Uvod u funkcionalno programiranje

Stefan Nožinić (stefan@lugons.org)

April 8, 2023

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(ListElem)
  );
  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 izraza

```
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 <- [0,0.01..10], x*x + y*y < 1
  ]
```

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)

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

Upotreba

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

fold

- krenemo od funkcije sum pa ćemo da je generalizujemo

Obična suma brojeva

```
sum :: [Int] -> Int
sum [] = 0
sum [x:xs] = x + (sum xs)
```

Obična suma brojeva - ali sada imamo drugačiji neutralni element!

```
sum :: Int -> [Int] -> Int
sum z []     = z
sum z [x:xs] = x + (sum z xs )
```

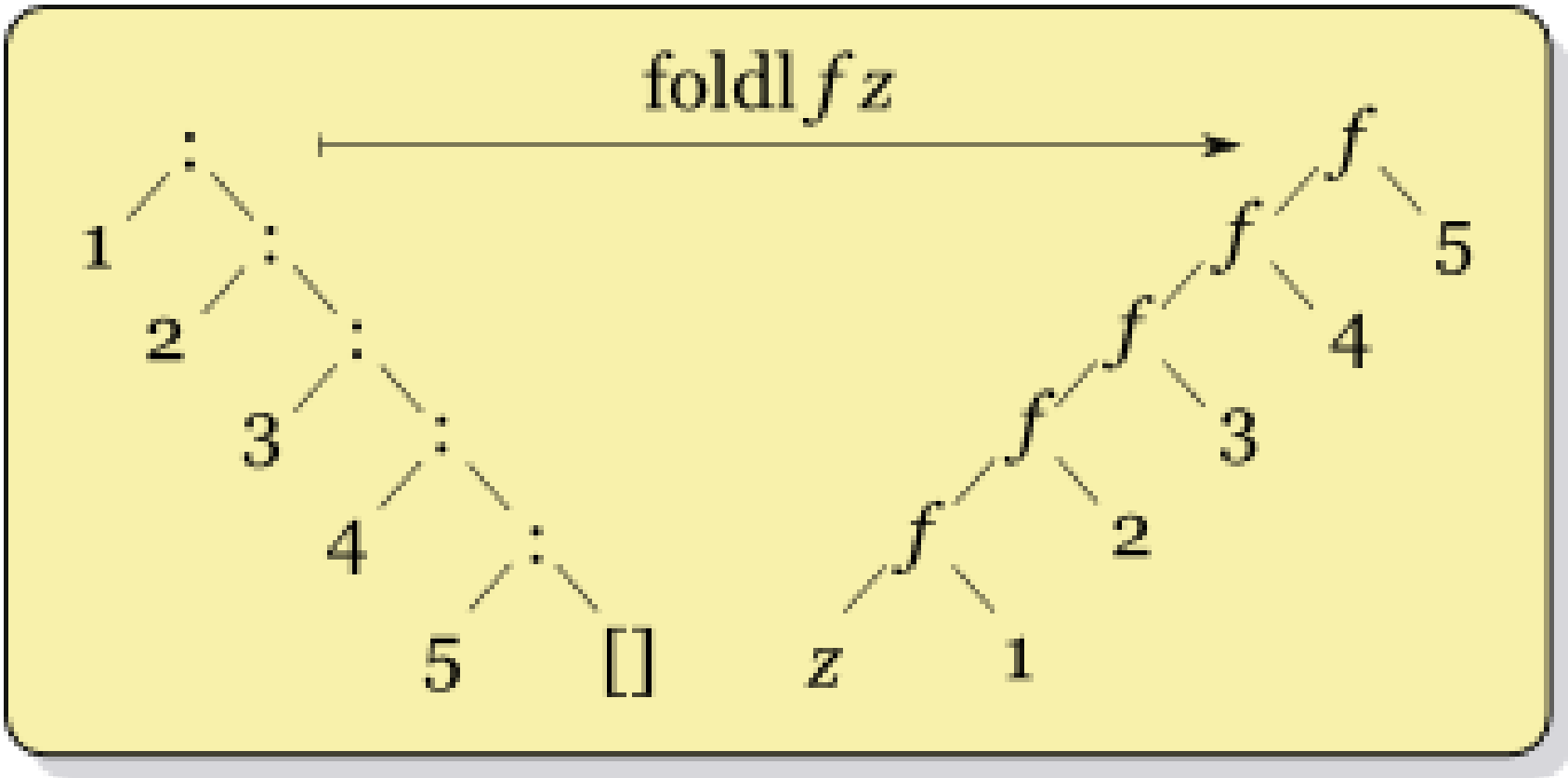
Zamenimo + sa generalnom funkcijom f

```
sum :: (Int -> Int -> Int) -> Int -> [Int] -> Int
sum f z []     = z
sum f z [x:xs] = f x  (sum f z xs )
```

