FUNKCIONALNO PROGRAMIRANJE

Stefan Mišković
Matematički fakultet, Beograd

Sadržaj

1 Uvod u Haskell

1.1 Pojam funkcije

U matematici, funkcija predstavlja pravilo koje svakom elementu jednog skupa pridružuje tačno jedan element drugog skupa. Ako funkcija f preslikava skup A u skup B, to zapisujemo kao $f : A \to B$. Skup A naziva se domen funkcije, a skup B kodomen. Ako funkcija f elementu $x \in A$ pridružuje vrednost $y \in B$, to zapisujemo kao $f(x) = y$.

U Haskellu, funkcija je preslikavanje koje prima jedan ili više argumenata i vraća tačno jednu vrednost. Funkcija se definiše jednačinom koja se sastoji od imena funkcije, imena njenih argumenata i izraza kojim se određuje rezultat u zavisnosti od tih argumenata.

Na primer, funkcija `double` koja za argument `x` izračunava vrednost `x + x` se može definisati na sledeći način:

```
double x = x + x
```

Kada se funkcija primeni na konkretnu vrednost, rezultat se dobija zamenom argumenta tom vrednošću i daljim izračunavanjem izraza. Na primer:

```
double 2
= 2 + 2
= 4
```

Na sličan način se izračunava i izraz sa ugnježdenom primenom funkcije:

```
double (double 2)
= double (2 + 2)
= double 4
= 4 + 4
= 8
```

Na ovaj način, izračunavanje u Haskellu se svodi na uzastopno pojednostavljivanje izraza dok se ne dobije konačna vrednost.

1.2 Osnove funkcionalnog programiranja

Funkcionalno programiranje je stil programiranja u kome se izračunavanje vrši primenom funkcija na argumente, a ne kao niz naredbi koje menjaju stanje memorije.

U imperativnim jezicima, kao što je C, program se sastoji od naredbi koje se izvršavaju redom i pri tome menjaju vrednosti promenljivih. Na primer, zbir brojeva od 1 do n može se izračunati korišćenjem promenljive koja se postepeno menja:

```
int sum = 0;
for (int i = 1; i <= n; i++) {
    sum = sum + i;
}
```

U ovom programu izračunavanje se zasniva na promeni stanja. Promenljiva `sum` se više puta menja, a rezultat se dobija kao poslednja dodeljena vrednost.

U funkcionalnom programiranju, isti problem se rešava drugačije – bez menjanja stanja. Umesto toga, rezultat se definiše kao vrednost izraza:

```
sum [1..n]
```

Ovde se ne opisuje postupak izračunavanja korak po korak, već se direktno zadaje izraz čija je vrednost traženi rezultat. Izračunavanje ovog izraza može se posmatrati kao postupno pojednostavljivanje. Na primer:

```
sum [1..5]
= sum [1,2,3,4,5]
= 1 + 2 + 3 + 4 + 5
= 15
```

Dakle, u funkcionalnom programiranju izračunavanje se zasniva na primeni funkcija, a ne na promeni stanja memorije. U Haskellu imena ne predstavljaju promenljive čija se vrednost menja, već vezivanja za određene vrednosti. Jednom dodeljena vrednost ostaje ista tokom celog izračunavanja.

1.3 Kratak istorijski pregled

Koreni funkcionalnog programiranja nalaze se u λ-računu, formalnom modelu računanja koji je razvijen tridesetih godina XX veka. λ-račun predstavlja matematički sistem za definisanje i primenu funkcija i smatra se teorijskom osnovom funkcionalnog programiranja.

Tokom narednih decenija razvijeni su brojni programski jezici koji su uvodili i unapređivali ideje funkcionalnog pristupa. Jedan od prvih takvih jezika bio je Lisp, koji se uglavnom smatra prvim funkcionalnim jezikom. Kasnije su razvijeni jezici poput ISWIM, koji je bio bliži matematičkom modelu funkcija, zatim FP, koji je implementirao ideje o funkcijama višeg reda, kao i ML, gde se uvode polimorfni tipovi i automatsko određivanje tipova. U ovom periodu razvijani su i jezici sa lenjom evaluacijom, među kojima se posebno ističe Miranda.

Krajem osamdesetih godina započet je razvoj Haskella sa ciljem objedinjavanja postojećih ideja funkcionalnog programiranja u jedinstven programski jezik. Njegova prva stabilna verzija definisana je 2003. godine, čime je postavljen standard za dalji razvoj.

1.4 Rad u interaktivnom okruženju

Za rad u programskom jeziku Haskell danas se koristi Glasgow Haskell Compiler (GHC). Pored kompajlera `ghc`, koji Haskell kôd prevodi u izvršne datoteke, GHC sistem sadrži i interaktivno okruženje `ghci`, namenjeno neposrednom radu sa izrazima i definicijama.

Za instalaciju Haskell okruženja najčešće se koristi alat GHCup[1], koji omogućava jednostavno instaliranje GHC sistema i pratećih alata. U ovoj fazi kursa koristićemo pre svega okruženje `ghci`, jer je ono najpogodnije za upoznavanje jezika.

Interaktivno okruženje pokreće se iz terminala komandom:

```
ghci
```

Nakon pokretanja pojavljuje se prompt oblika `ghci>`, koji označava da sistem čeka unos izraza. Izlazak iz okruženja vrši se komandom `:quit` ili `:q`.

U `ghci` okruženju mogu se direktno izračunavati aritmetički izrazi:

```
ghci> 2 + 3
```

[1] https://www.haskell.org/ghcup/

```
5

ghci> 5 * 10
50

ghci> 10 - 4
6

ghci> 5 / 2
2.5
```

Moguće je kombinovati više operatora u jednom izrazu:

```
ghci> 2 + 3 * 4
14

ghci> 2 * 3 ^ 4
162
```

Kao i u matematici, operatori imaju različite prioritete. Stepenovanje ima viši prioritet od množenja i deljenja, a oni imaju viši prioritet od sabiranja i oduzimanja. Na primer, izraz `2 * 3 ^ 4` tumači se kao `2 * (3 ^ 4)`.

Takođe, operatori imaju definisana pravila asocijativnosti. Stepenovanje je desno asocijativno, pa se izraz `2 ^ 3 ^ 2` tumači kao `2 ^ (3 ^ 2)`, dok su sabiranje, oduzimanje, množenje i deljenje levo asocijativni, pa se izraz `2 - 3 + 4` tumači kao `(2 - 3) + 4`.

Da bi se naglasio redosled izračunavanja, koriste se zagrade:

```
ghci> (2 + 3) * 4
20

ghci> (2 * 3) ^ 4
1296
```

Posebnu pažnju treba obratiti na negativne brojeve. Izraz:

```
ghci> 5 * -3
```

dovodi do greške, dok je ispravan zapis:

```
ghci> 5 * (-3)
-15
```

Razlog je to što se znak `-` u ovom kontekstu ne tumači kao deo broja, već kao operator.

Pored aritmetičkih izraza, u `ghci` okruženju moguće je raditi i sa logičkim vrednostima. Logičke vrednosti su `True` i `False`, a osnovni logički operatori su `&&` (logičko i), `||` (logičko ili) i `not` (negacija):

```
ghci> True && False
False

ghci> True || False
True
```

```
ghci> not True
False
```

Takođe, moguće je vršiti poređenje vrednosti. Operator `==` označava jednakost, dok `/=` označava nejednakost. Pored toga, standardni operatori poređenja nad brojevima su `<`, `<=`, `>` i `>=`:

```
ghci> 5 == 5
True

ghci> 5 /= 3
True

ghci> 10 > 4
True

ghci> 3 <= 2
False
```

Rad sa tekstualnim vrednostima takođe je podržan. Karakteri se zapisuju između jednostrukih, a stringovi između dvostrukih navodnika:

```
ghci> 'a'
'a'

ghci> "hello"
"hello"

ghci> "hello" == "hello"
True
```

Važno je napomenuti da su operacije definisane samo za odgovarajuće vrste vrednosti. Na primer, zapis:

```
ghci> True == 5
```

dovodi do greške, jer se poređenje može vršiti samo između vrednosti istog tipa. Ovakvo ponašanje proizilazi iz činjenice da svaka vrednost u Haskellu ima tačno određen tip, a operacije su definisane samo nad kompatibilnim tipovima.

Pored operatora, u `ghci` okruženju koriste se i predefinisane funkcije. Funkcije se u Haskellu pozivaju tako što se ime funkcije i argumenti razdvajaju razmacima, bez upotrebe zagrada kao u imperativnim jezicima. Na primer, funkcija `succ` vraća sledbenik broja:

```
ghci> succ 8
9
```

Funkcije mogu primati više argumenata. Na primer, funkcije `min` i `max` vraćaju manju, odnosno veću od dve zadate vrednosti:

```
ghci> min 9 10
9
```

```
ghci> max 100 101
101
```

Primena funkcije ima veći prioritet od svih aritmetičkih operatora. Na primer:

```
ghci> succ 9 + max 5 4 + 1
16
```

se tumači kao:

```
ghci> (succ 9) + (max 5 4) + 1
16
```

Ukoliko želimo drugačiji redosled izračunavanja, potrebno je koristiti zagrade:

```
ghci> succ (9 * 10)
91
```

Za razliku od uobičajenog prefiksnog zapisa, funkcije u Haskellu mogu se zapisivati i u infiksnom obliku, pri čemu se ime funkcije stavlja između argumenata i okružuje obrnutim apostrofima. Ovakav zapis često poboljšava čitljivost izraza. Na primer, funkcija `div` vrši celobrojno deljenje:

```
ghci> div 10 3
3

ghci> 10 `div` 3
3
```

1.5 Rad sa datotekama

Pored neposrednog unosa izraza u okruženju `ghci`, Haskell kôd se u praksi zapisuje u tekstualnim datotekama sa ekstenzijom `.hs`. Takva datoteka sastoji se od niza definicija kojima se imenima pridružuju vrednosti izraza, uključujući i funkcije. Na primer, sledeći sadržaj predstavlja jednostavnu Haskell datoteku:

```
daysInWeek = 7
hoursInDay = 24
hoursInWeek = daysInWeek * hoursInDay
```

Imena vrednosti i funkcija moraju početi malim slovom, a zatim mogu sadržati slova, cifre, donju crtu i apostrof. Na primer, ispravna su imena `myFun`, `fun1`, `arg_2` i `x'`. Određene reči imaju posebno značenje u jeziku i ne mogu se koristiti kao imena. To su: `as`, `class`, `data`, `default`, `deriving`, `do`, `else`, `hiding`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where` i `qualified`. Uobičajeno je i da imena listi (o kojima će kasnije biti reči) imaju sufiks `s`, na primer `ns` ili `xs`.

Za razliku od promenljivih u imperativnim jezicima, imena u Haskellu ne predstavljaju memorijske lokacije čija se vrednost menja tokom izvršavanja, već vezivanja imena za određene vrednosti. Jednom definisana vrednost ostaje ista, pa se izračunavanje zasniva na upotrebi definicija, a ne na promeni stanja.

Datoteka se može učitati u okruženje `ghci` komandom `:load`, odnosno skraćeno `:l`. Na primer, ako je datoteka sačuvana pod imenom `test.hs`, može se učitati na sledeći način:

```
ghci> :load test.hs
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, one module loaded.
```

Nakon učitavanja, definicije iz datoteke mogu se neposredno koristiti:

```
ghci> hoursInWeek
168
```

Pri tome su i dalje dostupne sve predefinisane funkcije i operatori standardnog okruženja. Ako se datoteka izmeni i ponovo sačuva, potrebno je ponovo učitati sadržaj komandom `:reload`, odnosno skraćeno `:r`:

```
ghci> :reload
Ok, one module loaded.
```

Pri tome se odbacuju i definicije koje su eventualno bile unete direktno u okruženju `ghci`, a nisu sačuvane u datoteci.

U datotekama se, pored jednostavnih definicija vrednosti, mogu definisati i funkcije. Na primer:

```
double x = x + x
quadruple x = double (double x)
```

U ovom primeru funkcija `quadruple` koristi prethodno definisanu funkciju `double`. Nove funkcije se mogu graditi kombinovanjem ranije definisanih i predefinisanih funkcija.

Nakon učitavanja datoteke u `ghci`, ovako definisane funkcije mogu se neposredno koristiti:

```
ghci> double 5
10

ghci> quadruple 3
12
```

Moguće je i kombinovati funkcije definisane u datoteci sa već postojećim funkcijama iz standardnog okruženja. Na primer:

```
ghci> succ (double 4)
9

ghci> max (double 3) (quadruple 2)
8
```

Pri pisanju Haskell datoteka važno je voditi računa o rasporedu koda. Haskell koristi pravilo nazubljivanja, što znači da definicije na istom nivou moraju počinjati u istoj koloni. Početak definicije ne treba uvlačiti bez potrebe, a kada se izraz prelama u više redova, nastavak mora biti više uvučen od početka izraza. Na primer, sledeći zapis je ispravan:

```
a = 10 * 10
```

```
  + 1
```

dok sledeći zapis nije:

```
a = 10 * 10
+ 1
```

Pri uvlačenju je preporučljivo koristiti razmake, a ne tabove, jer različiti editori mogu različito tumačiti tabulatore, a upotreba tabova može dovesti i do upozorenja pri kompajliranju.

Haskell podržava dve vrste komentara. Linijski komentari počinju sa `--` i traju do kraja reda, dok se višelinijski komentari nalaze između oznaka `{-` i `-}`. Na primer:

```
-- Sabira dva broja
add x y = x + y

{-
Ovo je
viselinijski komentar.
-}
```

2 Tipovi

2.1 Osnovni pojmovi o tipovima

Tip predstavlja kolekciju vrednosti. Svaka vrednost u Haskellu pripada tačno jednom tipu. Ako neka vrednost `v` pripada tipu `T`, to se označava zapisom `v :: T` i kaže se da vrednost `v` ima tip `T`. Na primer, vrednosti `True` i `False` imaju tip `Bool` (logičke vrednosti).

Haskell poseduje statički sistem tipova, što znači da se tip svakog izraza određuje pre njegovog izvršavanja. Time se mnoge greške mogu otkriti unapred. Na primer, izraz `2 + 3` je ispravan, dok izraz `2 + True` dovodi do greške, jer sabiranje nije definisano između broja i logičke vrednosti. Na taj način tipovi predstavljaju važan mehanizam za proveru ispravnosti programa.

Još jedna važna osobina Haskella je zaključivanje tipova (engl. *type inference*). U velikom broju slučajeva nije potrebno eksplicitno navoditi tipove vrednosti i funkcija, jer kompajler može sam da ih zaključi na osnovu izraza. Na primer, u prethodnim primerima mogli smo da pišemo izraze bez navođenja tipova, a da sistem ipak ispravno odredi njihovo značenje.

U okruženju `ghci` tip izraza može se prikazati komandom `:type`, odnosno skraćeno `:t`. Na primer:

```
ghci> :t True
True :: Bool

ghci> :t not True
not True :: Bool
```

Na ovaj način moguće je brzo proveriti tip bilo kog izraza. U nastavku ćemo se upoznati sa osnovnim tipovima u Haskellu i načinima njihovog korišćenja.

2.2 Osnovni tipovi

Haskell poseduje više osnovnih tipova koji su ugrađeni u sam jezik i koji se najčešće koriste u programima. U nastavku navodimo najvažnije među njima.

Tip `Bool` predstavlja kolekciju logičkih vrednosti `True` i `False`. Nad ovim vrednostima mogu se primenjivati logički operatori `&&` (i), `||` (ili) i funkcija `not` (negacija):

```
ghci> True && False
False

ghci> True || False
True

ghci> not True
False

ghci> (not False) && True
True
```

Logičke vrednosti mogu se i porediti. Operator `==` proverava da li su dve vrednosti jednake, dok `/=` proverava da li su različite. Rezultat ovakvog poređenja ponovo je logička vrednost:

```
ghci> True == True
```

```
True

ghci> True == False
False

ghci> True /= False
True
```

Tipovi `Int` i `Integer` predstavljaju cele brojeve. Tip `Int` koristi fiksnu količinu memorije, pa su njegove vrednosti ograničenog opsega, koji zavisi od arhitekture računara (tipično su to celi brojevi od -2^{63} do $2^{63}-1$). Nasuprot tome, tip `Integer` predstavlja cele brojeve proizvoljne veličine, pri čemu je jedino praktično ograničenje raspoloživa memorija računara.

