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.

10 Deklarisanje klasa i instanci

10.1 Izvođenje instanci pomoću deriving

Kada se uvede novi tip pomoću deklaracija `data` ili `newtype`, često želimo da se nad njegovim vrednostima odmah mogu koristiti neke već poznate operacije, kao što su poređenje, ispisivanje ili učitavanje iz niske karaktera. U mnogim jednostavnim slučajevima takve instance nije potrebno definisati ručno, jer Haskell omogućava njihovo automatsko izvođenje pomoću ključne reči `deriving`.

Najčešće se na ovaj način izvode instance klasa `Eq`, `Ord`, `Show` i `Read`. Ideja je sledeća: na osnovu same strukture tipa Haskell može automatski da zaključi kako se proverava jednakost, kako se vrednosti uređuju, kako se prikazuju i kako se ponovo čitaju iz tekstualnog zapisa.

Najpre pogledajmo jednostavan tip čije vrednosti predstavljaju dane vikenda:

```
data WeekendDay = Saturday | Sunday
  deriving (Eq)
```

Ovim zapisom kaže se da tip `WeekendDay` automatski postaje instanca klase `Eq`. To znači da se njegove vrednosti mogu porediti pomoću operatora `==` i `/=`.

Na primer:

```
ghci> Saturday == Saturday
True

ghci> Saturday == Sunday
False

ghci> Saturday /= Sunday
True
```

Pošto tip `WeekendDay` ima samo dva konstruktora bez argumenata, jednakost se ovde svodi na proveru da li su konstruktori isti.

Ako želimo da se vrednosti nekog tipa mogu i uređivati, možemo izvesti instancu klase `Ord`:

```
data WeekendDay = Saturday | Sunday
  deriving (Eq, Ord)
```

Sada se nad vrednostima ovog tipa mogu koristiti i operatori poređenja, kao i funkcije `compare`, `min` i `max`. Pri automatskom izvođenju klase `Ord`, poredak konstruktora određuje se njihovim redosledom u deklaraciji tipa. Zato ovde važi da je `Saturday < Sunday`.

Na primer:

```
ghci> Saturday < Sunday
True

ghci> Saturday >= Sunday
False

ghci> max Saturday Sunday == Sunday
True
```

Za prikazivanje vrednosti u čitljivom obliku koristi se klasa `Show`. Ako nju izvedemo, vrednosti datog tipa mogu se ispisivati:

```
data WeekendDay = Saturday | Sunday
  deriving (Eq, Ord, Show)
```

Na primer:

```
ghci> Saturday
Saturday

ghci> show Sunday
"Sunday"
```

Klasa `Read` omogućava obrnuti postupak: vrednost se može rekonstruisati iz odgovarajuće niske karaktera.

```
data WeekendDay = Saturday | Sunday
  deriving (Eq, Ord, Show, Read)
```

Na primer:

```
ghci> read "Saturday" :: WeekendDay
Saturday

ghci> read "Sunday" == Sunday
True
```

Primetimo da je u prvom primeru bilo potrebno eksplicitno naznačiti da rezultat treba da bude tipa `WeekendDay`.

U praksi se više instanci najčešće izvodi odjednom, kao u prethodnom primeru. Tada jedan isti tip istovremeno dobija mogućnost poređenja, uređivanja, ispisa i učitavanja.

Pogledajmo sada primer tipa čiji konstruktori imaju argumente:

```
data Shape = Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show, Read)
```

Pošto su izvedene instance klasa `Eq`, `Ord`, `Show` i `Read`, vrednosti tipa `Shape` mogu se porediti, uređivati, ispisivati i učitavati.

Na primer:

```
ghci> Circle 2.0 == Circle 2.0
True

ghci> Circle 2.0 == Rectangle 2.0 2.0
False

ghci> show (Rectangle 3.0 4.0)
"Rectangle 3.0 4.0"

ghci> read "Circle 5.0" :: Shape
Circle 5.0
```

Kod tipova čiji konstruktori imaju argumente, poređenje se vrši najpre prema redosledu konstruktora u deklaraciji tipa, a ako su oni isti, onda redom prema njihovim argumentima.

Na primer:

```
ghci> Circle 2.0 < Rectangle 1.0 1.0
True

ghci> Rectangle 2.0 3.0 < Rectangle 2.0 5.0
True
```

Ovakve instance mogu se izvesti samo ako i tipovi argumenata konstruktora već pripadaju odgovarajućim klasama. Ovde je to ispunjeno, jer tip `Float` već pripada klasama `Eq`, `Ord`, `Show` i `Read`.

Automatsko izvođenje instanci može se koristiti i kod parametrizovanih tipova. Na primer:

```
data Pair a = Pair a a
  deriving (Eq, Ord, Show)
```

Ovde je `Pair a` parametrizovan tip koji predstavlja par vrednosti istog tipa. Izvedene su instance klasa `Eq`, `Ord` i `Show`, pa se vrednosti tipa `Pair a` mogu porediti i ispisivati.

Na primer:

```
ghci> Pair 2 5
Pair 2 5

ghci> Pair 1 4 < Pair 1 7
True

ghci> Pair 'b' 'a' > Pair 'a' 'z'
True
```

Poređenje se i ovde vrši leksikografski, pri čemu se najpre porede prve komponente, a ako su one jednake, porede se druge. Pošto je tip `Pair a` parametrizovan, izvedene instance važe samo za one konkretne tipove `Pair a` kod kojih i tip `a` pripada odgovarajućim klasama. Drugim rečima, vrednosti tipa `Pair a` mogu se porediti i ispisivati samo ako isto važi za vrednosti tipa `a`.

Automatsko izvođenje instanci je veoma korisno kada prirodno ponašanje tipa već odgovara standardnim klasama. Na taj način novi tipovi brzo dobijaju osnovne mogućnosti, bez potrebe da se instance definišu ručno. Kada je potrebno neko specifično ponašanje koje se ne poklapa sa podrazumevanim, instance se mogu zadati i eksplicitno, što ćemo videti u narednoj podsekciji.

10.2 Deklarisanje instanci

U prethodnoj podsekciji videli smo da se za mnoge tipove instance standardnih klasa mogu automatski izvesti pomoću mehanizma `deriving`. Međutim, ponekad želimo da instancu zadajemo ručno. To je potrebno kada želimo ponašanje koje se razlikuje od podrazumevanog, ili kada iz nekog razloga ne želimo da se oslonimo na automatsko izvođenje.

Opšti oblik deklaracije instance jeste:

```
instance Klasa Tip where
  ...
```

Ovde se navodi da tip `Tip` postaje instanca klase `Klasa`, a u telu deklaracije se zadaju definicije odgovarajućih metoda. U Haskellu instance se deklarišu za tipove koji su uvedeni pomoću deklaracija `data` ili `newtype`.

Razmotrimo tip koji predstavlja stanje semafora:

```
data TrafficLight = Red | Yellow | Green
```

Ako želimo da se vrednosti ovog tipa mogu da porediti po jednakosti, možemo ručno deklarisati instancu klase `Eq`:

```
instance Eq TrafficLight where
  Red    == Red    = True
  Yellow == Yellow = True
  Green  == Green  = True
  _      == _      = False
```

Ova deklaracija kaže da su dve vrednosti jednake tačno kada su konstruisane istim konstruktorom. Na primer:

```
ghci> Red == Red
True

ghci> Red == Green
False
```

Ručno možemo deklarisati i instancu klase `Show`. Na primer:

```
instance Show TrafficLight where
  show Red    = "crveno"
  show Yellow = "zuto"
  show Green  = "zeleno"
```

Sada se vrednosti tipa `TrafficLight` ispisuju na način koji smo sami zadali:

```
ghci> Red
crveno

ghci> [Red, Green, Yellow]
[crveno,zeleno,zuto]
```

Ovaj primer pokazuje važnu razliku između ručnog deklarisanja instance i mehanizma `deriving`. Kada bismo koristili `deriving (Show)`, vrednost `Red` ispisivala bi se kao `Red`. Ručnim deklarisanjem instance možemo zadati drugačiji, pogodniji prikaz.

Kod nekih klasa nije neophodno eksplicitno definisati baš sve metode. Na primer, kod klase `Eq` dovoljno je zadati definiciju operatora `==`, jer se operator `/=` može dobiti iz njega. Slično tome, kod drugih klasa često postoje podrazumevane definicije nekih metoda. Zbog toga se u deklaraciji instance navode samo one definicije koje su zaista potrebne.

Ručno deklarisanje instanci moguće je i kod parametrizovanih tipova. Posmatrajmo ponovo tip

```
data Pair a = Pair a a
```

Ako želimo da vrednosti tipa `Pair a` mogu da se porede po jednakosti, možemo napisati:

```
instance Eq a => Eq (Pair a) where
  Pair x y == Pair u v = x == u && y == v
```

Ovde je klasno ograničenje `Eq a` neophodno, jer se jednakost parova svodi na poređenje njihovih komponenti. Zbog toga tip `a` mora biti instanca klase `Eq`.

Na primer:

```
ghci> Pair 1 2 == Pair 1 2
True

ghci> Pair 1 2 == Pair 2 1
False

ghci> Pair 'a' 'b' == Pair 'a' 'b'
True
```

Na sličan način možemo deklarisati i instancu klase `Show`:

```
instance Show a => Show (Pair a) where
  show (Pair x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
```

Sada se parovi ispisuju u obliku koji smo sami zadali:

```
ghci> Pair 3 5
(3, 5)

ghci> Pair True False
(True, False)
```

I ovde je potrebno klasno ograničenje, ovoga puta `Show a`, jer se za ispis para moraju ispisati i njegove komponente.

Važno je primetiti da se u deklaraciji instance za parametrizovan tip ne navodi sam konstruktor tipa `Pair`, već konkretan oblik `Pair a`. Razlog je to što se instance uvek deklarišu za konkretne tipove, a ne za same konstruktore tipova.

Ručno deklarisanje instanci zato nam daje veću kontrolu nego `deriving`. Ono omogućava da sami zadamo značenje operacija klase za novi tip, kao i da prilagodimo način poređenja, ispisivanja ili drugih oblika ponašanja potrebama konkretnog programa.

10.3 Deklarisanje klasa

Pored toga što u Haskellu možemo deklarisati instance već postojećih klasa, možemo uvoditi i sopstvene tipske klase. Na taj način opisujemo zajedničko ponašanje koje može biti pridruženo različitim tipovima.

Opšti oblik deklaracije klase je:

```
class ImeKlase a where
  ...
```

Ovde `a` predstavlja proizvoljan tip koji može biti instanca te klase, a u telu deklaracije navode se metode koje svaka instanca treba da podržava.

Na primer, možemo definisati klasu tipova čijim se vrednostima može pridružiti neka numerička veličina:

```
class Size a where
  size :: a -> Float
```

Ova deklaracija uvodi klasu `Size` sa jednom metodom `size`. Time se kaže da za svaki tip koji želimo da bude instanca ove klase moramo zadati način računanja njegove veličine.

Posmatrajmo tip koji predstavlja duž:

```
data Segment = Segment Float Float
  deriving (Show)
```

Ako vrednost tipa `Segment` tumačimo kao duž zadatu svojim krajevima na brojevnoj pravoj, njena veličina može se posmatrati kao dužina te duži. Zato možemo deklarisati instancu:

```
instance Size Segment where
  size (Segment x y) = abs (y - x)
```

Slično tome, za tip pravougaonika možemo definisati veličinu kao površinu:

```
data Rectangle = Rectangle Float Float
  deriving (Show)

instance Size Rectangle where
  size (Rectangle a b) = a * b
```

Sada se ista metoda `size` može primenjivati na vrednosti različitih tipova:

```
ghci> size (Segment 2 7)
5.0

ghci> size (Rectangle 3 4)
12.0
```

Ovaj primer pokazuje osnovnu ideju tipskih klasa: ista operacija može imati različite implementacije za različite tipove, ali se koristi pod istim imenom.

U deklaraciji klase moguće je zadati i podrazumevanu definiciju neke metode. Na primer, možemo definisati klasu tipova čije vrednosti imaju istaknut tekstualni prikaz:

```
class Pretty a where
  pretty :: a -> String
  prettyList :: [a] -> String
  prettyList xs = "[" ++ unwords (map pretty xs) ++ "]"
```

Ovde je metoda `prettyList` definisana pomoću metode `pretty`. To znači da pri deklarisanju instance nije neophodno zadavati `prettyList`, osim ako želimo drugačije ponašanje od podrazumevanog. Funkcija `unwords` spaja listu niski u jednu nisku, pri čemu između susednih niski ubacuje po jedan razmak.

Na primer, za semafor možemo napisati:

```
data TrafficLight = Red | Yellow | Green
```

```
instance Pretty TrafficLight where
  pretty Red = "crveno"
  pretty Yellow = "zuto"
  pretty Green = "zeleno"
```

Tada važi:

```
ghci> pretty Red
"crveno"

ghci> prettyList [Red, Green, Yellow]
"[crveno zeleno zuto]"
```

Dakle, dovoljno je bilo zadati samo metodu `pretty`, dok se metoda `prettyList` automatski dobija iz podrazumevane definicije. Ako je potrebno, podrazumevana definicija može se u konkretnoj instanci i zameniti novom definicijom.

Nova klasa može biti zasnovana na već postojećoj klasi. U tom slučaju se u zaglavlju deklaracije navodi klasno ograničenje. Na primer, možemo definisati klasu tipova čije vrednosti imaju kružni sledbenik:

```
class Eq a => Cycle a where
  next :: a -> a
```

Ova deklaracija kaže da tip može biti instanca klase `Cycle` samo ako je već instanca klase `Eq`. Drugim rečima, `Cycle` je izvedena iz klase `Eq`. To znači da svaka instanca klase `Cycle` mora ujedno biti i instanca klase `Eq`.

Na primer, posmatrajmo tip dana vikenda i početka radne nedelje:

```
data DayPart = Friday | Saturday | Sunday | Monday
  deriving (Eq, Show)
```

Tada možemo definisati instancu klase `Cycle` ovako:

```
instance Cycle DayPart where
  next Friday = Saturday
  next Saturday = Sunday
  next Sunday = Monday
  next Monday = Friday
```

Na primer:

```
ghci> next (next Sunday)
Friday

ghci> Friday == Monday
False
```

Pošto je `Cycle` izvedena iz `Eq`, tip `DayPart` mora prethodno imati instancu klase `Eq`. Zato je u deklaraciji tipa dodat deo `deriving (Eq, Show)`.

11 Ulaz i izlaz

11.1 Akcije

Do sada smo Haskell uglavnom koristili kroz interaktivno okruženje. Definisali smo funkcije, učitavali ih u okruženje i zatim neposredno ispitivali njihove vrednosti. Sada želimo da pišemo programe koji se mogu prevesti i samostalno pokrenuti. Takvi programi mogu da komuniciraju sa spoljnim svetom – da čitaju podatke sa standardnog ulaza, da ispisuju rezultate na standardni izlaz, da rade sa datotekama i slično.

Najjednostavniji primer takvog programa možemo napisati ovako:

```
main = putStrLn "Zdravo!"
```

Ako ovaj kôd sačuvamo u datoteci `hello.hs`, možemo ga prevesti i pokrenuti iz terminala:

```
$ ghc hello.hs
$ ./hello
Zdravo!
```

Ime `main` ima poseban status. Kada se pokrene prevedeni Haskell program, izvršava se upravo akcija zadata imenom `main`. Uobičajeno je da `main` ima tip `IO ()`. Taj tip se u praksi često ne navodi eksplicitno, ali ga ovde možemo zapisati radi jasnoće:

```
main :: IO ()
main = putStrLn "Zdravo!"
```

Da bismo razumeli ovaj program, pogledajmo najpre tip funkcije `putStrLn`:

```
putStrLn :: String -> IO ()
```

Ovaj tip kaže da `putStrLn` nije akcija sama po sebi, već funkcija koja uzima nisku i vraća akciju. Na primer, izraz

```
putStrLn "Zdravo!"
```

ima tip `IO ()`. To je akcija koja, kada se izvrši, ispisuje nisku `"Zdravo!"` na standardni izlaz i prelazi u novi red.

Dakle, vrednosti tipa `IO a` nazivamo akcijama. Akcija tipa `IO a`, kada se izvrši, može komunicirati sa spoljnim svetom i pritom vratiti rezultat tipa `a`.

Tip `()` naziva se jedinični tip. On ima samo jednu vrednost, koja se takođe zapisuje kao `()`. U tipu `IO ()` ovaj rezultat obično nema posebno značenje, već je važan efekat akcije, na primer ispisivanje teksta na standardni izlaz.

