# FUNKCIJSKO PROGRAMIRANJE

Komprehenzija listi

# Komprehenzija skupova

U matematici, oznakom za izdvajanje može se konstruirati novi skup iz starog

$$\{x^2 \mid x \in \{1...5\}\}$$

skup {1,4,9,16,25} je skup svih $x^2$ takvih da je x element skupa {1...5}.

# Komprehenzija listi

U Haskell-u postoji sličan način komprehenzije kojim se mogu stvoriti nove liste iz starih

```
[x^2 | x ← [1..5]]
```

Lista $[1,4,9,16,25]$ je lista svih $x^2$ takvih da je $x$ element liste $[1..5]$.

# Primjedba:

- Izraz $x \leftarrow [1..5]$ naziva se generator jer određuje kako će se generirati vrijednosti za $x$.

- Komprehenzija može koristiti više generatora koji se odvajaju zarezom. Primjerice:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

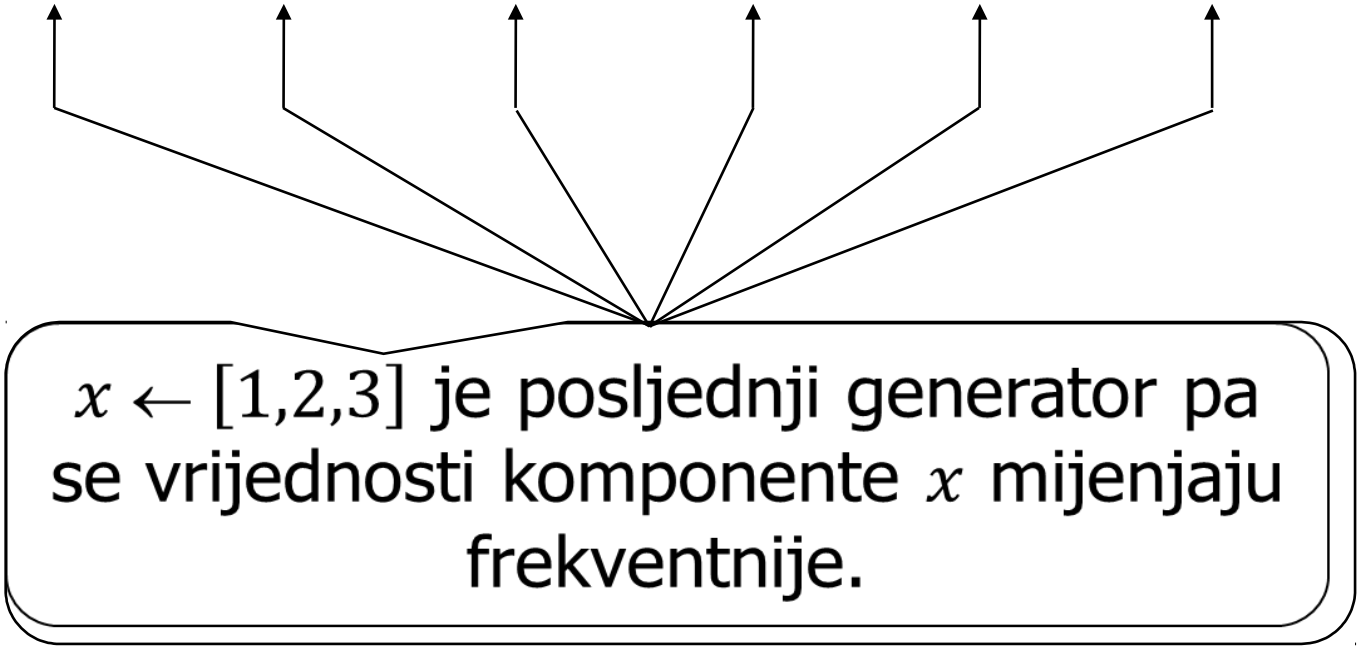
- Promjena redoslijeda generatora mijenja redoslijed elemenata u konačnoj listi:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- U slučaju više generatora imamo sličnu situaciju kao kod ugniježđenih petlji. Kod kasnije navedenih generatora, kao i kod dublje ugniježđenih listi, vrijednosti se učestalije mijenjaju

- Primjerice:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



# Zavisni generatori

Generatori koji su kasnije navedeni mogu zavisiti  o varijablama koje su uvedene u prije navedenim generatorima.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

Lista $[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]$
Sadrži sve parove $(x, y)$ takve da su $x, y$ elementi liste $[1..3]$ i $y \geq x$.

Koristeći zavisni generator možemo definirati funkciju koja konkatenira (spaja) listu sa drugom listom

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

Primjerice:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

# Čuvari

**Komprehenzija listi** može koristiti čuvare (eng. guards) koji će restringirati skup vrijednosti proizveden pomoću prethodnih generatora

```
[x | x ← [1..10], even x]
```

Lista [2,4,6,8,10] je lista svih brojeva x takvih da je x element liste [1..10] i x je paran.

