Definisanje tipova i klasa

Deklaracija tipa

Definisanje sinonima za tipove

```
type String = [Char]
```

```
type Pos = (Int,Int)

origin :: Pos
origin = (0,0)

left :: Pos -> Pos
left (x,y) = (x-1, y)
```

```
ghci> left (3,4)
(2,4)
```

```
type Pair a = (a,a)

copy :: a -> Pair a
copy x = (x,x)

mult :: Pair Int -> Int
mult (m,n) = m*n
```

```
ghci> copy 3
(3,3)
```

```
ghci> mult (3,4)
12
```

Deklaracija tipa

Mogu biti ugnježdene

- Na osnovi jednog tipa može se definisati drugi

```
type Pos = (Int,Int)
type Trans = Pos -> Pos
```

Ne mogu biti rekurzivne

- Ovakvu definiciju nije moguće „razviti“

```
type Tree = (Int, [Tree])
```

Deklaracija tipa

Mogu biti sa parametrom

```
type Pair a = (a,a)
```

Mogu imati više parametrara

```
type Assoc k v = [(k,v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k’,v) <- t, k == k’]
```

Deklaracija podataka

Definicija novog tipa navođenjem vrednosti (konstruktora)

```
data Bool = False | True
```

Mogu se posmatrati kao kontekstno slobodne gramatike

```haskell
data Answer = Yes | No | Unknown
    deriving (Show)

answers :: [Answer]
answers = [Yes, No, Unknown]

flip' :: Answer -> Answer
flip' Yes     = No
flip' No      = Yes
flip' Unknown = Unknown
```

```
ghci> flip' Yes

<interactive>:5:1: error:
    * No instance for (Show Answer) arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
ghci>
```

```
ghci> flip' Yes
No
```

Deklaracija podataka

```haskell
data Move = North | South | East | West
   deriving (Show)

move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East (x,y) = (x+1,y)
move West (x,y) = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

```
ghci> moves [North,North,West] (1,2)
(0,4)
```

Deklaracija podataka

```
data Shape = Circle Float | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

```
ghci> :type Circle
Circle :: Float -> Shape
ghci> area (square 5)
25.0
```

Deklaracija podataka

U slučajevima kada se koriste „zaključane“ biblioteke u kojima se definisani neki tipovi podataka

- Jednostavno je definisati nove tipove na osnovu već definisanih
- Jednostavno je definisati nove metode za definisane tipove
- Problem predstavlja definisanje podtipa koji ne može da se doda „zaključanom“ tipu

```
data Shape = ... | Polygon...   !!!
```

Deklaracija podataka sa parametrima

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

```
ghci> safehead []
Nothing
ghci> safehead [2,3,4]
Just 2
```

Rekurzivni tipovi

```
data Nat = Zero | Succ Nat
    deriving (Show)

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Peanovi brojevi

Kako bez Int definisati

```
add :: Nat -> Nat -> Nat
```

```
ghci> int2nat 4
Succ (Succ (Succ (Succ Zero)))
```

Aritmetički izrazi

```haskell
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
    deriving (Show)

e1 :: Expr
e1 = Add (Val 1) (Mul (Val 3) (Val 3))
```

```haskell
size :: Expr -> Int
size (Val n)    = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y
```

```haskell
eval :: Expr -> Int
eval (Val n)    = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

```
ghci> e1
Add (Val 1) (Mul (Val 3) (Val 3))
ghci> size e1
3
ghci> eval e1
10
```

Kako bi za ovakav tip podataka bila definisana funkcija fold i kako bi se koristila?

Binarna stabla

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

```
occurs :: Eq a => a -> Tree a -> Bool
```

- Pojavljivanje čvora u stablu

```
flatten :: Tree a -> [a]
```

- Formiranje sortirane liste od stabla

Iskazne formule

```
data Prop = Const Bool
          | Val Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

```
isTaut :: Prop -> Bool
```

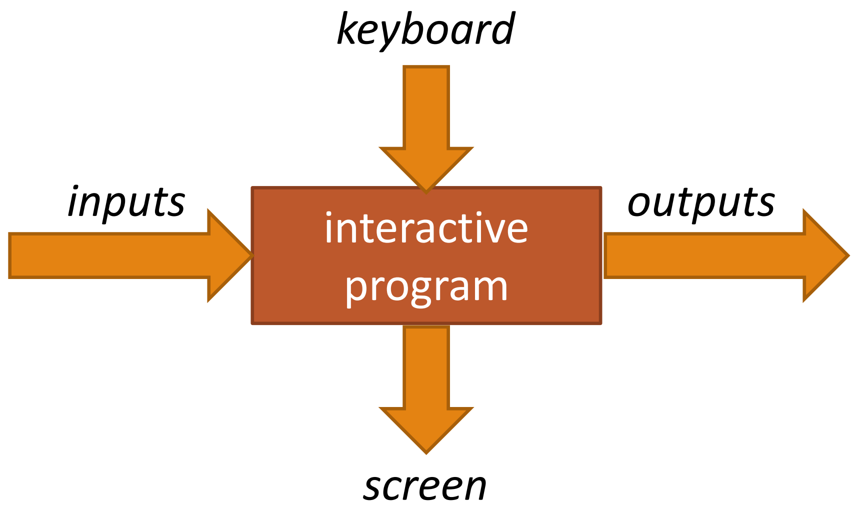
- Ispitivanje da li je formula tutologija
  - `eval – vrednost formule`
  - `vars – iskazna slova iz formule`
  - `bools – generisanje svih valuacija`
  - `substs – sve kombinacije vrednosti iskaznih slova i njihovih valuacija`

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
          where bss = bools (n-1)
```

