Curs 12

2023-2024

Programare Logică şi funcţională

Cuprins

Clasa de tipuri **Monad**

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

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

Clasa de tipuri **Monad**

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

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor
- în Control.**Monad** sunt definite
  - f >=>g = \x -> f x >>= g
  - (<=<) = **flip** (>=>)

Functor și Applicative definiți cu **return** și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)
   -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where          -- ma >>= \a -> return
    (f a)
  fmap f ma = pure f <*> ma       -- ma >>= (return . f)
```

Notația **do** pentru monade

Notaţia cu operatori	Notaţia **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

```
do
    x1 <- e1
    e2
    e3
```

Notația **do** pentru monade

De exemplu

```
e1    >>= \x1 ->
e2    >>= \x2 ->
e3    >>= \_  ->
e4    >>= \x4 ->
e5
```

devine

Notația **do** pentru monade

De exemplu

```
e1    >>= \x1 ->
e2    >>= \x2 ->
e3    >>= \_  ->
e4    >>= \x4 ->
e5
```

devine

```
do
    x1 <- e1
    x2 <- e2
    e3
    x4 <- e4
    e5
```

Exemple de efecte laterale

I/O	Monada **IO**
Logging	Monada Writer
Stare	Monada State
Excepții	Monada **Either**
Parțialitate	Monada **Maybe**
Nedeterminism	Monada [] (listă)
Memorie read-only	Monada Reader

Analiză sintactică

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

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

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

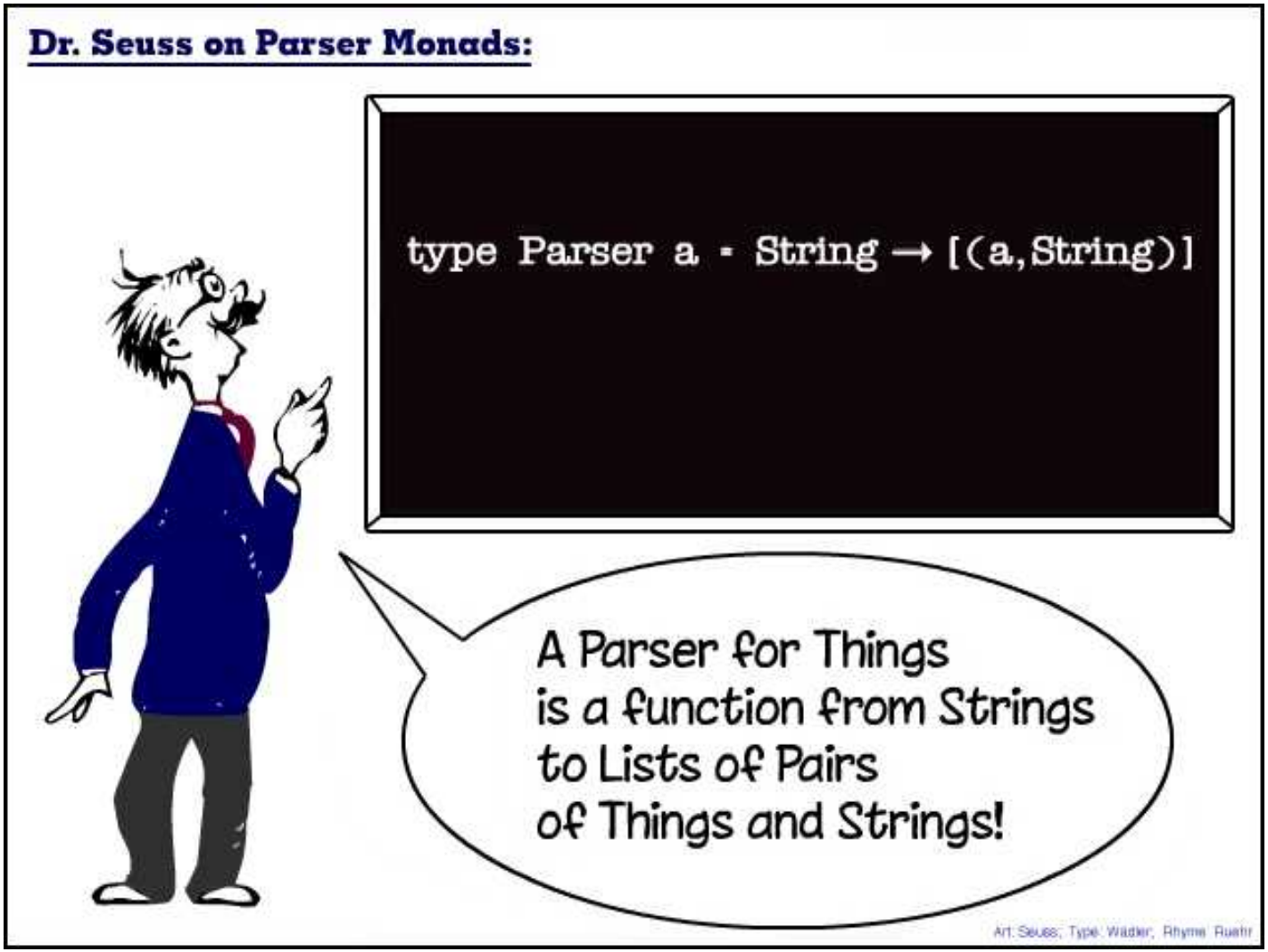
- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

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

- Dar dacă gramatica e ambiguă?
- Dar dacă intrarea nu corespunde nici unui element din a?

Tipul unui analizor sintactic



Tipul Parser

Tipul Parser