Nad vrednostima tipova `Int` i `Integer` mogu se koristiti standardni aritmetički operatori `+`, `-` i `*`, kao i operator `^` za stepenovanje. Mogu se koristiti i funkcije `div` i `mod`, koje predstavljaju celobrojno deljenje i ostatak pri deljenju:

```
ghci> 7 + 5
12

ghci> 7 - 5
2

ghci> 7 * 5
35

ghci> 7 ^ 2
49

ghci> 7 ^ (-2)
*** Exception: Negative exponent

ghci> div 7 2
3

ghci> mod 7 2
1
```

Operator `^` nije namenjen stepenovanju na negativan stepen kod celobrojnih vrednosti.

Funkcije `div` i `mod` se često zapisuju i u infiksnom obliku:

```
ghci> 7 `div` 2
3

ghci> 7 `mod` 2
1
```

Pri radu sa negativnim brojevima često je potrebno koristiti zagrade, jer znak `-` u tom slučaju ne predstavlja samo operator oduzimanja, već i negaciju broja:

```
ghci> 3 * (-2)
-6
```

Ako želimo da vidimo razliku između tipova `Int` i `Integer`, možemo eksplicitno zadati tip i posmatrati stepenovanje velikog broja:

```
ghci> 2 ^ 63 :: Int
-9223372036854775808

ghci> 2 ^ 63 :: Integer
9223372036854775808
```

U prvom slučaju dolazi do prekoračenja opsega tipa `Int`, pa rezultat nije ispravan. U drugom slučaju koristi se tip `Integer`, koji može da predstavi znatno veće vrednosti. Zbog toga je `Integer` pogodniji kada postoji mogućnost rada sa veoma velikim brojevima, dok je `Int` obično brži, jer odgovara načinu na koji procesor direktno obrađuje cele brojeve.

Vrednosti tipova `Int` i `Integer` mogu se i porediti operatorima `==`, `/=`, `<`, `<=`, `>` i `>=`:

```
ghci> 6 == 3 * 2
True

ghci> 2 < 3
True

ghci> 2 >= 3
False
```

Tipovi `Float` i `Double` koriste se za reprezentaciju realnih brojeva. Tip `Float` koristi jednostruku preciznost (32 bita), dok `Double` koristi dvostruku preciznost (64 bita). U praksi se češće koristi `Double`, jer daje preciznije rezultate.

Nad ovim tipovima mogu se koristiti operatori `+`, `-`, `*`, `/`, kao i operatori `^` i `**`. Pri tome, operator `**` dozvoljava i realne stepene:

```
ghci> 5 / 2
2.5

ghci> 2.0 ^ 3
8.0

ghci> 2 ** 3
8.0

ghci> 9 ** 0.5
3.0
```

U Haskellu su ugrađene i brojne matematičke funkcije, kao što su `sin`, `cos`, `log`, `sqrt` i druge, kao i konstanta `pi`:

```
ghci> pi
3.141592653589793

ghci> sqrt 2
1.4142135623730951
```

Tip `Char` predstavlja pojedinačne karaktere kodirane Unicode standardom. Karakteri se

zapisuju između jednostrukih navodnika:

```
ghci> 'a'
'a'

ghci> 'Z'
'Z'
```

Karakteri se mogu porediti pomoću standardnih operatora poređenja:

```
ghci> 'a' < 'b'
True
```

Tip `String` predstavlja niske karaktera. U Haskellu se niske zapisuju između dvostrukih navodnika:

```
ghci> "hello"
"hello"

ghci> "abc" == "abc"
True
```

2.3 Liste

Lista predstavlja uređenu kolekciju elemenata istog tipa. Zbog toga se kaže da su liste homogene strukture podataka. Elementi liste zapisuju se između uglastih zagrada i razdvajaju zarezima:

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

ghci> [True,False,True]
[True,False,True]
```

Svi elementi liste moraju biti istog tipa. Na primer, sledeći zapis nije ispravan:

```
ghci> [1,2,'a']
```

jer elementi nemaju isti tip.

Prazna lista se zapisuje kao `[]`. Lista može sadržati i druge liste kao elemente. Te liste mogu biti različitih dužina, ali moraju imati sve elemente istog tipa:

```
ghci> [[1,2],[3,4,5]]
[[1,2],[3,4,5]]
```

Ako su elementi liste tipa `A`, tada lista ima tip `[A]`. Na primer, lista celih brojeva ima tip `[Int]`, a lista karaktera tip `[Char]`.

Za rad sa listama koriste se posebni operatori.

Operator `++` služi za spajanje dve liste:

```
ghci> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Pri tome obe liste moraju biti istog tipa.

Operator `:` služi za dodavanje elementa na početak liste. Sa leve strane nalazi se element, a sa desne lista:

```
ghci> 1 : [2,3,4]
[1,2,3,4]
```

Dakle, operator `:` sa leve strane očekuje jedan element, a sa desne listu, dok operator `++` sa obe strane očekuje liste. Lista `[1,2,3]` može se zapisati i kao `1 : 2 : 3 : []`, što pokazuje da se lista konstruiše dodavanjem elemenata na početak prazne liste.

Elementima liste može se pristupiti pomoću operatora `!!`, pri čemu se indeksiranje vrši od nule:

```
ghci> [10,20,30,40] !! 2
30
```

Pokušaj pristupa elementu van opsega dovodi do greške.

Nad listama su definisane brojne funkcije. U nastavku su neke od najčešće korišćenih:

```
ghci> length [1,2,3,4]
4

ghci> head [10,20,30]
10

ghci> tail [10,20,30]
[20,30]

ghci> last [10,20,30]
30

ghci> init [10,20,30]
[10,20]

ghci> reverse [1,2,3]
[3,2,1]
```

Funkcija `length` vraća dužinu liste, `head` prvi element, `tail` sve osim prvog elementa, `last` poslednji element, `init` sve osim poslednjeg, dok `reverse` vraća listu u obrnutom redosledu. Pozivanje funkcija `head`, `tail`, `last` ili `init` nad praznom listom dovodi do greške.

Funkcije `sum` i `product` rade nad listama brojeva:

```
ghci> sum [1,2,3,4]
10

ghci> product [1,2,3,4]
24
```

Funkcija `elem` proverava da li se element nalazi u listi:

```
ghci> elem 3 [1,2,3,4]
True

ghci> elem 5 [1,2,3,4]
False
```

Funkcija `null` proverava da li je lista prazna:

```
ghci> null []
True

ghci> null [1,2]
False
```

U Haskellu su niske (tip `String`) liste karaktera. Na primer, niska `"abc"` ekvivalentna je listi `['a','b','c']`. Zbog toga se funkcije nad listama mogu primenjivati i na niske:

```
ghci> head "hello"
'h'

ghci> length "hello"
5
```

Liste se mogu porediti, pod uslovom da se elementi mogu porediti. Poređenje se vrši leksikografski, odnosno redom po elementima:

```
ghci> [1,2,3] == [1,2,3]
True

ghci> [1,2] < [1,3]
True

ghci> [1,2] < [1,2,3]
True
```

Poređenje se vrši na prvom mestu na kome se liste razlikuju, dok se preostali elementi ne uzimaju u obzir. Ako su svi upoređeni elementi jednaki, kraća lista se smatra manjom:

```
ghci> [1,2,100] < [1,3,0]
True

ghci> [1,2,1000] > [1,2,3]
True

ghci> [2,0,0] > [1,100,100]
True

ghci> [1,2,3] < [1,2,3,0]
True
```

Poređenje važi i za niske, pri čemu se koristi redosled karaktera u Unicode tabeli:

```
ghci> "abc" < "abd"
True

ghci> "abc" < "ab"
False

ghci> "abc" < "abcd"
True

ghci> "a" < "B"
False
```

2.4 Uređene n-torke

Uređene n-torke (engl. *tuples*) predstavljaju konačne kolekcije vrednosti koje mogu biti različitih tipova. Za razliku od listi, koje sadrže proizvoljan broj elemenata istog tipa, n-torke imaju fiksiran broj komponenti, pri čemu svaka komponenta može biti različitog tipa.

Komponente n-torke zapisuju se između običnih zagrada i razdvajaju zarezima:

```
ghci> (1,2)
(1,2)

ghci> ('a', True, 3.14)
('a',True,3.14)
```

Tip n-torke određen je tipovima njenih komponenti i njihovim redosledom. Na primer:

```
ghci> :t ('a', True)
('a', True) :: (Char, Bool)
```

Broj komponenti n-torke naziva se njena arnost. Postoji n-torka arnosti 0, koja se zapisuje kao `()` i naziva se prazna n-torka. n-torke arnosti 1 se ne koriste, jer se zapis `(x)` ne bi razlikovao od običnog izraza u zagradi.

Za razliku od listi, tip n-torke zavisi od broja komponenti, njihovih tipova i njihovog redosleda. Na primer, tipovi `(Int, Char)` i `(Char, Int)` nisu isti.

n-torke se koriste kada je unapred poznat broj vrednosti koje treba objediniti. Na primer:

```
ghci> ("Ana", 20)
("Ana",20)
```

Ovakva vrednost može predstavljati, na primer, ime i godine osobe.

Nad uređenim parovima (tj. n-torkama arnosti 2) definisane su funkcije `fst` i `snd`, koje vraćaju prvu, odnosno drugu komponentu para:

```
ghci> fst (5, 'a')
5

ghci> snd (5, 'a')
'a'
```

Ove funkcije su definisane samo za uređene parove. Za n-torke veće arnosti ne postoje ugrađene funkcije za izdvajanje komponenti.

n-torke se često kombinuju sa drugim strukturama podataka. Na primer, moguće je imati listu uređenih parova:

```
ghci> [(1,'a'), (2,'b'), (3,'c')]
[(1,'a'),(2,'b'),(3,'c')]
```

U ovom slučaju svi elementi liste moraju imati isti tip, odnosno biti n-torke iste arnosti i sa istim tipovima komponenti. Na primer, sledeći zapis nije ispravan:

```
ghci> [(1,'a'), (2,'b','c')]
```

jer se u istoj listi pojavljuju n-torke različite arnosti.

Uređene n-torke koriste se kada je potrebno objediniti konačan broj vrednosti koje mogu biti različitih tipova, pri čemu je broj komponenti unapred poznat.

3 Funkcije

3.1 Pojam i tip funkcije

U prethodnim poglavljima već smo koristili funkcije kao osnovni način izračunavanja u Haskellu. Sada ćemo preciznije razmotriti njihov pojam i tip.

U Haskellu, funkcija predstavlja pravilo koje vrednostima jednog tipa pridružuje vrednosti drugog tipa. Ako funkcija prima argument tipa `A` i vraća rezultat tipa `B`, kažemo da je njen tip `A -> B`. Na taj način tip funkcije određuje koju vrstu argumenta funkcija očekuje i kakvu vrednost vraća kao rezultat.

Za razliku od mnogih imperativnih jezika, funkcije u Haskellu nemaju bočne efekte. Njihova uloga je da za zadate argumente izračunaju rezultat i vrate ga kao vrednost. Zbog toga se ista funkcija, primenjena na iste argumente, uvek ponaša na isti način i daje isti rezultat. U Haskellu se, takođe, ne koriste funkcije bez povratne vrednosti u smislu u kome se u drugim jezicima koriste `void` funkcije.

Kao i druge vrednosti, i funkcije imaju svoje tipove. Haskell najčešće može automatski da zaključi tip funkcije, ali je u praksi poželjno da se tip navede eksplicitno. Na primer, funkcija koja svakom celom broju pridružuje njegov kvadrat može se definisati ovako:

```
square :: Int -> Int
square x = x * x
```

Iz deklaracije `square :: Int -> Int` vidimo da funkcija `square` prima argument tipa `Int` i vraća rezultat istog tipa.

Nova funkcija može se definisati i korišćenjem već postojećih funkcija. Na primer, funkcija koja svakom celom broju pridružuje broj uvećan za 2 može se zapisati na sledeći način:

```
addTwo :: Int -> Int
addTwo x = succ (succ x)
```

Ovde je funkcija `addTwo` definisana pomoću funkcije `succ`, koja vraća sledbenika zadatog broja.

Kada se funkcija primeni na argument, Haskell proverava da li se tip argumenta poklapa sa tipom koji funkcija očekuje. Ako funkcija `f` ima tip `A -> B`, a izraz `e` ima tip `A`, tada izraz `f e` ima tip `B`. Na primer, pošto `square` ima tip `Int -> Int`, izraz `square 3` ima tip `Int`, dok izraz `addTwo 5` takođe ima tip `Int`.

Ako se funkcija primeni na argument neodgovarajućeg tipa, dolazi do tipske greške. Na primer, izraz

```
not 3
```

nije ispravan, jer funkcija `not` očekuje argument tipa `Bool`, dok je `3` broj. Takvi izrazi nisu dobro tipizirani i Haskell ih odbacuje.

Važno je primetiti da funkcija ne mora biti definisana za svaku vrednost tipa njenog argumenta. Na primer, funkcija `head` vraća prvi element liste, ali za praznu listu rezultat nije definisan. Izraz

```
head []
```

zato dovodi do greške pri izvršavanju. Dakle, činjenica da funkcija ima određeni tip ne znači nužno da je definisana za sve vrednosti tog tipa.

Mnoge funkcije i konstante koje koristimo već su unapred definisane u standardnom okruženju Haskella, koje se naziva prelid (engl. *prelude*). U njemu su definisane i već pomenute funkcije kao što su `not`, `succ`, `head`, `fst`, `div` i `mod`. Ove definicije su automatski dostupne u svakom Haskell programu.

3.2 Funkcije sa više argumenata

U prethodnoj podsekciji razmatrali smo funkcije koje primaju jedan argument. Međutim, u Haskellu se često koriste i funkcije koje primaju više argumenata (primer su funkcije `div` i `mod`, koje smo već koristili).

Najjednostavniji način da se opiše funkcija koja koristi dva argumenta jeste da se ta dva argumenta objedine u uređeni par. Na primer, funkcija koja računa zbir dva cela broja može se definisati ovako:

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

Ovde funkcija `add` prima jedan argument tipa `(Int, Int)`, odnosno uređeni par celih brojeva, i vraća njihov zbir.

U Haskellu se, međutim, funkcije sa više argumenata najčešće zapisuju tako što se argumenti navode jedan po jedan. Ista funkcija može se zato definisati i na sledeći način:

```
add' :: Int -> Int -> Int
add' x y = x + y
```

Na prvi pogled izgleda kao da funkcija `add'` prima dva argumenta odjednom. Međutim, njen tip pokazuje da se ona može posmatrati drugačije: najpre prima jedan argument tipa `Int`, a kao rezultat vraća novu funkciju, koja zatim prima još jedan argument tipa `Int` i vraća konačan rezultat tipa `Int`.

Zbog toga zapis

```
Int -> Int -> Int
```

treba posmatrati kao

```
Int -> (Int -> Int)
```

odnosno strelica `->` u tipovima grupiše se zdesna nalevo. Sa druge strane, sama primena funkcije na argumente grupiše se sleva nadesno, pa se izraz

```
add' 2 3
```

tumači kao

```
(add' 2) 3
```

Ovakav zapis je u Haskellu uobičajeniji od zapisa sa uređenim parom, jer je fleksibilniji i prirodniji za dalji rad sa funkcijama.

Isti princip važi i za funkcije sa više od dva argumenta. Na primer, funkcija koja računa proizvod tri cela broja može se definisati na sledeći način:

```
mult :: Int -> Int -> Int -> Int
```

```
mult x y z = x * y * z
```

Ovaj tip se posmatra kao

```
Int -> (Int -> (Int -> Int))
```

a primena funkcije

```
mult 2 3 4
```

kao

```
((mult 2) 3) 4
```

Da ne bismo stalno pisali veliki broj zagrada, u Haskellu se podrazumeva upravo ovakvo grupisanje tipova i primena funkcija. Zbog toga se funkcije sa više argumenata obično zapisuju u kraćem obliku, bez dodatnih zagrada, osim kada je uređeni par zaista potreban kao jedna celina.

3.3 Uslovni izrazi

Jedan od osnovnih načina definisanja funkcija u Haskellu jeste korišćenje uslovnih izraza. Uslovni izraz omogućava da se, u zavisnosti od ispunjenosti nekog uslova, izaberu dve moguće vrednosti. Njegov opšti oblik je

```
if uslov then izraz1 else izraz2
```

pri čemu se izraz `izraz1` bira ako je uslov ispunjen, a izraz `izraz2` u suprotnom.