Ako program treba da izvrši više akcija, koristimo `do` notaciju:

```
main = do
  putStrLn "Zdravo!"
  putStrLn "Ovo je Haskell program."
  putStrLn "Kraj."
```

Svaka linija u ovom `do` bloku jeste akcija. Akcije se izvršavaju redom kojim su navedene, a ceo `do` blok predstavlja jednu složenu akciju. Pošto je poslednja akcija u ovom primeru tipa `IO ()`, i ceo blok ima tip `IO ()`.

Kada se blok sastoji od samo jedne akcije, eksplicitno navođenje `do` nije neophodno. Zbog toga su sledeće dve definicije ekvivalentne:

```
main = putStrLn "Zdravo!"
```

```
main = do
  putStrLn "Zdravo!"
```

Za ispisivanje na standardni izlaz često se koriste funkcije `putStrLn`, `putStr`, `putChar` i `print`:

```
putStrLn :: String -> IO ()
putStr   :: String -> IO ()
putChar  :: Char -> IO ()
print    :: Show a => a -> IO ()
```

Funkcija `putStrLn` ispisuje nisku i zatim prelazi u novi red. Funkcija `putStr` ispisuje nisku bez prelaska u novi red. Funkcija `putChar` ispisuje jedan karakter. Funkcija `print` prima vrednost tipa koji pripada klasi `Show`, pretvara je u nisku pomoću funkcije `show`, a zatim ispisuje dobijenu nisku.

Na primer:

```
main = do
  putStr "Rezultat"
  putStr ": "
  print (2 + 3)
  putChar '>'
  putChar ' '
  putStrLn "kraj"
```

Pokretanjem programa dobija se:

```
Rezultat: 5
> kraj
```

Za čitanje sa standardnog ulaza najčešće se koriste akcije `getLine` i `getChar`:

```
getLine :: IO String
getChar :: IO Char
```

Akcija `getLine` čita celu liniju unosa, dok akcija `getChar` čita jedan karakter. Da bismo rezultat neke akcije koristili u nastavku programa, koristimo strelicu `<-` unutar `do` bloka.

Na primer:

```
fullName :: String -> String -> String
fullName name surname = name ++ " " ++ surname

main = do
  putStrLn "Ime:"
  name <- getLine
  putStrLn "Prezime:"
  surname <- getLine
```

```
  putStrLn ("Puno ime: " ++ fullName name surname)
```

Linija

```
name <- getLine
```

znači: izvrši akciju `getLine`, uzmi njen rezultat i veži ga za ime `name`. Pošto je `getLine :: IO String`, ime `name` u nastavku `do` bloka ima tip `String`. Zato se može proslediti funkciji `fullName`, koja je obična čista funkcija tipa `String -> String -> String`.

Slično se koristi i `getChar`:

```
main = do
  putStrLn "Unesi karakter:"
  c <- getChar
  putStrLn ("Unet je karakter: " ++ [c])
```

Ovde je `c` vrednost tipa `Char`. Pošto je niska lista karaktera, jednočlana niska koja sadrži karakter `c` zapisuje se kao `[c]`.

Pored strelice `<-`, u `do` bloku možemo koristiti i `let`. Razlika je u tome što se `<-` koristi za izvršavanje akcije i vezivanje njenog rezultata, dok se `let` koristi za imenovanje običnih izraza. U `do` bloku uz `let` se ne piše `in`.

Na primer:

```
main = do
  putStrLn "Unesi tekst:"
  text <- getLine
  let reversed = reverse text
      message = "Obrnuto: " ++ reversed
  putStrLn message
```

Ovde se akcija `getLine` izvršava pomoću `<-`, a njen rezultat se vezuje za ime `text`. Sa druge strane, izrazi `reverse text` i `"Obrnuto: " ++ reversed` nisu akcije, već obične vrednosti tipa `String`. Zato se uvode pomoću `let`.

Akcije možemo definisati i izvan `main`. Na primer:

```
announce :: String -> IO ()
announce word = putStrLn ("Uneta rec: " ++ word)

main = do
  putStrLn "Unesi rec:"
  word <- getLine
  announce word
```

Funkcija `announce` nije akcija, već funkcija koja od niske pravi akciju. Izraz `announce word` ima tip `IO ()` i može se navesti kao linija u `do` bloku.

Možemo definisati i akciju koja učitava više podataka i vraća običnu vrednost kao rezultat. Na primer:

```
readFullName :: IO String
readFullName = do
  putStrLn "Ime:"
  name <- getLine
```

```
  putStrLn "Prezime:"
  surname <- getLine
  return (name ++ " " ++ surname)
```

Ova akcija ima tip `IO String`. Kada se izvrši, ona učitava dve linije i vraća jednu nisku. Poslednja linija u `do` bloku mora biti akcija, pa zato ne možemo samo napisati

```
name ++ " " ++ surname
```

jer taj izraz ima tip `String`. Funkcija `return` pretvara običnu vrednost u akciju koja tu vrednost vraća kao rezultat i njen tip je

```
return :: a -> IO a
```

Zato izraz `return (name ++  ++ surname)` ima tip `IO String`.

Akciju `readFullName` zatim možemo koristiti unutar veće akcije:

```
main = do
  person <- readFullName
  putStrLn ("Korisnik: " ++ person)
```

Ovde se akcija `readFullName` izvršava, njen rezultat se vezuje za ime `person`, a zatim se taj rezultat koristi za ispis.

Posebno treba naglasiti da `return` u Haskellu nije isto što i `return` u mnogim imperativnim jezicima. On ne prekida izvršavanje programa i ne izlazi iz funkcije. On samo uzima običnu vrednost i pravi akciju koja vraća tu vrednost.

Na primer:

```
main = do
  return ()
  putStrLn "Ova poruka se ipak ispisuje."
```

Akcija `return ()` ne proizvodi vidljiv efekat, a njen rezultat se zanemaruje. Posle nje se normalno izvršava naredna akcija.

Funkcija `return` može biti korisna kada u nekoj grani uslovnog izraza ne želimo da uradimo ništa, ali nam je ipak potrebna akcija odgovarajućeg tipa. Na primer, sledeći program učitava linije i ispisuje ih obrnuto, sve dok korisnik ne unese praznu liniju:

```
main = do
  line <- getLine
  if null line
    then return ()
    else do
      putStrLn (reverse line)
      main
```

U Haskellu obe grane uslovnog izraza moraju imati isti tip. U ovom primeru grana `else` je akcija tipa `IO ()`. Ona ispisuje obrnutu liniju i zatim ponovo izvršava `main`. Zato i grana `then` mora imati tip `IO ()`. Pošto u slučaju prazne linije ne želimo nikakav vidljiv efekat, koristimo `return ()`. Ovaj primer pokazuje i da akcije mogu biti rekurzivne.

Važno je jasno razlikovati funkcije i akcije. Čista funkcija u Haskellu za iste argumente uvek daje isti rezultat i nema sporedne efekte. Na primer, funkcija

```
reverse :: [a] -> [a]
```

samo računa novu listu iz postojeće liste. Ona ništa ne učitava, ništa ne ispisuje i ne menja spoljašnji svet.

Akcija, sa druge strane, može zavisiti od spoljnog sveta ili može imati efekat na njega. Rezultat akcije `getLine` zavisi od toga šta korisnik unese tokom izvršavanja programa. Akcija `putStrLn "Zdravo!"` ima efekat na spoljašnji svet tako što šalje tekst na standardni izlaz.

Zato je preciznije reći da `putStrLn` ne ispisuje tekst neposredno, već da prima nisku i vraća akciju. Ta akcija ispisuje tekst tek kada bude izvršena, na primer kao deo akcije `main`.

Tip `IO` omogućava da se u sistemu tipova jasno vidi koji delovi programa komuniciraju sa spoljnim svetom. Uobičajena praksa je da najveći deo programa bude napisan pomoću čistih funkcija, dok se akcije koriste tamo gde su zaista potrebni učitavanje, ispisivanje ili neki drugi oblik komunikacije sa okruženjem.

11.2 Rad sa datotekama

Pored čitanja sa standardnog ulaza i ispisivanja na standardni izlaz, programi često treba da čitaju podatke iz datoteka ili da rezultate zapisuju u njih. Za osnovni rad sa tekstualnim datotekama dovoljne su funkcije `readFile`, `writeFile` i `appendFile`.

Njihovi tipovi su:

```
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Tip `FilePath` je tipski sinonim za `String`. Njegova upotreba čini tipove funkcija čitljivijim. Iz tipa `FilePath -> IO String` jasno se vidi da je prvi argument putanja do datoteke, a ne proizvoljna niska.

Funkcija `readFile` prima putanju do datoteke i vraća akciju koja, kada se izvrši, učitava sadržaj te datoteke kao nisku. Na primer, ako se u istom direktorijumu kao program nalazi datoteka `ulaz.txt` sa sadržajem

```
Prva linija
Druga linija
Treca linija
```

njen sadržaj možemo učitati i ispisati ovako:

```
main = do
  contents <- readFile "ulaz.txt"
  putStrLn contents
```

Kao i kod akcije `getLine`, rezultat akcije `readFile "ulaz.txt"` dobijamo pomoću strelice `<-`. Pošto je

```
readFile "ulaz.txt" :: IO String
```

ime `contents` u nastavku `do` bloka ima tip `String`. Zato ga možemo obraditi običnim funkcijama za rad sa niskama i listama.

Na primer, broj linija u datoteci možemo izračunati pomoću funkcija `lines` i `length`:

```
main = do
  contents <- readFile "ulaz.txt"
  print (length (lines contents))
```

Funkcija `lines` deli nisku na listu linija, pa izraz `length (lines contents)` daje broj linija u učitanom tekstu.

Za zapisivanje u datoteku koristi se funkcija `writeFile`. Ona prima putanju do datoteke i nisku koju treba zapisati:

```
main = do
  writeFile "izlaz.txt" "Ovo je tekst zapisan iz Haskell programa.\n"
```

Pokretanjem ovog programa biće napravljena datoteka `izlaz.txt`, ako već ne postoji, i u nju će biti upisana zadata niska. Ako datoteka već postoji, njen prethodni sadržaj biće zamenjen novim sadržajem.

Često želimo da sadržaj jedne datoteke obradimo i rezultat upišemo u drugu datoteku. Na primer, sledeći program učitava tekst iz datoteke `ulaz.txt`, obrće redosled karaktera u celom tekstu i rezultat zapisuje u datoteku `izlaz.txt`:

```
main = do
  contents <- readFile "ulaz.txt"
  let result = reverse contents
  writeFile "izlaz.txt" result
```

Ako ne želimo da zamenimo postojeći sadržaj datoteke, već da novi tekst dodamo na njen kraj, koristi se funkcija `appendFile`. Na primer:

```
main = do
  putStrLn "Unesi novu liniju:"
  line <- getLine
  appendFile "izlaz.txt" (line ++ "\n")
```

Ovaj program učitava jednu liniju sa standardnog ulaza i dodaje je na kraj datoteke. Pošto `getLine` ne vraća znak za novi red, on se eksplicitno dodaje izrazom `line ++ "\n"`.

Funkcije `readFile`, `writeFile` i `appendFile` dovoljne su za mnoge jednostavne programe koji rade sa tekstualnim datotekama. Za složeniji rad postoje i funkcije koje eksplicitno otvaraju datoteke, rade sa rukovaocima datoteka i zatim ih zatvaraju, ali za osnovni nivo je najvažnije razumeti da se čitanje i pisanje datoteka u Haskellu takođe obavlja pomoću akcija.

11.3 Argumenti komandne linije

Programu se prilikom pokretanja iz terminala mogu proslediti dodatne niske, koje nazivamo argumentima komandne linije. One se navode posle imena programa i program ih zatim može koristiti tokom izvršavanja.

Na primer, ako je program preveden u izvršnu datoteku `info`, možemo ga pokrenuti ovako:

```
$ ./info prvi drugi "treci argument"
```

Tada su programu prosleđeni argumenti `"prvi"`, `đrugi"` i `"treci argument"`. Navodnici omogućavaju da više reči bude prosleđeno kao jedan argument.

Za rad sa argumentima komandne linije koriste se akcije `getArgs` i `getProgName`. One nisu deo Prelida, već se nalaze u modulu `System.Environment`. Zbog toga je na početku datoteke potrebno navesti odgovarajući uvoz:

```
import System.Environment
```

Modul je celina koja sadrži definicije funkcija, tipova, klasa i drugih imena. U ovom trenutku dovoljno je znati da se naredbom `import` u program uvode imena koja nisu automatski dostupna. Module ćemo detaljnije razmatrati u narednoj sekciji.

Akcije za rad sa argumentima komandne linije imaju sledeće tipove:

```
getArgs     :: IO [String]
getProgName :: IO String
```

Akcija `getArgs` vraća listu argumenata koji su prosleđeni programu, ne računajući ime samog programa. Akcija `getProgName` vraća ime programa kojim je program pokrenut.

Jednostavan program koji ispisuje ime programa i sve prosleđene argumente možemo napisati ovako:

```
import System.Environment

main = do
  program <- getProgName
  args <- getArgs
  putStrLn ("Ime programa: " ++ program)
  putStrLn "Argumenti:"
  print args
```

Ako datoteku prevedemo tako da izvršna datoteka ima ime `info`, možemo dobiti, na primer:

```
$ ./info
Ime programa: info
Argumenti:
[]

$ ./info ulaz.txt izlaz.txt
Ime programa: info
Argumenti:
["ulaz.txt","izlaz.txt"]

$ ./info "jedan argument" drugi
Ime programa: info
Argumenti:
["jedan argument","drugi"]
```

Pošto je rezultat akcije `getArgs` lista niski, nad njim možemo koristiti uobičajene funkcije za rad sa listama. Na primer, sledeći program očekuje da prvi argument bude ime datoteke, a zatim ispisuje sadržaj te datoteke:

```
import System.Environment

printFile :: FilePath -> IO ()
```

```
printFile path = do
  contents <- readFile path
  putStrLn contents

main = do
  args <- getArgs
  if null args
    then putStrLn "Nije navedeno ime datoteke."
    else printFile (head args)
```

Ako postoji datoteka `ulaz.txt`, program možemo pokrenuti ovako:

```
$ ./program ulaz.txt
```

U ovom primeru akcija `main` najpre učitava argumente komandne linije. Ako lista argumenata nije prazna, prvi argument se koristi kao putanja do datoteke. Sama akcija čitanja i ispisivanja datoteke izdvojena je u funkciju `printFile`, čime je `main` kraći i pregledniji.

Još jedan čest primer jeste program koji očekuje dva argumenta, ime ulazne i ime izlazne datoteke. Sledeći program učitava sadržaj prve datoteke i zapisuje ga u drugu, pri čemu se redosled karaktera u tekstu obrće:

```
import System.Environment

copyReversed :: FilePath -> FilePath -> IO ()
copyReversed input output = do
  contents <- readFile input
  writeFile output (reverse contents)

main = do
  args <- getArgs
  if length args < 2
    then putStrLn "Navedite ulaznu i izlaznu datoteku."
    else copyReversed (args !! 0) (args !! 1)
```

Na primer:

```
$ ./program ulaz.txt izlaz.txt
```

Argumenti komandne linije naročito su korisni za programe koji treba da se lako pokreću iz skripti, terminala ili drugih programa. Za razliku od interaktivnog unosa pomoću `getLine`, argumenti su poznati već u trenutku pokretanja programa.

11.4 Izuzeci

Pri komunikaciji sa spoljnim svetom mogu nastati greške koje se ne mogu u potpunosti predvideti iz samog programa. Na primer, datoteka koju pokušavamo da pročitamo možda ne postoji, program možda nema dozvolu za čitanje ili upis, ili se putanja može odnositi na nedostupan resurs. Takve situacije mogu dovesti do izuzetka i prekinuti izvršavanje programa.

Posmatrajmo jednostavan program koji pokušava da pročita sadržaj datoteke `podaci.txt` i ispiše broj njenih linija:

```
main = do
  contents <- readFile "podaci.txt"
  print (length (lines contents))
```

Ako datoteka postoji, program će ispisati broj linija. Međutim, ako datoteka `podaci.txt` ne postoji, akcija `readFile "podaci.txt"` dovodi do izuzetka i program se prekida s porukom o grešci.

Jedan način da se ovakva situacija obradi jeste upotreba funkcije `catch`. Ona se nalazi u modulu `Control.Exception`, pa je potrebno dodati odgovarajući uvoz:

```
import Control.Exception
```

Prvi argument funkcije `catch` je akcija koju pokušavamo da izvršimo. Drugi argument je funkcija za obradu greške. Ako se prva akcija izvrši bez greške, rezultat je isti kao da `catch` nije ni korišćen. Ako tokom izvršavanja prve akcije nastane izuzetak, on se prosleđuje funkciji za obradu greške.

Prethodni program možemo napisati ovako:

