Monadi i ostalo

Funktori

Uopštenje funkcije `map`

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

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

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 elementi tipa `a`, primenjuje funkciju i vraća strukturu tipa `f b`

`f` mora biti parametrizovan tip

Funktori

```
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]
```

```
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

```
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
```

```
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
```

```
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))
```

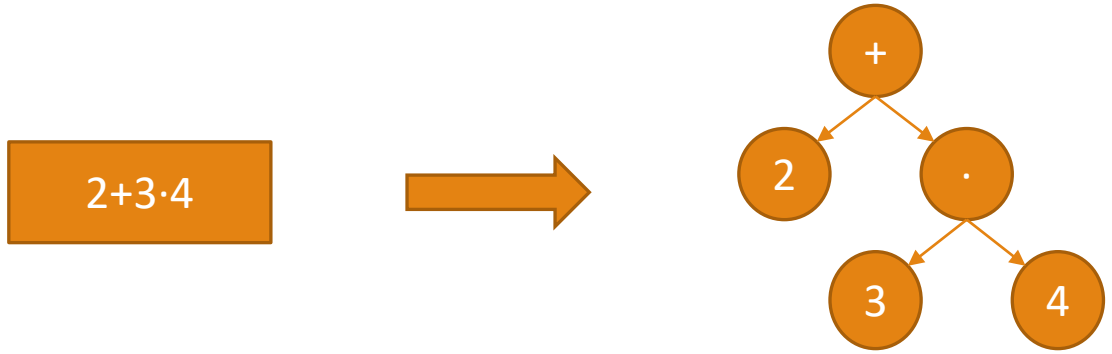
Tipovi i klase tipova

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

Parser

Parser

Program koji analizira tekst i određuje njegovu sintaksnu strukturu



Svaki kompajler, interpreter, OS, veb čitač,…. mora imati parser

```
type Parser = String -> Tree
```

Ovako definisan parser je “Black Box”

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

Vraća drvo i deo ulaza koji nije obrađen

Parser

Šta ako ulaz nije moguće parsirati u slučaju greške?

```
data Maybe                  ~> [ ]
     | Just a               ~> [a]
```

```
type Parser = String -> [(Tree, String)]
```

Različiti parseri će vratiti različite tipove stabla ili neku drugu vrednost

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

Osnovni parser

```
import Control.Applicative
import Data.Char
```

Da bismo Parser koristili kao instancu klase koristićemo dummy konstruktor

```
newtype Parser a = P (String -> [(a,String)])
```

Ovakav parser se primenjuje na ulazni string korišćenjem funkcije koja uklanja konstruktor

```
parse :: Parser a -> String -> [(a,String)]
parse (P p) inp = p inp
```

Osnovni parser

Prvi karakter ulaza

```
item :: Parser Char
item = P (\inp -> case inp of
          [] -> []
          (x:xs) -> [(x,xs)])
```

```
ghci> parse item ""
[]
ghci> parse item "abc"
[('a',"bc")]
```

Parseri sekvenci

Parser kao instanca funktora

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P (\inp -> case parse p inp of
                 [] -> []
                 [(v,out)] -> [(g v, out)])
```

```
ghci> parse (fmap toUpper item) "abc"
[('A',"bc")]
ghci> parse (fmap toUpper item) ""
[]
```

Parseri sekvenci

Parser kao instanca aplikativnog funktora

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure v = P (\inp -> [(v,inp)])
  -- <*> :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P (\inp -> case parse pg inp of
                  [] -> []
                  [(g,out)] -> parse (fmap g px) out)
```

```
three :: Parser (Char,Char)
three = pure g <*> item <*> item <*> item
          where g x y z = (x,z)
```

```
ghci> parse (pure 1) "abc"
[(1,"abc")]
ghci> parse three "abcdef"
[(('a','c'),"def")]
ghci> parse three "ab"
[]
```

Parseri sekvenci

Parser kao instanca aplikativnog monada

```
instance Monad Parser where
  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P (\inp -> case parse p inp of
                [] -> []
                [(v,out)] -> parse (f v) out)
```

```
three' :: Parser (Char,Char)
three' = do
            x <- item
            item
            z <- item
            return (x,z)
```

Donešenje odluka

Donošenje odluka nije specifično za parser, ali može da se generalizuje aplikativne tipove

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

instance Alternative Maybe where
  empty :: Maybe a
  empty = Nothing

  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> my = my
  (Just x) <|> _ = Just x
```

Donošenje odluka

```
instance Alternative Parser where
  -- empty :: Parser a
  empty = P (\inp -> [])
  -- (<|>) :: Parser a -> Parser a -> Parser a
  p <|> q = P (\inp -> case parse p inp of
                [] -> parse q inp
                [(v,out)] -> [(v,out)])
```

```
ghci> parse empty "abc"
[]
ghci> parse (item <|> return 'd') "abc"
[('a',"bc")]
ghci> parse (empty <|> return 'd') "abc"
[('d',"abc")]
```

