FUNKCIJSKO PROGRAMIRANJE

Definiranje funkcija

Uvjetni izrazi

Kao i u većini ostalih programskih jezik, funkcije se mogu definirati pomoću uvjetnih izraza

```
apsolutno :: Int → Int
apsolutno n = if n ≥ 0 then n else -n
```

Apsolutno prima cijeli broj n i vraća:

- n, ako je n nenegativan
- $-n$, ako je n negativan

Uvjetni izrazi se mogu ugnijezditi

```
predznak :: Int → Int
predznak n = if n < 0 then -1 else
             if n == 0 then 0 else 1
```

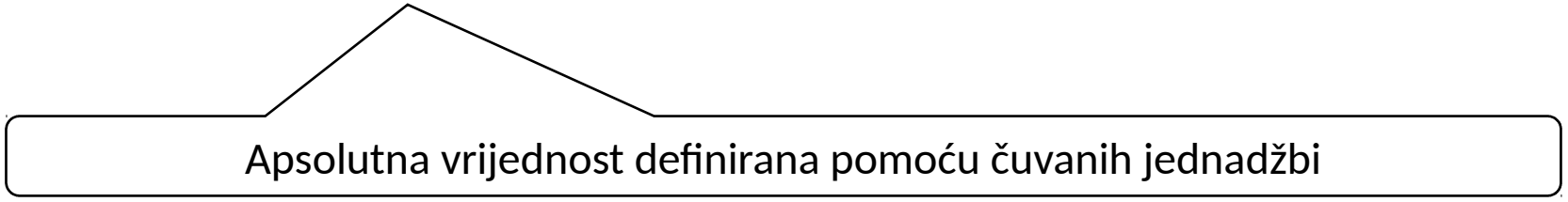
Primjedba:

- Za razliku od nekih drugih programskih jezika, u Haskellu mora uvijek postojati **else** grana
- Na taj način se izbjegava moguća dvosmislenost

Čuvane jednadžbe

- kao alternativa uvjetnim izrazima, mogu se koristiti čuvane jednadžbe (eng. guarded equations)
- Čuvari su logički izrazi na osnovu čije istinitosti se bira između rezultata istog tipa

```
apsolutno ::     Int → Int
apsolutno n| n ≥ 0     = n
           | otherwise = -n
```



Čuvane jednadžbe se koriste radi bolje čitljivost kod definiranja funkcija koje bi imali puno uvjetnih izraza

```
predznak n| n < 0     = -1
          | n == 0    = 0
          | otherwise = 1
```

Primjedba:

- Nije nužno završiti definiciju funkcije s **otherwise**, ali na taj način izbjegavamo grešku koja bi se dogodila u slučaju da niti jedan od čuvara nije istinit

Prepoznavanje obrazaca

- Mnoge funkcije se mogu jasno definirati koristeći **prepoznavanje obrazaca** (eng. pattern matching) nad njihovim argumentima

```
negiraj :: Bool → Bool
negiraj False = True
negiraj True  = False
```

Funkcija *negiraj* vrijednosti **False** pridružuje **True**, a vrijednosti **True** pridružuje **False**.

Funkcije se mogu definirati na različite načina koristeći prepoznavanje obrazaca.
Primjerice:

```
(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

može se definirati na **kompaktniji** način

```
True && True = True
_    && _    = False
```

U slučaju veznika **&&** nije potrebno evaluirati drugi argument ukoliko je prvi *False*. Slijedi efikasnija definicija:

```
True  && b = b
False && _ = False
```

Primjer:

- Donja povlaka _ je rezervirani simbol _ za obrazac u kojem će odgovarati bilo kojoj vrijednosti

- Obrasci se prepoznaju redom. Primjerice, sljedeća definicija veznika && uvijek vraća *False:*

```
_    && _    = False
True && True = True
```

- U obrascima se ne smiju ponavljati varijable više puta. Sljedeća definicija izazvat će grešku iako je logički korektna:

```
b && b = b
_ && _ = False
```

Obrasci za liste

Svaka neprazna lista konstruirana je uzastopnom primjenom „cons” operatora (:) koji dodaje elemente na početak liste.

[1,2,3,4]

Nastaje na sljedeći način 1:(2:(3:(4:[]))).

Funkcije nad listama mogu se definirati koristeći obrazac x:xs.

