Demonstrature 2

Primjer 1. Navedimo nekoliko primjera rekurzivnih funkcija:

a) $sum : \mathbb{N}_0 \to \mathbb{N}_0,\ sum(n) = \begin{cases} 0, & n = 0 \\ n + sum(n-1), & n > 0 \end{cases}$

b) $fact : \mathbb{N}_0 \to \mathbb{N},\ fact(n) = \begin{cases} 1, & n = 0 \\ n \cdot fact(n-1), & n > 0 \end{cases}$

c) $fibo : \mathbb{N}_0 \to \mathbb{N}_0,\ fibo(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fibo(n-1) + fibo(n-2), & n \geq 2 \end{cases}$

d) $gcd : \mathbb{N} \times \mathbb{N}_0 \to \mathbb{N},\ gcd(a,b) = \begin{cases} a, & b = 0 \\ gcd(b,\ a \bmod b), & b > 0 \end{cases}$

Svaka od ovih funkcija u svom tijelu ima **najmanje dva** različita slučaja. Razlog tome je što svaka rekurzivna funkcija mora imati nekakav zaustavni uvjet (zaustavni kriterij) pri kojem se više ne vrše rekurzivni pozivi. U suprotnom bi rekurzija pozivala samu sebe beskonačno mnogo puta i nikad ne bi vratila rezultat.

Primjer 2. Pogledajmo kako implementirati gore navedene funkcije u Haskellu:

```
-- a)
sum :: Int -> Int
sum 0 = 0
sum n = n + sum (n-1)

-- b)
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)

-- c)
-- zadatak za vježbu :)

-- d)
gcd :: Int -> Int -> Int
gcd a 0 = a
gcd a b = gcd b (mod a b)
```

Pisanje različitih uvjeta na ovaj način zove se ***pattern matching***. Funkcija primi argument, provjeri može li ga zapisati na način dan u prvoj liniji te ako može, računa rezultat funkcije onako kako je definirano iza znaka jednakosti. Ukoliko ne može, ide na sljedeću liniju i ponavlja postupak sve dok ne dođe do odgovarajućeg *patterna*.

Napomena 1. Primijetimo da smo u potpisu funkcije *fact* koristili `Integer` umjesto `Int`. Razlog tome je što ova funkcija raste jako brzo te će prerasti maksimalnu vrijednost koja stane u `Int` već za n koji je blizu 20. Ukoliko baš moramo poslati argument tipa `Int`, možemo definirati pomoćnu funkciju koja će služiti samo za pretvorbu argumenta u `Integer`:

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

fact :: Int -> Integer
fact n = fact' $ toInteger n
```

Primjer 3. Osim što funkcije u Haskellu mogu raditi s brojevima i slovima, također mogu primati i složenije objekte kao argumente. Prvi primjer takvog objekta je lista:

```
[a] = [] | a : [a]
```

Ova definicija znači da lista tipa `a` može biti zadana kao prazna lista ili kao podatak tipa `a` "naljepljen" na drugu listu tipa `a`. Ta lista je tada ili prazna lista ili je također zadana kao podatak tipa `a` naljepljen na neku treću listu tipa `a` itd. Pogledajmo neke primjere listi u Haskellu:

```
brojevi :: [Int]
brojevi = [1, 5, 2, -7]

v :: [Float]
v = [1, 2.5, -3.1]

slova :: [Char]
slova = ['M', 'a', 't', 'h', 'O', 'S']

rijec :: String
rijec = "MathOS"
```

Zapišimo listu `brojevi` onako kako je definirana, tj. onako kako ju Haskell vidi:

```
brojevi :: [Int]
brojevi = 1 : (5 : (2 : ((-7) : [])))
```

Slično možemo napraviti i za ostale navedene liste, no ne možemo niti za jednu beskonačnu listu.

Nadalje, konačne liste možemo i uspoređivati, ali samo ako su istog tipa. Pokušamo li usporediti liste `brojevi` i `v`, dobit ćemo pogrešku. Što će se dogoditi ako pokušamo usporediti listu `slova` i string `rijec`?

```
print $ rijec == slova
>> True
```

Razlog zašto je ovo uopće moguće je jednostavan: u Haskellu, lista tipa `Char` je isto što i podatak tipa `String`. Prije pokretanja cijelog koda, Haskell iz optimizacijskih razloga pretvara sve liste tipa `Char` u tip `String`, no tijekom izvođenja programa ih i dalje tretira kao liste kako bi mogao pristupiti svakom znaku zasebno.

Long story short: možete pisati `[Char]` umjesto `String` ili obrnuto - Haskellu je apsolutno svejedno.

Primjer 4. Prisjetimo se pojma kojeg smo uveli u Primjeru 2 - pattern matching. Kada definiramo funkcije koje primaju liste kao argument, najčešće ćemo koristiti sljedeći pattern matching:

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

len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

Ova dva patterna odgovaraju upravo onima iz definicije liste tipa `a`: lista je zadana ili kao prazna lista ili kao njen prvi element nalijepljen na sve ostale. Prvi element liste često zovemo ***head***, a ostale elemente ***tail***. Slično kao i u primjeru s prijašnjih demonstratura, sve argumente koje funkcija prima, ali ih ne koristimo u tijelu funkcije, možemo jednostavno zapisati kao *underscore*:

```
len :: [a] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

Primjer 5. Do sada smo vidjeli liste tipa `Int`, `Float` i `Char`. Mogu li liste kao tip imati listu? Odgovor je ***da***:

```
m :: [[Int]]
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

-- String je isto što i [Char]
rijeci :: [String]
rijeci = ["Odjel", "za", "matematiku"]
```

Matrice su u Haskellu često reprezentirane kao lista listi nekog numeričkog tipa (`Int` ili `Float`). Međutim, ako imamo listu koja kao elemente ima liste nekog tipa `a`, te liste ne moraju biti iste duljine, što možemo vidjeti i u listi `rijeci`. Imajte ovo na umu kada budete radili s ovakvim listama.