```haskell
newtype Parser a =
  Parser { apply :: String -> [(a, String)] }
```

```haskell
-- Folosirea unui parser
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) s = f s

-- Daca exista parsare, da prima varianta
parse :: Parser a -> String -> a
parse m s = head[x | (x,t) <- apply m s, t == ""]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter
anychar :: Parser Char
anychar = Parser f
  where
  f []     = []
  f (c:s) = [(c,s)]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter
anychar :: Parser Char
anychar = Parser f
  where
  f []     = []
  f (c:s) = [(c,s)]
```

```
*Main> parse anychar "a"
'a'
*Main> parse anychar "ab"
*** Exception: Prelude.head: empty list

*Main> apply anychar "abc"
[('a',"bc")]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                  = []
  f (c:s) | p c         = [(c, s)]
          | otherwise = []
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                  = []
  f (c:s) | p c         = [(c, s)]
          | otherwise = []

*Main> parse (satisfy isUpper) "A"
'A'
(0.01 secs, 52,760 bytes)
*Main> parse (satisfy isUpper) "a"
*** Exception: Prelude.head: empty list

*Main> apply (satisfy isUpper) "Ab"
[('A',"b")]
```

Parsare pentru caractere

```
-- Recunoasterea unui anumit caracter
char :: Char -> Parser Char
char c = satisfy (== c)
```

Parsare pentru caractere

```
-- Recunoasterea unui anumit caracter
char :: Char -> Parser Char
char c = satisfy (== c)
```

```
*Main> parse (char 'a') "a"
'a'
(0.00 secs, 52,824 bytes)
*Main> parse (char 'a') "ab"
*** Exception: Prelude.head: empty list

*Main>  apply (char 'a') "ab"
[('a',"b")]
```

Parsarea unui cuvânt cheie

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]
```

Parsarea unui cuvânt cheie

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]

*Main> parse (string "abc") "abc"
"abc"
*Main> parse (string "abc") "abcd"
*** Exception: Prelude.head: empty list

"*Main> apply  (string "abc") "abcd"
[("abc","d")]
```

Monada Parser

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

instance Monad Parser where
  return x  = Parser (\s -> [ (x, s) ])
  m >>= k   = Parser (\s -> [ (y, u)
                            | (x, t) <- apply m s
                            , (y, u) <- apply (k x)
                                 t
                            ])
```