Izvedeni tipovi

```
sat :: (Char -> Bool) -> Parser Char
sat p = do
           x <- item
           if p x then return x else empty
```

```
digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper
```

```
letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char -> Parser Char
char x = sat (== x)
```

Izvedeni tipovi

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

```
ghci> parse (char 'a') "abc"
[('a',"bc")]
ghci> parse (string "abc") "abcdef"
[("abc","def")]
ghci> parse (string "abc") "ab1234"
[]
```

```
ghci> parse (many digit) "123abc"
[("123","abc")]
ghci> parse (many digit) "abc"
[("","abc")]
ghci> parse (some digit) "abc"
[]
```

Izvedeni tipovi

```
ident :: Parser String
ident = do
          x <- lower
          xs <- many alphanum
          return (x:xs)
```

```
nat :: Parser Int
nat = do
        xs <- some digit
        return (read xs)
```

```
ghci> parse ident "abc def"
[("abc"," def")]
ghci> parse nat "123 abc"
[(123," abc")]
```

Izvedeni tipovi

```
space :: Parser ()
space = do
          many (sat isSpace)
          return ()
```

```
int :: Parser Int
int = do
        char '-'
        n <- nat
        return (-n)
      <|> nat
```

```
ghci> parse space " abc"
[((),"abc")]
ghci> parse int "-123 abc"
[(-123," abc")]
```

Obrada belina

```
token :: Parser a -> Parser a
token p = do
            space
            v <- p
            space
            return v
```

```
identifier :: Parser String
identifier = token ident
```

```
natural :: Parser Int
natural = token nat
```

```
integer :: Parser Int
integer = token int
```

```
symbol :: String -> Parser String
symbol xs = token (string xs)
```

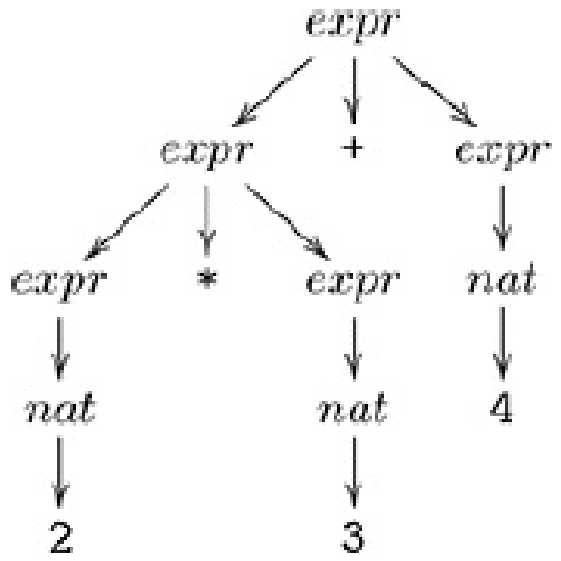
Aritmetički izrazi

expr ::= expr + expr | expr * expr | (expr) | nat

nat ::= 0 | 1 | 2 | …

Ovako definisana gramatika ne uzima u obzir činjenucu da je * većeg prioriteta od +