```
import System.IO.Error
import Control.Exception

countLines :: IO ()
countLines = do
  contents <- readFile "podaci.txt"
  print (length (lines contents))

handleError :: IOError -> IO ()
handleError e
  | isDoesNotExistError e = putStrLn "Datoteka ne postoji."
  | otherwise             = ioError e

main = countLines `catch` handleError
```

Izraz

```
countLines `catch` handleError
```

znači da se najpre pokušava izvršavanje akcije `countLines`. Ako se ona izvrši uspešno, program ispisuje broj linija. Ako tokom njenog izvršavanja nastane izuzetak, taj izuzetak se prosleđuje funkciji `handleError`.

U funkciji `handleError` koristi se `isDoesNotExistError` iz modula `System.IO.Error`, koji proverava da li je greška nastala zato što datoteka ne postoji. Ako jeste, ispisuje se odgovarajuća poruka. Ako nije, izuzetak se ponovo prosleđuje pomoću funkcije `ioError`. Time se izbegava da program neprimetno sakrije greške koje ne ume smisleno da obradi.

Na primer, ako datoteka postoji, program može dati rezultat:

```
$ ./program
12
```

Ako datoteka ne postoji, program ispisuje kontrolisanu poruku:

```
$ ./program
Datoteka ne postoji.
```

U praksi ne treba hvatati sve izuzetke bez razlikovanja. Bolje je obraditi samo one greške za koje program zaista zna šta treba da uradi, a ostale ponovo proslediti. U suprotnom, program može sakriti ozbiljnije greške i otežati njihovo pronalaženje.

Izuzetke u Haskellu posebno ima smisla razmatrati kod ulazno-izlaznih akcija, jer se tada program oslanja na spoljašnji svet. I čisti izrazi mogu dovesti do greške pri izvršavanju, na primer ako se primeni parcijalna funkcija na nedozvoljen argument. Ipak, u čistom delu programa često je bolje izbegavati takve situacije i koristiti tipove kao što je `Maybe`, jer se tada mogućnost neuspeha vidi neposredno u tipu funkcije.

12 Moduli

12.1 Uvoz modula

Modul je celina koja sadrži povezane definicije funkcija, tipova, klasa i drugih imena. Podela programa na module omogućava bolju organizaciju koda, jer se srodne definicije mogu izdvojiti u posebne celine i koristiti iz drugih delova programa.

Mnoge funkcije koje smo do sada koristili se nalaze u Prelidu. Taj modul se automatski uvozi u svaki Haskell program, pa se funkcije kao što su `map`, `filter`, `length`, `show`, `read` i `putStrLn` mogu koristiti bez posebnog navođenja. Imena iz drugih modula moraju se eksplicitno uvesti.

Uvoz modula navodi se na početku datoteke, pre definicija funkcija, tipova i klasa. Osnovni oblik uvoza je:

```
import ImeModula
```

Na primer, ako želimo da koristimo funkciju `nub` iz modula `Data.List`, možemo napisati:

```
import Data.List

numUniques :: Eq a => [a] -> Int
numUniques xs = length (nub xs)
```

Funkcija `nub` uklanja ponovljene elemente iz liste, pa funkcija `numUniques` računa broj različitih elemenata liste.

Na primer:

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

ghci> numUniques "banana"
3
```

Ovim oblikom uvoza sva imena koja modul `Data.List` izvozi postaju dostupna u programu. Ako želimo da uvezemo samo neka imena, možemo ih eksplicitno navesti:

```
import Data.List (nub)
```

Tada je iz modula `Data.List` dostupna samo funkcija `nub`. Ovaj oblik uvoza je koristan kada želimo da jasno naglasimo koja imena iz nekog modula zaista koristimo.

Moguće je uvesti i sva imena osim nekih navedenih. To se radi pomoću ključne reči `hiding`:

```
import Data.List hiding (nub)
```

Ovim se iz modula `Data.List` uvode sva imena osim funkcije `nub`. Takav oblik uvoza može biti koristan ako u programu već postoji sopstvena funkcija sa istim imenom.

Još jedan način za izbegavanje sukoba imena jeste kvalifikovani uvoz:

```
import qualified Data.List
```

Kada je modul uveden na ovaj način, njegova imena se koriste zajedno sa imenom modula:

```
import qualified Data.List
```

```
numUniques :: Eq a => [a] -> Int
numUniques xs = length (Data.List.nub xs)
```

Pošto puno ime modula može biti dugo, kvalifikovanom modulu se često daje kraće ime:

```
import qualified Data.List as L

numUniques :: Eq a => [a] -> Int
numUniques xs = length (L.nub xs)
```

U interaktivnom okruženju moduli se mogu uvesti naredbom `:m +`. Na primer:

```
ghci> :m + Data.List
```

Moguće je uvesti i više modula odjednom:

```
ghci> :m + Data.List Data.Char
```

Ako je učitana datoteka koja već sadrži odgovarajuće `import` naredbe, nije potrebno iste module uvoditi zasebno.

Dakle, najčešći oblici uvoza su:

```
import Data.List
import Data.List (nub)
import Data.List hiding (nub)
import qualified Data.List
import qualified Data.List as L
```

Prvi oblik uvodi sva imena iz modula. Drugi uvodi samo navedena imena. Treći uvodi sva imena osim navedenih. Četvrti i peti oblik uvode modul kvalifikovano, pa se njegova imena koriste sa prefiksom.

Detaljnije primere korisnih standardnih modula videćemo u narednoj podsekciji.

12.2 Pregled standardnih modula

Standardna biblioteka Haskella organizovana je kroz module. Modul predstavlja celinu u kojoj su definisane funkcije, tipovi, klase i druge vrednosti povezane određenom namenom. U nastavku ćemo prikazati nekoliko često korišćenih standardnih modula i osnovne načine njihove upotrebe.

Modul `Data.List` sadrži brojne funkcije za rad sa listama. Neke funkcije za liste, kao što su `map`, `filter`, `length`, `take`, `drop`, `zip` i `zipWith`, već su dostupne kroz Prelid. Međutim, mnoge dodatne funkcije nalaze se upravo u modulu `Data.List`.

Na primer:

```
import Data.List
```

Kao što smo već videli, funkcija `nub` uklanja ponovljene elemente iz liste:

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

ghci> nub "banana"
```

```
"ban"
```

Funkcija `sort` uređuje listu, pa tip elemenata mora pripadati klasi `Ord`:

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

ghci> sort "haskell"
"aehklls"
```

Funkcija `group` grupiše susedne jednake elemente. Često se koristi zajedno sa funkcijom `sort`, na primer za brojanje pojavljivanja elemenata:

```
ghci> group "mississippi"
["m","i","ss","i","ss","i","pp","i"]

ghci> map length (group (sort "mississippi"))
[4,1,2,4]
```

Ako želimo da proverimo da li se neka lista pojavljuje kao početak, kraj ili deo druge liste, možemo koristiti funkcije `isPrefixOf`, `isSuffixOf` i `isInfixOf`:

```
ghci> "Ha" `isPrefixOf` "Haskell"
True

ghci> "ell" `isSuffixOf` "Haskell"
True

ghci> "ske" `isInfixOf` "Haskell"
True
```

Funkcija `partition` deli listu na dva dela prema zadatom predikatu. Prvi deo sadrži elemente koji zadovoljavaju predikat, a drugi deo one koji ga ne zadovoljavaju:

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

Modul `Data.Char` sadrži funkcije za rad sa karakterima. Pošto su niske u Haskellu liste karaktera, ove funkcije se često koriste i pri obradi teksta.

Na primer:

```
import Data.Char
```

Neke funkcije iz ovog modula proveravaju kojoj vrsti karakter pripada:

```
isDigit    :: Char -> Bool
isAlpha    :: Char -> Bool
isAlphaNum :: Char -> Bool
isSpace    :: Char -> Bool
```

Na primer:

```
ghci> isDigit '7'
```

```
True

ghci> isAlpha 'A'
True

ghci> isAlphaNum '#'
False

ghci> isSpace ' '
True
```

Ove funkcije su posebno korisne uz funkcije višeg reda, na primer:

```
ghci> filter isAlphaNum "Haskell 3.0!"
"Haskell30"
```

Funkcije `toUpper` i `toLower` menjaju veličinu slova:

```
ghci> map toUpper "Haskell"
"HASKELL"

ghci> map toLower "Haskell"
"haskell"
```

Funkcije `ord` i `chr` prevode karakter u njegov numerički kôd i obratno:

```
ghci> ord 'a'
97

ghci> chr 97
'a'
```

Zbog toga se pomoću njih mogu definisati jednostavne transformacije nad tekstom. Na primer, sledeća funkcija pomera svaki karakter za zadati broj mesta u tabeli karaktera:

```
shiftChars :: Int -> String -> String
shiftChars n xs = map (\c -> chr (ord c + n)) xs
```

Na primer:

```
ghci> shiftChars 1 "abc"
"bcd"

ghci> shiftChars (-1) "bcd"
"abc"
```

Modul `Data.Map` sadrži strukturu podataka koja predstavlja preslikavanje ključeva u vrednosti. Takve strukture nazivaju se mape, rečnici ili asocijativne tabele. Za razliku od obične liste parova, mapa je pogodnija za efikasno traženje vrednosti po ključu.

Pošto `Data.Map` sadrži funkcije čija se imena poklapaju sa imenima iz Prelida, ovaj modul se najčešće uvozi kvalifikovano:

```
import qualified Data.Map as Map
```

Mapu možemo napraviti iz liste uređenih parova pomoću funkcije `Map.fromList`:

```
grades :: Map.Map String Int
grades = Map.fromList
  [ ("Ana", 10)
  , ("Marko", 8)
  , ("Jelena", 9)
  ]
```

Vrednost pridruženu ključu možemo potražiti pomoću funkcije `Map.lookup`:

```
ghci> Map.lookup "Ana" grades
Just 10

ghci> Map.lookup "Petar" grades
Nothing
```

Rezultat je tipa `Maybe Int`, jer traženi ključ može, ali i ne mora postojati u mapi. Ako ključ postoji, dobija se rezultat oblika `Just v`, a ako ne postoji, dobija se `Nothing`.

Nova veza ključ–vrednost može se dodati pomoću funkcije `Map.insert`:

```
ghci> Map.insert "Petar" 7 grades
fromList [("Ana",10),("Jelena",9),("Marko",8),("Petar",7)]
```

Prazna mapa dobija se pomoću vrednosti `Map.empty`:

```
emptyGrades :: Map.Map String Int
emptyGrades = Map.empty
```

Za rad sa mapama ključevi se najčešće moraju porediti, zbog čega moraju pripadati klasi `Ord`. Razlog je to što su mape interno organizovane tako da se ključevi mogu uređivati i efikasno pretraživati.

Modul `Data.Set` sadrži strukturu podataka za predstavljanje skupova. Skup čuva elemente bez ponavljanja. Kao i kod mapa, osnovne operacije nad skupovima zahtevaju da se elementi mogu porediti, tj. da pripadaju klasi `Ord`.

Ovaj modul se takođe najčešće uvozi kvalifikovano:

```
import qualified Data.Set as Set
```

Skup možemo napraviti iz liste pomoću funkcije `Set.fromList`:

```
ghci> Set.fromList [1,2,1,3,2,4]
fromList [1,2,3,4]
```

Duplikati se pritom uklanjaju. Obrnuto, skup se može pretvoriti u listu pomoću funkcije `Set.toList`:

```
ghci> Set.toList (Set.fromList [3,1,3,2])
[1,2,3]
```

Pripadnost elementa skupu proverava se pomoću funkcije `Set.member`:

```
ghci> Set.member 2 (Set.fromList [1,2,3])
True

ghci> Set.member 5 (Set.fromList [1,2,3])
False
```

Nad skupovima se mogu koristiti i uobičajene skupovne operacije. Na primer:

```
ghci> Set.union (Set.fromList [1,2,3]) (Set.fromList [3,4,5])
fromList [1,2,3,4,5]

ghci> Set.intersection (Set.fromList [1,2,3]) (Set.fromList [3,4,5])
fromList [3]

ghci> Set.difference (Set.fromList [1,2,3]) (Set.fromList [3,4,5])
fromList [1,2]
```

Skupovi su korisni kada je važno da se elementi ne ponavljaju ili kada često proveravamo da li se neki element nalazi u kolekciji.

Pored ovih modula, u praksi se često koriste i drugi standardni moduli. Na primer, modul `System.Environment` sadrži funkcije za pristup argumentima komandne linije i drugim informacijama iz okruženja programa, modul `System.IO.Error` funkcije za obradu ulazno-izlaznih grešaka, a modul `System.Directory` funkcije za rad sa sistemom datoteka, kao što su provera postojanja, brisanje i preimenovanje datoteka. Neke od ovih modula već smo koristili u prethodnoj sekciji. U većim programima izbor modula zavisi od konkretnog problema koji se rešava.

12.3 Definisanje sopstvenih modula

Pored korišćenja postojećih, u Haskellu možemo definisati i sopstvene module. To omogućava da se veći program podeli u više manjih celina i da se povezane definicije smeste na jedno mesto. Na taj način se program lakše čita, održava i ponovo koristi.

Modul se najčešće definiše u posebnoj datoteci. Na početku datoteke navodi se ime modula i lista imena koja taj modul izvozi. Na primer, možemo napraviti datoteku `Measurements.hs` sa sledećim sadržajem:

```
module Measurements
  ( rectangleArea
  , rectanglePerimeter
  , circleArea
  , circleCircumference
  ) where

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b

rectanglePerimeter :: Float -> Float -> Float
rectanglePerimeter a b = 2 * (a + b)
```

```
circleArea :: Float -> Float
circleArea r = pi * square r

circleCircumference :: Float -> Float
circleCircumference r = 2 * pi * r

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

Prva linija deklaracije kaže da se definiše modul `Measurements`. Ime modula mora početi velikim slovom. Kada se modul uvozi iz druge datoteke, ime datoteke treba da prati ime modula. Zato se modul `Measurements` nalazi u datoteci `Measurements.hs`.

U zagradama posle imena modula navode se imena koja modul izvozi:

```
module Measurements
  ( rectangleArea
  , rectanglePerimeter
  , circleArea
  , circleCircumference
  ) where
```

Izvezena imena čine javni deo modula. To znači da će druga datoteka koja uveze modul `Measurements` moći neposredno da koristi funkcije `rectangleArea`, `rectanglePerimeter`, `circleArea` i `circleCircumference`.

Funkcija `square` je takođe definisana u modulu, ali nije navedena u listi izvezenih imena. Zbog toga ona ostaje pomoćna funkcija dostupna samo unutar samog modula. Na primer, funkcija `circleArea` koristi `square`, ali korisnik modula ne mora da zna kako je površina kruga interno izračunata.

Ovaj izbor je važan pri organizaciji programa. Modul može sadržati pomoćne definicije koje služe za implementaciju, ali ne moraju biti deo njegovog javnog interfejsa. Time se jasnije razdvaja ono što modul nudi drugim delovima programa od načina na koji je to interno realizovano.

Modul možemo koristiti iz druge datoteke pomoću naredbe `import`. Na primer, u datoteci `Main.hs` možemo napisati:

```
import Measurements

main = do
  print (rectangleArea 3 4)
  print (rectanglePerimeter 3 4)
  print (circleArea 2)
  print (circleCircumference 2)
```

Ako su datoteke `Main.hs` i `Measurements.hs` u istom direktorijumu, program se može prevesti komandom:

```
$ ghc Main.hs
```

Prilikom prevođenja, kompajler pronalazi modul `Measurements`, prevodi ga i omogućava da se njegova izvezena imena koriste u glavnom programu.

Kao što je već rečeno, svako ime definisano u modulu može se koristiti unutar tog modula,

ali se iz drugih modula mogu koristiti samo ona imena koja su izvezena. Ako želimo da izvezemo sva imena definisana u modulu, lista izvezenih imena može se izostaviti:

```
module Measurements where
```

Ipak, u većim programima često je bolje eksplicitno navesti šta se izvozi. Tako se jasnije vidi koji deo modula je namenjen za upotrebu spolja, a koje definicije su samo pomoćne.

Haskell podržava i hijerarhijska imena modula. Na primer, modul `Measurements.Circle` se može nalaziti u datoteci `Circle.hs`, unutar direktorijuma `Measurements`. Takva organizacija je korisna u većim projektima, kada program ima više povezanih celina.

13 Funktori

13.1 Klasa Functor

Jedan od čestih obrazaca u radu sa listama jeste primena iste funkcije na svaki element liste. Na primer, funkciju koja duplira svaki element liste možemo definisati rekurzivno:

```
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = 2 * x : doubleList xs
```

Ova definicija ne menja oblik liste. Prazna lista ostaje prazna, a u nepraznoj listi funkcija se primenjuje na glavu liste, dok se rep obrađuje rekurzivno. Redosled elemenata se pritom ne menja.

Prethodnu funkciju možemo zapisati kraće preko bibliotečke funkcije `map`:

```
doubleList :: [Int] -> [Int]
doubleList = map (*2)
```

Ako ovu funkciju primenimo na konkretnu listu, dobijamo:

```
ghci> doubleList [1,2,3,4]
[2,4,6,8]
```

Funkcija `map` opisuje ideju preslikavanja elemenata liste. Međutim, slična ideja može se primeniti i na druge parametrizovane tipove, ne samo na liste. Parametrizovani tipovi za koje je definisana ovakva operacija preslikavanja nazivaju se funktori. U Haskellu se taj pojam izražava klasom `Functor`.

Osnovni deo klase `Functor` može se prikazati ovako:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Funkcija `fmap` prima funkciju tipa `a -> b` i vrednost tipa `f a`, a kao rezultat vraća vrednost tipa `f b`. Dakle, ona omogućava da se funkcija primeni na vrednosti koje se nalaze unutar neke strukture ili konteksta označenog sa `f`.

Važno je primetiti da `f` u ovoj definiciji nije konkretan tip kao `Int`, `Bool` ili `String`. Iz izraza `f a` i `f b` vidi se da `f` mora biti konstruktor tipa koji prima jedan tipski argument. Na primer, lista je takav konstruktor tipa, jer od tipa `Int` pravi tip `[Int]`, od tipa `String` pravi tip `[String]`, i tako dalje.

Liste su najjednostavniji primer funktora. Za liste je `fmap` upravo funkcija `map`:

```
instance Functor [] where
  fmap = map
```

U deklaraciji instance piše `[]` zato što se instanca ne definiše za jedan konkretan tip liste, kao što je `[Int]`, već za sam konstruktor liste.

Zato se `fmap` nad listama ponaša isto kao `map`:

```
ghci> fmap (*2) [1,2,3,4]
[2,4,6,8]
```

```
ghci> fmap length ["Haskell", "FP", ""]
[7,2,0]
```

U prvom primeru funkcija `(*2)` primenjuje se na svaki broj u listi. U drugom primeru funkcija `length` primenjuje se na svaku nisku u listi, pa se od liste niski dobija lista brojeva.

Prema tome, za liste je `fmap` samo opštiji naziv za već poznatu funkciju `map`. Razlog za uvođenje klase `Functor` jeste to što se ista osnovna ideja može definisati i za druge parametrizovane tipove. Takve primere razmotrićemo u nastavku.

13.2 Primeri funktora

U prethodnoj podsekciji videli smo da su liste funktori i da je za liste funkcija `fmap` zapravo funkcija `map`. Međutim, funktori nisu ograničeni samo na liste.

Još jedan važan primer jeste tipski konstruktor `Maybe`. Podsetimo se da se vrednost tipa `Maybe a` može shvatiti kao vrednost tipa `a` koja možda postoji:

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

Instanca klase `Functor` za tipski konstruktor `Maybe` definisana je na sledeći način:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Ako je vrednost oblika `Nothing`, nema vrednosti na koju bi se funkcija primenila, pa rezultat ostaje `Nothing`. Ako je vrednost oblika `Just x`, funkcija se primenjuje na `x`, a rezultat se ponovo pakuje u konstruktor `Just`.

Na primer:

```
ghci> fmap (*2) (Just 3)
Just 6

ghci> fmap (*2) Nothing
Nothing

ghci> fmap not (Just False)
Just True
```

Ovaj primer pokazuje da `fmap` čuva strukturu vrednosti. U slučaju `Maybe`, to znači da se odsustvo vrednosti ne zamenjuje proizvoljnom vrednošću, već ostaje predstavljeno konstruktorom `Nothing`.

Funktor ne mora predstavljati kolekciju elemenata u uobičajenom smislu. Na primer, tipski konstruktor `IO` takođe je instanca klase `Functor`. Vrednost tipa `IO a` predstavlja akciju koja, kada se izvrši, može dati rezultat tipa `a`. Funkcija `fmap` tada ne deluje na elemente kolekcije, već na rezultat akcije.

Instanca klase `Functor` za `IO` može se opisati ovako:

```
instance Functor IO where
  fmap f action = do
    result <- action
    return (f result)
```

Ova definicija kaže da se od postojeće akcije pravi nova akcija. Nova akcija najpre izvršava akciju `action`, zatim funkciju `f` primenjuje na njen rezultat, a dobijenu vrednost vraća kao svoj rezultat.

Na primer, program koji učitava liniju, zatim je obrće i ispisuje možemo napisati ovako:

```
main = do
  line <- getLine
  let reversed = reverse line
  putStrLn reversed
```

Pošto se u ovom programu funkcija `reverse` primenjuje samo na rezultat akcije `getLine`, isti program možemo zapisati kraće pomoću `fmap`:

```
main = do
  line <- fmap reverse getLine
  putStrLn line
```

Izraz `fmap reverse getLine` ima tip `IO String`. To je akcija koja učitava liniju sa standardnog ulaza, ali kao rezultat daje obrnutu učitanu liniju. Kada se ta akcija izvrši pomoću `<-`, ime `line` se vezuje za već obrnut tekst.

Na primer, ako korisnik unese tekst `haskell`, program ispisuje:

```
lleksah
```

Možemo definisati i sopstveni parametrizovani tip čiji je tipski konstruktor instanca klase `Functor`. Posmatrajmo jednostavan tip za trodimenzionalni vektor:

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

Vrednost tipa `Vector3 a` sadrži tri vrednosti tipa `a`. Zbog toga je prirodno da `fmap` primeni zadatu funkciju na svaku od te tri vrednosti:

```
instance Functor Vector3 where
  fmap f (Vector3 x y z) = Vector3 (f x) (f y) (f z)
```

Na primer:

```
ghci> fmap (*2) (Vector3 1 2 3)
Vector3 2 4 6

ghci> fmap even (Vector3 1 2 3)
Vector3 False True False

ghci> fmap length (Vector3 "ab" "haskell" "xyz")
Vector3 2 7 3
```

U ovom primeru `fmap` ne menja oblik vektora. Vektor i dalje ima tri komponente, ali se funkcija primenjuje na svaku od njih. Zato se tip rezultata može promeniti iz `Vector3 a` u `Vector3 b`, dok sama struktura ostaje ista.

Ovi primeri pokazuju da se isti interfejs, predstavljen funkcijom `fmap`, može koristiti nad različitim tipovima. Kod lista `fmap` obrađuje svaki element liste, kod `Maybe` obrađuje postojeću

vrednost ako ona postoji, kod `IO` obrađuje rezultat akcije, a kod tipa `Vector3` obrađuje svaku komponentu vektora.

13.3 Zakoni funktora

Da bi neki tipski konstruktor bio smislena instanca klase `Functor`, nije dovoljno samo definisati funkciju `fmap` odgovarajućeg tipa. Funkcija `fmap` treba da se ponaša kao preslikavanje u okviru datog konteksta. To ponašanje se opisuje pomoću dva zakona funktora.

Prvi zakon kaže da preslikavanje identičke funkcije ne sme promeniti vrednost:

```
fmap id = id
```

Drugim rečima, ako funkcija `id` ne menja običnu vrednost, onda ni `fmap id` ne treba da menja vrednost unutar funktora.

Na primer:

```
ghci> fmap id [1,2,3]
[1,2,3]

ghci> fmap id (Just 5)
Just 5

ghci> fmap id Nothing
Nothing
```

Drugi zakon kaže da `fmap` čuva kompoziciju funkcija:

```
fmap (g . h) = fmap g . fmap h
```

To znači da je svejedno da li najpre sastavimo funkcije `g` i `h`, pa zatim jednom primenimo `fmap`, ili prvo primenimo `fmap h`, a zatim `fmap g`.

Na primer, izraz

```
fmap (not . even) [1,2,3]
```

daje isti rezultat kao izraz

```
(fmap not . fmap even) [1,2,3]
```

U oba slučaja dobija se:

```
[True,False,True]
```

Zaista, prvi izraz najpre posmatra kompoziciju `not . even`, pa tu funkciju primenjuje na svaki element liste. Drugi izraz najpre funkcijom `even` preslikava listu celih brojeva u listu logičkih vrednosti, a zatim na svaku od tih vrednosti primenjuje funkciju `not`. Rezultat mora biti isti.

Ovi zakoni nisu provereni od strane kompajlera. Haskell može proveriti da li definicija funkcije `fmap` ima odgovarajući tip, ali ne može automatski proveriti da li ona zadovoljava navedene jednakosti. Zato je odgovornost programera da pri definisanju instance klase `Functor` obezbedi da se `fmap` zaista ponaša kao preslikavanje.

Kod lista, zakoni obezbeđuju da `fmap` samo primenjuje funkciju na elemente liste, bez promene same strukture liste. To znači da se elementi ne smeju dodavati, izbacivati niti preuređivati.

Sama činjenica da neka funkcija ima odgovarajući tip nije dovoljna. Zakoni funktora preciziraju očekivano ponašanje funkcije `fmap`. Standardne instance, kao što su instance za liste, `Maybe` i `IO`, zadovoljavaju ove zakone. Kada definišemo sopstvenu instancu klase `Functor`, treba proveriti da li se i ona ponaša u skladu sa njima.

13.4 Klasa Applicative

Funkcija `fmap` omogućava da funkciju jednog argumenta primenimo na vrednost koja se nalazi u nekom funktorskom kontekstu. Na primer:

```
ghci> fmap (*2) (Just 3)
Just 6

ghci> fmap (*2) Nothing
Nothing
```

Međutim, sama funkcija `fmap` nije dovoljna kada želimo da funkciju sa dva ili više argumenata primenimo na više vrednosti koje se nalaze u istom kontekstu. Posmatrajmo, na primer, sabiranje dve vrednosti tipa `Maybe Int`. Jedan način je da posebno obradimo sve slučajeve:

```
maybeAdd :: Maybe Int -> Maybe Int -> Maybe Int
maybeAdd (Just x) (Just y) = Just (x + y)
maybeAdd _        _        = Nothing
```

Na primer:

```
ghci> maybeAdd (Just 2) (Just 3)
Just 5

ghci> maybeAdd (Just 2) Nothing
Nothing
```

Ova definicija je ispravna, ali nije dovoljno opšta. Za svaku novu funkciju od dva argumenta morali bismo da pišemo sličan kôd. Još veći problem nastaje kod funkcija sa tri ili više argumenata.

Pogledajmo zašto običan `fmap` ne rešava ceo problem. Definišimo najpre funkciju za sabiranje celih brojeva:

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

Kao što je poznato, tip `Int -> Int -> Int` možemo čitati kao `Int -> (Int -> Int)`. Drugim rečima, kada funkcija `add` dobije prvi argument, rezultat je funkcija koja očekuje još jedan argument.

Zato primena funkcije `fmap` na `add` i jednu vrednost tipa `Maybe Int` daje vrednost koja sadrži funkciju:

```
ghci> :t fmap add (Just 2)
fmap add (Just 2) :: Maybe (Int -> Int)
```

Izraz `fmap add (Just 2)` možemo shvatiti kao vrednost tipa `Maybe (Int -> Int)` koja, u slučaju uspeha, sadrži funkciju dobijenu delimičnom primenom funkcije `add` na argument `2`. Drugim rečima, u kontekstu `Maybe` sada imamo funkciju koja očekuje još jedan ceo broj.

Zato nam je, pored funkcije `fmap`, potrebna još jedna operacija. Za tip `Maybe` možemo je najpre opisati ovako:

```
maybeApply :: Maybe (a -> b) -> Maybe a -> Maybe b
maybeApply (Just f) (Just x) = Just (f x)
maybeApply _        _        = Nothing
```

Funkcija `maybeApply` primenjuje funkciju koja se nalazi u `Maybe` kontekstu na vrednost koja se takođe nalazi u `Maybe` kontekstu. Ako postoje i funkcija i argument, dobija se `Just` rezultat. Ako je bar jedna od tih vrednosti `Nothing`, rezultat je `Nothing`.

Na primer:

```
ghci> maybeApply (fmap add (Just 2)) (Just 3)
Just 5

ghci> maybeApply (fmap add (Just 2)) Nothing
Nothing
```

Klasa `Applicative` uopštava upravo ovu ideju. Njen osnovni deo može se prikazati ovako:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Stvarna definicija klase `Applicative` u standardnoj biblioteci sadrži još neke metode, ali su za osnovnu upotrebu najvažnije funkcija `pure` i operator `<*>`.

Ograničenje `Functor f =>` znači da svaki aplikativni funktor mora prethodno biti funktor. Funkcija `pure` običnu vrednost stavlja u dati kontekst, dok operator `<*>` primenjuje funkciju koja je u nekom kontekstu na vrednost koja je u istom tom kontekstu.

Pošto svaka instanca klase `Applicative` mora biti i instanca klase `Functor`, između funkcija `fmap`, `pure` i operatora `<*>` postoji očekivana veza:

```
fmap f x = pure f <*> x
```

Leva strana kaže da se funkcija `f` preslikava preko vrednosti `x`. Desna strana najpre funkciju `f` stavlja u aplikativni kontekst pomoću `pure`, a zatim je operatorom `<*>` primenjuje na vrednost `x`. Drugim rečima, aplikativni funktori proširuju mogućnosti funktora, jer pored običnog preslikavanja funkcije nad jednim kontekstom omogućavaju i primenu funkcija na više vrednosti u istom kontekstu.

Za tip `Maybe` instanca klase `Applicative` može se opisati ovako:

```
instance Applicative Maybe where
  pure = Just

  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

Funkcija `pure` za `Maybe` pravi vrednost oblika `Just`. Operator `<*>` radi isto što i prethodno

uvedena funkcija `maybeApply`. Ako je sa leve strane `Just f`, funkcija `f` se primenjuje na desni argument pomoću `fmap`. Ako je sa leve strane `Nothing`, rezultat je `Nothing`.

Sada prethodne primere možemo zapisati pomoću operatora `<*>`:

```
ghci> fmap add (Just 2) <*> Just 3
Just 5

ghci> fmap add (Just 2) <*> Nothing
Nothing
```

Funkcija `pure` omogućava da i samu funkciju stavimo u kontekst. Zato se sabiranje dve možda-vrednosti može zapisati ovako:

```
ghci> pure add <*> Just 2 <*> Just 3
Just 5

ghci> pure add <*> Just 2 <*> Nothing
Nothing

ghci> pure (*) <*> Just 4 <*> Just 5
Just 20
```

Operator `<*>` je levoasocijativan, pa se izraz

```
pure add <*> Just 2 <*> Just 3
```

čita kao:

```
(pure add <*> Just 2) <*> Just 3
```

Najpre se funkcija `add` stavlja u kontekst pomoću `pure`. Zatim se primenjuje na vrednost `Just 2`, čime se dobija funkcija koja očekuje još jedan argument. Nakon toga se ta funkcija primenjuje na `Just 3`, pa se dobija rezultat `Just 5`.

U praksi se često koristi i operator `<$>`, koji je infiksni oblik funkcije `fmap`:

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

Zato se izraz

```
pure add <*> Just 2 <*> Just 3
```

može zapisati kraće kao:

```
ghci> add <$> Just 2 <*> Just 3
Just 5
```

Ako neki od argumenata ne postoji, rezultat je `Nothing`:

```
ghci> add <$> Just 2 <*> Nothing
Nothing
```

Obični funktori omogućavaju da funkciju jednog argumenta primenimo na vrednost u nekom

kontekstu. Aplikativni funktori omogućavaju da funkcije sa više argumenata primenjujemo na više vrednosti koje se nalaze u istom kontekstu. Kod tipa `Maybe`, taj kontekst predstavlja mogućnost odsustva vrednosti, pa se neuspeh automatski prenosi kroz ceo izraz.

13.5 Primeri aplikativnih funktora

U nastavku ćemo razmatriti još dva važna aplikativna funktora, liste i `IO`. Instanca za liste može se opisati ovako:

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Funkcija `pure` stavlja vrednost u listu sa jednim elementom. Operator `<*>` uzima listu funkcija i listu argumenata, pa primenjuje svaku funkciju iz prve liste na svaki element iz druge liste.

Na primer:

```
ghci> pure (*2) <*> [1,2,3]
[2,4,6]

ghci> [(+1), (*2)] <*> [10,20]
[11,21,20,40]
```

U drugom primeru funkcija `(+1)` primenjuje se na oba elementa liste `[10,20]`, a zatim se funkcija `(*2)` primenjuje na oba elementa iste liste.