Monada Parser

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]
```

e echivalent cu

```
string :: String -> Parser String
string []     = return []
string (x:xs) = do  y  <- char x
                    ys <- string xs
                    return (y:ys)
```

Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy (`elem` ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
          where f s = apply p1 s ++ apply p2 s
```

Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy (`elem` ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
          where f s = apply p1 s ++ apply p2 s

*Main> apply (alt digit abcP) "1sd"
[('1',"sd")]
*Main> apply (alt digit abcP) "Asd"
[('A',"sd")]
*Main> apply (alt digit abcP) "dsd"
[]
*Main> parse (alt digit abcP) "A"
'A'
*Main> parse (alt digit abcP) "1"
'1'
```

Recunoașterea unui caracter cu o proprietate

```
failP :: Parser a
failP = Parser (\s -> [])

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
              c <- anychar
              if (p c) then (return c) else failP
```

Recunoașterea unui caracter cu o proprietate

```
failP :: Parser a
failP = Parser (\s -> [])

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
              c <- anychar
              if (p c) then (return c) else failP
```

Definiţia fără monade este:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where     f []                  = []
            f (c:s) | p c         = [(c, s)]
                    | otherwise = []
```

Recunoașterea unei secvențe repetitive

```
-- Steluta Kleene (zero, una sau mai multe
    repetitii)
many :: Parser a -> Parser [a]
many p  = alt  (some p)     (return "")

*Main> parse (return "") ""
""

-- cel putin o repetitie
some :: Parser a -> Parser [a]
some p = do    x <- p
               xs <- many p
               return (x:xs)
```

Recunoașterea unui numar întreg

```
-- Recunoasterea unui numar natural
decimal :: Parser Int
decimal = do  s <- some digit
              return (read s)
```

Recunoașterea unui numar întreg

```
-- Recunoasterea unui numar natural
decimal :: Parser Int
decimal = do  s <- some digit
              return (read s)

-- Recunoasterea unui numar negativ
negdecimal :: Parser Int
negdecimal =  do
                 char '-'
                 n <- decimal
                 return (-n)
```

Recunoașterea unui numar întreg

```
-- Recunoasterea unui numar natural
decimal :: Parser Int
decimal = do  s <- some digit
              return (read s)

-- Recunoasterea unui numar negativ
negdecimal :: Parser Int
negdecimal =  do
                 char '-'
                 n <- decimal
                 return (-n)

-- Recunoasterea unui numar intreg
integer :: Parser Int
integer = alt decimal  negdecimal
```

Recunoașterea unui identificator

Cum arată un identificator

Un identificator este definit de doi parametri

- felul primului caracter (e.g., începe cu o literă)
- felul restului caracterelor (e.g., literă sau cifră)

Recunoașterea unui identificator

Cum arată un identificator

Un identificator este definit de doi parametri

- felul primului caracter (e.g., începe cu o literă)
- felul restului caracterelor (e.g., literă sau cifră)

Dat fiind un parser pentru felul primului caracter și un parser pentru felul următoarelor caractere putem parsa un identificator:

```
-- Recunoasterea unui identificator
iden :: Parser Char -> Parser Char -> Parser String
iden firstCh nextCh = do    c <- firstCh
                            s <- many nextCh
                            return (c : s)
```

Recunoașterea unui identificator

Cum arată un identificator

Un identificator este definit de doi parametri

- felul primului caracter (e.g., începe cu o literă)
- felul restului caracterelor (e.g., literă sau cifră)

Dat fiind un parser pentru felul primului caracter și un parser pentru felul următoarelor caractere putem parsa un identificator:

```
-- Recunoasterea unui identificator
iden :: Parser Char -> Parser Char -> Parser String
iden firstCh nextCh = do    c <- firstCh
                            s <- many nextCh
                            return (c : s)
```