Interaktivni programi

Uvod

Haskell programi su čisto funkcijski/matematički
No side effects



ReadLine – nije matematička funkcija
Interactive programs have side effects

Current "state of world" as argument

Tip IO

```
type IO = World -> World
```

```
type IO a = World -> (a, World)
```

`IO a` – akcija koja vraća tip a

- `IO Char` – vraća karakter
- `IO ()` -- akcija koja ne vraća vrednost

Standarda biblioteka sadrži veliki broj IO akcija

- `getChar :: IO Char     <~>    getChar :: () -> IO Char`
- `putChar :: Char -> IO ()`
- `return :: a -> IO a`

Sekvence

Niz akcija koje se spajaju u jednu korišćenjem ključne reči do

```
do v1 <- a1
   v2 <- a2
   ...
   vn <- an
   return (f v1 v2 ... vn)
```

```
a :: IO (Char, Char)
a = do x <- getChar
       getChar
       y <- getChar
       return (x,y)
```

```
ghci> a
a
b
('a','b')
```

Učitavanje i štampanje stringova

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)

b :: IO String
b = do x <- getLine
       return x
```

```
ghci> b
Haskell
"Haskell"
```

Učitavanje i štampanje stringova

```
putStr          :: String -> IO ()
putStr []       = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

```
ghci> putStr "abcd"
abcdghci>
```

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

```
ghci> putStrLn "abcd"
abcd
```

Interaktivni program

```haskell
strlen :: IO ()
strlen = do putStr "Enter a string "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

```
ghci> strlen
Enter a string Haskell
The string has 7 characters
```

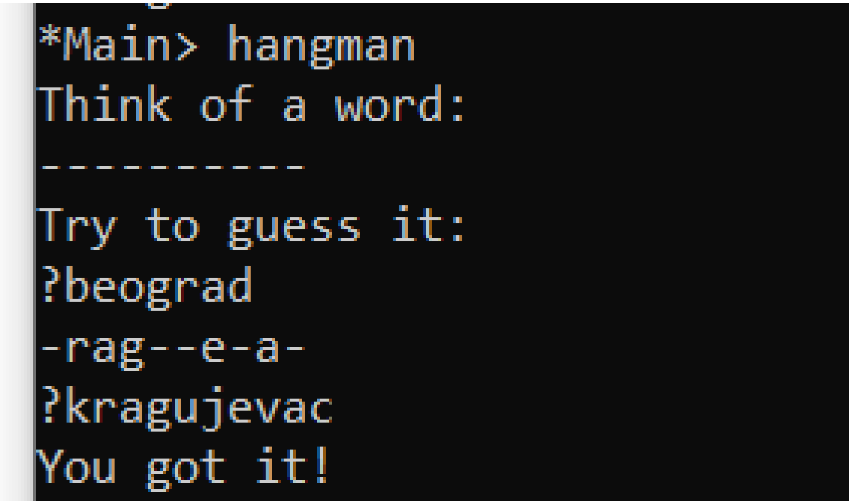
Izračunavanje akcija izvršava „*side effects*“ dok je konačni rezultat zanemaren

Hangman

Top-down pristup

```
import System.IO
```

```haskell
hangman :: IO ()
hangman = do putStrLn "Think of a word:"
             word <- sgetLine
             putStrLn "Try to guess it:"
             play word
```



Hangman

Prihvatanje reči, ali bez prikazivanja karaktera

```
sgetLine :: IO String
sgetLine = do x <- getCh
              if x == '\n' then
                 do putChar x
                    return []
              else
                 do putChar '-'
                    xs <- sgetLine
                    return (x:xs)
```

Hangman

Prihvatanje karaktera, bez prikazivanja

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           x <- getChar
           hSetEcho stdin True
           return x
```

Hangman

Glavna petlja

```
play :: String -> IO ()
play word = do putStr "?"
               guess <- getLine
               if guess == word then
                 putStrLn "You got it!"
               else
                  do putStrLn (match word guess)
                     play word
```

```
match :: String -> String -> String
match xs ys = [if elem x ys then x else '-'| x <- xs]
```

Kako proširiti brojačem pokušaja?