Uslov mora biti logičkog tipa `Bool`, a oba moguća rezultata moraju biti istog tipa. Na taj način i ceo uslovni izraz ima jednoznačno određen tip. Za razliku od nekih imperativnih jezika, u Haskellu se grana `else` ne može izostaviti. Razlog je taj što je `if then else` izraz, pa zato uvek mora imati vrednost.

Na primer, funkcija koja vraća apsolutnu vrednost celog broja može se definisati ovako:

```
absolute :: Int -> Int
absolute n = if n >= 0 then n else -n
```

Ako je broj `n` nenegativan, rezultat je sam broj `n`, a inače rezultat je njegov suprotan broj.

Uslovni izrazi mogu biti i ugnježdeni. Na primer, funkcija koja određuje znak celog broja može se zapisati na sledeći način:

```
sign :: Int -> Int
sign n =
  if n < 0 then -1
  else if n == 0 then 0
  else 1
```

Ovde se najpre proverava da li je broj negativan. Ako jeste, rezultat je `-1`. U suprotnom se proverava da li je broj jednak nuli. Ako jeste, rezultat je `0`, a u svim ostalim slučajevima rezultat je `1`.

Pošto je `if then else` izraz, može se koristiti i kao deo složenijeg izraza. Na primer:

```
increase :: Int -> Int
increase n = (if n <= 10 then n else 10) + 1
```

U ovoj definiciji najpre se, u zavisnosti od vrednosti broja `n`, bira jedna od dve vrednosti, a zatim se na dobijeni rezultat dodaje `1`. Zagrade su ovde potrebne zato što se sabiranje vrši nad vrednošću celog uslovnog izraza.

Uslovni izrazi predstavljaju najjednostavniji oblik grananja u Haskellu. U narednim podsekcijama videćemo i druge načine definisanja funkcija sa više mogućih ishoda, koji su u mnogim situacijama pregledniji i pogodniji za upotrebu.

3.4 Ograđene definicije

Kao alternativa uslovnim izrazima, funkcije se u Haskellu mogu definisati i pomoću ograđenih definicija (engl. *guards*). Kod ovakvih definicija navodi se niz logičkih uslova, a vrednost funkcije određuje prvi uslov koji je zadovoljen. Opšti oblik je

```
f x1 x2 ... xn
  | uslov1 = izraz1
  | uslov2 = izraz2
  ...
  | uslovN = izrazN
```

gde `x1, x2, ..., xn` predstavljaju argumente funkcije `f`, svaki uslov je izraz tipa `Bool`, a svi izrazi na desnoj strani moraju biti istog tipa.

Ograđene definicije su naročito korisne kada funkcija ima više mogućih slučajeva, jer su tada često preglednije od ugnježdenih `if then else` izraza. Na primer, funkcija `absolute` iz prethodne podsekcije može se definisati i ovako:

```
absolute :: Int -> Int
absolute n
  | n >= 0 = n
  | n < 0  = -n
```

U ovom primeru najpre se proverava uslov `n >= 0`. Ako je on zadovoljen, rezultat je `n`. U suprotnom se proverava naredni uslov `n < 0`, pa je u tom slučaju rezultat `-n`. Uslovi se, dakle, proveravaju redom, a vrednost funkcije određuje prva grana čiji je uslov tačan.

U mnogim slučajevima poslednja grana služi da obuhvati sve preostale mogućnosti. Tada se umesto posebnog uslova često koristi oznaka `otherwise`, koja je unapred definisana kao vrednost `True`. Na primer, funkcija koja određuje znak celog broja može se zapisati na sledeći način:

```
sign :: Int -> Int
sign n
  | n < 0     = -1
  | n == 0    = 0
  | otherwise = 1
```

Ovde se najpre proverava da li je broj negativan, zatim da li je jednak nuli, a poslednja grana obuhvata sve ostale slučajeve. Zbog toga je redosled grana važan.

Ako nijedan uslov nije zadovoljen, a nije navedena završna grana koja pokriva preostale slučajeve, doći će do greške pri izvršavanju. Na primer, definicija

```
describeSign :: Int -> String
describeSign n
  | n > 0 = "Pozitivan"
  | n < 0 = "Negativan"
```

nije potpuna, jer vrednost `0` nije obuhvaćena nijednim uslovom.

Ograđene definicije mogu se koristiti i kod funkcija sa više argumenata. Na primer, funkcija koja vraća najveći od tri cela broja može se definisati na sledeći način:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  | otherwise        = z
```

Ovde se uslovi ponovo proveravaju redom, a rezultat funkcije određuje prva grana čiji je uslov tačan.

Ograđene definicije predstavljaju pregledan način za definisanje funkcija sa više slučajeva. U odnosu na uslovne izraze, njihova glavna prednost je u tome što se niz uslova i odgovarajućih vrednosti može zapisati jasnije i čitljivije.

3.5 Lokalna definisanja pomoću where i let

Pri definisanju funkcija često se javlja potreba da se neki pomoćni izrazi izdvoje i imenuju, kako bi definicija bila preglednija i kako bi se izbeglo ponavljanje. U Haskellu se to može uraditi pomoću lokalnih definisanja. Najčešće se za to koriste konstrukcije `where` i `let ... in`.

Konstrukcija `where` koristi se uz definiciju funkcije. Njome se na kraju definicije mogu navesti pomoćna imena koja važe samo u okviru te definicije. Na primer:

```
describe :: Int -> String
describe n
  | absN == 0 = "Nula"
  | absN <= 9 = "Jednocifren"
  | otherwise = "Visecifren"
  where
    absN = absolute n
```

Ovde je ime `absN` definisano lokalno i može se koristiti u svim granama funkcije. Time se izraz `absolute n` zapisuje samo jednom, pa definicija postaje preglednija. U ovoj definiciji korišćena je funkcija `absolute` iz prethodne podsekcije.

Pomoću `where` može se navesti i više lokalnih definicija:

```
interval :: Int -> String
interval n
  | n < lower = "Mali"
  | n > upper = "Veliki"
  | otherwise = "U intervalu"
  where
```

```
    lower = 10
    upper = 20
```

Lokalna imena definisana pomoću `where` vidljiva su samo unutar funkcije uz koju stoje.

Sličnu ulogu ima i konstrukcija `let ... in`. Njome se takođe uvode lokalna imena, ali za razliku od `where`, ona predstavlja izraz. Opšti oblik je

```
let ime1 = izraz1
    ime2 = izraz2
    ...
in izraz
```

Imena definisana između `let` i `in` važe samo u izrazu koji dolazi posle reči `in`. Na primer:

```
surface :: Int
surface =
  let a = 10
      b = 5
      c = 3
  in 2 * (a * b + b * c + c * a)
```

Ovde su imena `a`, `b` i `c` uvedena samo radi izračunavanja izraza koji sledi posle `in`. Bez tih lokalnih definicija isti izraz bi bio manje pregledan.

Pošto `let ... in` predstavlja izraz, može se koristiti i unutar tela funkcije:

```
increase :: Int -> Int
increase n = let m = n + 1 in m * m
```

Na taj način se pomoćno ime uvodi samo tamo gde je potrebno.

Glavna razlika između konstrukcija `where` i `let ... in` jeste u tome što se `where` vezuje za celu definiciju funkcije, dok je `let ... in` vezan samo za jedan izraz. Zbog toga je `where` pogodniji kada se isto pomoćno ime koristi u više grana funkcije, dok je `let ... in` prirodniji kada je pomoćno ime potrebno samo u jednom izrazu.

Kao i kod drugih Haskell definicija, pri pisanju lokalnih definisanja važno je pravilno nazubljivanje. Definicije koje pripadaju istom `where` bloku, odnosno istom `let ... in` bloku, treba da budu poravnate u istoj koloni.

Lokalne definicije mogu se zapisati i u jednom redu, razdvojene znakom `;`. Na primer:

```
surface' = let a = 10; b = 5; c = 3 in 2 * (a * b + b * c + c * a)

interval' n =
  if n < lower then "Mali" else if n > upper then "Veliki" else "U intervalu"
  where lower = 10; upper = 20
```

3.6 Podudaranje oblika

Podudaranje oblika (engl. *pattern matching*) je način definisanja funkcija kod koga se vrednost funkcije određuje na osnovu oblika njenih argumenata. Umesto da se u telu funkcije naknadno proveravaju uslovi, oblici argumenata navode se neposredno u definiciji funkcije. Ako se prvi navedeni oblik poklopi sa argumentom, uzima se odgovarajuća vrednost. U suprotnom se

proverava naredni oblik, i tako redom.

Na primer, funkcija slična bibliotečkoj funkciji `not` može se definisati ovako:

```
not' :: Bool -> Bool
not' False = True
not' True  = False
```

Ovde se vrednost argumenta poredi sa dve moguće logičke vrednosti. Ako je argument `False`, rezultat je `True`, a ako je argument `True`, rezultat je `False`.

Pri podudaranju oblika redosled slučajeva je važan. Na primer, funkcija

```
f :: Int -> Int
f 0 = 1
f _ = 0
```

preslikava `0` u `1`, a sve ostale brojeve u `0`. U drugoj liniji korišćen je džoker (engl. *wildcard*) `_`, koji predstavlja proizvoljnu vrednost koju ne želimo da vezujemo za ime. Kada bi redosled bio obrnut, prvi slučaj bi važio za sve argumente, pa druga linija nikada ne bi bila upotrebljena.

Ako se neki mogući oblik argumenta ne obuhvati nijednim slučajem, funkcija neće biti definisana za sve vrednosti svog tipa. Na primer, definicija

```
isZero :: Int -> Bool
isZero 0 = True
```

nije potpuna, jer ne određuje vrednost funkcije za argumente različite od nule.

Podudaranje oblika može se koristiti i kod funkcija sa više argumenata. Na primer, funkcija slična bibliotečkom operatoru `&&` može se definisati ovako:

```
and' :: Bool -> Bool -> Bool
and' True  True  = True
and' _     _     = False
```

Ovde prva linija obuhvata jedini slučaj u kome je rezultat `True`, dok druga linija pokriva sve preostale mogućnosti. Treba napomenuti da se džoker `_` može pojaviti više puta u istoj jednačini, jer ne vezuje vrednost za ime. Nasuprot tome, u jednoj jednačini nije dozvoljeno koristiti isto ime za više argumenata. Zbog toga zapis poput

```
g x x = x
```

nije ispravan.

Podudaranje oblika je naročito korisno pri radu sa uređenim n-torkama i listama. Na primer, funkcije slične bibliotečkim funkcijama `fst` i `snd`, ali definisane za uređene trojke, mogu se zapisati na sledeći način:

```
fst3 :: (Int, Int, Int) -> Int
fst3 (x, _, _) = x

snd3 :: (Int, Int, Int) -> Int
snd3 (_, y, _) = y
```

Ovde se uređena trojka dekonstruše na svoje komponente, pri čemu se džoker koristi za one komponente koje nas ne zanimaju.

Slično tome, moguće je vršiti podudaranje i nad listama. Na primer, funkcija koja proverava da li je lista celih brojeva prazna može se definisati ovako:

```
isEmpty :: [Int] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Prazna lista se prepoznaje obrascem `[]`, dok druga linija obuhvata sve neprazne liste.

Za neprazne liste često je korisno rastaviti listu na prvi element i ostatak liste. To se radi pomoću operatora `:`. Na primer, funkcije slične bibliotečkim funkcijama `head` i `tail` mogu se definisati ovako:

```
head' :: [Int] -> Int
head' (x:_) = x

tail' :: [Int] -> [Int]
tail' (_:xs) = xs
```

U izrazu `(x:xs)` ime `x` označava prvi element liste, a `xs` ostatak liste. Zagrade su ovde potrebne zato što primena funkcije ima veći prioritet od operatora `:`. Bez zagrada, zapis ne bi imao željeno značenje.

Podudaranjem oblika moguće je razlikovati i posebne oblike listi, na primer praznu, jednočlanu i dvočlanu listu:

```
sumSmall :: [Int] -> Int
sumSmall []     = 0
sumSmall [x]    = x
sumSmall [x, y] = x + y
sumSmall _      = 1
```

Ova funkcija praznoj listi dodeljuje `0`, jednočlanoj listi njen element, dvočlanoj listi zbir elemenata, a svim ostalim listama vrednost `1`.

Pri podudaranju oblika nad listama nije moguće koristiti operator `++`. Na primer, nije dopušteno zapisati obrazac poput `xs ++ ys`, jer Haskell ne može jednoznačno da odredi gde se završava prva, a gde počinje druga lista.

Podudaranje oblika predstavlja pregledan i prirodan način definisanja funkcija u zavisnosti od oblika argumenata. Posebno je korisno kada se radi sa n-torkama i listama, jer omogućava da se podaci neposredno razlože na svoje sastavne delove.

3.7 Case izrazi

Podudaranje oblika do sada smo koristili pri definisanju funkcija. U Haskellu je, međutim, moguće koristiti ga i neposredno u izrazima, pomoću `case` konstrukcije. Opšti oblik je

```
case izraz of
  obrazac1 -> izraz1
  obrazac2 -> izraz2
  ...
```

Pri tome se vrednost izraza `izraz` redom poredi sa navedenim obrascima. Rezultat `case` izraza jeste vrednost koja odgovara prvom obrascu koji se poklopi.

Na primer, funkcija koja proverava da li je lista prazna može se definisati i pomoću `case` izraza:

```
isEmpty' :: [Int] -> Bool
isEmpty' xs = case xs of
  [] -> True
  _  -> False
```

Ovde se argument `xs` poredi najpre sa obrascem `[]`. Ako je lista prazna, rezultat je `True`, a u suprotnom se bira druga grana i rezultat je `False`.

Prednost `case` izraza je u tome što omogućava podudaranje oblika i onda kada ne definišemo novu funkciju, već želimo da ga upotrebimo neposredno u nekom izrazu. Na primer:

```
describeList :: [Int] -> String
describeList xs = "Lista je " ++ case xs of
  []  -> "prazna."
  [x] -> "jednoclana."
  _   -> "duza."
```

Ovde se pomoću `case` izraza određuje odgovarajući tekst na osnovu oblika liste, a zatim se taj tekst koristi kao deo šireg izraza.

Kao i kod podudaranja oblika u definicijama funkcija, redosled obrazaca je važan. Takođe, svi izrazi na desnoj strani strelice moraju biti istog tipa, kako bi i ceo `case` izraz imao jednoznačno određen tip.

`Case` izrazi predstavljaju prirodno proširenje podudaranja oblika. Oni su naročito korisni kada želimo da ispitamo oblik neke vrednosti unutar izraza, a ne samo pri definisanju funkcije.

3.8 Lambda izrazi

Pored definisanja funkcija pomoću jednačina, u Haskellu je moguće konstruisati funkcije i pomoću lambda izraza. Lambda izraz sadrži argumente funkcije i izraz koji određuje rezultat, ali pri tome ne dodeljuje funkciji ime. Zbog toga se lambda izrazi često nazivaju i anonimne funkcije.

Opšti oblik lambda izraza je

```
\x -> izraz
```

gde `x` predstavlja argument funkcije, a `izraz` telo funkcije. Simbol `\` predstavlja zapis grčkog slova λ.

Na primer, funkcija koja svakom celom broju pridružuje njegov dvostruki iznos može se zadati lambda izrazom

```
\x -> 2 * x
```

Pošto lambda izraz predstavlja funkciju, može se primeniti neposredno na argument. Na primer,

```
(\x -> 2 * x) 5
```

ima vrednost `10`.

Lambda izraz se može i dodeliti nekom imenu, čime dobijamo uobičajenu definiciju funkcije. Na primer:

```
double :: Int -> Int
double = \x -> 2 * x
```

Ova definicija ima isto značenje kao i ranije definicije oblika `double x = 2 * x`. Ipak, u praksi se za obične definicije funkcija najčešće koristi upravo taj jednostavniji zapis, dok se lambda izrazi koriste onda kada želimo da funkciju zadamo neposredno, bez posebnog imenovanja.