**Primjer:** Koristeći čuvare možemo definirati funkciju koja će pozitivnom cijelom broju pridružiti listu njegovih djelitelja:

```
factors :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

Primjerice:

```
> factors 15

[1,3,5,15]
```

Koristeći listu djelitelja možemo jednostavno utvrditi je li broj prost:

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

Primjerice:

```
> prime 15
False

> prime 7
True
```

Koristeći čuvare možemo definirati i funkciju koja će vratiti listu svih prostih brojeva do zadanog broja n:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

Primjerice:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

# Zip funkcija

Zip je funkcija koja dvjema listama pridružuje listu parova u kojima je prva komponenta element prve liste, a druga komponenta odgovarajući element druge liste.

```
zip :: [a] → [b] → [(a,b)]
```

Primjerice:

```
> zip [’a’,’b’,’c’] [1,2,3,4]

[(’a’,1),(’b’,2),(’c’,3)]
```

Koristeći zip funkciju moguće je definirati funkciju koja vraća listu svih parova susjednih elemenata liste:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

Primjerice:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Koristeći *pairs* funkciju možemo definirati funkciju koja provjerava je li lista sortirana:

```
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

Primjerice:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Koristeći *zip* funkciju moguće je definirati funkciju positions koja vraća listu svih pozicija zadane vrijednosti:

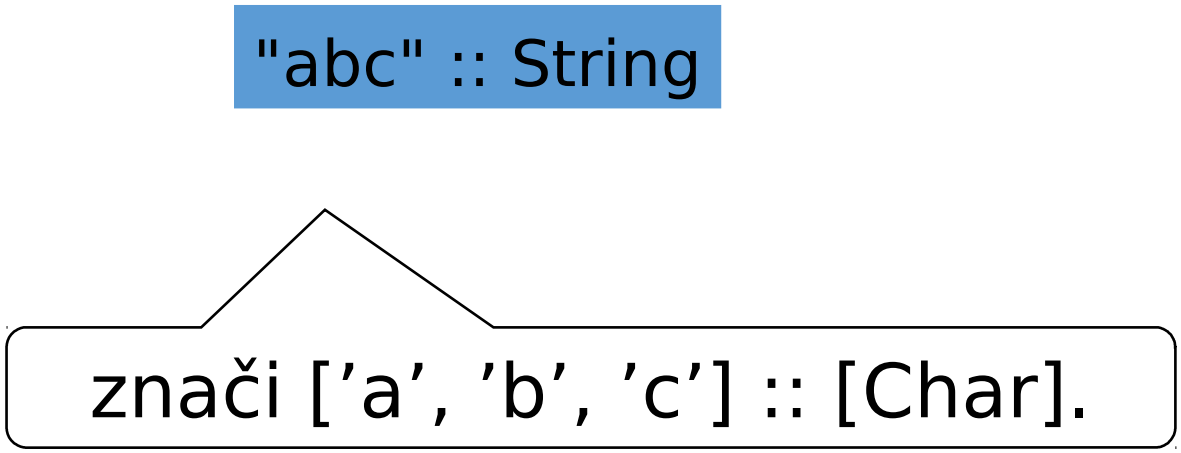
```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..], x == x']
```

Primjerice:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

# Komprehenzija stringova

Iako smo **string** zadavali kao niz znakova unutar dvostrukih navodnika, interno su stringovi reprezentirani kao liste znakova



Kako je string (znakovni niz) specijalna vrsta liste, bilo koja polimorfna funkcija koja se izvršava nad listama može se primijeniti i na stringove. Primjerice:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Slično, komprehenzija listi može se koristiti i u definiranju funkcija na stringovima, kao što je primjerice funkcija koja vraća broj pojavljivanja znakova:

```
count :: Char → String → Int
count x xs = length [x’ | x’ ← xs, x == x’]
```

Primjerice:

```
> count ‘a’ "Odjel za matematiku"
3
```

# Vježbe

**Zadatak 1.** Za uređenu trojkupozitivnih cijelih brojeva kažemo da je Pitagorina trojka ako vrijedi .  Koristeći komprehenziju listi definiramo funkciju:

```
pitagora:: Int → [(Int,Int,Int)]
```

koja cijelom broju $n$ pridružuje sve Pitagorine trojke u skupu $[1..n]$. Primjerice:

```
> Pitagora 5
[(3,4,5),(4,3,5)]
```

**Zadatak 2.** Pozitivan cijeli broj je **savršen** ukoliko je jednak sumi svih svojih djelitelja, isključujući samog sebe. Koristeći komprehenziju listi, definirajte funkciju:

```
savrsen:: Int → [Int]
```

Koja vraća listu svih savršenih brojeva do zadanog broja $n$. Primjerice:

```
> savrsen 500

[6,28,496]
```

**Zadatak 3.** Skalarni produkt dvije liste cijelih brojeva duljine *n* definiramo kao sumu produkta odgovarajućih komponenti:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Koristeći komprehenziju listi definirajte funkciju koja će vratiti skalarni produkt dvije liste.