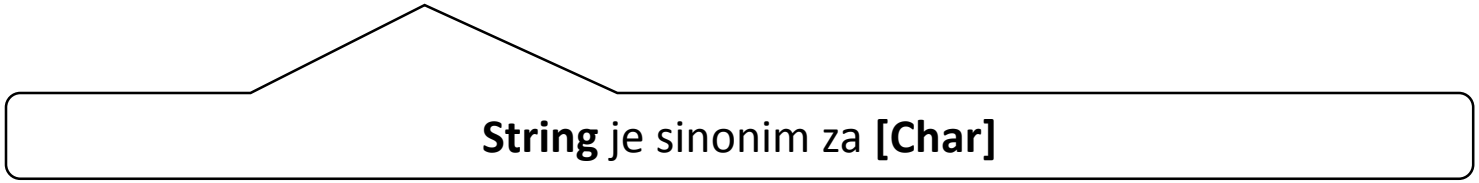
FUNKCIJSKO PROGRAMIRANJE

Deklariranje tipova

Deklariranje tipova

U Haskellu se može koristiti **type declaration** za definiranje postojećeg tipa

```
type String = [Char]
```



Deklariranje tipova se također može koristiti kako bi drugi tipovi bili pregledniji ili se lakše čitali. Primjerice:

```
type Pos = (Int,Int)
```

Možemo definirati:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Kao i kod definiranja funkcija, tipovi imaju parametre. Primjerice:

```
type Pair a = (a,a)
```

Možemo definirati:

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

Tipovi se mogu ugnijezditi:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

Tipovi **ne mogu** biti rekurzivni:

```
type Tree = (Int,[Tree])
```

Deklaracija podataka

U potpunosti nov tip može se definirati navodeći njegove vrijednosti:

```
data Bool = False | True
```

Bool je novi tip koji može imati vrijednosti *False* i *True*

Primjedbe:

- Dvije vrijednosti False i True zovu se konstruktori tipa Bool.
- Ime tipa i konstruktora uvijek mora početi velikim slovom.

Vrijednosti novih tipova mogu se koristiti na isti način kao i vrijednosti ugrađenih tipova. Primjerice, za

```
data Answer = Yes | No | Unknown
```

možemo definirati:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

Konstruktori u deklaraciji podataka također imaju parametre. Primjerice, za:

```
data Shape = Circle Float
           | Rect Float Float
```

možemo definirati:

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Primjedba:

- *Shape* ima vrijednosti *Circle r* gdje je *r* tipa *float*, i *Rect x y* gdje su *x* i *y* tipa *float*.
- *Circle* i *Rect* možemo promatrati kao funkcije koje konstruiraju vrijednosti tipa *Shape*:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```

Deklaracija podataka također može imati parametre. Primjerice za:

```
data Maybe a = Nothing | Just a
```

možemo definirati:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Rekurzivni tipovi

U Haskellu, **novi tipovi** mogu se deklarirati „u terminima njih samih”. Drugim riječima, novi tip može biti rekurzivan:

```
data Nat = Zero | Succ Nat
```

Nat je novi tip, s konstruktorima Zero :: Nat i Succ :: Nat → Nat.

Primjedba:

- Vrijednost tipa *Nat* je ili *Zero*, ili je oblika *Succ n* gdje je *n :: Nat*.
- Odnosno, *Nat* sadrži sljedeći beskonačan niz vrijednosti:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- Možemo promatrati vrijednosti tipa *Nat* kao prirodne brojeve, gdje *Zero* predstavlja *0*, i *Succ* predstavlja funkciju sljedbenika *1+*.

- Primjerice, vrijednost

```
Succ (Succ (Succ Zero))
```

Predstavlja prirodan broj

```
1 + (1 + (1 + 0))   =   3
```

Koristeći rekurzije, jednostavno se mogu definirati funkcije koje konvertiraju vrijednosti tipova *Nat* i *Int*:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Dva prirodna broja mogu se zbrajati tako da se najprije pretvore u cijele brojeve, zbroje, a zatim rezultat pretvori u prirodan broj:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

Jednostavnije je koristiti rekurziju kako bismo izbjegli potrebu pretvaranja:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Primjerice:

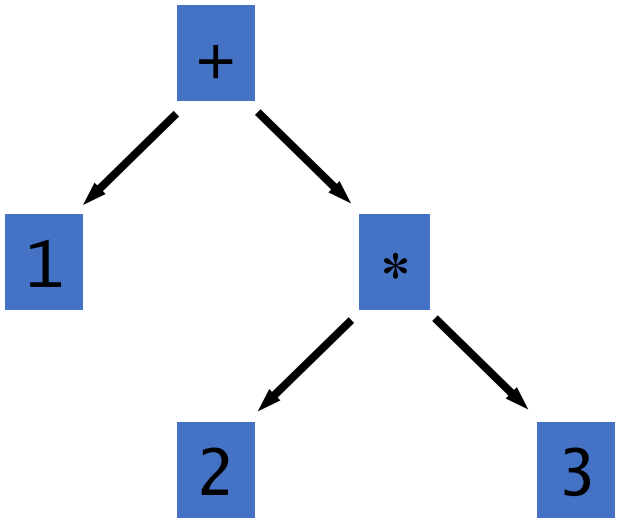
```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

Primjedba:

- Rekurzivna definicija za add odgovara pravilu: $0 + n = n$ i $(1 + m) + n = 1 + (m + n)$.

Aritmetički izrazi

Koristite jednostavnu formu aritmetičkih izraza koji se sastoje od cijelih brojeva te operacija zbrajanja i množenja.