Exemplu:

```
ide = iden (satisfy isAlpha)(satisfy isAlphaNum)
```

Eliminarea spațiilor

Ignorarea spațiilor

```
skipSpace :: Parser ()
skipSpace = do  _ <- many (satisfy isSpace)
                return ()
```

Eliminarea spațiilor

Ignorarea spațiilor

```
skipSpace :: Parser ()
skipSpace = do  _ <- many (satisfy isSpace)
                return ()
```

Ignorarea spațiilor de dinainte și după

```
token :: Parser a -> Parser a
token p = do  skipSpace
              x <- p
              skipSpace
              return x
```

Modulul Exp

```
module Exp where

import Monad
import Parser

data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp
         deriving (Eq,Show)

evalExp    :: Exp -> Int
evalExp    (Lit n)     = n
evalExp    (e :+: f) = evalExp e + evalExp f
evalExp    (e :*: f) = evalExp e * evalExp f
```

Recunoașterea unei expresii

```
parseExp :: Parser Exp
parseExp = alt  parseLit (alt parseAdd  parseMul)
  where
  parseLit = do   n <- integer
                  return (Lit n)
  parseAdd = do   char '('
                  d <- token parseExp
                  char '+'
                  e <- token parseExp
                  char ')'
                  return (d :+: e)
  parseMul = do   char '('
                  d <- token parseExp
                  char '*'
                  e <- token parseExp
                  char ')'
                  return (d :*: e)
```

Recunoașterea și evaluarea unei expresii

```
*Exp> parse parseExp "(1 + (2 * 3))"
Lit 1 :+: (Lit 2 :*: Lit 3)
*Exp> evalExp (parse parseExp "(1 + (2 * 3))")
7
*Exp> parse parseExp "( ( 1 + 2 ) * 3 )"
(Lit 1 :+: Lit 2) :*: Lit 3
*Exp> evalExp (parse parseExp "( ( 1 + 2 ) * 3 )")
9
```

Testare - QuickCheck

Testare QuickCheck - Exemplu

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool  -- precizam tipul
prdef xs = (myreverse xs == reverse xs)
wrongpr :: [Int] -> Bool  -- precizam tipul
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```

Testare QuickCheck - Exemplu

```
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]


prdef xs = (myreverse xs == reverse xs)
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.
```

Testare QuickCheck - Exemplu

```
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]


prdef xs = (myreverse xs == reverse xs)
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
+++ OK, passed 100 tests.
```

Ce se întâmplă?

Testare QuickCheck - Exemplu

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

> verboseCheck wrongpr
...
Passed:
[(),(),(),(),(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),(),(),(),(),(),(),(),()]
...
```

Trebuie să precizăm tipul datelor testate!

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)

wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs


> quickCheck prdef1
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)

wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs


> quickCheck prdef1
error:
        No instance for (Arbitrary Season)
```

Testare QuickCheck

- Generarea testelor aleatoare depinde de tipul de date.
- Tipurile de date care pot fi testate cu QuickCheck trebuie să fie instanţe ale clasei Arbitrary:

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a este o monadă:

```
instance Monad Gen where
  return a       = Gen (\n r -> a)
  Gen m >>= k = Gen (\n r0 ->
                          let (r1,r2) = split r0
                              Gen m'  = k (m n r1)
                          in m' n r2)
```

- La acest curs vom folosi o variantă a implementării originale: https://www.cse.chalmers.se/~rjmh/QuickCheck/

Testare QuickCheck

- Tipurile de date care pot fi testate cu QuickCheck trebuie să fie instanţe ale clasei Arbitrary:

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a poate fi tratat ca un tip abstract, datele de tip Gen a pot fi definite cu ajutorul combinatorilor:

```
choose :: Random a => (a, a) -> Gen a
oneof :: [Gen a] -> Gen a
elements :: [a] -> Gen a
....
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

instance Arbitrary Season where
  arbitrary = elements [Spring, Summer, Autumn, Winter]
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

instance Arbitrary Season where
  arbitrary =  elements[Spring, Summer, Autumn, Winter]

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)
wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs

> quickCheck prdef1
+++ OK, passed 100 tests.

> quickCheck wrongpr1
*** Failed! Falsified (after 3 tests):
[Winter,Summer]
```