Lambda izrazi mogu imati i više argumenata. Na primer:

```
add :: Int -> Int -> Int
add = \x -> (\y -> x + y)
```

Ova definicija jasno pokazuje ono što smo ranije videli kod funkcija sa više argumenata: funkcija najpre prima jedan argument, a zatim vraća novu funkciju koja prima sledeći argument. Prethodna definicija se može zapisati i kraće kao

```
add = \x y -> x + y
```

Lambda izrazi su naročito korisni kada funkciju želimo da zadamo neposredno, bez uvođenja posebnog imena. Na primer, izraz

```
(\x -> x + 1) 7
```

predstavlja primenu anonimne funkcije na broj `7`. Pri tome se argument `7` uvrštava u telo funkcije umesto parametra `x`, pa se dobija izraz `7 + 1`, čija je vrednost `8`. Na taj način lambda izrazi omogućavaju da funkciju zapišemo i odmah upotrebimo, bez potrebe da joj prethodno dodelimo ime.

Lambda izrazi predstavljaju osnovni način zapisivanja anonimnih funkcija u Haskellu. Iako se za većinu običnih definicija koriste jednačine i podudaranje oblika, lambda izrazi su korisni kada želimo da funkciju zapišemo neposredno i bez uvođenja novog imena.

4 Tipske promenljive i tipske klase

4.1 Tipske promenljive i polimorfni tipovi

Do sada smo u tipskim deklaracijama uglavnom koristili konkretne tipove, kao što su `Int`, `Bool` ili `[Int]`. Međutim, u Haskellu je moguće definisati i funkcije koje se mogu primenjivati nad vrednostima različitih tipova. Takve funkcije u svojim tipovima sadrže tipske promenljive.

Tipska promenljiva je oznaka koja predstavlja proizvoljan tip. Za razliku od imena konkretnih tipova, koja počinju velikim slovom, imena tipskih promenljivih pišu se malim slovom. Najčešće se koriste imena `a`, `b`, `c` i slično.

Na primer, funkcija koja bilo koju vrednost smešta u jednočlanu listu može se definisati ovako:

```
singleton :: a -> [a]
singleton x = [x]
```

Ovde tipska promenljiva `a` označava proizvoljan tip. Zbog toga funkcija `singleton` može primiti argument bilo kog tipa, a kao rezultat vraća jednočlanu listu čiji je element tog istog tipa.

Tip koji sadrži jednu ili više tipskih promenljivih naziva se polimorfni tip, a funkcija sa takvim tipom polimorfna funkcija. Tako je `a -> [a]` polimorfni tip, a `singleton` polimorfna funkcija.

Mnoge funkcije iz prelida imaju polimorfne tipove. Na primer:

```
reverse :: [a] -> [a]
fst     :: (a, b) -> a
```

Iz tipa funkcije `reverse` vidi se da ona prima listu elemenata nekog tipa `a` i vraća listu elemenata tog istog tipa. Funkcija `fst` prima uređeni par čije su komponente proizvoljnih tipova `a` i `b`, a vraća prvu komponentu, dakle vrednost tipa `a`.

Važno je primetiti da ista tipska promenljiva, kada se pojavljuje na više mesta u istom tipu, označava isti tip. Na primer, u tipu `[a] -> [a]` obe pojave promenljive `a` odnose se na isti tip. Zato funkcija `reverse` za listu celih brojeva vraća obrnutu listu, a za listu znakova obrnutu nisku.

Sa druge strane, različite tipske promenljive ne moraju označavati različite tipove, već samo nezavisne tipove. Na primer, u tipu `(a, b) -> a` promenljive `a` i `b` mogu predstavljati proizvoljne tipove, koji mogu biti isti, ali ne moraju.

Polimorfni tip funkcije često daje dobar uvid u njeno ponašanje. Tako se iz tipa funkcije `fst` odmah vidi da ona iz uređenog para izdvaja prvu komponentu, dok se iz tipa funkcije `reverse` vidi da ona ne menja tip elemenata liste, već samo njihov raspored.

Moguće je definisati i nešto složenije polimorfne funkcije. Na primer, funkcija koja od dve vrednosti pravi uređeni par može se definisati na sledeći način:

```
makePair :: a -> b -> (a, b)
makePair x y = (x, y)
```

Ovde tipske promenljive `a` i `b` predstavljaju tipove prve i druge komponente para.

Polimorfni tipovi omogućavaju da se jednom definisana funkcija koristi nad velikim brojem različitih tipova. Zbog toga su polimorfne funkcije veoma važne u Haskellu i često se pojavljuju među funkcijama iz prelida.

4.2 Klasna ograničenja i preopterećeni tipovi

U prethodnoj podsekciji videli smo da tipska promenljiva može predstavljati proizvoljan tip. Međutim, ponekad je potrebno da tipska promenljiva ne označava bilo koji tip, već samo tipove koji podržavaju određene operacije. Za to se u Haskellu koriste tipske klase i klasna ograničenja.

Tipska klasa predstavlja kolekciju tipova koji podržavaju određene operacije. Za tip koji pripada nekoj klasi kaže se da je instanca te klase. Na primer, numerički tipovi pripadaju klasi `Num`.

Kada je potrebno naglasiti da neka tipska promenljiva ne predstavlja proizvoljan tip, već samo tipove iz određene klase, koristi se klasno ograničenje. Ako je `C` ime klase, a `a` tipska promenljiva, tada zapis `C a` označava da tip `a` pripada klasi `C`.

U tipskim deklaracijama klasna ograničenja pišu se ispred samog tipa, odvojena simbolom `=>`. Na primer, tipski potpis operatora sabiranja ima oblik

```
(+) :: Num a => a -> a -> a
```

Ovaj tipski potpis znači da operator `+` prima dve vrednosti istog tipa `a` i vraća rezultat tog istog tipa, pri čemu tip `a` mora biti instanca klase `Num`. Drugim rečima, operator sabiranja može se koristiti nad bilo kojim numeričkim tipom.

Pošto se operator `+` uobičajeno koristi u infiksnom obliku, pri navođenju njegovog tipa stavlja se u zagrade, čime se posmatra kao obična funkcija. Na isti način i druge numeričke funkcije imaju klasna ograničenja:

```
(*)    :: Num a => a -> a -> a
negate :: Num a => a -> a
abs    :: Num a => a -> a
```

Tip koji sadrži jedno ili više klasnih ograničenja naziva se preopterećen tip (engl. *overloaded type*), a funkcija sa takvim tipom preopterećena funkcija (engl. *overloaded function*). Tako je tip `Num a => a -> a -> a` preopterećen tip, a operator `+` preopterećena funkcija.

Preopterećenost se ne javlja samo kod funkcija, već i kod numeričkih literala. Na primer, broj `5` može se posmatrati kao vrednost proizvoljnog numeričkog tipa:

```
5 :: Num a => a
```

To znači da ista konstanta može, u zavisnosti od konteksta u kome se koristi, biti interpretirana kao vrednost tipa `Int`, `Integer`, `Float` ili `Double`.

Klasna ograničenja omogućavaju da se tipske promenljive koriste na precizniji način, tj. ne nad svim tipovima, već samo nad onima koji pripadaju odgovarajućoj klasi. U narednoj podsekciji upoznaćemo najvažnije osnovne tipske klase u Haskellu.

4.3 Osnovne tipske klase

Haskell poseduje više ugrađenih tipskih klasa. U nastavku su opisane najvažnije osnovne klase koje se najčešće koriste pri radu sa standardnim tipovima i funkcijama iz prelida.

Klasa `Eq` sadrži tipove čije se vrednosti mogu porediti na jednakost i nejednakost. Njene osnovne operacije su operatori `==` i `/=`. Tipovi `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` i `Double` pripadaju klasi `Eq`. Isto važi i za liste i n-torke, ukoliko tipovi njihovih elemenata, odnosno komponenti, pripadaju istoj klasi. Na primer:

```
ghci> True == False
False

ghci> "abc" == "abc"
True

ghci> [1,2] /= [1,2,3]
True
```

Funkcijski tipovi u opštem slučaju ne pripadaju klasi `Eq`, jer nije moguće na opšti način porediti dve funkcije na jednakost.

Klasa `Ord` sadrži tipove čije se vrednosti mogu uređivati. Svaki tip koji pripada ovoj klasi mora ujedno pripadati i klasi `Eq`. Osnovne operacije klase `Ord` su operatori `<`, `<=`, `>`, `>=`, kao i funkcije `min` i `max`. Tipovi `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` i `Double` pripadaju klasi `Ord`. Pod analognim uslovima kao kod klase `Eq`, isto važi i za liste i n-torke. Na primer:

```
ghci> 3 < 5
True

ghci> min 'a' 'c'
'a'

ghci> "ana" < "anja"
True
```

Kod listi i n-torki poređenje se vrši leksikografski.

Klasa `Show` sadrži tipove čije se vrednosti mogu predstaviti kao niske karaktera. Njena osnovna funkcija jeste `show`. Na primer:

```
ghci> show True
"True"

ghci> show 123
"123"

ghci> show [1,2,3]
"[1,2,3]"
```

Klasa `Show` je naročito korisna u interaktivnom radu, jer omogućava prikazivanje vrednosti u čitljivom obliku.

Klasa `Read` je na izvestan način dualna klasi `Show`. Ona sadrži tipove čije se vrednosti mogu rekonstruisati iz niske karaktera pomoću funkcije `read`. Na primer:

```
ghci> read "False" :: Bool
False

ghci> read "[1,2,3]" :: [Int]
[1,2,3]
```

Pošto ista niska može odgovarati različitim tipovima, često je potrebno eksplicitno naznačiti

očekivani tip rezultata. Ipak, tip se nekada može zaključiti i iz konteksta. Na primer, u izrazu

```
not (read "False")
```

iz primene funkcije `not` sledi da izraz `read "False"` mora biti tipa `Bool`. Ako niska ne predstavlja ispravnu vrednost očekivanog tipa, primena funkcije `read` dovodi do greške pri izvršavanju.

Klasa `Num` sadrži tipove čije se vrednosti mogu obrađivati osnovnim aritmetičkim operacijama. U ovu klasu spadaju tipovi `Int`, `Integer`, `Float` i `Double`. Najvažnije operacije su `+`, `-`, `*`, kao i funkcije `negate`, `abs` i `signum`. Na primer:

```
ghci> 2 + 3
5

ghci> abs (-4)
4

ghci> signum (-7)
-1
```

Klasa `Num` ne sadrži operaciju deljenja, već se deljenje obrađuje posebnim numeričkim klasama.

Klasa `Integral` sadrži celobrojne tipove, kao što su `Int` i `Integer`. Pored osnovnih numeričkih operacija, ovi tipovi podržavaju i celobrojno deljenje i ostatak pri deljenju pomoću funkcija `div` i `mod`. Na primer:

```
ghci> 8 `div` 3
2

ghci> 8 `mod` 3
2
```

Klasa `Fractional` sadrži tipove koji podržavaju razlomljeno deljenje, kao što su `Float` i `Double`. Osnovne operacije ove klase su `/` i `recip`. Na primer:

```
ghci> 9.0 / 2.0
4.5

ghci> recip 4.0
0.25
```

Korisna funkcija pri radu sa numeričkim tipovima jeste `fromIntegral`, čiji je tip

```
fromIntegral :: (Integral a, Num b) => a -> b
```

Ovaj tipski potpis pokazuje da funkcija `fromIntegral` uzima vrednost tipa `a`, pri čemu tip `a` mora pripadati klasi `Integral`, i vraća vrednost tipa `b`, pri čemu tip `b` mora pripadati klasi `Num`. Drugim rečima, funkcija `fromIntegral` prevodi celobrojnu vrednost u numerički tip koji odgovara kontekstu.

U ovom tipskom potpisu pojavljuju se dva klasna ograničenja. Prvo ograničenje, `Integral a`, kaže da argument mora biti celobrojnog tipa, kao što su `Int` ili `Integer`. Drugo ograničenje, `Num b`, kaže da rezultat mora biti nekog numeričkog tipa, na primer `Int`, `Integer`, `Float` ili `Double`.

Funkcija `fromIntegral` naročito je korisna kada treba kombinovati celobrojne i razlomljene vrednosti. Na primer, funkcija `length` vraća rezultat tipa `Int`, pa izraz

```
length [1,2,3,4] + 3.2
```

nije ispravan, jer se u njemu pokušava sabiranje vrednosti tipa `Int` i vrednosti tipa `Double`. Međutim, izraz

```
fromIntegral (length [1,2,3,4]) + 3.2
```

jeste ispravan, jer funkcija `fromIntegral` prevodi rezultat funkcije `length` u odgovarajući numerički tip.

5 Liste i izlistavanje

5.1 Rasponi

Pri radu sa listama često je potrebno konstruisati listu uzastopnih vrednosti, na primer svih brojeva od 1 do 10 ili svih karaktera od `'a'` do `'z'`. U Haskellu se takve liste mogu zadati pomoću raspona (engl. *ranges*).

Najjednostavniji oblik raspona dobija se navođenjem početne i krajnje vrednosti, razdvojenih dvema tačkama. Na primer:

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]

ghci> [-3..3]
[-3,-2,-1,0,1,2,3]
```

Ako je krajnja vrednost manja od početne, a korak nije posebno naveden, rezultat je prazna lista. Na primer:

```
ghci> [10..1]
[]
```

Moguće je zadati i raspon sa korakom različitim od 1. U tom slučaju navode se prva dva elementa raspona, na osnovu kojih se određuje korak. Na primer:

```
ghci> [1,3..11]
[1,3,5,7,9,11]

ghci> [0,10..50]
[0,10,20,30,40,50]

ghci> [10,8..0]
[10,8,6,4,2,0]
```

Važno je primetiti da krajnja vrednost ne mora nužno biti deo rezultujuće liste. Na primer:

```
ghci> [1,5..11]
[1,5,9]
```

Sledeći član bi bio `13`, što prelazi zadatu granicu.

Rasponi se mogu koristiti i nad karakterima. Na primer:

```
ghci> ['A'..'D']
"ABCD"

ghci> ['a','c'..'k']
"acegik"
```

Pošto je lista karaktera u Haskellu isto što i niska, rezultati ovakvih izraza prikazuju se kao niske.

Treba biti oprezan pri korišćenju raspona sa racionalnim brojevima, jer zbog načina predstavljanja takvih brojeva u računaru rezultat može biti neočekivan. Na primer:

```
ghci> [0.1,0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Zbog toga se rasponi sa tipovima `Float` i `Double` uglavnom koriste pažljivo.

Ako se izostavi krajnja vrednost, dobija se beskonačan raspon. Na primer, izraz

```
[1..]
```

predstavlja beskonačnu listu svih prirodnih brojeva počev od `1`. Takve liste imaju smisla u Haskellu zbog lenjog izračunavanja – elementi liste izračunavaju se tek kada su potrebni.

Rasponi su korisni pri radu sa listama brojeva i karaktera, a naročito su pogodni za zadavanje pravilnih nizova vrednosti.

5.2 Još neke funkcije za rad sa listama

U prethodnim sekcijama već smo pomenuli neke osnovne funkcije za rad sa listama. Ovde ćemo navesti još nekoliko predefinisanih funkcija koje su korisne pri konstruisanju i obradi listi.

Funkcija `take` uzima kao prvi argument broj elemenata koje treba izdvojiti, a kao drugi listu iz koje se ti elementi uzimaju. Rezultat je početni deo liste zadate dužine. Na primer:

```
ghci> take 5 [1..]
[1,2,3,4,5]

ghci> take 6 [10,20..]
[10,20,30,40,50,60]
```

Funkcija `drop` radi slično, ali umesto izdvajanja početnog dela liste odbacuje zadati broj početnih elemenata. Na primer:

```
ghci> drop 3 [1,2,3,4,5,6]
[4,5,6]

ghci> drop 2 "abcdef"
"cdef"
```

Funkcija `cycle` uzima konačnu listu i od nje pravi beskonačnu listu tako što njene elemente ponavlja kružno. Na primer:

```
ghci> take 9 (cycle "abc")
"abcabcabc"
```

Pošto funkcija `cycle` daje beskonačnu listu, u praksi se najčešće koristi zajedno sa funkcijama poput `take`, kojima se izdvaja samo konačan početni deo takve liste.