Aplikativni zapis posebno je koristan kada želimo da primenimo funkciju više argumenata na više lista. Na primer:

```
ghci> pure (*) <*> [1,2] <*> [10,20]
[10,20,20,40]
```

Isti izraz možemo zapisati i pomoću operatora `<$>`:

```
ghci> (*) <$> [1,2] <*> [10,20]
[10,20,20,40]
```

Ovaj izraz računa sve proizvode u kojima se prvi činilac bira iz liste `[1,2]`, a drugi iz liste `[10,20]`. Dakle, ne množe se elementi po pozicijama, već se razmatraju sve kombinacije elemenata iz dve liste. Zato se rezultat može shvatiti kao:

```
[1 * 10, 1 * 20, 2 * 10, 2 * 20]
```

Isti postupak mogli bismo zapisati i pomoću izlistavanja:

```
products :: [Int] -> [Int] -> [Int]
products xs ys = [x * y | x <- xs, y <- ys]
```

Aplikativni zapis iste funkcije je nešto kraći:

```
products :: [Int] -> [Int] -> [Int]
products xs ys = (*) <$> xs <*> ys
```

Na primer:

```
ghci> products [1,2,3] [10,20]
[10,20,20,40,30,60]
```

Tipski konstruktor `IO` takođe je instanca klase `Applicative`. Vrednost tipa `IO a` predstavlja akciju koja, kada se izvrši, daje rezultat tipa `a`. Instanca klase `Applicative` za `IO` može se opisati ovako:

```
instance Applicative IO where
  pure = return

  mf <*> mx = do
    f <- mf
    x <- mx
    return (f x)
```

Funkcija `pure` pravi akciju koja ne obavlja nikakav ulazno-izlazni rad, već samo vraća datu vrednost kao rezultat. Operator `<*>` pravi novu akciju od dve akcije. Prva akcija daje funkciju, druga akcija daje argument, a rezultat nove akcije dobija se primenom te funkcije na taj argument.

Na primer, program koji učitava dve linije i ispisuje njihovu konkatenaciju možemo napisati ovako:

```
main = do
  x <- getLine
  y <- getLine
  putStrLn (x ++ y)
```

Pošto se ovde funkcija `(++)` primenjuje na rezultate dve akcije `getLine`, isti program možemo zapisati i aplikativno:

```
main = do
  text <- (++) <$> getLine <*> getLine
  putStrLn text
```

Dakle, izraz

```
(++) <$> getLine <*> getLine
```

ima tip `IO String`. To je akcija koja najpre učitava jednu liniju, zatim drugu liniju, a kao rezultat daje njihovu konkatenaciju.

Prethodni primeri pokazuju različita značenja aplikativne primene. Kod tipa `Maybe`, aplikativni funktor omogućava rad sa vrednostima koje možda ne postoje. Kod lista omogućava rad sa više mogućih vrednosti. Kod tipa `IO` omogućava da se rezultati više akcija kombinuju običnom funkcijom. U svim slučajevima, ista sintaksa izražava istu opštu ideju: čista funkcija primenjuje se na argumente koji se nalaze u nekom kontekstu.

I za aplikativne funktore postoje zakoni, koje ovde nećemo posebno navoditi. Kao i kod funktora, Haskell proverava tipove, dok je poštovanje zakona odgovornost programera koji definiše instancu.

14 Monade

14.1 Klasa Monad

U prethodnoj sekciji videli smo da funktori i aplikativni funktori omogućavaju rad sa vrednostima koje se nalaze u nekom kontekstu. Funkcija `fmap` omogućava da običnu funkciju primenimo na vrednost u funktorskom kontekstu:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Aplikativni funktori proširuju ovu ideju. Operator `<*>` omogućava da funkciju koja je i sama u kontekstu primenimo na vrednost u istom tom kontekstu:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Međutim, postoji još jedan čest obrazac. Pretpostavimo da imamo vrednost u nekom kontekstu, ali da funkcija koju želimo da primenimo ne vraća običnu vrednost, već ponovo vraća vrednost u istom tipu konteksta. Drugim rečima, imamo vrednost tipa `m a` i funkciju tipa `a -> m b`. Želimo da ih povežemo tako da dobijemo rezultat tipa `m b`.

Upravo ovu ideju izražava operator `>>=`:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Operator `>>=` naziva se operator vezivanja (engl. *bind*). On povezuje monadnu vrednost sa funkcijom koja takođe vraća monadnu vrednost. Pri tome se vodi računa o značenju konteksta u kome se vrednosti nalaze.

Jedan od najjednostavnijih primera je tip `Maybe`. Podsetimo se, vrednost tipa `Maybe a` predstavlja vrednost tipa `a` koja možda postoji. Ako imamo vrednost `Just x`, izračunavanje je uspelo i rezultat je `x`. Ako imamo vrednost `Nothing`, izračunavanje nije dalo rezultat.

Posmatrajmo najpre sledeću funkciju:

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing  _ = Nothing
applyMaybe (Just x) f = f x
```

Funkcija `applyMaybe` prima vrednost tipa `Maybe a` i funkciju koja od obične vrednosti tipa `a` pravi vrednost tipa `Maybe b`. Ako je prvi argument `Nothing`, nema vrednosti koju bismo mogli da prosledimo funkciji, pa je rezultat takođe `Nothing`. Ako je prvi argument oblika `Just x`, funkcija se primenjuje na vrednost `x`. Na primer:

```
ghci> applyMaybe (Just 3) (\x -> Just (x + 1))
Just 4

ghci> applyMaybe Nothing (\x -> Just (x + 1))
Nothing

ghci> applyMaybe (Just 3) (\x -> if x > 0 then Just x else Nothing)
Just 3

ghci> applyMaybe (Just (-3)) (\x -> if x > 0 then Just x else Nothing)
Nothing
```

Dakle, ako je početna vrednost `Nothing`, izračunavanje se ne nastavlja. Ako je početna vrednost oblika `Just x`, funkcija se primenjuje na `x`, ali i ona sama može vratiti `Nothing`.

Klasa `Monad` uopštava ovaj obrazac. Njen osnovni deo može se prikazati ovako:

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Stvarna definicija klase `Monad` u standardnoj biblioteci sadrži još neke elemente, ali su za osnovnu upotrebu najvažniji operator `>>=` i funkcija `return`.

Ograničenje `Applicative m =>` znači da svaka monada mora prethodno biti aplikativni funktor. Funkcija `return` običnu vrednost stavlja u monadni kontekst. U savremenom Haskellu ona ima isto značenje kao funkcija `pure`, ali se naziv `return` tradicionalno koristi u monadnom programiranju.

Podsetimo se da `return` u Haskellu ne znači isto što i naredba `return` u imperativnim jezicima. Ona ne prekida izvršavanje funkcije, već samo pravi monadnu vrednost od obične vrednosti.

Za tip `Maybe`, instanca klase `Monad` može se opisati ovako:

```
instance Monad Maybe where
  Nothing  >>= _ = Nothing
  Just x   >>= f = f x

  return x = Just x
```

Ova definicija je ista kao prethodno uvedena funkcija `applyMaybe`, samo što sada koristimo opšti operator `>>=`. Ako je leva strana `Nothing`, rezultat je `Nothing`. Ako je leva strana `Just x`, vrednost `x` se prosleđuje funkciji sa desne strane.

Na primer:

```
ghci> Just 9 >>= \x -> return (x * 10)
Just 90

ghci> Nothing >>= \x -> return (x * 10)
Nothing

ghci> Just 5 >>= \x -> if even x then Just x else Nothing
Nothing

ghci> Just 6 >>= \x -> if even x then Just x else Nothing
Just 6
```

U prvom primeru vrednost `9` se nalazi u konstruktoru `Just`, pa se prosleđuje funkciji koja vraća `Just 90`. U drugom primeru nema vrednosti koju bismo mogli proslediti funkciji, pa rezultat ostaje `Nothing`. U trećem i četvrtom primeru funkcija sa desne strane proverava dodatni uslov i sama odlučuje da li će vratiti uspešan rezultat ili neuspeh.

Ovakav obrazac posebno je koristan kada imamo više uzastopnih izračunavanja koja mogu da ne uspeju. Posmatrajmo jednostavan tip aritmetičkih izraza koji sadrže celobrojne vrednosti i operaciju deljenja:

```
data Term = Const Int | Quot Term Term
```

Funkciju koja računa vrednosti ovakvih izraza možemo definisati na sledeći način:

```
eval :: Term -> Int
eval (Const n) = n
eval (Quot t u) = eval t `div` eval u
```

Ova definicija je jednostavna, ali nije bezbedna. Ako se pri izračunavanju desnog podizraza dobije nula, doći će do greške deljenja nulom. Zato možemo definisati bezbednu verziju deljenja:

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

Sada evaluator treba da vraća vrednost tipa `Maybe Int`, jer izračunavanje izraza može da ne uspe:

```
evalSafe :: Term -> Maybe Int
evalSafe (Const n) = Just n
evalSafe (Quot t u) =
  evalSafe t >>= \n ->
  evalSafe u >>= \m ->
  safeQuot n m
```

Definicija se čita sledećim redom. Najpre se izračunava levi podizraz `t`. Ako njegovo izračunavanje ne uspe, ceo rezultat je `Nothing`. Ako uspe, dobijena vrednost se imenuje sa `n`. Zatim se izračunava desni podizraz `u`. Ako to izračunavanje ne uspe, ceo rezultat je `Nothing`. Ako uspe, dobijena vrednost se imenuje sa `m`. Na kraju se primenjuje funkcija `safeQuot n m`, koja takođe može vratiti `Nothing` ako je `m` jednako nuli.

Na primer:

```
ghci> evalSafe (Quot (Const 10) (Const 2))
Just 5

ghci> evalSafe (Quot (Const 10) (Const 0))
Nothing

ghci> evalSafe (Quot (Const 10) (Quot (Const 1) (Const 0)))
Nothing
```

U poslednjem primeru neuspeh se dešava pri izračunavanju desnog podizraza. Pošto operator `>>=` za `Maybe` automatski prenosi `Nothing`, nije potrebno da u svakoj grani ručno proveravamo da li je prethodni korak uspeo.

Prema tome, monada `Maybe` omogućava da se niz izračunavanja koja mogu da ne uspeju piše kao običan niz koraka. Mogućnost neuspeha ostaje vidljiva u tipu rezultata, ali se samo prosleđivanje neuspeha kroz izračunavanje prepušta definiciji operatora `>>=`.

14.2 Do notacija

Operator `>>=` omogućava da povežemo više monadnih izračunavanja. Međutim, ako takvih izračunavanja ima više, zapis pomoću operatora `>>=` i lambda izraza može postati manje pregledan. Zbog toga Haskell uvodi posebnu sintaksu, koja se naziva `do` notacija.

U prethodnoj podsekciji definisali smo funkciju `evalSafe`, koja računa vrednost izraza i vraća `Nothing` ako se negde pojavi deljenje nulom. Pomoću `do` notacije ista funkcija može se zapisati ovako:

```
evalSafe :: Term -> Maybe Int
evalSafe (Const n) = Just n
evalSafe (Quot t u) = do
  n <- evalSafe t
  m <- evalSafe u
  safeQuot n m
```

Ovaj zapis ima isto značenje kao zapis pomoću operatora `>>=`. Prvi red u `do` bloku izračunava levi podizraz `t`. Ako je rezultat tog izračunavanja oblika `Just n`, vrednost `n` se koristi u nastavku. Ako je rezultat `Nothing`, ceo `do` izraz ima vrednost `Nothing`.

Slično tome, drugi red izračunava desni podizraz `u`. Ako i to izračunavanje uspe, njegova vrednost se imenuje sa `m`. Poslednji red zatim primenjuje funkciju `safeQuot` na dobijene vrednosti. Pošto `safeQuot n m` takođe ima tip `Maybe Int`, ono može vratiti ili uspešan rezultat, ili `Nothing` ako je `m` jednako nuli.

Važno je razumeti da zapis

```
n <- evalSafe t
```

ne znači da se vrednost izdvaja iz konteksta `Maybe`. Ako bi to bilo moguće, izgubili bismo informaciju o tome da izračunavanje može da ne uspe. Umesto toga, ovaj zapis je samo čitljiviji oblik monadnog vezivanja. Monada određuje šta se dešava sa kontekstom, a u slučaju `Maybe` to znači da se `Nothing` automatski prenosi kroz ostatak izračunavanja.

Ovaj primer pokazuje tipičnu upotrebu `do` notacije za tip `Maybe`. Program se piše kao niz koraka koji zavise jedan od drugog, dok se mogućnost neuspeha ne obrađuje ručno u svakom koraku. Ako neki korak vrati `Nothing`, dalji koraci se ne izvršavaju na uobičajen način, već ceo izraz dobija vrednost `Nothing`.

Do sada smo `do` notaciju najčešće koristili kod programa koji rade sa ulazom i izlazom. Na primer:

```
main :: IO ()
main = do
  line <- getLine
  putStrLn line
```

Sada možemo preciznije objasniti zašto je takav zapis moguć. Tip `IO` je takođe instanca klase `Monad`. Vrednost tipa `IO a` predstavlja akciju koja, kada se izvrši, može dati rezultat tipa `a`. Zbog toga se i akcije tipa `IO` mogu povezivati pomoću `do` notacije.

U prethodnom primeru akcija `getLine` ima tip `IO String`. Kada se ona izvrši, njen rezultat se imenuje sa `line`. Zatim se vrednost `line` koristi u narednoj akciji, `putStrLn line`. Ceo `do` blok predstavlja jednu veću akciju tipa `IO ()`.

Prema tome, `do` notacija nije posebna sintaksa samo za `IO`. Ona se može koristiti sa svakim tipom koji je instanca klase `Monad`. Kod tipa `Maybe`, ona povezuje izračunavanja koja mogu da ne uspeju. Kod tipa `IO`, ona povezuje akcije koje se izvršavaju u spoljašnjem svetu. U oba slučaja, `do` notacija je pregledniji zapis za povezivanje monadnih izračunavanja.

14.3 Liste kao monade

U prethodnim primerima videli smo da tip `Maybe` možemo posmatrati kao kontekst mogućeg neuspeha. Kod lista je značenje konteksta drugačije. Listu možemo posmatrati kao vrednost sa više mogućih rezultata.

Na primer, lista `[1,2,3]` može se shvatiti kao izračunavanje koje može dati rezultat `1`, `2` ili `3`. U tom smislu liste predstavljaju nedeterministička izračunavanja, odnosno izračunavanja sa više mogućih ishoda.

Za liste se instanca klase `Monad` može opisati ovako:

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
```

Funkcija `return` stavlja običnu vrednost u listu sa jednim elementom. To je najmanji kontekst liste koji i dalje sadrži datu vrednost.

Operator `>>=` za liste radi na sledeći način. Funkcija `f` primenjuje se na svaki element liste `xs`. Pošto `f` za svaki element vraća listu, primenom funkcije `map` najpre se dobija lista listi. Funkcija `concat` zatim spaja listu listi u jednu listu.

Na primer:

```
ghci> map (\x -> [x, -x]) [1,2,3]
[[1,-1],[2,-2],[3,-3]]

ghci> concat [[1,-1],[2,-2],[3,-3]]
[1,-1,2,-2,3,-3]
```

Dakle, izraz

```
[1,2,3] >>= \x -> [x, -x]
```

daje rezultat

```
[1,-1,2,-2,3,-3]
```

U ovom primeru funkcija `\x -> [x, -x]` svakom broju pridružuje dva moguća rezultata, sam taj broj i njegovu suprotnu vrednost. Zato se od liste `[1,2,3]` dobija lista svih rezultata nastalih primenom te funkcije na svaki element.

Ista definicija operatora `>>=` za liste može se zapisati i pomoću izlistavanja:

```
xs >>= f = [y | x <- xs, y <- f x]
```

Ovaj zapis kaže da za svaki element `x` iz liste `xs` posmatramo sve elemente `y` iz liste `f x`. Svi tako dobijeni elementi skupljaju se u jednu rezultujuću listu.