```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

Primjerice, izraz s prethodnog slajda može se prikazati na sljedeći način:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Koristeći rekurzije možemo jednostavno definirati funkcije koje će evaluirati izraz:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Primjedba:

- Prethodna tri konstruktora su sljedećeg tipa:

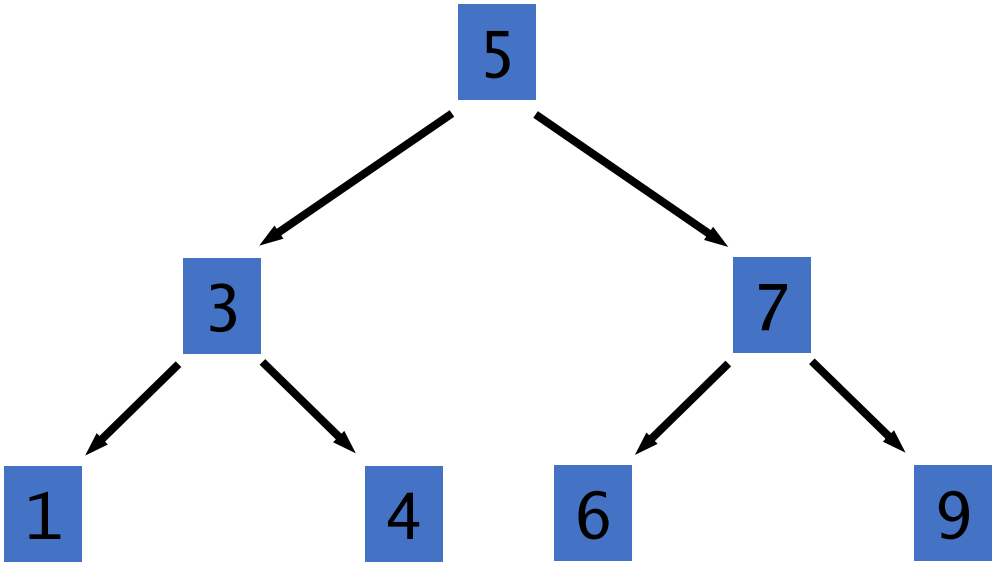
```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- Funkcije na izrazima mogu se definirati zamjenjujući konstruktore drugim funkcijama koristeći odgovarajuću *folde* funkciju. Primjerice:

```
eval = folde id (+) (*)
```

Binarna stabla

U računarstvu se često podaci spremaju u strukturu podataka binarno stablo.



Koristeći rekurzivni pristup kod definiranja novih tipova, binarno stablo se može definirati na sljedeći način:

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)
```

Primjerice, stablo s prethodnog slajda može se reprezentirati na sljedeći način:

```
t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
         (Node (Leaf 6) 7 (Leaf 9))
```

Sada ćemo definirati funkciju koja ispituje nalazi li se dana vrijednost u binarnom stablu:

```
occurs :: Ord a ⇒ a → Tree a → Bool
occurs x (Leaf y)      = x == y
occurs x (Node l y r) = x == y
                        || occurs x l
                        || occurs x r
```

U najgorem slučaju, ako se vrijednost na nalazi u stablu, ova funkcija obilazi **cijelo** stablo.

Razmotrimo funkciju **flatten** koja vraća listu vrijednosti sadržanih u stablu:

```
flatten :: Tree a → [a]
flatten (Leaf x)     = [x]
flatten (Node l x r) = flatten l
                       ++ [x]
                       ++ flatten r
```

(Binarno) stablo zovemo (binarno) stablo pretraživanja ukoliko funkcija flatten vraća sortiranu listu vrijednosti. Primjer našeg stabla je binarno stablo pretraživanja jer funkcija flatten vraća [1,3,4,5,6,7,9].

Binarno stablo pretraživanja ima vrlo važno svojstvo na osnovu kojeg uvijek možemo odlučiti u kojem od dva podstabla se nalazi tražena vrijednost:

```
occurs x (Leaf y)                  = x == y
occurs x (Node l y r) | x == y = True
                      | x < y  = occurs x l
                      | x > y  = occurs x r
```

Je li ovakav pristup efikasniji?

Uvijek nastavljamo u jedno od podstabala pa pretraživanje slijedi put od korijena do listova.

Zadaci za vježbu

(1) Koristeći rekurziju i funkciju *add*, definirajte funkciju koja *množi* dva prirodna broja.

(2) Binarno stablo je potpuno ukoliko su za bilo koja dva vrha veličine podstabala, ukorijenjenih u tim vrhovima, jednake. Definirajte funkciju koja će ispitati je li binarno stablo potpuno.