Funkcija `repeat` od jedne vrednosti pravi beskonačnu listu u kojoj se ta vrednost neprekidno ponavlja. Na primer:

```
ghci> take 8 (repeat 5)
[5,5,5,5,5,5,5,5]
```

Kada nam nije potrebna beskonačna lista, već samo konačan broj ponavljanja istog elementa, pogodnije je koristiti funkciju `replicate`. Ona kao prvi argument prima broj ponavljanja,

a kao drugi element koji se ponavlja. Na primer:

```
ghci> replicate 5 'a'
"aaaaa"

ghci> replicate 4 10
[10,10,10,10]
```

Još jedna korisna funkcija jeste `zip`. Ona od dve liste pravi jednu listu uređenih parova, tako što uparuje njihove odgovarajuće elemente. Na primer:

```
ghci> zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]

ghci> zip [1..4] ["jedan","dva","tri","cetiri"]
[(1,"jedan"),(2,"dva"),(3,"tri"),(4,"cetiri")]
```

Ako liste nisu iste dužine, funkcija `zip` uparuje elemente samo dok za to postoje odgovarajući elementi u obe liste. Na primer:

```
ghci> zip [10,20,30,40] ['x','y']
[(10,'x'),(20,'y')]
```

Funkcija `zip` je korisna kada želimo da elemente neke liste povežemo sa njihovim rednim brojevima. Na primer:

```
ghci> zip [1..] ["Pon","Uto","Sre","Cet"]
[(1,"Pon"),(2,"Uto"),(3,"Sre"),(4,"Cet")]
```

Ovde je prva lista beskonačna, ali se zahvaljujući lenjom izračunavanju iz nje uzima samo onoliko elemenata koliko je potrebno za uparivanje sa drugom listom.

5.3 Izlistavanje

U matematici se često koristi zapis za zadavanje skupova pomoću svojstava njihovih elemenata. Na primer, skup kvadrata prirodnih brojeva od 1 do 5 može se zapisati u obliku

$$\{x^2 \mid x \in \mathbb{N},\ 1 \leq x \leq 5\}.$$

Time se označava skup svih vrednosti oblika x^2, pri čemu promenljiva x uzima prirodne vrednosti od 1 do 5. U Haskellu postoji slična sintaksa za zadavanje listi, koja se naziva izlistavanje (engl. *list comprehension*).

Osnovni oblik izlistavanja je

```
[izraz | generator]
```

Pri tome se sa desne strane uspravne crte navodi iz koje liste se uzimaju vrednosti, a sa leve strane izraz koji određuje elemente rezultujuće liste. Na primer:

```
ghci> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

U ovom slučaju promenljiva `x` redom uzima vrednosti iz liste `[1..5]`, a u rezultujuću listu upisuju se njihovi kvadrati.

Izraz oblika `x <- xs` naziva se generator. On određuje da promenljiva `x` redom uzima elemente liste `xs`. Na primer:

```
ghci> [x | x <- [3,1,4,1,5]]
[3,1,4,1,5]
```

Ovde izlistavanje samo ponavlja elemente polazne liste, jer je izraz sa leve strane isti kao promenljiva iz generatora.

U izlistavanju se mogu koristiti i uslovi. Oni se navode sa desne strane uspravne crte, posle generatora, i razdvajaju se zarezima. Na primer:

```
ghci> [x | x <- [1..10], even x]
[2,4,6,8,10]

ghci> [x^2 | x <- [1..10], even x]
[4,16,36,64,100]
```

U prvom primeru izdvajaju se svi parni brojevi iz liste `[1..10]`, a u drugom njihovi kvadrati.

Moguće je navesti i više uslova. Tada u rezultujuću listu ulaze samo oni elementi koji zadovoljavaju sve navedene uslove. Na primer:

```
ghci> [x | x <- [1..20], even x, x `mod` 3 == 0]
[6,12,18]
```

Izlistavanje može sadržati i više generatora. Tada se grade sve odgovarajuće kombinacije elemenata iz navedenih listi. Na primer:

```
ghci> [(x,y) | x <- [1,2,3], y <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Redosled generatora je važan, jer utiče na redosled elemenata u rezultujućoj listi. Na primer:

```
ghci> [(x,y) | y <- ['a','b'], x <- [1,2,3]]
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]
```

Kasniji generatori mogu zavisiti od ranijih. Na primer:

```
ghci> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Ovde vrednosti koje promenljiva `y` može da uzme zavise od trenutne vrednosti promenljive `x`.

Džoker `_` može se koristiti i u generatorima kada neka vrednost ne treba da se vezuje za ime. Na primer, funkcija koja računa dužinu liste može se definisati ovako:

```
length' :: [a] -> Int
length' xs = sum [1 | _ <- xs]
```

Ovde se svaki element liste zamenjuje brojem `1`, a zatim se svi ti brojevi saberu.

Pošto su niske u Haskellu liste karaktera, izlistavanje se može koristiti i pri radu sa niskama.

Na primer:

```
removeNonUppercase :: String -> String
removeNonUppercase xs = [x | x <- xs, x >= 'A' && x <= 'Z']
```

Tako, na primer, izraz `removeNonUppercase "HaSkElL"` ima vrednost `"HSEL"`.

Izlistavanje predstavlja pregledan način za konstruisanje novih listi iz postojećih. Posebno je korisno kada želimo da nad elementima neke liste izvršimo jednostavnu transformaciju, izdvajanje po uslovu ili kombinovanje elemenata iz više listi. Zbog svoje sažetosti i preglednosti, ova sintaksa se veoma često koristi pri radu sa listama i niskama.

5.4 Cezarova šifra

Kao primer primene funkcija nad listama, raspona, funkcije `zip` i izlistavanja, razmotrićemo jednostavnu Cezarovu šifru. Ideja ove šifre jeste da se svako veliko slovo engleskog alfabeta zameni slovom koje se nalazi za n ($0 \leq n \leq 25$) mesta dalje u alfabetu, pri čemu se po dolasku do kraja alfabeta nastavlja od slova `A`. Tako, na primer, za pomeraj 3 slovo `A` prelazi u `D`, slovo `B` u `E`, a slovo `Z` u `C`.

Najpre ćemo za zadati pomeraj n konstruisati listu parova koja svakom slovu pridružuje odgovarajuće šifrovano slovo:

```
cipher :: Int -> [(Char, Char)]
cipher n = zip letters shifted
  where
    shifted = drop n letters ++ take n letters
    letters = ['A'..'Z']
```

Ovde je `letters` lista svih velikih slova engleskog alfabeta. Izraz `drop n letters` uklanja prvih `n` slova, dok izraz `take n letters` uzima upravo tih prvih `n` slova. Njihovim nadovezivanjem dobija se alfabet pomeren za `n` mesta ulevo. Funkcija `zip` zatim uparuje svako slovo sa njegovom šifrovanom verzijom.

Na primer:

```
ghci> cipher 3
[('A','D'),('B','E'),('C','F'),('D','G'),('E','H'),('F','I'),('G','J'),
('H','K'),('I','L'),('J','M'),('K','N'),('L','O'),('M','P'),('N','Q'),
('O','R'),('P','S'),('Q','T'),('R','U'),('S','V'),('T','W'),('U','X'),
('V','Y'),('W','Z'),('X','A'),('Y','B'),('Z','C')]
```

Sada možemo definisati funkciju koja šifruje jedan karakter. Ako se prosleđeni karakter pojavljuje kao prva komponenta nekog para u listi `cipher n`, tada vraćamo odgovarajuću drugu komponentu. U suprotnom vraćamo isti karakter, čime se razmaci i drugi znakovi ostavljaju nepromenjenim.

```
encodeChar :: Int -> Char -> Char
encodeChar n c
  | matches == [] = c
  | otherwise     = snd (head matches)
  where
    matches = [(x,y) | (x,y) <- cipher n, x == c]
```

Na primer:

```
ghci> encodeChar 3 'A'
'D'

ghci> encodeChar 3 'Z'
'C'

ghci> encodeChar 3 ' '
' '
```

Šifrovanje cele niske sada je jednostavno, jer je dovoljno šifrovati svaki njen karakter. To možemo uraditi izlistavanjem:

```
encode :: Int -> String -> String
encode n xs = [encodeChar n x | x <- xs]
```

Na primer:

```
ghci> encode 5 "HELLO WORLD"
"MJQQT BTWQI"
```

Dešifrovanje se može definisati potpuno analogno. Ovoga puta tražimo par u kome je dati karakter druga komponenta, a vraćamo prvu:

```
decodeChar :: Int -> Char -> Char
decodeChar n c
  | matches == [] = c
  | otherwise     = fst (head matches)
  where
    matches = [(x,y) | (x,y) <- cipher n, y == c]
```

Dešifrovanje cele niske opet dobijamo primenom na svaki karakter:

```
decode :: Int -> String -> String
decode n xs = [decodeChar n x | x <- xs]
```

Na primer:

```
ghci> decode 5 "MJQQT BTWQI"
"HELLO WORLD"
```

6 Rekurzija

6.1 Osnovni koncepti

Rekurzija je način definisanja funkcija pri kome se vrednost funkcije izražava pomoću vrednosti te iste funkcije na jednostavnijim argumentima. Da bi takva definicija bila ispravna, potrebno je navesti i bazne slučajeve, odnosno slučajeve u kojima se rezultat dobija neposredno, bez daljih rekurzivnih poziva.

U Haskellu je rekurzija naročito važna, jer se mnoge funkcije prirodno definišu upravo na ovaj način. Umesto da se problem rešava ponavljanjem koraka pomoću petlji, on se svodi na isti takav, ali jednostavniji problem.

Jedan od klasičnih primera jeste faktorijel prirodnog broja. Podsetimo se,

$$0! = 1, \qquad n! = n \cdot (n-1)!$$

za svako $n > 0$. Ova definicija se neposredno prevodi u Haskell:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

U ovoj definiciji prva jednačina zadaje bazni slučaj, a druga opisuje kako se vrednost funkcije za argument `n` dobija preko vrednosti iste funkcije za manji argument `n - 1`.

Na primer, izračunavanje izraza `factorial 4` odvija se ovako:

```
factorial 4
= 4 * factorial 3
= 4 * (3 * factorial 2)
= 4 * (3 * (2 * factorial 1))
= 4 * (3 * (2 * (1 * factorial 0)))
= 4 * (3 * (2 * (1 * 1)))
= 24
```

Vidimo da se pri svakom koraku argument smanjuje za `1`, sve dok se ne dođe do baznog slučaja.

Još jedan važan primer jeste Fibonačijev niz. Njegovi prvi članovi su

$$0, 1, 1, 2, 3, 5, 8, \ldots$$

pri čemu je svaki naredni član jednak zbiru prethodna dva. Funkcija koja vraća n-ti Fibonačijev broj može se definisati ovako:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Ovde postoje dva bazna slučaja, za argumente `0` i `1`. Za svaku veću vrednost rezultat se dobija sabiranjem prethodna dva Fibonačijeva broja.

Važno je primetiti da rekurzivna definicija mora biti postavljena tako da se rekurzivni pozivi približavaju nekom baznom slučaju. U suprotnom, izračunavanje se ne bi završilo. Zato se rekurzivna definicija po pravilu sastoji od baznih slučajeva i opšteg slučaja koji problem svodi na jednostavnije instance istog problema.

6.2 Rekurzivne funkcije nad listama

Rekurzija je naročito prirodna pri radu sa listama. Razlog je to što se svaka lista može posmatrati na dva osnovna načina: ili je prazna, ili je neprazna i tada je oblika `x:xs`, gde je `x` prvi element liste, a `xs` njen ostatak. Zbog toga rekurzivne definicije nad listama najčešće imaju jedan bazni slučaj za praznu listu i jedan rekurzivni slučaj za nepraznu listu.

Na primer, funkcija slična bibliotečkoj funkciji `sum`, koja računa zbir elemenata liste brojeva, može se definisati ovako:

```
sum' :: Num a => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

U ovoj definiciji bazni slučaj kaže da je zbir prazne liste jednak `0`, dok se zbir neprazne liste dobija kao zbir njenog prvog elementa i zbira ostatka liste.

Na primer, izračunavanje izraza `sum' [2,3,4]` odvija se ovako:

```
sum' [2,3,4]
= 2 + sum' [3,4]
= 2 + (3 + sum' [4])
= 2 + (3 + (4 + sum' []))
= 2 + (3 + (4 + 0))
= 9
```

Slično tome, funkcija slična bibliotečkoj funkciji `maximum`, koja vraća najveći element neprazne liste, može se definisati na sledeći način:

```
maximum' :: Ord a => [a] -> a
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Ovde je bazni slučaj jednočlana lista: njen najveći element jeste njen jedini element. U rekurzivnom slučaju najveći element liste dobija se kao veći od brojeva `x` i `maximum' xs`. Primetimo da ova funkcija nije definisana za praznu listu.

Na isti način može se definisati i funkcija slična bibliotečkoj funkciji `length`:

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

Dužina prazne liste jednaka je `0`, dok se dužina neprazne liste dobija tako što se na dužinu njenog repa doda `1`. U rekurzivnom slučaju koristi se džoker `_`, jer vrednost prvog elementa nije važna za izračunavanje dužine.

Funkcija slična bibliotečkoj funkciji `reverse` može se definisati ovako:

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Bazni slučaj kaže da je obrnuta prazna lista opet prazna lista. U rekurzivnom slučaju prvo se obrće rep liste, a zatim se prvi element dodaje na kraj dobijene liste.

Na primer, izračunavanje izraza `reverse' [1,2,3]` izgleda ovako:

```
reverse' [1,2,3]
= reverse' [2,3] ++ [1]
= (reverse' [3] ++ [2]) ++ [1]
= ((reverse' [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= [3,2,1]
```

Rekurzija se može koristiti i kod funkcija koje rade sa više listi istovremeno. Na primer, funkcija slična bibliotečkoj funkciji `zip` može se definisati na sledeći način:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' [] _ = []
zip' _ [] = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

Ovde postoje dva bazna slučaja: ako je bilo koja od dve liste prazna, rezultat je prazna lista. U rekurzivnom slučaju formira se par od prvih elemenata obeju listi, a zatim se funkcija primenjuje na njihove repove.

Rekurzivne definicije nad listama najčešće prate istu osnovnu ideju: najpre se obrade najjednostavniji oblici liste, a zatim se opšti slučaj svodi na isti problem nad kraćom listom. Upravo zbog toga rekurzija i podudaranje oblika veoma prirodno idu zajedno pri radu sa listama.

6.3 Sortiranje pomoću rekurzije

Rekurzija se prirodno može koristiti i za definisanje funkcija koje sortiraju liste. Ideja je da se sortiranje složenije liste svede na sortiranje njenih jednostavnijih delova.

Jedan od najjednostavnijih pristupa zasniva se na pomoćnoj funkciji koja ubacuje novi element na odgovarajuće mesto u već sortiranu listu. Funkcija slična bibliotečkoj funkciji `insert` može se definisati ovako:

```
insert' :: Ord a => a -> [a] -> [a]
insert' x [] = [x]
insert' x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert' x ys
```

U baznom slučaju, umetanje elementa u praznu listu daje jednočlanu listu. Ako je lista neprazna, poredi se element `x` sa njenim prvim elementom `y`. Ako je `x <= y`, tada se `x` postavlja na početak liste. U suprotnom, prvi element ostaje na svom mestu, a umetanje se nastavlja u rep liste.

Pomoću ove funkcije može se definisati sortiranje umetanjem:

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert' x (isort xs)
```

Bazni slučaj kaže da je prazna lista već sortirana. U rekurzivnom slučaju najpre se sortira rep liste, a zatim se prvi element umeće na odgovarajuće mesto u tako dobijenu sortiranu listu.