Testare QuickCheck - ADT

Definiţi o instanţă a clasei Arbitrary pentru

```
data ElemIB = I Int | B Bool
              deriving (Show, Eq)

instance Arbitrary ElemIB where
    arbitrary = do
                  x <- arbitrary
                  y <- arbitrary
                  elements [I x, B y]
```

Testare QuickCheck - ADT

Definiţi o instanţă a clasei Arbitrary pentru

```
data ElemIB = I Int | B Bool
              deriving (Show, Eq)

instance Arbitrary ElemIB where
    arbitrary = do
                  x <- arbitrary
                  y <- arbitrary
                  elements [I x, B y]

wrongpr3 :: [ElemIB] -> Bool
wrongpr3 xs = myreverse xs == xs

>quickCheck wrongpr3
*** Failed! Falsified (after 8 tests):
[I (-2),I (-3)]
```

Testare QuickCheck - ADT

```
wrongpr3 :: [ElemIB] -> Bool
wrongpr3 xs = myreverse xs == xs

>quickCheck wrongpr3
*** Failed! Falsified (after 8 tests):
[I (-2),I (-3)]
> quickCheck wrongpr3
*** Failed! Falsified (after 3 tests):
[B True,B False]
> quickCheck wrongpr3
*** Failed! Falsified (after 3 tests):
[B True,I (-2)]
```

Exerciţii - exemplu examen

Exerciţiu

Se considera urmatoarea reprezentare pentru arbori binari:

```
data Binar a = Gol | Nod (Binar a) a (Binar a)

exemplu :: Binar Integer
exemplu = Nod
            (Nod (Nod Gol 2 Gol) 4 (Nod Gol 5 Gol))
            7
            (Nod Gol 9 Gol)
```

Exerciţiu

Un drum în acest arbore îl reprezentam ca o secveţă de direcţii:

```
data Binar a = Gol | Nod (Binar a) a (Binar a)

data Directie = Stanga | Dreapta

type Drum = [Directie]

exemplu :: Binar Integer
exemplu = Nod
            (Nod (Nod Gol 2 Gol) 4 (Nod Gol 5 Gol))
            7
            (Nod Gol 9 Gol)

[Stanga, Dreapta, Stanga] :: Drum
```

Exerciţiu

a) Dat fiind un drum în arbore, determinaţi informaţia din nodul la care se ajunge parcurgând arborele dupa direcţiile date. Daca se ajunge la un nod gol se va întoarce **Nothing**.

```
test1, test2 :: Bool
test1 = plimbare [Stanga, Dreapta] exemplu == Just 5
test2 = plimbare [Dreapta, Stanga] exemplu == Nothing
```

Exerciţiu

a) Dat fiind un drum în arbore, determinaţi informaţia din nodul la care se ajunge parcurgând arborele dupa direcţiile date. Daca se ajunge la un nod gol se va întoarce **Nothing**.

```
test1, test2 :: Bool
test1 = plimbare [Stanga, Dreapta] exemplu == Just 5
test2 = plimbare [Dreapta, Stanga] exemplu == Nothing

plimbare :: Drum -> Binar a -> Maybe a
plimbare _ Gol = Nothing
plimbare [] (Nod _ x _) = Just x
plimbare (Stanga:is) (Nod st _ _) = plimbare is st
plimbare (Dreapta:is) (Nod _ _ dr) = plimbare is dr
```

Exerciţiu

b) Pentru arbori cu elemente de tip **Integer** definiţi o proprietate care verfica că nodul final al unui drum este gol sau conţine un element pozitiv:

```
propArb :: Binar Integer -> Drum -> Bool
propArb = undefined
```