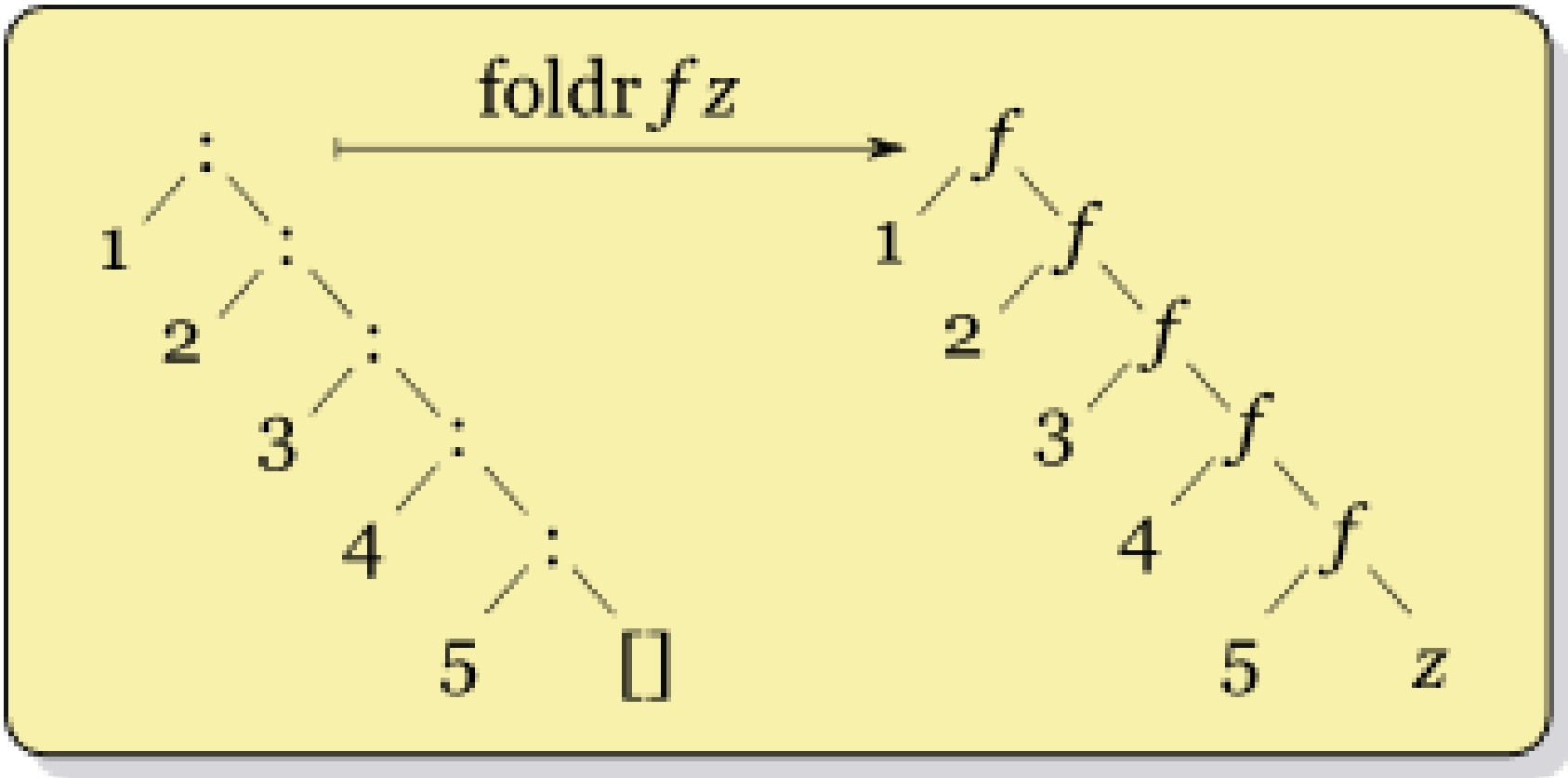
E al sad nam ne treba više Int, sad može šta hoćemo!

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

Ilustracija fold-a



Ilustracija desnog fold-a



Upotreba

```
*Main> reduce (+) 0 [1,2,3,4,5]
15
*Main>
```

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)
else x:(norepeat (y:xs))
```

Upotreba

```
*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]
  ++ (qsort [x | x<-l, x>y]) where y = (head l)
```

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
```

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
```

Binarno stablo

```
append :: TreeNode -> Int -> TreeNode
append EmptyNode x = Node EmptyNode EmptyNode x
append n x = let v = value n
  r = right n
  l = left n
  in if x < v
  then Node (append l x) r v
  else Node l (append r x) v

fromlist :: [Int] -> TreeNode -> TreeNode
fromlist [] n = n
fromlist l n = let x = head l
  in fromlist (tail l) (append n x)

bstsort :: [Int] -> [Int]
bstsort l = tolist (fromlist l EmptyNode)
```

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

Unit monada

```
data Unit a = Unit a deriving (Show)

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

Maybe monada

```
data Maybe a = Just a | Nothing

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
```

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!

State

```
data State s a = State (s -> (a,s))

instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f =
      State $ \s -> let (a, newState) = h s
                        (State g) = f a
                        in  g newState
```

Zaključak

- Deklarativni stil
- Ne mora Haskell - principi primenljivi na druge jezike!
- Manje bagova
- Lakše testirati softver