Drugi poznat rekurzivni postupak sortiranja jeste brzo sortiranje. Funkcija koja implementira algoritam brzog sortiranja može se definisati na sledeći način:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [y | y <- xs, y <= x]
    larger  = [y | y <- xs, y > x]
```

Bazni slučaj je ponovo prazna lista. Ako je lista neprazna, njen prvi element `x` uzima se kao razdvajajući element. Zatim se ostali elementi dele na dve liste: u listi `smaller` nalaze se svi elementi manji ili jednaki od `x`, a u listi `larger` svi elementi strogo veći od `x`. Nakon toga se obe liste rekurzivno sortiraju, a rezultat se dobija njihovim spajanjem sa elementom `x` između njih.

Kod brzog sortiranja važno je primetiti da se u rekurzivnom slučaju funkcija poziva dva puta: jednom nad listom manjih ili jednakih elemenata, a drugi put nad listom većih elemenata. Time se početni problem razlaže na dva manja problema istog tipa.

Sortiranje umetanjem i brzo sortiranje lepo pokazuju da se različite ideje sortiranja mogu izraziti vrlo pregledno pomoću rekurzije. U prvom slučaju lista se gradi postepenim umetanjem elemenata na pravo mesto, dok se u drugom slučaju lista najpre razdvaja na manje delove, koji se zatim zasebno sortiraju.

6.4 Uzajamna rekurzija

Do sada smo posmatrali primere u kojima je funkcija definisana rekurzivno pomoću same sebe. Moguća je i nešto drugačija situacija, u kojoj se dve funkcije definišu rekurzivno jedna pomoću druge. Takva pojava naziva se uzajamna rekurzija.

U prelidu već postoje funkcije `even` i `odd`, koje određuju da li je ceo broj paran, odnosno neparan. Ovde ćemo definisati njima slične funkcije `even'` i `odd'` samo radi ilustracije uzajamne rekurzije. Ovakve definicije nisu najjednostavniji način da se proveri parnost broja, ali jasno pokazuju kako dve funkcije mogu biti definisane jedna pomoću druge.

Ideja je sledeća: broj `0` je paran, a svaki drugi broj je paran ako i samo ako je njegov prethodnik neparan. Slično tome, broj `0` nije neparan, a svaki drugi broj je neparan ako i samo ako je njegov prethodnik paran. To se može zapisati ovako:

```
even' :: Int -> Bool
even' 0 = True
even' n = odd' (n - 1)

odd' :: Int -> Bool
odd' 0 = False
odd' n = even' (n - 1)
```

Ovde su bazni slučajevi dati za argument `0`. U ostalim slučajevima funkcija `even'` poziva funkciju `odd'`, a funkcija `odd'` poziva funkciju `even'`. Na taj način svaka od njih svodi problem na proveru manjeg broja, sve dok se ne dođe do baznog slučaja.

Na primer, izračunavanje izraza `even' 4` odvija se ovako:

```
even' 4
= odd' 3
= even' 2
```

```
= odd' 1
= even' 0
= True
```

Slično, za izraz `odd' 4` dobija se:

```
odd' 4
= even' 3
= odd' 2
= even' 1
= odd' 0
= False
```

Važno je primetiti da su ove definicije ispravne samo za nenegativne brojeve. Za negativne argumente rekurzivni pozivi ne bi vodili ka baznom slučaju, pa se izračunavanje ne bi završilo.

Uzajamna rekurzija pokazuje da rekurzivna definicija ne mora uvek biti zasnovana na jednoj funkciji. Ponekad je prirodnije da se više međusobno povezanih pojmova definiše istovremeno, tako što se njihove definicije oslanjaju jedna na drugu.

6.5 Hanojske kule

Kao završni primer razmotrimo problem Hanojskih kula. Na raspolaganju su tri štapa, koje ćemo označiti sa `A`, `B` i `C`, i n diskova različitih veličina. Na početku su svi diskovi složeni na štapu `A`, od najmanjeg pri vrhu do najvećeg pri dnu. Cilj je da se svi diskovi prebace na štap `C`, pri čemu se u svakom potezu sme pomeriti samo jedan disk, i veći disk nikada ne sme biti postavljen na manji.

Rešenje ovog problema možemo predstaviti kao listu poteza. Svaki potez biće uređen par štapova koji označava sa kog se štapa disk prebacuje i na koji se štap postavlja. Na primer, par `('A','C')` označava potez u kome se gornji disk prebacuje sa štapa `A` na štap `C`. Zbog toga će jedno rešenje biti predstavljeno listom tipa `[(Char, Char)]`.

Ako treba prebaciti samo jedan disk, rešenje je neposredno: dovoljno je napraviti jedan potez sa početnog na krajnji štap. To je bazni slučaj. Suština problema je, dakle, u tome kako prebaciti više od jednog diska.

Pretpostavimo zato da treba prebaciti n diskova sa nekog početnog štapa na neki krajnji štap, uz pomoć trećeg štapa. Tada se problem prirodno razlaže na tri koraka:

1. najpre se gornjih $n-1$ diskova prebaci sa početnog na pomoćni štap;
2. zatim se najveći disk prebaci sa početnog na krajnji štap;
3. na kraju se tih $n-1$ diskova prebaci sa pomoćnog na krajnji štap.

Ključna ideja je da su prvi i treći korak opet isti problem, samo za manji broj diskova. Upravo zato je ovaj zadatak pogodan za rekurzivno rešavanje.

Pomoćna funkcija zato treba da primi broj diskova, početni štap, krajnji štap i pomoćni štap, a da kao rezultat vrati listu svih potrebnih poteza:

```
move :: Int -> Char -> Char -> Char -> [(Char, Char)]
move 1 source target _ = [(source, target)]
move n source target auxiliary =
  part1 ++ [(source, target)] ++ part2
```

```
  where
    part1 = move (n - 1) source auxiliary target
    part2 = move (n - 1) auxiliary target source
```

U baznom slučaju, kada postoji samo jedan disk, rezultat je jednočlana lista sa odgovarajućim potezom. U rekurzivnom slučaju najpre se izračunavaju potezi kojima se $n-1$ diskova prebacuje na pomoćni štap, zatim se dodaje potez kojim se najveći disk prebacuje na krajnji štap, a potom se izračunavaju potezi kojima se tih $n-1$ diskova prebacuje sa pomoćnog na krajnji štap.

Na osnovu ove pomoćne funkcije lako se definiše i glavna funkcija, koja rešava početni problem: prebacivanje svih diskova sa štapa `A` na štap `C`, uz štap `B` kao pomoćni:

```
hanoi :: Int -> [(Char, Char)]
hanoi n = move n 'A' 'C' 'B'
```

Na primer, za dva diska dobijamo:

```
ghci> hanoi 2
[('A','B'),('A','C'),('B','C')]
```

Za tri diska dobijamo:

```
ghci> hanoi 3
[('A','C'),('A','B'),('C','B'),('A','C'),('B','A'),('B','C'),('A','C')]
```

Problem Hanojskih kula lepo pokazuje snagu rekurzije: složen zadatak razlaže se na dva manja zadatka istog tipa i jedan jednostavan potez između njih. Upravo ovakvo svođenje problema na manje instance istog problema predstavlja jedno od osnovnih svojstava rekurzivnog pristupa.

7 Funkcije višeg reda

7.1 Funkcije kao argumenti

Prethodno smo videli da se funkcije sa više argumenata u Haskellu tumače tako da argumente primaju jedan po jedan. Zbog toga se funkcije oblika `A -> B -> C` mogu posmatrati kao funkcije koje primaju argument tipa `A` i vraćaju funkciju tipa `B -> C`. Takve funkcije nazivaju se karirane funkcije (engl. *curried functions*).

Pored toga što funkcija može vratiti drugu funkciju kao rezultat, ona može i primiti funkciju kao argument. Formalno, funkcije koje primaju funkcije kao argumente ili vraćaju funkcije kao rezultate nazivaju se funkcije višeg reda. U praksi se, međutim, izraz funkcija višeg reda često koristi pre svega za funkcije koje primaju druge funkcije kao argumente, jer za funkcije koje vraćaju funkcije već imamo pojam kariranih funkcija. U ovoj sekciji pažnja će biti usmerena upravo na funkcije koje primaju druge funkcije kao argumente.

Jednostavan primer je funkcija koja zadatu funkciju primenjuje dva puta na istu vrednost:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Prvi argument funkcije `twice` jeste funkcija tipa `a -> a`, a drugi argument vrednost tipa `a`. Rezultat se dobija tako što se funkcija `f` najpre primeni na vrednost `x`, a zatim još jednom na dobijeni rezultat.

Na primer:

```
ghci> twice reverse [1,2,3]
[1,2,3]

ghci> twice (\x -> x + 3) 10
16
```

U prvom primeru funkcija `reverse` primenjuje se dva puta na istu listu, pa se dobija početna lista. U drugom primeru lambda izraz `\x -> x + 3` najpre preslikava broj `10` u `13`, a zatim broj `13` u `16`.

Funkcije višeg reda omogućavaju da se opšti obrasci računanja izdvoje i zapišu jednom, a zatim primenjuju u različitim situacijama. Zahvaljujući tome, mnoge operacije nad listama mogu se zapisati kraće i preglednije.

7.2 Funkcije map, filter i zipWith

Među najvažnijim funkcijama višeg reda za rad sa listama nalaze se funkcije `map`, `filter` i `zipWith`. One omogućavaju da se česte operacije nad listama zapišu pregledno i bez eksplicitne rekurzije, iako se same mogu definisati upravo rekurzivno.

Funkcija `map` primenjuje zadatu funkciju na svaki element liste i vraća listu dobijenih rezultata. Njen tip je

```
map :: (a -> b) -> [a] -> [b]
```

Prvi argument funkcije `map` jeste funkcija tipa `a -> b`, a drugi lista elemenata tipa `[a]`. Rezultat je lista tipa `[b]`, dobijena primenom zadate funkcije na svaki element polazne liste.

Na primer:

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]

ghci> map even [1,2,3,4]
[False,True,False,True]

ghci> map reverse ["abc","def","ghi"]
["cba","fed","ihg"]
```

U prvom primeru koristi se zapis `(+1)`, koji označava funkciju koja svom argumentu dodaje `1`. Slično, izrazi poput `(*2)` ili `(/2)` takođe označavaju funkcije dobijene parcijalnom primenom operatora.

Funkcija `map` može se primeniti i na ugnježdene liste. Na primer:

```
ghci> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

Ovde spoljašnja funkcija `map` prolazi kroz listu listi, dok unutrašnja funkcija `map (+1)` u svakoj od tih listi uvećava svaki element za `1`.

Funkcija analogna bibliotečkoj funkciji `map` može se definisati i pomoću izlistavanja:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = [f x | x <- xs]
```

Ista ideja može se zapisati i rekurzivno:

```
map'' :: (a -> b) -> [a] -> [b]
map'' f [] = []
map'' f (x:xs) = f x : map'' f xs
```

Funkcija `filter` izdvaja iz liste samo one elemente koji zadovoljavaju dati uslov. Njen tip je

```
filter :: (a -> Bool) -> [a] -> [a]
```

Prvi argument funkcije `filter` jeste predikat, odnosno funkcija koja za dati element vraća logičku vrednost. U rezultujućoj listi ostaju samo oni elementi za koje je vrednost predikata `True`.

Na primer:

```
ghci> filter even [1..10]
[2,4,6,8,10]

ghci> filter (>5) [1..10]
[6,7,8,9,10]

ghci> filter (/=' ') "abc def ghi"
"abcdefghi"
```

Funkcija analogna bibliotečkoj funkciji `filter` može se definisati pomoću izlistavanja:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p xs = [x | x <- xs, p x]
```

Može se definisati i rekurzivno:

```
filter'' :: (a -> Bool) -> [a] -> [a]
filter'' p [] = []
filter'' p (x:xs)
  | p x       = x : filter'' p xs
  | otherwise = filter'' p xs
```

Funkcije `map` i `filter` često se koriste zajedno. Na primer, zbir kvadrata parnih brojeva iz liste može se definisati ovako:

```
sumSqEven :: [Int] -> Int
sumSqEven xs = sum (map (^2) (filter even xs))
```

Ovde se najpre funkcijom `filter even` izdvajaju parni brojevi iz liste, a zatim se funkcijom `map (^2)` svaki od njih kvadrira. Na kraju se dobijeni brojevi sabiraju funkcijom `sum`.

Funkcija `zipWith` predstavlja uopštenje funkcije `zip`. Dok `zip` od dve liste pravi listu parova, `zipWith` nad odgovarajućim elementima dve liste primenjuje zadatu funkciju. Njen tip je

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Prvi argument funkcije `zipWith` jeste funkcija koja prima dva argumenta, drugi i treći argument su dve liste, a rezultat je lista dobijena primenom te funkcije na odgovarajuće elemente polaznih listi.

Na primer:

```
ghci> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]

ghci> zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]

ghci> zipWith (++) ["foo","bar","baz"] ["1","2","3"]
["foo1","bar2","baz3"]
```

Funkcija analogna bibliotečkoj funkciji `zipWith` može se definisati rekurzivno ovako:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Kao i funkcija `zip`, funkcija `zipWith` obrađuje elemente dveju listi redom, sve dok se jedna od njih ne iscrpi. Zbog toga dužina rezultujuće liste odgovara dužini kraće od polaznih listi.

7.3 Funkcije foldr i foldl

Mnoge funkcije nad listama imaju zajedničku strukturu: praznoj listi pridružuje se neka početna vrednost, a neprazna lista obrađuje se tako što se njen prvi element kombinuje sa rezultatom dobijenim obradom ostatka liste. Funkcije `foldr` i `foldl` izdvajaju upravo taj obrazac u opšti oblik, pa omogućavaju da se brojne funkcije nad listama zapišu kraće i preglednije.

Funkcija `foldr` obrađuje listu zdesna nalevo. Ona prima tri argumenta: binarnu funkciju, početnu vrednost i listu. Na primer, izraz

```
foldr (+) 0 [1,2,3]
```

odgovara izrazu

```
1 + (2 + (3 + 0))
```

Dakle, funkcija `foldr` polazi od kraja liste i postepeno ugrađuje elemente ulevo.

Zbog toga se zbir elemenata liste može definisati ovako:

```
sum' :: Num a => [a] -> a
sum' = foldr (+) 0
```

Slično tome, mogu se definisati i druge poznate funkcije:

```
product' :: Num a => [a] -> a
product' = foldr (*) 1

or' :: [Bool] -> Bool
or' = foldr (||) False

and' :: [Bool] -> Bool
and' = foldr (&&) True
```

U svim ovim primerima obrazac je isti: navodi se operacija kojom se elementi kombinuju i vrednost koja odgovara praznoj listi.

Funkcija `foldl` radi slično, ali listu obrađuje sleva nadesno. I ona prima binarnu funkciju, početnu vrednost i listu. Na primer, izraz

```
foldl (+) 0 [1,2,3]
```

odgovara izrazu

```
((0 + 1) + 2) + 3
```

Ovde se, dakle, rezultat akumulira počev od leve strane liste.

Zbir se zato može definisati i pomoću funkcije `foldl`:

```
sum'' :: Num a => [a] -> a
sum'' = foldl (+) 0
```

U ovom slučaju funkcije `foldr` i `foldl` daju isti rezultat. Međutim, to u opštem slučaju ne mora biti tako. Na primer:

```
ghci> foldr (/) 1 [2,3]
```

```
0.6666666666666666

ghci> foldl (/) 1 [2,3]
0.16666666666666666
```

Zaista, ovde važi:

```
foldr (/) 1 [2,3] = 2 / (3 / 1)
foldl (/) 1 [2,3] = (1 / 2) / 3
```

Pošto deljenje nije asocijativno, različit redosled grupisanja daje različite rezultate.

Funkcije `foldr` i `foldl` ne moraju davati samo brojeve ili logičke vrednosti. Njima se mogu definisati i funkcije čiji je rezultat nova lista. Na primer, obrtanje liste može se zapisati pomoću funkcije `foldl` ovako:

```
reverse' :: [a] -> [a]
reverse' = foldl (\xs x -> x : xs) []
```

Ovde je početna vrednost prazna lista. Zatim se elementi redom uzimaju sleva nadesno i svaki novi element dodaje se na početak akumulirane liste. Zbog toga se redosled elemenata obrće. Na primer:

```
foldl (\xs x -> x : xs) [] [1,2,3]
= foldl (\xs x -> x : xs) (1:[]) [2,3]
= foldl (\xs x -> x : xs) (2:[1]) [3]
= foldl (\xs x -> x : xs) (3:[2,1]) []
= [3,2,1]
```

Slično tome, funkcija `length` može se definisati pomoću obe funkcije:

```
length' :: [a] -> Int
length' = foldr (\_ n -> 1 + n) 0

length'' :: [a] -> Int
length'' = foldl (\n _ -> n + 1) 0
```

U prvoj definiciji funkcija `foldr` prolazi kroz listu zdesna nalevo i za svaki element uvećava broj za `1`. Vrednost samog elementa nije važna, pa se koristi džoker `_`. U drugoj definiciji funkcija `foldl` koristi akumulator koji se pri svakom koraku uvećava za `1`. U oba slučaja rezultat je broj elemenata liste.

Ponekad nije zgodno zadavati posebnu početnu vrednost. Tada se koriste funkcije `foldr1` i `foldl1`, koje ne dobijaju početnu vrednost kao poseban argument, već je uzimaju iz same liste, `foldr1` od njenog poslednjeg, a `foldl1` od njenog prvog elementa. Na primer, maksimum i minimum neprazne liste mogu se definisati ovako:

```
maximum' :: Ord a => [a] -> a
maximum' = foldl1 max

minimum' :: Ord a => [a] -> a
minimum' = foldl1 min
```

Na primer:

```
ghci> maximum' [3,1,4,2]
4

ghci> minimum' [3,1,4,2]
1
```

Za razliku od funkcija `foldr` i `foldl`, funkcije `foldr1` i `foldl1` nisu definisane za praznu listu, jer tada ne postoji element koji bi mogao poslužiti kao početna vrednost.

7.4 Kompozicija funkcija

Operator kompozicije `.` omogućava da se dve funkcije spoje u jednu novu funkciju. Ako su `g` i `f` funkcije odgovarajućih tipova, tada izraz `f . g` označava funkciju koja najpre primenjuje `g`, a zatim na dobijeni rezultat primenjuje `f`.

Operator kompozicije može se zadati na sledeći način:

```
(f . g) x = f (g x)
```

Ekvivalentno, može se pisati i

```
f . g = \x -> f (g x)
```

pri čemu drugi zapis eksplicitno pokazuje da kompozicija dve funkcije ponovo daje funkciju.

Na primer, ako želimo funkciju koja broj najpre udvostručuje, a zatim mu menja znak, možemo napisati:

```
ghci> (negate . (*2)) 3
-6
```

Kompozicija je naročito korisna kada želimo da pojednostavimo ugnježdene primene funkcija. Na primer, definicija

```
odd' x = not (even x)
```

može se zapisati kraće kao

```
odd' = not . even
```

Slično tome, funkcija koja dva puta primenjuje zadatu funkciju može se zapisati kao

```
twice f = f . f
```

Kompozicija se može koristiti i za pregledniji zapis izraza sa više funkcija. Na primer, funkcija koja računa zbir kvadrata parnih brojeva iz liste može se definisati kao

```
sumSqEven :: [Int] -> Int
sumSqEven = sum . map (^2) . filter even
```

Ova definicija znači da se iz liste najpre izdvajaju parni elementi, zatim se svaki od njih kvadrira, a na kraju se svi tako dobijeni brojevi sabiraju.

Važno je primetiti da je kompozicija funkcija asocijativna, odnosno da za funkcije odgovarajućih tipova važi

```
f . (g . h) = (f . g) . h
```

Pored operatora kompozicije, u Haskellu se često koristi i operator `$`. Njegova definicija je:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Na prvi pogled, operator `$` ne uvodi novu operaciju, jer izraz `f $ x` ima isto značenje kao `f x`. Njegov značaj je u tome što ima veoma nizak prioritet. Zbog toga se često koristi da zameni spoljašnje zagrade u izrazima sa više ugnježdenih primena funkcija.

Na primer, umesto

```
sum (map (^2) [1..5])
```

može se pisati

```
sum $ map (^2) [1..5]
```

Slično tome, izraz

```
sum (map (^2) (filter odd [1..10]))
```

može se zapisati preglednije kao

```
sum $ map (^2) $ filter odd [1..10]
```

Operator `$` grupiše se zdesna nalevo, pa se poslednji izraz tumači isto kao i odgovarajući izraz sa zagradama. Zato se `$` najčešće koristi kao sintaksna pogodnost za smanjenje broja zagrada, dok operator `.` služi za građenje nove funkcije kompozicijom postojećih funkcija.

8 Definisanje novih tipova

8.1 Tipski sinonimi

Najjednostavniji način da se uvede novo ime za već postojeći tip je upotreba deklaracije `type`. Na taj način se ne konstruiše novi tip, već se samo postojećem tipu dodeljuje drugo ime. Takvo ime naziva se tipski sinonim.

Opšti oblik deklaracije je:

```
type NovoIme = PostojeciTip
```

Ime tipskog sinonima mora počinjati velikim slovom. Pošto se ovde ne uvodi novi tip, već samo novo ime za stari, vrednosti tog tipa ostaju iste kao i ranije.

Najpoznatiji primer iz prelida jeste:

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

Ova deklaracija kaže da je `String` samo drugo ime za tip `[Char]`, odnosno za liste karaktera.

Glavna svrha tipskih sinonima jeste pregledniji zapis tipova. Na primer, uređeni par realnih brojeva može predstavljati tačku u ravni, pa možemo napisati:

```
type Point = (Float, Float)
```

Tada definicija funkcije za translaciju tačke može izgledati ovako:

```
shiftX :: Float -> Point -> Point
shiftX d (x, y) = (x + d, y)
```

Bez tipskog sinonima isti tip bi morao da se piše direktno kao `(Float, Float)`, što je manje pregledno i teže za čitanje.

Tipski sinonimi mogu se definisati i pomoću drugih tipskih sinonima. Na primer, ako želimo da uvedemo tip za funkcije koje preslikavaju tačke u tačke, možemo napisati:

```
type Transform = Point -> Point
```

Sada `Transform` označava tip funkcija koje menjaju položaj tačke.

Tipski sinonimi mogu biti i parametrizovani. Na primer, sinonim za uređeni par elemenata istog tipa može se definisati ovako:

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

Tako su `Pair Int` i `Pair Bool` samo kraći zapisi za tipove `(Int, Int)` i `(Bool, Bool)`.

Mogući su i sinonimi sa više parametara. Na primer, asocijativnu listu, odnosno listu parova ključ–vrednost, možemo zapisati ovako:

```
type Assoc k v = [(k, v)]
```

Ovde `Assoc k v` označava listu parova u kojoj se vrednostima tipa `k` pridružuju vrednosti tipa `v`. Na primer, funkcija `containsKey` može se definisati ovako:

```
containsKey :: Eq k => k -> Assoc k v -> Bool
containsKey k xs = or [k == k' | (k', _) <- xs]
```

Ovaj primer pokazuje i jednu važnu osobinu tipskih sinonima: oni ne uvode nove konstruktore niti nove vrednosti, već samo omogućavaju da se postojeći tipovi koriste pod pogodnijim imenima.

Važno je naglasiti da tipski sinonim ne pravi novi tip. Na primer, `String` i `[Char]` predstavljaju isti tip, baš kao što `Point` i `(Float, Float)` predstavljaju isti tip. Zato se sve funkcije koje važe za originalni tip mogu bez ikakvih izmena koristiti i nad njegovim tipskim sinonimom. Zbog toga se tipski sinonimi koriste radi jasnijeg značenja i preglednijeg zapisa, a ne radi uvođenja novih tipovskih ograničenja.

8.2 Deklaracije pomoću data

Za razliku od tipskih sinonima, koji samo uvode novo ime za već postojeći tip, deklaracije pomoću ključne reči `data` uvode zaista nov tip. Takvi tipovi nazivaju se algebarski tipovi podataka. Njihove vrednosti zadaju se pomoću konstruktora, pa se na taj način precizno određuje koji oblici podataka pripadaju novom tipu.

Opšti oblik ovakve deklaracije je:

```
data ImeTipa = Konstruktor1 | Konstruktor2 | ...
```

Ime tipa i imena konstruktora moraju počinjati velikim slovom. Znak `|` čita se kao „ili”, pa ovakva deklaracija kaže da se vrednosti novog tipa mogu graditi na više različitih načina.

Na primer, tip koji predstavlja strane sveta možemo definisati ovako:

```
data Direction = North | South | East | West
```

Ovde tip `Direction` ima tačno četiri vrednosti: `North`, `South`, `East` i `West`. Na njima se zatim mogu definisati funkcije kao i na vrednostima ugrađenih tipova. Na primer, funkcija koja vraća suprotan smer može se zapisati ovako:

```
opposite :: Direction -> Direction
opposite North = South
opposite South = North
opposite East  = West
opposite West  = East
```

Možemo definisati i funkciju koja proverava da li je smer horizontalan:

```
isHorizontal :: Direction -> Bool
isHorizontal East = True
isHorizontal West = True
isHorizontal _    = False
```

U ovim definicijama konstruktori se koriste u podudaranju oblika, isto kao što smo ranije koristili `[]` ili `(x:xs)` pri radu sa listama.

Dobro je primetiti da su i neki poznati bibliotečki tipovi definisani na isti način. Na primer, logički tip `Bool` može se posmatrati kao tip uveden deklaracijom

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

Deklaracije pomoću `data` nisu ograničene samo na konstruktore bez argumenata. Konstruktor može imati i argumente, čime se dobijaju vrednosti koje u sebi nose dodatne podatke. Na

primer, oblik možemo predstaviti ovako:

```
data Shape = Circle Float | Rectangle Float Float
```

Ova deklaracija kaže da je vrednost tipa `Shape` ili oblika `Circle r`, gde je `r` poluprečnik kruga, ili oblika `Rectangle a b`, gde su `a` i `b` dužine stranica pravougaonika.

Sada možemo definisati funkciju koja računa površinu:

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle a b) = a * b
```

Ovde je važno razlikovati ime tipa od imena konstruktora. U prethodnom primeru `Shape` je ime tipa, dok su `Circle` i `Rectangle` konstruktori tog tipa. Zbog toga funkcija `area` ima tip `Shape -> Float`, a ne `Circle -> Float`, jer `Circle` nije tip.

Konstruktori sa argumentima ponašaju se kao funkcije koje od svojih argumenata grade vrednosti novog tipa. U prethodnom primeru konstruktor `Circle` prima jedan argument tipa `Float` i vraća vrednost tipa `Shape`, dok konstruktor `Rectangle` prima dva argumenta tipa `Float` i takođe vraća vrednost tipa `Shape`. To se može proveriti u interaktivnom okruženju:

```
ghci> :t Circle
Circle :: Float -> Shape

ghci> :t Rectangle
Rectangle :: Float -> Float -> Shape
```

Ipak, konstruktori ne služe za izračunavanje, već za građenje podataka. Na primer, izraz `abs (-2)` može se izračunati i svesti na `2`, jer je `abs` obična funkcija definisana jednačinama. Sa druge strane, izraz `Circle 2` ne predstavlja račun koji treba dalje pojednostavljivati, već gotovu vrednost tipa `Shape`. Dakle, konstruktor od zadatih argumenata samo gradi vrednost odgovarajućeg tipa.

Pošto sistem podrazumevano ne zna kako da prikaže vrednosti novog tipa, pokušaj njihovog ispisa može dovesti do greške. Zbog toga se često uz deklaraciju tipa dodaje i deo `deriving (Show)`, čime se automatski omogućava prikazivanje vrednosti tog tipa. Na primer:

```
data Shape = Circle Float | Rectangle Float Float
  deriving (Show)
```

Sada se vrednosti tipa `Shape` mogu ispisivati:

```
ghci> Circle 2
Circle 2.0

ghci> Rectangle 3 4
Rectangle 3.0 4.0
```

Bez dodatka `deriving (Show)` ovakvi izrazi bi mogli da se konstruišu i koriste u programima, ali njihov prikaz ne bi bio automatski podržan. Za sada je dovoljno da `deriving (Show)` posmatramo kao praktičan dodatak koji omogućava ispisivanje vrednosti novog tipa. Kasnije ćemo preciznije objasniti šta ovakvo izvođenje znači.

Na kraju, važno je naglasiti da tip uveden pomoću `data` nije samo drugo ime za neki stari

tip, kao kod deklaracije `type`. Čak i kada novi tip koristi poznate tipove kao sastavne delove, on ostaje zaseban tip sa sopstvenim konstruktorima.

8.3 Parametrizovani tipovi

Pomoću deklaracije `data` mogu se definisati i tipovi koji zavise od drugih tipova. Oni se nazivaju parametrizovani tipovi.

Važan primer iz prelida jeste možda-tip `Maybe`. On je definisan ovako:

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

Ovde je `Maybe` konstruktor tipa, dok je `a` tipski parametar. Tek kada se parametar `a` zameni nekim konkretnim tipom, dobija se konkretan tip, na primer `Maybe Int`, `Maybe Char` ili `Maybe String`. Sa druge strane, `Nothing` i `Just` su konstruktori vrednosti. Vrednosti tipa `Maybe a` zato imaju jedan od dva oblika: ili su `Nothing`, ili su oblika `Just x`, gde je `x` neka vrednost tipa `a`.

Na primer:

```
ghci> :t Just 'a'
Just 'a' :: Maybe Char

ghci> :t Just 5
Just 5 :: Num a => Maybe a

ghci> :t Nothing
Nothing :: Maybe a
```

Vrednost `Nothing` predstavlja odsustvo rezultata, dok `Just x` predstavlja uspešan rezultat koji sadrži vrednost `x`. Zbog toga je tip `Maybe a` veoma koristan kada želimo da predstavimo izračunavanje koje može, ali ne mora dati rezultat.

Na primer, možemo definisati totalnu verziju deljenja:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)
```

Ako je drugi argument nula, funkcija vraća `Nothing`. U suprotnom, vraća količnik upakovan konstruktorom `Just`.

Na primer:

```
ghci> safeDiv 10 2
Just 5

ghci> safeDiv 10 0
Nothing
```

Slično tome, možemo definisati i totalnu verziju funkcije `head`:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x
```

Ovde se za praznu listu vraća `Nothing`, dok se za nepraznu listu prvi element vraća u obliku `Just x`.

Na primer:

```
ghci> safeHead [3,4,5]
Just 3

ghci> safeHead []
Nothing
```

Parametrizovani tipovi nisu ograničeni samo na `Maybe`. Na primer, možemo definisati tip trodimenzionalnih vektora čije su sve koordinate istog tipa:

```
data Vector3 a = Vector3 a a a deriving (Show)
```

Ovde je `Vector3` konstruktor tipa, a istoimeni `Vector3` sa desne strane konstruktor vrednosti. Tako se dobijaju konkretni tipovi kao što su `Vector3 Int`, `Vector3 Float` i `Vector3 Double`.

Na primer, sabiranje vektora može se definisati ovako:

```
addV :: Num a => Vector3 a -> Vector3 a -> Vector3 a
addV (Vector3 x1 y1 z1) (Vector3 x2 y2 z2) =
    Vector3 (x1 + x2) (y1 + y2) (z1 + z2)
```

Na primer:

```
ghci> addV (Vector3 1 2 3) (Vector3 4 5 6)
Vector3 5 7 9

ghci> addV (Vector3 1.5 2.0 3.5) (Vector3 0.5 1.0 2.5)
Vector3 2.0 3.0 6.0
```

Ovaj primer pokazuje kako se kod parametrizovanih tipova isti oblik podatka može koristiti sa različitim tipovima elemenata, bez potrebe da se za svaki slučaj uvodi poseban tip.

8.4 Deklaracije pomoću newtype

Ako želimo da uvedemo novi tip koji je predstavljen jednom vrednošću nekog već postojećeg tipa, možemo koristiti deklaraciju `newtype`. Opšti oblik je:

```
newtype NoviTip = Konstruktor PostojeciTip
```

Ovime se uvodi zaista novi tip, a ne samo novo ime za stari tip. Zbog toga se vrednosti novog tipa ne mogu automatski mešati sa vrednostima tipa na kome su zasnovane.

Na primer, možemo definisati tip koji predstavlja identifikator studenta:

```
newtype StudentId = StudentId Int
```

Svaka vrednost tipa `StudentId` ima oblik `StudentId n`, gde je `n` vrednost tipa `Int`. Konstruktor `StudentId` koristi se za formiranje vrednosti novog tipa, ali i za njihovo raspakivanje pomoću podudaranja oblika.

Na primer, funkcija koja uvećava identifikator za `1` može se definisati ovako:

```
nextId :: StudentId -> StudentId
nextId (StudentId n) = StudentId (n + 1)
```

Ovde se vrednost najpre raspakuje obrascem `(StudentId n)`, zatim se broj `n` uveća, a dobijeni rezultat se ponovo upakuje konstruktorom `StudentId`.

Važno je primetiti da tip `StudentId` nije isto što i tip `Int`. Na primer, funkcija

```
isFirst :: StudentId -> Bool
isFirst (StudentId n) = n == 1
```

očekuje argument tipa `StudentId`, pa joj se ne može direktno proslediti običan ceo broj. Dakle, iako novi tip koristi istu unutrašnju reprezentaciju kao `Int`, sistem tipova ih jasno razlikuje.

Upravo u tome je glavna razlika između deklaracija `type`, `data` i `newtype`. Deklaracija `type` uvodi samo drugo ime za postojeći tip. Na primer:

```
type StudentId = Int
```

Ovde `StudentId` i `Int` predstavljaju isti tip. Takva deklaracija poboljšava čitljivost, ali ne uvodi nikakvu novu zaštitu na nivou tipova.