Folosind quickCheck testaţi că drumurile unui arbore au această proprietate sau determinaţi un contraexemplu:

```
> quickCheck (propArb exemplu1)
OK, passed 100 tests.
```

Exerciţiu

```
propArb :: Binar Integer -> Drum -> Bool
propArb arb  d = fromMaybe 0 (plimbare d arb) >= 0

*Main> propArb exemplu1 [Stanga, Stanga]
True
```

Exerciţiu

```
propArb :: Binar Integer -> Drum -> Bool
propArb arb  d = fromMaybe 0 (plimbare d arb) >= 0

*Main> propArb exemplu1 [Stanga, Stanga]
True
```

Pentru a folosi QuickCheck trebuie sa definim o instanţă a clasei Arbitrary:

```
instance Arbitrary Directie where
    arbitrary =  elements [Dreapta, Stanga]


> quickCheck (propArb exemplu1)
OK, passed 100 tests.
```

Exerciţiu

```
instance Arbitrary Directie where
    arbitrary =  elements [Dreapta, Stanga]

> quickCheck (propArb exemplu1)
OK, passed 100 tests.

cexemplu = Nod
             (Nod (Nod Gol(-2)Gol) 4 (Nod Gol 5 Gol))
             7
             (Nod Gol 9 Gol)

> quickCheck (propArb cexemplu)
Falsifiable, after 64 tests:
[Stanga,Stanga]
```

Exerciţiu

c) Presupunem că arborii conţin informaţie de tip (Cheie, Valoare) şi că sunt arbori de cautare dupa cheie (elementele din subarborele stâng au cheia mai mica decât cheia din radacina, iar cele din subarborele drept au cheia mai mare decat cea din radacina).

```
type Cheie = Integer
type Valoare = Float

ex :: Binar (Integer, Float)
ex =  Nod
                (Nod
                    (Nod Gol (2, 3.5) Gol)
                    (4, 1.2)
                    (Nod  Gol (5, 2.4) Gol))
                (7, 1.9)
                (Nod Gol (9, 0.0) Gol)
```

Exerciţiu

(continuare enunţ)
Definim monada Writer specializata la String:

```
newtype WriterStr a = Writer{runWriter::(a,String)}

instance Monad WriterStr where
  return x = Writer (x, "")
  ma >>= k = let (x, logx) = runWriter ma
                 (y, logy) = runWriter (k x)
             in Writer (y, logx ++ logy)

tell :: String -> WriterStr ()
tell s = Writer ((), s)
```

Exerciţiu

(continuare enunţ)
Scrieţi o funcţie care caută în arbore valoarea corespunzatoare unei chei date, întoarce aceasta valoare (daca exista) şi are ca efect lateral înregistrarea drumului parcurs.
Punctajul maxim se va obţine pentru varianta monadică.

```
cauta :: Cheie-> Binar (Cheie, Valoare)->
                        WriterStr (Maybe Valoare)


test3, test4 :: Bool
test3 = runWriter (cauta 5 ex) ==
                            (Just 2.4,"Stanga;Dreapta;")
test4 = runWriter (cauta 8 ex) ==
                            (Nothing, "Dreapta;Stanga;")
```

Exerciţiu

Scrieţi o funcţie care caută în arbore valoarea corespunzatoare unei chei date, întoarce aceasta valoare (daca exista) şi are ca efect lateral înregistrarea drumului parcurs. Punctajul maxim se va obţine pentru varianta monadică.

```
cauta :: Cheie -> Binar (Cheie, Valoare)
                        -> WriterStr (Maybe Valoare)
cauta cheie Gol = return Nothing
cauta cheie (Nod st (cheie', valoare) dr)
    | cheie == cheie' = return (Just valoare)
    | cheie < cheie' = do
                          tell "Stanga;"
                          cauta cheie st
    | otherwise = do
                     tell "Dreapta;"
                     cauta cheie dr
```

Succes în sesiune!