Pogledajmo sada primer sa dve liste. Funkcija koja formira sve parove elemenata iz dve liste može se napisati pomoću `do` notacije:

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do
  x <- xs
  y <- ys
  return (x,y)
```

Na primer:

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

U ovom primeru `x` redom dobija vrednosti iz liste `xs`. Za svaku izabranu vrednost `x`, promenljiva `y` redom dobija vrednosti iz liste `ys`. Poslednji red, `return (x,y)`, od svakog tako dobijenog para pravi listu sa jednim elementom. Pošto se u monadi liste svi dobijeni rezultati skupljaju u jednu listu, konačan rezultat je lista svih parova elemenata iz `xs` i `ys`.

Ista funkcija može se zapisati i pomoću izlistavanja:

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

Ova dva zapisa daju isti rezultat. Izlistavanja su specifična za liste, dok se `do` notacija može koristiti sa svakom monadom. Kod tipa `Maybe`, `do` notacija povezuje korake koji mogu da ne uspeju. Kod lista, ona povezuje korake koji mogu imati više mogućih rezultata.

Prazna lista u ovom kontekstu predstavlja izračunavanje bez rezultata. Zato se ponaša slično kao `Nothing` kod tipa `Maybe`:

```
ghci> [] >>= \x -> [x, -x]
[]

ghci> [1,2,3] >>= \x -> []
[]
```

U prvom primeru nema nijednog elementa koji bi se prosledio funkciji. U drugom primeru funkcija za svaki element vraća praznu listu, pa se na kraju ne dobija nijedan rezultat.

14.4 Zakoni monada

Kao i kod funktora, za monade nije dovoljno samo definisati funkcije odgovarajućih tipova. Da bi instanca klase `Monad` bila smislena, operator `>>=` i funkcija `return` treba da zadovoljavaju određene zakone. Ti zakoni opisuju očekivano ponašanje monadnog vezivanja.

Haskell može proveriti da li definicije imaju odgovarajuće tipove, ali ne može automatski proveriti da li zakoni važe. Zato je odgovornost programera da, kada definiše novu instancu klase `Monad`, obezbedi da se ona ponaša u skladu sa ovim pravilima. Standardne instance, kao što su instance za `Maybe`, liste i `IO`, zadovoljavaju zakone monada.

Prvi zakon naziva se levi identitet:

```
return x >>= f = f x
```

Ovaj zakon kaže da `return` ne dodaje nikakav suštinski efekat pre primene funkcije. Ako vrednost `x` stavimo u monadni kontekst pomoću `return`, a zatim je prosledimo funkciji `f`, rezultat treba da bude isti kao da smo odmah primenili `f` na `x`.

Na primer, za monadu `Maybe` važi:

```
ghci> return 3 >>= \x -> Just (x + 1)
Just 4

ghci> (\x -> Just (x + 1)) 3
```

```
Just 4
```

U prvom izrazu `return 3` pravi vrednost `Just 3`, a zatim se broj `3` prosleđuje funkciji sa desne strane. U drugom izrazu ista funkcija se neposredno primenjuje na broj `3`. Zato oba izraza daju isti rezultat.

Drugi zakon naziva se desni identitet:

```
m >>= return = m
```

Ovaj zakon kaže da povezivanje monadne vrednosti sa funkcijom `return` ne menja tu vrednost. Drugim rečima, ako iz monadnog izračunavanja dobijenu vrednost samo vratimo nazad pomoću `return`, nismo promenili značenje izračunavanja.

Na primer:

```
ghci> Just 5 >>= return
Just 5

ghci> Nothing >>= return
Nothing

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

Kod tipa `Maybe`, vrednost `Just 5` ostaje `Just 5`, a `Nothing` ostaje `Nothing`. Kod lista, funkcija `return` svaki element stavlja u listu sa jednim elementom, a operator `>>=` zatim sve te liste spaja u početnu listu.

Treći zakon je asocijativnost:

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

Ovaj zakon kaže da način grupisanja uzastopnih monadnih vezivanja ne sme promeniti rezultat. Leva strana najpre povezuje `m` sa funkcijom `f`, a zatim dobijeni rezultat povezuje sa funkcijom `g`. Desna strana isto povezivanje zapisuje tako što ceo nastavak izračunavanja stavlja u funkciju koja se primenjuje na vrednost dobijenu iz `m`. Iz te vrednosti najpre se računa `f x`, a zatim se taj rezultat povezuje sa `g`.

Posmatrajmo sledeće dve funkcije:

```
safeRecip :: Double -> Maybe Double
safeRecip 0 = Nothing
safeRecip x = Just (1 / x)

safeSqrt :: Double -> Maybe Double
safeSqrt x
  | x >= 0    = Just (sqrt x)
  | otherwise = Nothing
```

Funkcija `safeRecip` računa recipročnu vrednost broja, ali vraća `Nothing` ako je argument jednak nuli. Funkcija `safeSqrt` računa kvadratni koren, ali vraća `Nothing` ako je argument negativan.

Sada uporedimo dva načina grupisanja istog niza vezivanja:

```
ghci> (Just 4 >>= safeRecip) >>= safeSqrt
```

```
Just 0.5

ghci> Just 4 >>= (\x -> safeRecip x >>= safeSqrt)
Just 0.5
```

U oba izraza polazi se od vrednosti `Just 4`. Najpre se računa recipročna vrednost broja `4`, pa se dobija `Just 0.25`. Zatim se računa kvadratni koren broja `0.25`, pa je konačan rezultat `Just 0.5`.

Ako neki korak ne uspe, neuspeh se u oba grupisanja prenosi na isti način:

```
ghci> (Just 0 >>= safeRecip) >>= safeSqrt
Nothing

ghci> Just 0 >>= (\x -> safeRecip x >>= safeSqrt)
Nothing
```

U ovom slučaju funkcija `safeRecip` vraća `Nothing`, jer recipročna vrednost nule nije definisana. Zbog toga se ni u jednom grupisanju ne dobija vrednost nad kojom bi se zatim računao kvadratni koren.

Zakon asocijativnosti obezbeđuje da ovakva promena zagrada ne menja značenje programa. Zato monadna izračunavanja možemo pisati kao niz koraka, bez brige da će drugačije grupisanje istih vezivanja promeniti rezultat.

15 Algoritmi nad listama i nizovima

15.1 Sortiranje spajanjem

Sortiranje spajanjem je algoritam koji se zasniva na strategiji podeli pa vladaj. Ideja je da listu podelimo na dve približno jednake polovine, rekurzivno sortiramo obe polovine, a zatim dobijene sortirane liste spojimo u jednu.

Najpre ćemo definisati pomoćnu funkciju koja spaja dve već sortirane liste. Ako je jedna od listi prazna, rezultat je druga lista. Ako su obe liste neprazne, poredimo njihove prve elemente. Manji element stavljamo na početak rezultata, a zatim rekurzivno spajamo preostale elemente.

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

Ograničenje `Ord a =>` znači da elementi liste moraju pripadati tipu čije vrednosti mogu da se porede. To nam je potrebno jer u funkciji `merge` koristimo operator `<=`.

Na primer:

```
ghci> merge [1,4,7] [2,3,8]
[1,2,3,4,7,8]

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

Važno je da ulazne liste budu sortirane. Funkcija `merge` ne sortira pojedinačne liste, već samo spaja dve liste za koje već znamo da su sortirane.

Sada možemo definisati sortiranje spajanjem. Prazna lista i lista sa jednim elementom već su sortirane. U ostalim slučajevima listu delimo na dve polovine, sortiramo ih rekurzivno i zatim spajamo.

```
mergeSort :: Ord a => [a] -> [a]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  =
  merge (mergeSort left) (mergeSort right)
  where
    n = length xs `div` 2
    (left, right) = splitAt n xs
```

Funkcija `splitAt n xs` deli listu `xs` na dve liste. Prva sadrži prvih `n` elemenata, a druga preostale elemente. U funkciji `mergeSort` te dve liste su približno iste dužine.

Na primer:

```
ghci> splitAt 3 [7,2,5,1,4,3]
([7,2,5],[1,4,3])
```

Zato se pri pozivu

```
mergeSort [7,2,5,1,4,3]
```

lista najpre deli na `[7,2,5]` i `[1,4,3]`. Obe polovine se zatim sortiraju istim postupkom, a na kraju se dobijene sortirane liste spajaju.

Na primer:

```
ghci> mergeSort [7,2,5,1,4,3]
[1,2,3,4,5,7]

ghci> mergeSort ["Haskell", "C", "Python", "Ada"]
["Ada","C","Haskell","Python"]

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

U drugom primeru sortiramo liste niski. To je moguće zato što i niske pripadaju klasi `Ord`, pa se mogu porediti leksikografski.

Dakle, lista se deli sve dok ne dobijemo liste dužine nula ili jedan. Takve liste su već sortirane. Zatim se one redom spajaju u sve veće sortirane liste, dok se na kraju ne dobije sortirana početna lista.

Prednost sortiranja spajanjem je u tome što u svakom nivou rekurzije ukupno obrađujemo sve elemente liste, a broj nivoa rekurzije je reda $\log n$. Zato je vremenska složenost algoritma $O(n \log n)$, gde je n dužina liste.

15.2 Brojanje inverzija

Neka je data lista elemenata $a_1, a_2, \ldots, a_n$, koji mogu da se porede. Inverzija predstavlja par elemenata (a_i, a_j) za koje je $i < j$ i $a_i > a_j$.

Na primer, u listi

```
[3,1,2]
```

postoje dve inverzije. To su parovi $(3, 1)$ i $(3, 2)$, jer se broj `3` nalazi pre oba manja broja. Par $(1, 2)$ nije inverzija, jer su brojevi u rastućem redosledu.

Broj inverzija može se shvatiti kao mera nesortiranosti liste. Sortirana lista nema nijednu inverziju, dok lista sortirana u opadajućem poretku ima najveći mogući broj inverzija.

Najjednostavnije rešenje bilo bi da proverimo sve parove elemenata. Međutim, takav algoritam ima složenost $O(n^2)$. Efikasnije rešenje može se dobiti malom izmenom sortiranja spajanjem.

Ideja je sledeća. Kao i kod sortiranja spajanjem, listu delimo na dve polovine. Inverzije zatim delimo na tri vrste. Prve su inverzije koje se nalaze potpuno u levoj polovini. Druge su inverzije koje se nalaze potpuno u desnoj polovini. Treće su inverzije u kojima je prvi element iz leve polovine, a drugi iz desne polovine. Prve dve vrste brojimo rekurzivno, a treću vrstu brojimo tokom spajanja sortiranih polovina.

Zato ćemo definisati funkciju koja istovremeno spaja dve sortirane liste i broji inverzije između njih. Pretpostavimo da su obe liste već sortirane. Ako je prvi element leve liste manji ili jednak prvom elementu desne liste, onda taj element ne pravi inverziju sa elementima desne liste, pa ga samo prebacujemo u rezultat. Ako je prvi element desne liste manji od prvog elementa leve liste, onda on pravi inverziju sa svim preostalim elementima leve liste.

```
mergeAndCount :: Ord a => [a] -> [a] -> ([a], Int)
mergeAndCount xs ys = merge xs ys (length xs)
  where
    merge [] ys _ = (ys, 0)
    merge xs [] _ = (xs, 0)
    merge (x:xs) (y:ys) nLeft
      | x <= y =
          let
            (zs, c) = merge xs (y:ys) (nLeft - 1)
          in
            (x:zs, c)
      | otherwise =
          let
            (zs, c) = merge (x:xs) ys nLeft
          in
            (y:zs, nLeft + c)
```

Funkcija `mergeAndCount` vraća par. Prva komponenta je spojena sortirana lista, a druga komponenta je broj inverzija u kojima je jedan element iz prve, a drugi iz druge liste.

U pomoćnoj funkciji `merge` treći argument, `nLeft`, predstavlja broj preostalih elemenata u levoj listi. Kada uzmemo element iz leve liste, broj preostalih elemenata leve liste smanjuje se za jedan. Kada uzmemo element iz desne liste zato što je manji od prvog elementa leve liste, on pravi inverziju sa svih `nLeft` preostalih elemenata leve liste.

Na primer:

```
ghci> mergeAndCount [3,5,7] [2,6]
([2,3,5,6,7],4)
```

U ovom primeru postoje četiri inverzije između dve liste. Broj 2 pravi inverzije sa 3, 5 i 7, a broj 6 pravi inverziju sa 7.

Sada možemo definisati funkciju koja sortira celu listu i istovremeno broji sve inverzije:

```
sortAndCount :: Ord a => [a] -> ([a], Int)
sortAndCount []  = ([], 0)
sortAndCount [x] = ([x], 0)
sortAndCount xs  =
  let
    n = length xs `div` 2
    (left, right) = splitAt n xs

    (sortedLeft, countLeft) = sortAndCount left
    (sortedRight, countRight) = sortAndCount right

    (sorted, countSplit) =
      mergeAndCount sortedLeft sortedRight
  in
    (sorted, countLeft + countRight + countSplit)
```

Funkcija `sortAndCount` vraća sortiranu listu i broj inverzija u početnoj listi. Inverzije u levoj polovini broje se vrednošću `countLeft`, inverzije u desnoj polovini vrednošću `countRight`, a

inverzije između leve i desne polovine vrednošću `countSplit`.

Ako nas zanima samo broj inverzija, možemo definisati posebnu funkciju koja uzima drugu komponentu rezultata:

```
inversionCount :: Ord a => [a] -> Int
inversionCount xs = snd (sortAndCount xs)
```

Na primer:

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

ghci> inversionCount [1,2,3,4]
0

ghci> inversionCount [4,3,2,1]
6
```

U prvoj listi inverzije su $(3, 1)$ i $(3, 2)$. Druga lista je već sortirana, pa nema inverzija. U trećoj listi svaki par elemenata je u pogrešnom redosledu, pa za četiri elementa dobijamo

$$\frac{4 \cdot 3}{2} = 6$$

inverzija.

Funkcija radi i kada postoje jednaki elementi. Pošto inverziju čini samo par za koji je levi element strogo veći od desnog, jednaki elementi se ne broje kao inverzije:

```
ghci> inversionCount [2,2,1]
2

ghci> inversionCount [1,1,1]
0
```

U listi `[2,2,1]` oba pojavljivanja broja `2` prave inverziju sa brojem `1`, pa je rezultat `2`. U listi `[1,1,1]` nema strogo većeg elementa koji se nalazi pre manjeg, pa je broj inverzija jednak nuli.

Složenost ovog algoritma je $O(n \log n)$. Kao i kod sortiranja spajanjem, lista se deli na polovine, a na svakom nivou rekurzije svi elementi se obrađuju tokom spajanja. Dodatno brojanje inverzija ne menja red složenosti, jer se obavlja u istom prolasku u kome se liste spajaju.

15.3 Binarna pretraga

Binarna pretraga je algoritam za traženje elementa u sortiranoj kolekciji. Umesto da elemente proveravamo redom, u svakom koraku posmatramo srednji element. Ako je tražena vrednost jednaka srednjem elementu, pretraga se završava. Ako je tražena vrednost manja, nastavljamo pretragu u levoj polovini. Ako je veća, nastavljamo pretragu u desnoj polovini.

Za ovakav algoritam potreban nam je efikasan pristup elementu po indeksu. Liste u Haskellu nisu pogodne za to, jer se do elementa na poziciji i dolazi prolaskom kroz prethodne elemente. Zato pristup elementu liste po indeksu nije konstantan. Binarna pretraga se prirodnije implementira nad nizom, gde se elementu sa zadatim indeksom pristupa direktno.

U Haskellu možemo koristiti nizove iz modula `Data.Array`. Zato ga na početku programa treba uvesti na sledeći način:

```
import Data.Array (Array, listArray, bounds, (!))
```

Tip `Array Int a` predstavlja niz čiji su indeksi tipa `Int`, a elementi tipa `a`. Funkcija `listArray` pravi niz od liste elemenata, pri čemu zadajemo opseg indeksa. Funkcija `bounds` vraća donju i gornju granicu indeksa, a operator `!` služi za pristup elementu niza po indeksu.

Na primer:

```
ghci> let arr = listArray (0,5) [1,3,4,7,9,12] :: Array Int Int

ghci> bounds arr
(0,5)

ghci> arr ! 3
7
```

U ovom primeru niz ima indekse od `0` do `5`. Element sa indeksom `3` jednak je `7`.

Sada možemo definisati binarnu pretragu. Funkcija će primati sortiran niz i vrednost koju tražimo. Ako se tražena vrednost nalazi u nizu, funkcija će vratiti njen indeks u konstruktoru `Just`. Ako se vrednost ne nalazi u nizu, rezultat će biti `Nothing`.