2 * 3 + 4



Aritmetički izrazi

expr ::= expr + expr | term

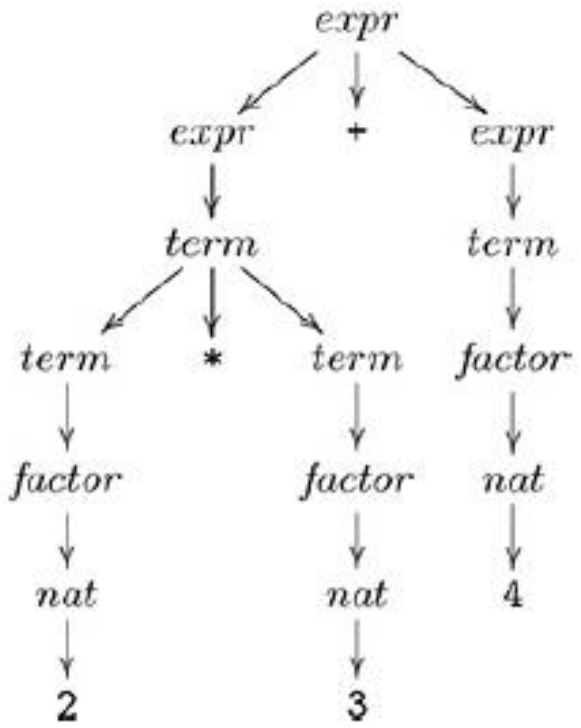
term ::= term * term | factor

factor ::= (expr) | nat

nat ::= 0 | 1 | 2 | …

Prioritet je rešen, ali nije uzeta u obzir desna asocijativnost za * i +

2 * 3 + 4



Aritmetički izrazi

expr ::= term + expr | term

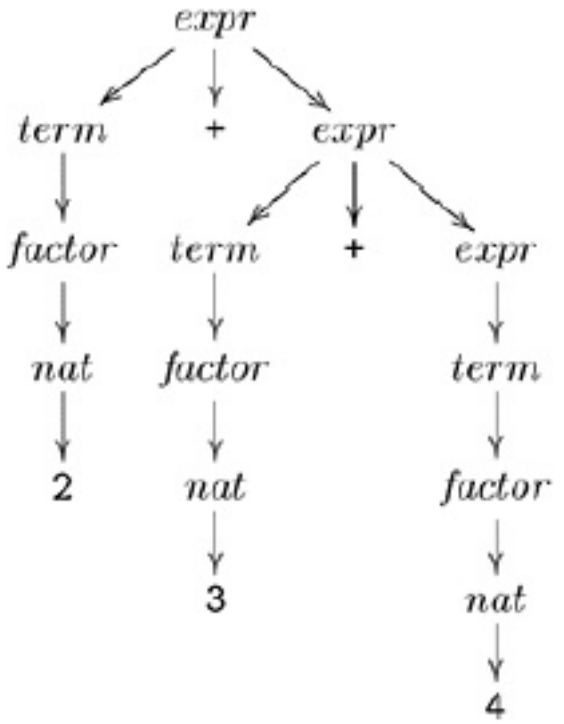
term ::= factor * term | factor

factor ::= (expr) | nat

nat ::= 0 | 1 | 2 | …

Dobijena gramatika je nedvosmislena

2 + 3 + 4



Aritmetički izrazi

expr ::= term (+ expr | ε)

term ::= factor (* term | ε)

factor ::= (expr) | nat

nat ::= 0 | 1 | 2 | …

```
expr :: Parser Int
expr = do
         t <- term
         do
            symbol "+"
            e <- expr
            return (t + e)
            <|> return t
```

```
term :: Parser Int
term = do
         f <- factor
         do
             symbol "*"
             t <- term
             return (f * t)
             <|> return f
```

```
factor :: Parser Int
factor = do
             symbol "("
             e <- expr
             symbol ")"
             return e
         <|> natural
```

Aritmetički izrazi

```
eval' :: String -> Int
eval' xs = case (parse expr xs) of
             [(n,[])]  -> n
             [(_,out)] -> error ("Unused input " ++ out)
             []        -> error "Invalid input"
```

```
ghci> eval' "2*3+4"
10
ghci> eval' "2*(3+4)"
14
```

```
ghci> eval' "2*3^4"
*** Exception: Unused input ^4
CallStack (from HasCallStack):
  error, called at parser.hs:176:27 in main:Main
ghci> eval' "one plus two"
*** Exception: Invalid input
CallStack (from HasCallStack):
  error, called at parser.hs:177:27 in main:Main
ghci>
```

Kalkulator

```
box :: [String]
box = ["+---------------+",
       "|               |",
       "+---+---+---+---+",
       "| q | c | d | = |",
       "+---+---+---+---+",
       "| 1 | 2 | 3 | + |",
       "+---+---+---+---+",
       "| 4 | 5 | 6 | - |",
       "+---+---+---+---+",
       "| 7 | 8 | 9 | * |",
       "+---+---+---+---+",
       "| 0 | ( | ) | / |",
       "+---+---+---+---+"]
```

Kalkulator

```
buttons :: String
buttons = standard ++ extra
   where
      standard = "qcd=123+456-789*0()/"
      extra = "QCD \ESC\BS\DEL\n"
```

```
showbox :: IO ()
showbox = sequence_ [writeAt (1,y) b | (y,b) <- zip [1..] box]
```

```
display xs = do
               writeAt (3,2) (replicate 13 ' ')
               writeAt (3,2) (reverse (take 13 (reverse xs)))
```

Kalkulator

```
calc :: String -> IO ()
calc xs = do
            display xs
            c <- getCh
            if elem c buttons then
                  process c xs
            else
                  do
                    beep
                    calc xs
```

```
process :: Char -> String -> IO ()
process c xs | elem c "qQ\ESC"  = quit
             | elem c "dD\BS\DEL" = delete xs
             | elem c "=\n"       = eval xs
             | elem c "cC"        = clear
             | otherwise          = press c xs
```

Kalkulator

```
quit :: IO ()
quit = goto (1,14)
```

```
delete :: String -> IO ()
delete [] = calc []
delete xs = calc (init xs)
```

```
eval :: String -> IO ()
eval xs = case parse expr xs of
            [(n,[])]  -> calc (show n)
            _         -> do
                           beep
                           calc xs
```

Kalkulator

```
clear :: IO ()
clear = calc []
```

```
press :: Char -> String -> IO ()
press c xs = calc (xs ++ [c])
```

```
run :: IO ()
run = do
        cls
        showbox
        clear
```