Deklaracija `data` uvodi potpuno novi tip i u opštem slučaju može imati više konstruktora, pri čemu svaki konstruktor može imati proizvoljno mnogo argumenata. Deklaracija `newtype` takođe uvodi novi tip, ali je namenjena samo jednostavnom slučaju: novi tip ima tačno jedan konstruktor sa tačno jednim argumentom.

Zbog toga se `newtype` može posmatrati kao specijalni slučaj deklaracije `data`, namenjen situacijama kada želimo jednostavan novi tip bez dodatne strukture.

Ako želimo da vrednosti novog tipa možemo i ispisivati, možemo dopisati `deriving (Show)`:

```
newtype StudentId = StudentId Int deriving (Show)
```

Tada, na primer, važi:

```
ghci> nextId (StudentId 7)
StudentId 8
```

Deklaracija `newtype` je korisna kada želimo da zadržimo jednostavnu reprezentaciju nekog podatka, ali da pritom uvedemo poseban tip sa jasnijim značenjem. Time se dobija pregledniji kod i sprečava slučajno mešanje vrednosti koje imaju istu reprezentaciju, ali različitu namenu.

8.5 Zapisi

Kada konstruktor nekog tipa ima više argumenata, često nije odmah jasno koja pozicija odgovara kom podatku. To može otežati i čitanje i pisanje funkcija koje rade sa takvim vrednostima. Na primer, deklaracija

```
data Person = Person String String Int String
```

sama po sebi ne govori jasno šta predstavljaju pojedina polja. Možemo pretpostaviti da su to, na primer, ime, prezime, godine i grad, ali se to iz same deklaracije ne vidi neposredno.

Zbog toga Haskell dozvoljava upotrebu zapisa (engl. *record syntax*). Kod zapisa svako polje dobija svoje ime. Na primer, prethodni tip može se zapisati ovako:

```
data Person = Person
  { firstName :: String
  , lastName  :: String
  , age       :: Int
  , city      :: String
  } deriving (Show)
```

Ovde su `firstName`, `lastName`, `age` i `city` imena polja. Ovakav zapis je pregledniji, jer se iz same deklaracije odmah vidi šta svako polje predstavlja.

Vrednosti ovog tipa mogu se i dalje zadavati na uobičajen način, navođenjem argumenata konstruktora redom:

```
p1 :: Person
p1 = Person "Ana" "Anic" 20 "Beograd"
```

Moguće je, međutim, koristiti i posebnu sintaksu za zapise, u kojoj se eksplicitno navode imena polja:

```
p2 :: Person
p2 = Person
  { firstName = "Marko"
  , lastName  = "Markovic"
  , age       = 22
  , city      = "Novi Sad"
  }
```

Prednost ovakvog zapisa je u tome što je odmah jasno koja vrednost pripada kom polju. Pored toga, pri ovakvom zadavanju vrednosti nije neophodno voditi računa o redosledu polja.

Jedna od važnih posledica upotrebe zapisa jeste to da Haskell automatski definiše funkcije za pristup poljima. U ovom primeru dobijaju se funkcije

```
firstName :: Person -> String
lastName  :: Person -> String
age       :: Person -> Int
city      :: Person -> String
```

Na primer:

```
ghci> firstName p1
"Ana"

ghci> age p2
22
```

Dakle, imena polja mogu se koristiti kao obične funkcije koje iz vrednosti izdvajaju odgovarajuću komponentu.

Zapisi su korisni i kada želimo da promenimo samo jedno polje postojeće vrednosti. Na primer, ako želimo da promenimo grad osobe `p1`, možemo napisati:

```
p3 :: Person
p3 = p1 { city = "Nis" }
```

Pri tome sve ostale komponente ostaju iste kao kod vrednosti `p1`. Na primer:

```
ghci> p3
Person {firstName = "Ana", lastName = "Anic", age = 20, city = "Nis"}
```

Zapisi se mogu koristiti i u podudaranju oblika. Na primer, funkcija koja vraća puno ime osobe može se definisati ovako:

```
fullName :: Person -> String
fullName (Person { firstName = f, lastName = l }) = f ++ " " ++ l
```

Ovde se izdvajaju samo polja koja su nam potrebna, dok ostala polja uopšte ne moraju da se pominju. To često daje pregledniji kod nego običan zapis konstruktora sa više argumenata.

Zapisi su zato naročito korisni kada tip sadrži više polja i kada želimo da kod bude što jasniji i pregledniji. Oni ne uvode novu vrstu tipa, već predstavljaju samo pogodniji način zapisivanja nekih `data` deklaracija.

9 Rekurzivni tipovi podataka

9.1 Prirodni brojevi kao rekurzivni tip

Rekurzivni tipovi podataka su tipovi u čijoj se definiciji pojavljuje sam tip koji se definiše. To znači da se njihove vrednosti konstruišu postepeno: od jednostavnijih vrednosti istog tipa grade se složenije. Kao i kod rekurzivnih funkcija, i ovde postoji bazni slučaj, kao i pravilo kojim se od već konstruisanih vrednosti dobijaju nove.

Jedan od najjednostavnijih primera jeste tip prirodnih brojeva. Možemo ga definisati ovako:

```
data Nat = Zero | Next Nat
  deriving (Show)
```

Ova deklaracija kaže da je vrednost tipa `Nat` ili `Zero`, ili oblika `Next n`, gde je `n` neka vrednost istog tipa. Konstruktor `Zero` predstavlja bazni slučaj, dok konstruktor `Next` od jednog prirodnog broja gradi njegovog sledbenika.

Pošto je dodat deo `deriving (Show)`, vrednosti ovog tipa mogu se ispisivati. Prvih nekoliko vrednosti su:

```
Zero
Next Zero
Next (Next Zero)
Next (Next (Next Zero))
Next (Next (Next (Next Zero)))
```

One prirodno odgovaraju brojevima `0`, `1`, `2`, `3` i `4`.

Kako se prirodni brojevi ovde predstavljaju drugačije nego pomoću ugrađenog tipa `Int`, korisno je definisati funkcije koje povezuju ova dva zapisa.

Funkcija koja vrednost tipa `Nat` prevodi u odgovarajući ceo broj može se definisati ovako:

```
toInt :: Nat -> Int
toInt Zero = 0
toInt (Next n) = 1 + toInt n
```

Bazni slučaj kaže da vrednosti `Zero` odgovara broj `0`. U rekurzivnom slučaju broj predstavljen oblikom `Next n` dobija se tako što se najpre izračuna vrednost broja `n`, a zatim se na rezultat doda `1`.

U suprotnom smeru možemo definisati funkciju koja ceo broj prevodi u vrednost tipa `Nat`:

```
fromInt :: Int -> Nat
fromInt n
  | n <= 0    = Zero
  | otherwise = Next (fromInt (n - 1))
```

Ovde je prirodno da se sve vrednosti manje ili jednake nuli preslikaju u `Zero`. Za pozitivan broj `n`, najpre se konstruiše vrednost koja odgovara broju `n - 1`, a zatim se na nju primeni konstruktor `Next`.

Na primer:

```
ghci> toInt (Next (Next Zero))
2
```

```
ghci> fromInt 3
Next (Next (Next Zero))
```

Nad ovako definisanim prirodnim brojevima mogu se zadavati i operacije neposredno, bez prevođenja na tip `Int`. Na primer, sabiranje se može definisati ovako:

```
plus :: Nat -> Nat -> Nat
plus Zero n = n
plus (Next m) n = Next (plus m n)
```

Prva jednačina kaže da je zbir broja `Zero` i broja `n` jednak upravo broju `n`. Druga jednačina pokazuje da se sabiranje vrši tako što se konstruktori `Next` iz prvog sabirka jedan po jedan prenose na rezultat. Kada se dođe do završnog `Zero`, na njegovo mesto dolazi drugi sabirak.

Na primer, izračunavanje izraza

```
plus (Next (Next Zero)) (Next Zero)
```

odvija se ovako:

```
plus (Next (Next Zero)) (Next Zero)
= Next (plus (Next Zero) (Next Zero))
= Next (Next (plus Zero (Next Zero)))
= Next (Next (Next Zero))
```

pa dobijena vrednost zaista odgovara broju `3`. To možemo i neposredno proveriti:

```
ghci> plus (Next (Next Zero)) (Next Zero)
Next (Next (Next Zero))

ghci> toInt (plus (Next (Next Zero)) (Next Zero))
3
```

Ovaj primer pokazuje osnovnu ideju rekurzivnih tipova podataka: vrednosti se ne navode pojedinačno, već se konstruišu korak po korak, polazeći od bazne vrednosti i primenjujući odgovarajuće konstruktore. Ista ideja prirodno se primenjuje i na složenije strukture, kao što su povezane liste i binarna stabla.

9.2 Povezane liste

Povezana lista je još jedan prirodan primer rekurzivnog tipa podataka. Intuitivno, lista je ili prazna, ili se sastoji od prvog elementa i ostatka liste. Upravo ta ideja neposredno vodi do rekurzivne definicije tipa.

Naš tip povezanih listi možemo definisati ovako:

```
data List a = Empty | Node a (List a)
  deriving (Show)
```

Ova deklaracija kaže da je vrednost tipa `List a` ili prazna lista `Empty`, ili oblika `Node x xs`, gde je `x` element tipa `a`, a `xs` ostatak liste. Konstruktor `Empty` predstavlja bazni slučaj, dok konstruktor `Node` od jednog elementa i jedne liste gradi novu listu.

Pošto je dodat deo `deriving (Show)`, vrednosti ovog tipa mogu se ispisivati. Na primer:

```
ghci> Node 1 (Node 2 (Node 3 Empty))
Node 1 (Node 2 (Node 3 Empty))

ghci> Node True (Node False Empty)
Node True (Node False Empty)
```

Iz ovih primera se vidi da je lista ili prazna, ili se dobija tako što se jedan element dodaje na početak već postojeće liste. Na primer, izraz

```
Node 1 (Node 2 (Node 3 Empty))
```

predstavlja listu čiji je prvi element `1`, a ostatak liste je

```
Node 2 (Node 3 Empty)
```

Pošto je i sama struktura liste rekurzivna, funkcije nad njom prirodno se definišu rekurzijom. Na primer, dužina liste može se definisati ovako:

```
len :: List a -> Int
len Empty = 0
len (Node _ xs) = 1 + len xs
```

Bazni slučaj kaže da je dužina prazne liste jednaka `0`. U rekurzivnom slučaju zanemaruje se prvi element liste, a dužina se dobija tako što se na dužinu ostatka liste doda `1`.

Na primer, izračunavanje izraza

```
len (Node 'a' (Node 'b' (Node 'c' Empty)))
```

odvija se ovako:

```
len (Node 'a' (Node 'b' (Node 'c' Empty)))
= 1 + len (Node 'b' (Node 'c' Empty))
= 1 + (1 + len (Node 'c' Empty))
= 1 + (1 + (1 + len Empty))
= 1 + (1 + (1 + 0))
= 3
```

Možemo definisati i funkciju koja nadovezuje jednu listu na kraj druge:

```
append :: List a -> List a -> List a
append Empty ys = ys
append (Node x xs) ys = Node x (append xs ys)
```

Ako je prva lista prazna, rezultat je druga lista. Ako je prva lista neprazna, njen prvi element ostaje na početku rezultata, a ostatak se dobija rekurzivnim nadovezivanjem repa prve liste i druge liste.

Na primer:

```
ghci> append (Node 1 (Node 2 Empty)) (Node 3 (Node 4 Empty))
Node 1 (Node 2 (Node 3 (Node 4 Empty)))
```

Ovaj primer pokazuje da se i povezane liste prirodno opisuju rekurzivno: prazna lista pred-

stavlja bazni slučaj, dok se svaka neprazna lista dobija dodavanjem jednog elementa na početak neke manje liste. Zbog toga se funkcije nad listama najčešće definišu upravo rekurzijom i podudaranjem oblika.

9.3 Binarna stabla

Još jedan važan primer rekurzivnih tipova podataka jesu binarna stabla. Za razliku od povezanih listi, koje imaju linearnu strukturu, kod binarnog stabla svaki čvor može imati najviše dva podstabla, levo i desno. Zbog toga su stabla pogodna za hijerarhijsko organizovanje podataka.

Tip binarnog stabla možemo definisati na sledeći način:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
  deriving (Show)
```

Ova deklaracija kaže da je vrednost tipa `Tree a` ili prazno stablo `Empty`, ili čvor oblika `Node x l r`, gde je `x` vrednost u korenu čvora, a `l` i `r` levo i desno podstablo.

Na primer, vrednost

```
Node 5
  (Node 3
    (Node 1 Empty Empty)
    (Node 4 Empty Empty))
  (Node 7
    Empty
    (Node 9 Empty Empty))
```

predstavlja binarno stablo čiji je koren `5`, levo podstablo ima koren `3`, a desno koren `7`.

Pošto je dodat deo `deriving (Show)`, ovakve vrednosti mogu se ispisivati. Na primer:

```
ghci> Node 2 Empty (Node 5 Empty Empty)
Node 2 Empty (Node 5 Empty Empty)
```

Kao i kod povezanih listi, funkcije nad stablima prirodno se definišu rekurzijom. Jedna od najjednostavnijih takvih funkcija jeste funkcija koja računa visinu stabla.

```
height :: Tree a -> Int
height Empty = 0
height (Node _ l r) = 1 + max (height l) (height r)
```

Visina praznog stabla jednaka je `0`. Ako stablo nije prazno, njegova visina dobija se tako što se uzme veća od visina levog i desnog podstabla, a zatim se na nju doda `1` zbog korena.

Na primer:

```
ghci> height Empty
0

ghci> height (Node 5 Empty Empty)
1
```

Posebno su interesantna uređena binarna stabla pretrage. To su stabla kod kojih je svaki element u levom podstablu manji od vrednosti u čvoru, a svaki element u desnom podstablu

veći od nje. Zahvaljujući tome, pri pretrazi nije potrebno obilaziti celo stablo, već se u svakom koraku može odlučiti da li treba nastaviti ulevo ili udesno.

Umetanje elementa u takvo stablo može se definisati ovako:

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node y l r)
  | x < y     = Node y (insert x l) r
  | x > y     = Node y l (insert x r)
  | otherwise = Node y l r
```

Ako je stablo prazno, novi element postaje koren stabla. Ako stablo nije prazno, novi element se poredi sa vrednošću u korenu. Ako je manji, umeće se u levo podstablo, a ako je veći, umeće se u desno. U slučaju jednakosti stablo ostaje nepromenjeno, pa se isti element ne dodaje više puta.

Na primer:

```
ghci> insert 5 Empty
Node 5 Empty Empty

ghci> insert 3 (insert 5 Empty)
Node 5 (Node 3 Empty Empty) Empty

ghci> insert 7 (insert 3 (insert 5 Empty))
Node 5 (Node 3 Empty Empty) (Node 7 Empty Empty)
```

Na sličan način može se definisati i funkcija koja proverava da li se dati element nalazi u uređenom stablu:

```
contains :: Ord a => a -> Tree a -> Bool
contains _ Empty = False
contains x (Node y l r)
  | x == y    = True
  | x < y     = contains x l
  | otherwise = contains x r
```

Ako je stablo prazno, traženi element se u njemu ne nalazi. Ako stablo nije prazno, najpre se proverava da li je tražena vrednost jednaka korenu. Ako nije, poređenje određuje u kom podstablu pretragu treba nastaviti.

Na primer, ako definišemo

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

tada dobijamo:

```
ghci> contains 4 t
True

ghci> contains 8 t
False

ghci> height t
3
```

Vrednost `4` pronalazi se tako što se iz korena `5` prelazi ulevo, a zatim iz čvora `3` udesno. Sa druge strane, pri traženju vrednosti `8` najpre se iz korena `5` prelazi udesno, zatim iz čvora `7` ponovo udesno, a potom se dolazi do praznog stabla, pa je rezultat `False`.

Ovaj primer pokazuje osnovnu prednost uređenih binarnih stabala: zahvaljujući uređenosti, u svakom koraku pretrage odbacuje se jedna cela grana stabla. Zbog toga se pretraga i umetanje mogu definisati prirodno i efikasno, koristeći istu rekurzivnu strukturu kojom je definisan i sam tip podataka.