```
binarySearch :: Ord a => Array Int a -> a -> Maybe Int
binarySearch arr x = search left right
  where
    (left, right) = bounds arr

    search l r
      | l > r           = Nothing
      | arr ! mid == x  = Just mid
      | arr ! mid < x   = search (mid + 1) r
      | otherwise       = search l (mid - 1)
      where
        mid = (l + r) `div` 2
```

Pomoćna funkcija `search` pretražuje deo niza između indeksa `l` i `r`. Ako je `l > r`, posmatrani interval je prazan. To znači da traženi element nije pronađen, pa funkcija vraća `Nothing`.

U suprotnom računamo srednji indeks `mid` i poredimo element `arr ! mid` sa traženom vrednošću `x`. Ako je `arr ! mid == x`, element je pronađen i vraćamo `Just mid`. Ako je `arr ! mid < x`, pošto je niz sortiran, traženi element može biti samo desno od indeksa `mid`. Zato nastavljamo pretragu u intervalu od `mid + 1` do `r`. Ako je `arr ! mid > x`, nastavljamo pretragu u levoj polovini.

Na primer:

```
ghci> let arr = listArray (0,5) [1,3,4,7,9,12] :: Array Int Int

ghci> binarySearch arr 7
Just 3

ghci> binarySearch arr 5
Nothing
```

```
ghci> binarySearch arr 1
Just 0

ghci> binarySearch arr 12
Just 5
```

U prvom primeru broj 7 nalazi se na indeksu 3, pa je rezultat `Just 3`. U drugom primeru broj 5 ne postoji u nizu, pa je rezultat `Nothing`. Poslednja dva primera pokazuju da algoritam ispravno pronalazi i krajnje elemente niza.

Možemo definisati i pomoćnu funkciju koja kreira niz od liste:

```
arrayFromList :: [a] -> Array Int a
arrayFromList xs = listArray (0, length xs - 1) xs
```

Tada prethodne primere možemo pisati kraće:

```
ghci> let arr = arrayFromList [1,3,4,7,9,12]

ghci> binarySearch arr 9
Just 4

ghci> binarySearch arr 2
Nothing
```

Funkcija `binarySearch` vraća indeks pronađenog elementa, ali se iz nje lako može dobiti i funkcija koja proverava samo da li element postoji u nizu:

```
contains :: Ord a => Array Int a -> a -> Bool
contains arr x =
  case binarySearch arr x of
    Just _  -> True
    Nothing -> False
```

Ako je rezultat binarne pretrage oblika `Just _`, element postoji u nizu. Donja crta znači da nas konkretan indeks u tom slučaju ne zanima. Ako je rezultat `Nothing`, element nije pronađen.

Na primer:

```
ghci> contains arr 9
True

ghci> contains arr 2
False
```

Važno je naglasiti da binarna pretraga pretpostavlja da je niz sortiran. Ako niz nije sortiran, poređenje sa srednjim elementom ne govori pouzdano u kojoj polovini treba nastaviti pretragu, pa algoritam ne mora dati ispravan rezultat.

U svakom koraku binarne pretrage odbacuje se približno polovina preostalog intervala. Zato je broj koraka reda $\log n$, gde je n broj elemenata u nizu. Pošto je pristup elementu niza po indeksu konstantan, ukupna vremenska složenost binarne pretrage nad nizom je $O(\log n)$.

Kada bismo isti postupak pokušali da sprovedemo nad listom koristeći pristup elementu po

indeksu, izgubili bismo ovu složenost, jer pristup srednjem elementu liste nije konstantan. Zbog toga su nizovi prirodniji izbor za binarnu pretragu.

15.4 Prefiksne sume

Prefiksne sume su jednostavna i veoma korisna tehnika za brzo računanje zbira uzastopnih delova liste ili niza. Ideja je da unapred izračunamo zbirove svih početnih delova date kolekcije. Nakon toga se zbir bilo kog segmenta može dobiti oduzimanjem dve prethodno izračunate vrednosti.

Neka je data lista brojeva $a_0, a_1, \ldots, a_{n-1}$. Definišemo niz prefiksnih suma p tako da je $p_0 = 0$ i $p_{i+1} = a_0 + a_1 + \cdots + a_i$. Na primer, za listu `[3,1,4,2]` prefiksne sume su `[0,3,4,8,10]`. Prva vrednost po definiciji iznosi `0`, a zatim redom dobijamo `3`, `3 + 1 = 4`, `3 + 1 + 4 = 8` i `3 + 1 + 4 + 2 = 10`.

U Haskellu se ovakva lista može dobiti pomoću funkcije `scanl`:

```
prefixSumsList :: Num a => [a] -> [a]
prefixSumsList xs = scanl (+) 0 xs
```

Funkcija `scanl` je slična funkciji `foldl`. Razlika je u tome što `foldl` vraća samo konačnu vrednost akumulatora, dok `scanl` vraća listu svih vrednosti koje akumulator dobija tokom prolaska kroz listu. U izrazu `scanl (+) 0 xs`, početna vrednost akumulatora je `0`, a zatim se elementi liste `xs` redom dodaju na akumulator. Zbog toga je:

```
ghci> prefixSumsList [3,1,4,2]
[0,3,4,8,10]

ghci> prefixSumsList [5,10,20]
[0,5,15,35]
```

Prefiksne sume su posebno korisne kada treba više puta računati zbir elemenata na različitim intervalima. Ako želimo zbir elemenata od indeksa l do indeksa r, uključujući oba kraja, onda važi da je

$$a_l + a_{l+1} + \cdots + a_r = p_{r+1} - p_l.$$

Razlog je jednostavan. Vrednost p_{r+1} sadrži zbir svih elemenata od početka do indeksa r, dok p_l sadrži zbir elemenata pre indeksa l. Kada oduzmemo te dve vrednosti, ostaje upravo zbir segmenta od l do r.

Na primer, za listu `[3,1,4,2]` prefiksne sume su `[0,3,4,8,10]`. Zbir elemenata od indeksa `1` do indeksa `3` jednak je `1 + 4 + 2 = 7`. Pomoću prefiksnih suma dobijamo isti rezultat:

$$p_4 - p_1 = 10 - 3 = 7.$$

Pošto želimo brz pristup prefiksnim sumama po indeksu, prirodno je da ih čuvamo u nizu. Koristićemo ranije uvedenu funkciju `arrayFromList`:

```
prefixSums :: Num a => [a] -> Array Int a
prefixSums xs = arrayFromList (scanl (+) 0 xs)
```

Funkcija `prefixSums` od liste brojeva pravi niz prefiksnih suma. Ako početna lista ima n elemenata, niz prefiksnih suma ima $n + 1$ elemenata, jer sadrži i početnu vrednost `0`.

Na primer:

```
ghci> let pref = prefixSums [3,1,4,2]

ghci> pref ! 0
0

ghci> pref ! 1
3

ghci> pref ! 4
10
```

Sada možemo definisati funkciju koja računa zbir segmenta. Pretpostavljamo da su indeksi ispravni, odnosno da važi $0 \leq l \leq r < n$, gde je n dužina početne liste.

```
rangeSum :: Num a => Array Int a -> Int -> Int -> a
rangeSum pref l r = pref ! (r + 1) - pref ! l
```

Funkcija `rangeSum pref l r` računa zbir elemenata od indeksa `l` do indeksa `r`. Niz `pref` je niz prefiksnih suma, pa se rezultat dobija kao razlika dve vrednosti.

Na primer:

```
ghci> let pref = prefixSums [3,1,4,2]

ghci> rangeSum pref 1 3
7

ghci> rangeSum pref 0 2
8

ghci> rangeSum pref 2 2
4
```

U prvom primeru računamo zbir `1 + 4 + 2`. U drugom primeru računamo zbir `3 + 1 + 4`. U trećem primeru segment se sastoji od samo jednog elementa, pa je rezultat `4`.

Priprema niza prefiksnih suma zahteva jedan prolazak kroz početnu listu, pa je njena složenost $O(n)$. Nakon toga se svaki upit za zbir segmenta računa pomoću dva pristupa nizu i jednog oduzimanja, pa je složenost pojedinačnog upita $O(1)$.

Prema tome, prefiksne sume su korisne kada nad istom kolekcijom treba izračunati veliki broj zbirova uzastopnih segmenata. Umesto da za svaki upit ponovo prolazimo kroz odgovarajući deo liste, jednom izračunamo pomoćni niz i zatim svaki upit rešavamo neposredno.

15.5 Maksimalni zbir segmenta

Neka je data lista brojeva. Segment liste je njen uzastopni deo. Problem maksimalnog zbira segmenta sastoji se u tome da pronađemo najveći mogući zbir nekog nepraznog segmenta. Na primer, u listi `[4,-6,3,2,-1,4,-5]` najveći zbir ima segment `[3,2,-1,4]`, jer je njegov zbir jednak `8`. Nijedan drugi neprazan segment ove liste nema veći zbir.

Ovaj problem možemo povezati sa prefiksnim sumama. Neka su prefiksne sume date vrednostima $p_0, p_1, \ldots, p_n$, gde je $p_0 = 0$, a p_i predstavlja zbir prvih i elemenata liste. Zbir segmenta

od indeksa l do indeksa r iznosi $p_{r+1} - p_l$. Dakle, tražimo najveću razliku dve prefiksne sume $p_j - p_i$, pri čemu mora da važi $i < j$. Uslov $i < j$ znači da segment mora biti neprazan.

Ideja algoritma je sledeća. Dok prolazimo kroz prefiksne sume sleva nadesno, za svaku trenutnu prefiksnu sumu p_j želimo da od nje oduzmemo najmanju prefiksnu sumu koja se pojavila pre nje. Tako dobijamo najbolji segment koji se završava neposredno pre pozicije j. Ako tokom prolaska čuvamo najmanju prefiksnu sumu viđenu do sada, svaki korak možemo obraditi u konstantnom vremenu.

Na primer, za listu `[4,-6,3,2,-1,4,-5]` prefiksne sume su `[0,4,-2,1,3,2,6,1]`. Kada dođemo do prefiksne sume `6`, najmanja prethodna prefiksna suma je `-2`. Razlika je `6 - (-2) = 8`, što odgovara segmentu `[3,2,-1,4]`.

Pošto segment treba da bude neprazan, za praznu listu on ne postoji. Zato ćemo rezultat predstaviti tipom `Maybe`. Ako je lista prazna, rezultat je `Nothing`. Ako nije prazna, rezultat je oblika `Just s`, gde je `s` maksimalni zbir segmenta.

Koristićemo funkciju `prefixSumsList` iz prethodne podsekcije:

```
maxSegmentSum :: (Num a, Ord a) => [a] -> Maybe a
maxSegmentSum xs =
  case prefixSumsList xs of
    p0:p1:ps -> Just (bestSum (min p0 p1) (p1 - p0) ps)
    _        -> Nothing
  where
    bestSum _ best [] = best
    bestSum minPrefix best (p:ps) =
      bestSum (min minPrefix p) (max best (p - minPrefix)) ps
```

Ograničenje `Num a =>` potrebno je zato što sabiramo i oduzimamo vrednosti, a ograničenje `Ord a =>` zato što poredimo zbirove pomoću funkcija `min` i `max`.

U izrazu `p0:p1:ps` izdvajamo prve dve prefiksne sume. To je moguće samo ako početna lista ima bar jedan element. Tada je `p1 - p0` zbir segmenta koji se sastoji od prvog elementa, pa ga uzimamo kao početnu najbolju vrednost.

Prvi argument `minPrefix` pomoćne funkcije `bestSum` predstavlja najmanju prefiksnu sumu viđenu do sada. Drugi, `best`, predstavlja najbolji zbir segmenta pronađen do tog trenutka.

Kada obradimo novu prefiksnu sumu `p`, vrednost `p - minPrefix` predstavlja najbolji zbir segmenta koji se završava u trenutnoj poziciji. Zatim ažuriramo najbolji pronađeni zbir pomoću `max`, a najmanju prefiksnu sumu pomoću `min`.

Na primer:

```
ghci> maxSegmentSum [4,-6,3,2,-1,4,-5]
Just 8

ghci> maxSegmentSum [1,2,3]
Just 6

ghci> maxSegmentSum [-5,-2,-7]
Just (-2)

ghci> maxSegmentSum []
Nothing
```

U prvom primeru najbolji segment je `[3,2,-1,4]`, čiji je zbir `8`. U drugom primeru najbolji

segment je cela lista. U trećem primeru svi brojevi su negativni, pa je najbolji segment onaj koji sadrži samo najveći element, odnosno `-2`. Za praznu listu ne postoji neprazan segment, pa je rezultat `Nothing`.

Složenost algoritma je $O(n)$, gde je n dužina liste. Prefiksne sume se računaju jednim prolaskom kroz listu, a zatim se i kroz njih prolazi samo jednom. U svakom koraku čuvamo samo najmanju prethodnu prefiksnu sumu i najbolji zbir do sada.

Prema tome, maksimalni zbir segmenta možemo izračunati linearno. Veza sa prefiksnim sumama pokazuje zašto je dovoljno u svakom trenutku znati samo najmanju prefiksnu sumu viđenu ranije. Najbolji segment koji se završava na trenutnoj poziciji dobija se oduzimanjem upravo te vrednosti od trenutne prefiksne sume.

15.6 Najmanji broj apoena

Neka su dati apoeni, odnosno dozvoljene novčane vrednosti, i neka je data vrednost koju želimo da dobijemo. Cilj je da tu vrednost predstavimo pomoću najmanjeg mogućeg broja apoena. Svaki apoen možemo koristiti proizvoljan broj puta.

Na primer, ako su apoeni `[1,3,4]`, vrednost `6` možemo dobiti kao `3 + 3`, pa su dovoljna dva apoena. Pohlepni postupak, u kome bismo uvek birali najveći mogući apoen, ovde ne daje najbolje rešenje, jer bismo tako izabrali `4 + 1 + 1`, što koristi tri apoena.

Ovaj problem je pogodan za dinamičko programiranje, jer se najbolje rešenje za neku vrednost može dobiti pomoću najboljih rešenja za manje vrednosti. Neka je d_s najmanji broj apoena potreban da se dobije vrednost s. Ako je poslednji upotrebljeni apoen c, onda pre toga treba dobiti vrednost $s - c$. Zato važi:

$$d_s = 1 + \min\{d_{s-c} \mid c \text{ je apoen i } c \leq s\}.$$

Naravno, u minimum ulaze samo one vrednosti $s-c$ koje je moguće dobiti. Za početnu vrednost važi $d_0 = 0$, jer za vrednost nula nije potreban nijedan apoen.

U Haskellu ćemo koristiti tip `Maybe Int`. Ako neku vrednost nije moguće dobiti zadatim apoenima, rezultat će biti `Nothing`. Ako jeste moguće, rezultat će biti oblika `Just k`, gde je `k` najmanji broj potrebnih apoena.

Najpre definišemo pomoćnu funkciju koja povećava vrednost unutar `Maybe` za jedan:

```
addOne :: Maybe Int -> Maybe Int
addOne Nothing  = Nothing
addOne (Just x) = Just (x + 1)
```

Ako neku manju vrednost nije moguće dobiti, onda se ni dodavanjem još jednog apoena iz nje ne dobija ispravno rešenje. Zato `Nothing` ostaje `Nothing`. Ako je manja vrednost dobijena pomoću `x` apoena, onda se dodavanjem jednog apoena dobija rešenje sa `x + 1` apoena.

Zatim definišemo funkciju koja iz liste vrednosti tipa `Maybe Int` bira najmanju postojeću vrednost:

```
minimumMaybe :: [Maybe Int] -> Maybe Int
minimumMaybe xs =
  case values of
    [] -> Nothing
    _  -> Just (minimum values)
  where
    values = [x | Just x <- xs]
```

Lista `values` sadrži sve vrednosti `x` za koje se u početnoj listi pojavio element oblika `Just x`. Vrednosti `Nothing` se izostavljaju. Ako posle toga nema nijedne vrednosti, rezultat je `Nothing`. U suprotnom uzimamo najmanju od njih.

Sada možemo definisati glavni algoritam. Pretpostavićemo da su svi apoeni pozitivni celi brojevi i da je tražena vrednost nenegativna.

