Uvod u funkcionalno programiranje

Stefan Nožinić (stefan@lugons.org)

January 10, 2021

Uvod

- LISP 1950te
- Deklarativni stil
- Imutabilnost
- Thread-safeness

Primer promene stanja u C

```
#include <stdio.h>

int f( int* x )
{
        int res = (*x) + 1;
        (*x)++;
        return res;
}

int main() {
        int x = 5;
        printf("%d\n", f(&x));
        printf("%d\n", f(&x));
        return 0;
}
```

- Funkcija f povećava x za jedan i vraća x+1
- f menja stanje od x
- Ako ne vidimo definiciju funkcije f, ne znamo šta ona zapravo radi sa vrednostima ulaznih parametara!
- Da li uvek možemo da tvrdimo da je f(x) == f(x)?

Prednosti nemenjanja stanja promenljivih

- Kompajler će ispisati grešku kada pokušamo promeniti stanje
- Svaka funkcija za iste ulazne parametre vraća istu povratnu vrednost
- Moguće lako paralelizovati program jer nema promene stanja promenljive

Data in - data out model

- Primenjivo nezavisno od jezika
- Svaka funkcija vraća neku vrednost
- Za isti ulaz, funkcija uvek vraća isti izlaz
- Funkcija ne zavisi od globalnog okruženja (globalne promenljive, IO, ...)
- Funkcija ne modifikuje sopstveni ulaz
- Moguće primeniti i u OOP kodu - svaki setter vraća novi objekat
- Primer: lista

Lista u C

```
typedef struct ListElem_t
{
        int value;
        struct ListElem_t* next;
} ListElem;
```

Konstruktor elementa liste

```
ListElem* create_element(int value, ListElem* next)
{
        ListElem* elem = (ListElem*) malloc(sizeof(
        elem->value = value;
        elem->next = next;
        return elem;
}
```

Lista struktura

```
typedef struct
{
        ListElem* head;
} List;
```

Konstruktor liste

```
List* create_list(ListElem* head)
{
        List* l = (List*) malloc(sizeof(List));
        l->head = head;
        return l;
}
```

Dodavanje elementa u listu

```
List* push(List* l, int value)
{
        return create_list(create_element(
                value, l->head
        ));
}
```

Rep liste

```
List* tail(List* l)
{
        return create_list(l->head->next);
}
```

print funkcija za ispis liste - zbog debug-a

```
void print(List* l)
{
        if (l->head == NULL) printf("\n");
        else
        {
                printf("%d ", l->head->value);
                print(tail(l));
        }
}
```

main

```
int main() {
        List *l = create_list(NULL);
        push(l, 5);
        print(l);
        List* l2 = push(l, 5);
        print(l2);

        print(tail(l2));
        print(push(l2, 10));
        return 0;
}
```

- Pure functional
- Funkcije ne modifikuju parametre
- Funkcije UVEK vraćaju rezultat
- Funkcija može primiti funkciju kao parametar
- Funkcija može vratiti funkciju kao rezultat
- Kompajlira se ali se može i interpretirati interaktivno
- ghc i ghci
- ".hs" ekstenzija se obično koristi za fajlove sa kodom
- Strogo tipovan - svaka vrednost mora imati svoj tip i jasno se zna šta je kog tipa tokom kompajliranja

Primer funkcije koja računa kvadrat izlaza

```
sq :: Int -> Int
sq x = x*x
```

Primer upotrebe alata ghci

```
ghci example.hs
>>> sq 5
25
>>>
```

Primer funkcije koja računa zbir ulaznih parametara

```
add :: Int -> Int -> Int
add a b = a+b
```

- Funkcija u Haskell-u prima samo jedan parametar
- Funkcija ‘add‘ je funkcija koja prima jedan parametar i koja vraća funkciju koja prima drugi parametar i vraća zbir ta dva parametra
- inc = add 1 je funkcija koja vraća ulaz uvećan za jedan

- Jednostruko-povezane liste
- l = [1,2,3]
- head l
- tail l
- reverse l
- last l
- l = l1 ++ l2
- 5 : l
- range = [1..5]
- range = [1, 1.01..5]

```
squares = [x*x | x <- [1..10]]

squaresDiv3 = [x*x | x <- [1..10], x*x `mod` 3 == 0

unitcircle = [(x,y) | x <- [0, 0.01..10], y <- [0,0
```

Pattern matching

- Fukciju je moguće definisati specifično za neke ulazne parametre
- Neka vrsta zamene za "if" tako da kod izgleda lepo
- Može se koristiti za "otpakivanje" podataka u složenim tipovima kao što je lista

Faktorijel - primer pattern matching-a

```
fact :: Int -> Int
fact 0 = 1
fact n = n * (fact (n-1))
```

