e) []   -- výsledok
()
```

tree.hs

stav

výsledok

Prečíslovanie binárneho stromu

```
reindex :: Tree a -> State Int (Tree Int)        -- stav a výstupná hodnota
reindex Nil             = return Nil
reindex (Node value left right) =
                        do {
                                    i <- get;
                                    put (i+1);
                                    ileft <- reindex left;
                                    iright <- reindex right;
                                    return (Node i ileft iright) }
```

```
> e'
Node "d" (Node "c" (Node "a" Nil Nil) (Node "b" Nil Nil)) (Node "c" (Node
"a" Nil Nil) (Node "b" Nil Nil))

> evalState (reindex e') 0
Node 0 (Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)) (Node 4 (Node 5 Nil Nil)
(Node 6 Nil Nil))

> execState (reindex e') 0
7
```

tree.hs

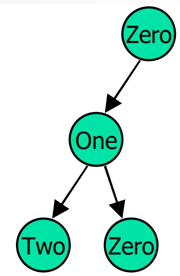
stav výsledok

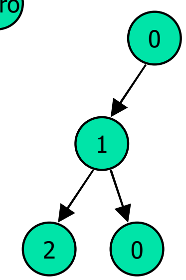
Prečíslovanie stromu 2

```
type Table a = [a]

numberTree :: Eq a => Tree a -> State (Table a) (Tree Int)
numberTree Nil                  = return Nil
numberTree (Node x t1 t2)       =  do   num <- numberNode x
                                        nt1  <- numberTree t1
                                        nt2  <- numberTree t2
                                        return (Node num nt1 nt2)
   where
   numberNode :: Eq a => a -> State (Table a) Int
   numberNode x                 = do    table <- get
                                        (newTable, newPos) <- return (addNode x table)
                                        put newTable
                                        return newPos

   addNode:: (Eq a) => a -> Table a -> (Table a, Int)
   addNode x table                      = case (findIndexInList (== x) table) of
                                             Nothing -> (table ++ [x], length table)
                                             Just i  -> (table, i)
```





tree.hs

Prečíslovanie stromu 2

```
numTree :: (Eq a) => Tree a -> Tree Int
numTree t = evalState (numberTree t) []
```

```
> numTree ( Node "Zero"
                 (Node "One" (Node "Two" Nil Nil)
                 (Node "One" (Node "Zero" Nil Nil) Nil)) Nil)

Node 0 (Node 1 (Node 2 Nil Nil) (Node 1 (Node 0 Nil Nil) Nil)) Nil
```

tree.hs

Piškvorky

```
size = 11
data PiskyState = PiskyState { playground::[[Tile]], onTurn::Bool, generator::StdGen }
data Tile = Empty | X | O deriving (Eq, Show)

-- player True/False
nextPlayer :: Bool->Bool
nextPlayer x = not x

-- player's sign
sign :: Bool -> Tile
sign True = X
sign False = O

-- no Empty on board
finish :: State PiskyState Bool
finish = do  pstate <- get
             return $ any (==Empty) ( concat (playground pstate))
```

Piskvorky.hs

Piškvorky

```
move :: State PiskyState (Int,Int)
move = do pstate <- get
          let free = [ (i,j) | i<-[0..size-1], j<-[0..size-1], (playground pstate)!!i!!j == Empty]
          let gen = generator pstate
          let (r, gen') = randomR (0, length free - 1) gen
          put $ pstate { generator = gen' }
          return $ free !! r

update :: (Int,Int) -> State PiskyState ()
update (row,col) = do pstate <- get
                      let player = onTurn pstate
                      let s = sign player
                      let newPG = [[ if i == row && j == col then s else (playground pstate)!!i!!j
                                             | j<-[0..size-1]] | i<-[0..size-1]]
                      put $ pstate { onTurn = nextPlayer player, playground=newPG }

oneTurn :: State PiskyState Bool
oneTurn = do { (row, col) <- move; update (row, col) ; finish }
```

Piskvorky.hs

Piškvorky

```
pinit :: StdGen -> PiskyState
pinit gen =  PiskyState (take size (repeat (take size ( repeat Empty))))
                        True
                        gen
main:: IO()
main = do g <- getStdGen
          let istate = pinit g
          putStr (show $ execState (sequence $ take 100 $ repeat oneTurn) istate)
```

```
> main
.oxxox.xxxx
oooo.xo.oox
xxoooooooxoo
oxxooooo.ox
x.ox.xo.xoo
oooxxxoxoxx
oxooo.x..xx
x.xooxox.xo
xoxx.oxx.oo
.xxo.x.xxoo
.xxxox..xox
```

Piskvorky.hs