```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

head – pridružuje nepraznoj listi njezin prvi element
tail – pridružuje nepraznoj listi novi listu koja sadrži sve elemente osim prvog

Primjedba:

- x:xs obrazac prepoznaje se samo na nepraznim listama:

```
> head []
*** Exception: empty list
```

- x:xs obrazac mora se staviti u zagrade jer primjena funkcije ima veći prioritet od cons operatora (:). Primjerice, ovakva definicija prouzročit će grešku:

```
head x:_ = x
```

Primjeri:

- Funkcija *test* vraća *True* ukoliko je prvi znak u listi ‘a’, a inače *False*

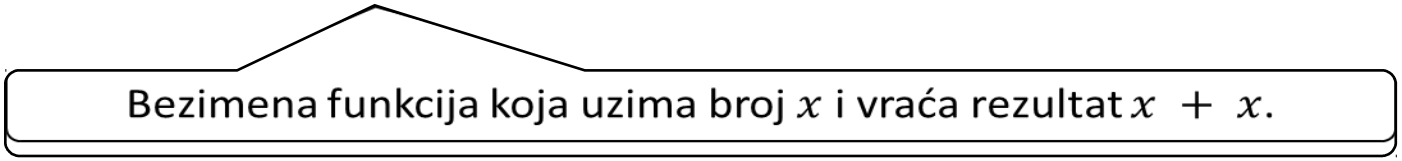
```
test :: [Char] -> Bool
test ['a', _, _] = True
test _ = False

test :: [Char] -> Bool
test ('a':_) = True
test _ = False
```

Lambda izrazi

Funkcije se mogu konstruirati i bez davanja imena funkciji koristeći lambda izraze.

$$\lambda x \rightarrow x + x$$



Primjedbe:

- U lambda izrazima koristimo grčko slovo λ (lambda), koje u kodu pišemo kao znak \ (eng. backslash)
- U matematici se bezimene funkcije pišu pomoću simbola $\mapsto$, kao primjerice $x \mapsto x + x$.
- U Haskellu se korištenje λ simbola za bezimene funkcije bazira na lambda računu, odnosno teoriji funkcija na kojoj je nastala paradigma funkcijskog programiranja.

Zbog čega su lambda izrazi korisni?

Lambda izrazi mogu poslužiti za formalno razumijevanje kaskadnih funkcija

Primjerice:

```
dodaj x y = x + y
```

znači

```
dodaj = λx → (λy → x + y)
```

Lambda izrazi su korisni prilikom definiranja funkcija koje vraćaju funkcije kao rezultat

Primjerice, funkciju

```
const :: a → b → a
const x _ = x
```

prirodnije je definirati kao:

```
const :: a → (b → a)
const x = λ_ → x
```

Lambda izrazi mogu se koristiti prilikom definiranja funckija koje se referenciraju samo jednom:

Primjerice, funkcija:

```
neparni n = map f [0..n-1]
        where
          f x = x*2 + 1
```

može se jednostavnije definirati kao:

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Sekcije

Operator koji se primjenjuje na dva argumenta (binarni operator) može se konvertirati u kaskadnu funkciju koja se u zagradama piše ispred svoja dva argumenta

Primjerice:

```
> 1+2
3

> (+) 1 2
3
```

Ovo pravilo također dozvoljava da se jedan od argumenata operatora stavlja u zagradu.

Primjerice:

```
> (1+) 2
3

> (+2) 1
3
```

Općenito, ako je ⊕ operator, onda funkcije oblika (⊕), (x⊕) and (⊕y) zovemo sekcije.

Why Are Sections Useful?

Korisne funkcije mogu se definirati koristeći sekcije. Primjerice:

(1+) - funkcija sljedbenika

(1/) - funkcija koja vraća recipročnu vrijednost argumenta

(*2) - funkcija koja udvostručuje vrijednost argumenta

(/2) - funkcija koja vraća dva puta manju vrijednost argumenta

Zadaci za vježbu

Zadatak 1. Koristeći funkcije iz biblioteke, definirajte funkciju halve :: [a] -> ([a], [a]) koja razdvaja listu parne duljine u dvije liste jednake duljine. Primjerice:

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

Zadatak 2. Neka je zadana funkcija safetail koja se ponaša na isti način kao i tail, osim što praznoj listi pridružuje praznu listu. Definirajte safetail koristeći:

(a) uvjetne izraze,
(b) čuvane jednadžbe,
(c) prepoznavanje obrazaca.

Napomena: funkcija null:: [a] ⟶ Bool može se koristiti za provjeru je li lista prazna

Zadatak 3. Napišite tri različite definicije logičkog operatora || koristeći prepoznavanje obrazaca.

Zadatak 4. Redefinirajte sljedeću verziju operatora && koristeći uvjetne izraze umjesto prepoznavanja obrazaca.

```
True && True = True
_    && _    = False
```

Zadatak 5. Učinite isto kao u prethodnom zadatku za sljedeću verziju:

```
True  && b = b
False && _ = False
```

Zadatak 6. Pokažite kako se sljedeća kaskadna funkcija može bolje razumijeti ukoliko se definira pomoću lambda izraza.