Funkcija koja vraća jedinicu ako je bar jedan ulaz jednak nuli

```
anyzero :: Int -> Int -> Int
anyzero 0 x = 1
anyzero x 0 = 1
anyzero x y = 0
```

Neke funkcije koje rade sa listama

```
mymap :: (a -> b) -> [a] -> [b]
mymap _ [] = []
mymap f (x:xs) = (f x) : (mymap f xs)


reduce :: (b -> a -> b) -> b -> [a] -> b
reduce _ z [] = z
reduce f z (x:xs) = reduce f (f z x) xs

myfilter :: (a -> Bool) -> [a] -> [a]
myfilter _ [] = []
myfilter f (x:xs) = if (f x) then x : (myfilter f x
        else myfilter f xs

*Main> mymap (+ 1) [1,2,3]
[2,3,4]
*Main> filter (\x -> x `mod` 2 == 0) [1,2,3,4,5,6]
```

Funkcija koja vraća listu gde dva ista elementa nisu jedan do drugog

```
norepeat :: (Eq a) => [a] -> [a]
norepeat [] = []
norepeat (x:[]) = [x]
norepeat (x:y:xs) = if x == y then norepeat (x:xs)

*Main> norepeat [1,1,2,5,5,6]
[1,2,5,6]
*Main> norepeat [1,1,2,5,5,6, 6]
[1,2,5,6]
*Main> norepeat [1,2,5,5,6, 6]
[1,2,5,6]
*Main> norepeat [1,2,5]
[1,2,5]
*Main>
```

Quick sort

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort l = (qsort [x | x<-l, x < y]) ++ [y] ++ (qsor
```

Razdvajanje stringa u listu reči

```
findFirstWord :: [Char] -> ([Char], [Char])
findFirstWord "" = ("", "")
findFirstWord " " = ("", "")
findFirstWord s =
    let (remaining, others) = findFirstWord $ tail
        x = head s
    in if x == ' ' then ("", tail s) else (x : rema

split :: [Char] -> [[Char]]
split "" = []
split s = let (x, y) = findFirstWord s
          in [x] ++ (split y)
```

Funkcija koja vraća dužinu stringa

```
strlen :: String -> Int
strlen s
    | s == "" = 0
        | otherwise = 1 + (strlen $ tail s)
```

Tipovi podataka

- Definišu se tako što se navede ime tipa i njegovi konstruktori
- data NoviTip = konstruktor1 — konstruktor2 — ...
- data Shape = Rectangle Int Int — Circle Int — Square Int

Primer funkcije nad tipom

```
data Shape = Circle Float | Rectangle Float Float
area :: Shape -> Float
area (Circle r) = r*r*3.14
area (Rectangle a b) = a*b
```

Primer - Zipper

```
data Zipper a = Zip [a] a [a] deriving (Show)

right :: Zipper a -> Zipper a
right (Zip l focus []) = Zip l focus []
right (Zip l focus r) = Zip (focus:l) (head r ) (ta


left :: Zipper a -> Zipper a
left (Zip [] focus r) = Zip [] focus r
left (Zip l focus r) = Zip (tail l) (head l) (focus
```

```
*Main> z = Zip [3, 2, 1] 4 [5, 6, 7]
*Main> right z
Zip [4,3,2,1] 5 [6,7]
*Main> left $ right z
Zip [3,2,1] 4 [5,6,7]
*Main> left $ left $ right z
Zip [2,1] 3 [4,5,6,7]
*Main> left $ left $ left $ right z
Zip [1] 2 [3,4,5,6,7]
*Main> left $ left $ left $ left $ right z
Zip [] 1 [2,3,4,5,6,7]
*Main> left $ left $ left $ left $ left $ right z
Zip [] 1 [2,3,4,5,6,7]
*Main>
```

Binarno stablo

```
data TreeNode = Node {
          left :: Node,
          right :: Node,
          value :: Int
      } | EmptyNode

tolist :: TreeNode -> [Int]
tolist EmptyNode = []
tolist n = let l = tolist $ left n
               r = tolist $ right n
               v = value n
           in l ++ [v] ++ r

append :: TreeNode -> Int -> Node
append EmptyNode x = Node EmptyNode EmptyNode x
append n x = let v = value n
                 r = right n
                 l = left n
```

Monade

- Način da se promeni stanje
- U realnom svetu, pojavljuje se potreba za moguč=ćnošću da se stanje može promeniti
- Primer: IO
- Monada uokviruje postojeći tip sa dodatnim informacijama koje se provlače kroz izračunavanje.
- Potrebno implementirati:
  - Tip podataka
  - return funkciju - a - ¿ m a
  - bind funkciju - m a -¿ (a -¿ m a) -¿ m a
- return ne utiče na izračunavanje
- bind vezuje dva izračunavanja - poput kompozicije

Primer * funkcije sa celim brojevima koje loguju operacije