Monadi i ostalo

Funktori

Uopštenje funkcije `map`

Ideja mapiranja funkcija na svaki element ne mora biti ograničena na liste

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Funktori – klasa tipova koji podržavaju upšteno mapiranja

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

`fmap` prihvata funkciju `a -> b` i na strukturu tipa `f a`, čiji su lementi tipa `a`, primenjuje funkciju i vraća strukturu tipa `f b`

`f` mora biti parametrizovan tip

Funktori

```haskell
instance Functor [] where
-- fmap :: (a -> b) -> [a] -> [b]
   fmap = map
```

```
ghci> map (+1) [1,3,4,7]
[2,4,5,8]
ghci> fmap (+1) [1,3,4,7]
[2,4,5,8]
```

```haskell
instance Functor Maybe where
-- fmap :: (a -> b) -> Maybe a -> Maybe b
   fmap _ Nothing = Nothing
   fmap g (Just x) = Just (g x)
```

```
ghci> fmap (+1) Nothing
Nothing
ghci> fmap (^2) (Just 3)
Just 9
ghci> fmap not (Just False)
Just True
```

```
ghci> fmap (++"!") getLine
hi
"hi!"
```

Funktori

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Show

instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node left right) = Node (fmap f left) (fmap f right)
```

```
ghci> fmap length (Leaf "abc")
Leaf 3
ghci> fmap even (Node (Leaf 1) (Leaf 2))
Node (Leaf False) (Leaf True)
ghci> fmap (\x -> x*2) (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 4)
```

Zakoni za funktore

```
fmap id     = id

fmap (g.h) = fmap g . fmap h
```

```
ghci> fmap (not.even) [1,2]
[True,False]
ghci> (fmap not . fmap even) [1,2]
[True,False]
```

Aplikativni funktori - Applicatives

Uopštenje funktora na funkcije sa više argumenata

Definisati hijerarhiju za `fmap`, npr.

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
...
```

Ili curring

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

`pure` – konvertuje tip `a` u tip `f a`

`<*>` - uopšten oblik primene funkcije gde su funkcija argument, vrednosti argumenta i vrednost rezultata su `f` strukture

Aplikativni funktori - Applicatives

```
pure g <*> x1 <*> x2 <*> ... <*> xn
```

Svaki argument `xi` ima tip `f ai`, a rezultat je tipa `f b`

```
fmap0 :: a -> f a
fmap0 = pure

fmap1 :: (a -> b) -> f a -> f b
fmap1 g x = pure g <*> x

fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
fmap3 g x y z = pure g <*> x <*> y <*> z

...
```

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Aplikativni funktori - Applicatives

Klasa funktora koja podržava pure i <*> naziva se aplikativni funktor

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

```
ghci> pure (+1) <*> Just 1
Just 2
ghci> pure (+) <*> Just 1 <*> Just 2
Just 3
ghci> pure (+) <*> Nothing <*> Just 2
Nothing
```

```
ghci> pure (\x -> \y -> \z -> x*y*z) <*> Just 1 <*> Just 2 <*> Just 3
Just 6
```

```
ghci> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

Aplikativni zakoni

```
pure id <*> x    = x
pure (g x)       = pure g <*> pure x
x <*> pure y     = pure (\g -> g y) <*> x
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

Monadi

```haskell
data Expr = Val Int | Div Expr Expr
eval :: Expr -> Int
eval (Val n)   = n
eval (Div x y) = eval x `div` eval y
```

```
ghci> eval (Div (Val 1) (Val 0))
*** Exception: divide by zero
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
            Nothing -> Nothing
            Just n -> case eval y of
                  Nothing -> Nothing
                  Just m -> safediv n m
```

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```
ghci> eval (Div (Val 1) (Val 0))
Nothing
```

Monadi

Rešenje preko aplikativnih funktora?

```
eval :: Expr -> Maybe Int
eval (Val n) = pure n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

Neodgovarajući tipovi!!!

```
(>>==) :: m a -> (a -> m b) -> m b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x -> f x
```

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m
```

Monadi

Uopšten oblik

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

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

    return = pure
```

IO akcije, funkcije nad stablima, izrazima, do notacija, Maybe, …

Zakoni za monade

```
return x >>= f     = f x
mx >>= return      = mx
(mx >>= f) >>= g   = mx >>= (\x -> (f x >>=g))
```

Tipovi i klase tipova

Type	Typeclasses
Bool	Eq, Ord, Show, Read, Enum, Bounded
Char	Eq, Ord, Show, Read, Enum, Bounded
Int	Eq, Ord, Show, Read, Enum, Bounded, Num, Real, Integral
Integer	Eq, Ord, Show, Read, Enum, Num, Real, Integral
Float	Eq, Ord, Show, Read, Enum, Num, Real, Fractional, RealFrac, Floating, RealFloat
Double	Eq, Ord, Show, Read, Enum, Num, Real, Fractional, RealFrac, Floating, RealFloat
Word	Eq, Ord, Show, Read, Enum, Bounded, Num, Real, Integral
Ordering	Eq, Ord, Show, Read, Enum, Bounded, Semigroup, Monoid
()	Eq, Ord, Show, Read, Enum, Bounded, Semigroup, Monoid
Maybe a	Eq, Ord, Show, Read, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
[a]	Eq, Ord, Show, Read, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
(a,b)	Eq, Ord, Show, Read, Bounded, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
a->b	Semigroup, Monoid, Functor, Applicative, Monad
IO	Semigroup, Monoid, Functor, Applicative, Monad
IOError	Eq, Show