```
data Debuggable = Debuggable (Int, [[Char]]) derivir


ret :: Int -> Debuggable
ret x = Debuggable (x, [])

bind :: Debuggable -> (Int -> Debuggable) -> Debugg
bind (Debuggable a b) f = Debuggable (y, z ++ b) wh

inc :: Int -> Debuggable
inc x = Debuggable (x+1, ["Number increased by one"
```

```
*Main> ret 5
Debuggable (5,[])
*Main> inc 5
Debuggable (6,["Number increased by one"])
*Main> bind (ret 5) inc
Debuggable (6,["Number increased by one"])
*Main> bind (bind (ret 5) inc) inc
Debuggable (7,["Number increased by one","Number in
*Main>
```

- Haskell ima sintaksni šećer za monade
- pomoću do-bloka je moguće pisati kod koji liči na kod u imperativnim jezicima
- Potrebno implementirati monade tako da ih Haskell prepozna kao takve

Implementacija monade tako da možemo koristiti Haskellov sintaksni šećer

```
import Control.Applicative -- Otherwise you can't d
import Control.Monad (liftM, ap)

data Debuggable a = Debuggable (a, [[Char]]) derivin
-- needed since newer versions of GHC
instance Functor Debuggable where
  fmap = liftM

instance Applicative Debuggable where
  pure  = return
  (<*>) = ap
-----------------------------------

instance Monad Debuggable where
  return x = Debuggable (x, [])
  (>>=) (Debuggable (a,b)) f = Debuggable (y, z ++
```

Primer upotrebe return i bind funkcija

```
*Main> (inc 5) >>= (\x -> (return x) >>= inc)
Debuggable (7,["Number increased by one","Number in
*Main>
```

do-blokovi

```
inc2 :: Int -> Debuggable Int
inc2 num = do
  x <- inc num
  y <- inc x
  return y
```

Unit monada

```
import Control.Applicative -- Otherwise you can't d
import Control.Monad (liftM, ap)

data Unit a = Unit a deriving (Show)

instance Functor Unit where
  fmap = liftM

instance Applicative Unit where
  pure  = return
  (<*>) = ap

instance Monad Unit where
  return x = Unit x
  (>>=) (Unit x) f = f x
```

Maybe monada

```
import Control.Applicative -- Otherwise you can't d
import Control.Monad (liftM, ap)

data Maybe a = Just a | Nothing

instance Functor Maybe where
  fmap = liftM

instance Applicative Maybe where
  pure  = return
  (<*>) = ap

instance Monad Maybe where
  return x = Just x
  (>>=) (Nothing) _ = Nothing
  (>>=) (Just x) f = f x
```

IO monada - hello world

```
main :: IO ()
main = putStrLn "Hello world!"

ghc hello.hs -o hello -dynamic
```

Unos podataka preko stadardnog ulaza

```
main :: IO ()
main = do
  putStrLn "Enter your name"
  name <- getLine
  putStrLn $ "Your name is " ++ name
```

WTF?????

- IO monada je hak
- IO moada je izlaz u realni svet
- IO monada čuva redni broj tako da funkcije se ne optimizuju od strane kompajlera već se uvek izvršavaju!

Naš printStrLn

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign.C.String
import Foreign.C.Types

foreign import ccall safe "prototypes.h my_print_li
    cMyPrintLine :: CString -> IO ()

myPrintLine :: String -> IO ()
myPrintLine str = do
    c_str <- newCString str
    cMyPrintLine c_str



main = do
    myPrintLine "hello!"
```

C kod

```c
#include <stdio.h>
#include "functions.h"


void my_print_line(char* str)
{
        printf("%s\n", str);
}
```

header

```
extern int my_rand();
extern void my_print_line(char*);
```

Implementacija rand funkcije

```
#include <stdlib.h>
#include <time.h>

int my_rand()
{
        time_t t;
        srand((unsigned) time(&t));
        return 1 + rand() % 5;
}
```

```
foreign import ccall safe "prototypes.h my_rand"
    cMyRand :: IO CInt

myRand :: IO Int
myRand = do
    r <- cMyRand
    return ((fromInteger . toInteger) r)


guessToStr :: Int -> String
guessToStr 1 = "1"
guessToStr 2 = "2"
guessToStr 3 = "3"
guessToStr 4 = "4"
guessToStr 5 = "5"
guessToStr 6 = "6"


main = do
    myPrintLine "hello!"
```

Zadaci

- JSON parser
- Parsiranje izraza (5+2, (3+2)*5, 5*2 - 1, ...)
- Web server sa REST API-em koji komunicira sa bazom