```
minCoins :: [Int] -> Int -> Maybe Int
minCoins coins target = table ! target
  where
    table = arrayFromList [value s | s <- [0 .. target]]

    value 0 = Just 0
    value s =
      minimumMaybe [addOne (table ! (s - c)) | c <- coins, c <= s]
```

Funkcija `minCoins` prima listu apoena i traženu vrednost. Niz `table` čuva rešenja za sve vrednosti od `0` do `target`. Na indeksu `s` nalazi se najmanji broj apoena potreban da se dobije vrednost `s`, ili `Nothing` ako to nije moguće.

Za vrednost `0` rezultat je `Just 0`. Za pozitivnu vrednost `s` razmatramo sve apoene `c` koji nisu veći od `s`. Ako kao poslednji apoen uzmemo `c`, onda prethodno treba dobiti vrednost `s - c`. Zato gledamo vrednost `table ! (s - c)` i na nju primenjujemo funkciju `addOne`. Od svih mogućnosti biramo najmanju pomoću funkcije `minimumMaybe`.

U ovoj definiciji niz `table` zavisi od svojih ranijih vrednosti. Iako u kodu nema eksplicitnog rekurzivnog poziva funkcije `minCoins`, rekurzija se pojavljuje kroz pristupe vrednostima oblika `table ! (s - c)`. To je moguće zbog lenjog izračunavanja u Haskellu. Vrednosti u nizu ne moraju biti izračunate odmah, već se računaju onda kada su potrebne. Pri računanju vrednosti za `s`, koriste se samo vrednosti `s - c`, gde je `c` pozitivan apoen, pa je `s - c < s`. Zato se zavisnosti uvek kreću ka manjim vrednostima i završavaju se u osnovnom slučaju `value 0 = Just 0`.

Na primer:

```
ghci> minCoins [1,3,4] 6
Just 2

ghci> minCoins [1,3,4] 10
Just 3

ghci> minCoins [2,5] 3
Nothing

ghci> minCoins [2,5] 0
Just 0
```

U prvom primeru vrednost `6` dobijamo kao `3 + 3`, pa su potrebna dva apoena. U drugom primeru vrednost `10` možemo dobiti kao `4 + 3 + 3`, pa su dovoljna tri apoena. U trećem primeru vrednost `3` nije moguće dobiti apoenima `2` i `5`. Za vrednost `0` nije potreban nijedan apoen.

Ako među apoenima postoji apoen `1`, tada se svaka nenegativna vrednost može dobiti. Ako takvog apoena nema, neke vrednosti mogu biti nedostižne, pa je zato tip `Maybe Int` prirodan izbor.

Složenost algoritma je $O(nS)$, gde je n broj apoena, a S tražena vrednost. Za svaku vrednost od `0` do `S` razmatramo sve apoene. Memorijska složenost je $O(S)$, jer čuvamo po jedno rešenje za svaku vrednost od `0` do `S`.

15.7 Najduži rastući podniz

Neka je data lista elemenata koji mogu da se porede. Podniz dobijamo tako što iz liste izbacimo neke elemente, pri čemu redosled preostalih elemenata ostaje isti. Za razliku od segmenta, podniz ne mora biti uzastopan deo liste.

Problem najdužeg rastućeg podniza sastoji se u tome da pronađemo podniz u kome su elementi strogo rastući i čija je dužina najveća moguća. Na primer, u listi `[3,1,5,2,6,4,9]` jedan najduži rastući podniz je `[1,2,4,9]`. Postoje i drugi rastući podnizovi iste dužine, na primer `[1,2,6,9]`.

Ovaj problem je pogodan za dinamičko programiranje, jer se rešenje za svaku poziciju može izračunati pomoću rešenja za prethodne pozicije. Ako elemente liste označimo sa $a_0, a_1, \ldots, a_{n-1}$, za svaku poziciju i računaćemo vrednost d_i, gde je d_i dužina najdužeg rastućeg podniza koji se završava baš na poziciji i.

Da bi se rastući podniz završio elementom a_i, prethodni element u tom podnizu može biti samo neki a_j za koji važi $j < i$ i $a_j < a_i$. Zato, ako takve pozicije postoje, važi:

$$d_i = 1 + \max\{d_j \mid j < i,\ a_j < a_i\}.$$

Ako takva pozicija ne postoji, onda je $d_i = 1$, jer sam element a_i čini rastući podniz dužine jedan.

Pored dužine najboljeg podniza, pamtićemo i indeks prethodnog elementa u tom podnizu. Ako prethodnik ne postoji, pamtićemo `Nothing`, a ako postoji, pamtićemo njegov indeks u obliku `Just j`. Tako ćemo kasnije moći da rekonstruišemo i sam podniz, a ne samo njegovu dužinu.

Pošto nam je potreban pristup elementima po indeksu, listu ćemo pretvoriti u niz. Koristićemo ranije uvedenu funkciju `arrayFromList`. Biće nam potrebne i funkcije `elems` i `indices` iz modula `Data.Array`, pa uvoz možemo proširiti ovako:

```
import Data.Array (Array, listArray, bounds, elems, indices, (!))
```

Funkcija `elems` vraća listu svih elemenata niza, dok funkcija `indices` vraća listu svih njegovih indeksa.

Najpre definišemo pomoćnu funkciju koja za zadatu poziciju pronalazi najbolji prethodni indeks:

```
bestPrevious ::
  Ord a =>
  Array Int a -> Array Int (Int, Maybe Int) -> Int -> Maybe Int
bestPrevious arr info i =
  case candidates of
    [] -> Nothing
    _  -> Just (best candidates)
  where
    candidates = [j | j <- [0 .. i - 1], arr ! j < arr ! i]

    best [j] = j
    best (j:js)
```

```
      | len j >= len k = j
      | otherwise      = k
      where
        k = best js

    len j = fst (info ! j)
```

Funkcija `bestPrevious` ima tri argumenta. Prvi argument, `arr`, jeste niz početnih elemenata. Drugi argument, `info`, jeste niz već izračunatih podataka za pojedinačne pozicije. Na poziciji `j` u tom nizu nalazi se par `(duzina, prethodnik)`. Treći argument, `i`, jeste pozicija za koju trenutno tražimo najboljeg prethodnika.

Lista `candidates` sadrži sve indekse `j` koji dolaze pre pozicije `i` i za koje važi da je `arr ! j < arr ! i`. To su upravo pozicije sa kojih možemo produžiti rastući podniz elementom `arr ! i`.

Ako je lista `candidates` prazna, ne postoji prethodni element koji može stajati pre `arr ! i` u rastućem podnizu, pa funkcija vraća `Nothing`. U suprotnom, među kandidatima biramo onaj indeks na kome se završava najduži rastući podniz. Zato se vraća `Just (best candidates)`.

Pomoćna funkcija `best` bira najbolji indeks iz neprazne liste kandidata. Ako lista ima samo jedan element, taj element je najbolji. Ako je lista oblika `j:js`, najpre se rekurzivno pronađe najbolji indeks `k` u ostatku liste `js`, a zatim se porede dužine podnizova koji se završavaju na indeksima `j` i `k`. Funkcija `len` iz niza `info` uzima prvu komponentu odgovarajućeg para, odnosno dužinu najboljeg podniza koji se završava na datoj poziciji.

Sada možemo izračunati sve potrebne podatke:

```
lisData :: Ord a => [a] -> Array Int (Int, Maybe Int)
lisData xs = info
  where
    arr = arrayFromList xs
    (lo, hi) = bounds arr

    info = listArray (lo, hi) [value i | i <- [lo .. hi]]

    value i =
      case bestPrevious arr info i of
        Nothing -> (1, Nothing)
        Just j  -> (fst (info ! j) + 1, Just j)
```

Funkcija `lisData` prima početnu listu i vraća niz podataka. Na svakoj poziciji tog niza nalazi se par. Prva komponenta para je dužina najdužeg rastućeg podniza koji se završava na toj poziciji, a druga komponenta je prethodni indeks u tom podnizu.

Najpre se početna lista pretvara u niz `arr`. Zatim pomoću `bounds arr` dobijamo najmanji i najveći indeks tog niza. Pošto funkcija `arrayFromList` pravi niz indeksiran od nule, za listu dužine n granice će biti `(0, n - 1)`.

Niz `info` pravimo pomoću funkcije `listArray`. Za svaki indeks `i` računamo vrednost `value i`. Ako funkcija `bestPrevious arr info i` vrati `Nothing`, onda se na poziciji `i` ne može produžiti nijedan raniji rastući podniz. Zato je odgovarajući par `(1, Nothing)`. Ako vrati `Just j`, onda se najbolji podniz koji se završava na poziciji `j` produžava elementom na poziciji `i`. Nova dužina je zato `fst (info ! j) + 1`, a prethodni indeks je `Just j`.

Kao i u prethodnoj podsekciji, i ovde definišemo niz pomoću njegovih ranijih vrednosti. U definiciji `info` koristimo samu vrednost `info`, što je moguće zbog lenjog izračunavanja u

Haskellu. Pri računanju podatka za poziciju `i` koriste se samo ranije pozicije `j < i`, pa se zavisnosti kreću sleva nadesno i dobro su definisane.

Na primer:

```
ghci> lisData [3,1,5,2,6,4,9]
array (0,6)
  [(0,(1,Nothing)),(1,(1,Nothing)),(2,(2,Just 0)),
   (3,(2,Just 1)),(4,(3,Just 2)),(5,(3,Just 3)),
   (6,(4,Just 4))]
```

Na indeksu `6` završava se rastući podniz dužine `4`. Njegov prethodni element je na indeksu `4`, zatim prethodni na indeksu `2`, a zatim na indeksu `0`. Tako se dobija podniz `[3,5,6,9]`. To je jedan od najdužih rastućih podnizova date liste.

Dužinu najdužeg rastućeg podniza dobijamo kao najveću od svih izračunatih dužina:

```
lisLength :: Ord a => [a] -> Int
lisLength [] = 0
lisLength xs = maximum [len | (len, _) <- elems (lisData xs)]
```

Za praznu listu rezultat je `0`. Za nepraznu listu računamo niz `lisData xs`, iz svakog para uzimamo prvu komponentu, odnosno dužinu, i zatim funkcijom `maximum` uzimamo najveću vrednost.

Na primer:

```
ghci> lisLength [3,1,5,2,6,4,9]
4

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

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

ghci> lisLength []
0
```

Ako želimo da rekonstruišemo jedan najduži rastući podniz, najpre pronalazimo indeks na kome se završava podniz najveće dužine:

```
bestIndex :: Array Int (Int, Maybe Int) -> Int
bestIndex info = best (indices info)
  where
    best [i] = i
    best (i:is)
      | len i >= len k = i
      | otherwise      = k
      where
        k = best is

    len i = fst (info ! i)
```

Funkcija `indices info` vraća listu svih indeksa niza `info`. Pomoćna funkcija `best` zatim

bira onaj indeks na kome je dužina najveća. Kao i ranije, `len i` označava prvu komponentu para `info ! i`, odnosno dužinu najboljeg rastućeg podniza koji se završava na poziciji `i`.

Kada znamo završni indeks, podniz rekonstruišemo praćenjem prethodnika:

```
reconstruct :: Array Int a -> Array Int (Int, Maybe Int) -> Int -> [a]
reconstruct arr info i =
  case snd (info ! i) of
    Nothing -> [arr ! i]
    Just j  -> reconstruct arr info j ++ [arr ! i]
```

Funkcija `reconstruct` prima niz početnih elemenata, niz izračunatih podataka i indeks na kome se završava podniz koji želimo da rekonstruišemo. Ako je druga komponenta para `info ! i` jednaka `Nothing`, onda trenutni element nema prethodnika i podniz se sastoji samo od `arr ! i`. Ako je ta komponenta oblika `Just j`, najpre rekonstruišemo podniz koji se završava na indeksu `j`, a zatim na kraj dodajemo trenutni element `arr ! i`.

Konačno, funkcija koja vraća jedan najduži rastući podniz glasi:

```
lis :: Ord a => [a] -> [a]
lis [] = []
lis xs = reconstruct arr info start
  where
    arr = arrayFromList xs
    info = lisData xs
    start = bestIndex info
```

Za praznu listu rezultat je prazna lista. Za nepraznu listu najpre pravimo niz početnih elemenata `arr` i niz podataka `info`. Zatim pomoću `bestIndex info` pronalazimo indeks na kome se završava najduži rastući podniz i odatle pokrećemo rekonstrukciju.

Na primer:

```
ghci> lis [3,1,5,2,6,4,9]
[3,5,6,9]

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

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

ghci> lis []
[]
```

U prvom primeru postoji više najdužih rastućih podnizova dužine `4`, a funkcija vraća jedan od njih. U opadajućoj listi svaki pojedinačni element čini rastući podniz dužine `1`, pa funkcija vraća jedan takav podniz.

Složenost ovog algoritma je $O(n^2)$. Za svaku poziciju i posmatramo sve prethodne pozicije $j < i$, pa ukupan broj poređenja raste kvadratno sa dužinom liste. Memorijska složenost je $O(n)$, jer čuvamo po jedan podatak za svaku poziciju. Za problem najdužeg rastućeg podniza, postoji i efikasniji algoritam složenosti $O(n \log n)$, ali je njegova implementacija nešto složenija i nećemo je prikazivati.

16 Grafovski algoritmi

16.1 Pretraga u dubinu

Pretraga u dubinu je jedan od osnovnih algoritama za obilazak grafa. Polazimo od zadatog čvora, zatim biramo jednog njegovog neposećenog suseda i nastavljamo obilazak iz tog čvora. Kada iz nekog čvora više ne možemo da pređemo u neposećenog suseda, vraćamo se unazad i nastavljamo pretragu iz prethodnih čvorova.

U ovoj sekciji grafove ćemo predstavljati listama susedstva smeštenim u niz. Čvorovi će biti označeni celim brojevima, a na indeksu v nalaziće se lista suseda čvora v:

```
type Graph = Array Int [Int]
```

Dakle, vrednost tipa `Graph` jeste niz čiji su indeksi čvorovi grafa, a elementi su liste suseda odgovarajućih čvorova. Za rad sa nizovima koristićemo ranije uvedeni modul `Data.Array`. Pored toga, za pamćenje posećenih čvorova koristićemo skupove iz modula `Data.Set`:

```
import Data.Array (Array, listArray, (!))
import qualified Data.Set as Set
```

Modul `Data.Set` uvozimo kvalifikovano, što znači da se njegove funkcije pozivaju sa prefiksom `Set`. Tako, `Set.empty` predstavlja prazan skup, `Set.member` proverava da li element pripada skupu, a `Set.insert` dodaje element u skup.

Na primer, usmereni graf definisan sa

$$0 \to 1, \quad 0 \to 2, \quad 1 \to 3, \quad 2 \to 3, \quad 2 \to 4, \quad 4 \to 5$$

možemo predstaviti na sledeći način:

```
graph1 :: Graph
graph1 = listArray (0, 5)
  [ [1, 2]  -- susedi cvora 0
  , [3]     -- susedi cvora 1
  , [3, 4]  -- susedi cvora 2
  , []      -- susedi cvora 3
  , [5]     -- susedi cvora 4
  , []      -- susedi cvora 5
  ]
```

Komentari u kodu pokazuju koja lista odgovara kom čvoru. Na primer, pošto je na indeksu `0` lista `[1,2]`, iz čvora `0` postoje grane ka čvorovima `1` i `2`. Pošto je na indeksu `3` prazna lista, čvor `3` nema izlaznih grana.

Sada možemo definisati pretragu u dubinu. Funkcija će primati graf i početni čvor, a vraćaće listu čvorova redom kojim su prvi put posećeni:

```
dfs :: Graph -> Int -> [Int]
dfs graph start = reverse order
  where
    (_, order) = visit Set.empty [] start

    visit visited order v
      | Set.member v visited = (visited, order)
```

```
      | otherwise =
          foldl visitNeighbour
                (Set.insert v visited, v : order)
                (graph ! v)

    visitNeighbour (visited, order) u =
      visit visited order u
```

Pomoćna funkcija `visit` ima tri argumenta. Prvi argument, `visited`, jeste skup već posećenih čvorova. Drugi argument, `order`, jeste lista dosad posećenih čvorova. Treći argument, `v`, jeste čvor koji trenutno obrađujemo.

Ako je čvor `v` već posećen, ne radimo ništa i vraćamo postojeći par `(visited, order)`. Bez provere posećenosti mogli bismo isti čvor obilaziti više puta, a u grafu sa ciklusom mogli bismo upasti u beskonačnu rekurziju.

Ako čvor `v` nije posećen, dodajemo ga u skup posećenih čvorova i u listu obilaska. Zatim redom obilazimo njegove susede, koje dobijamo izrazom `graph ! v`. Za to koristimo funkciju `foldl`.

Početno stanje za obilazak suseda jeste par u kome je čvor `v` već označen kao posećen. Funkcija `visitNeighbour` zatim redom obrađuje susede čvora `v`. Svaki obilazak može promeniti skup posećenih čvorova i redosled obilaska, pa se ažurirani par prenosi dalje kroz listu suseda.

Zbog efikasnosti, čvorove dodajemo na početak liste `order`. Zbog toga je redosled u toj listi obrnut, pa na kraju primenjujemo funkciju `reverse`.

Na primer:

```
ghci> dfs graph1 0
[0,1,3,2,4,5]
```

Obilazak počinje u čvoru `0`. Najpre se ide ka čvoru `1`, zatim ka čvoru `3`. Kada iz čvora `3` više ne može da se nastavi, pretraga se vraća unazad i zatim iz čvora `0` prelazi ka čvoru `2`. Iz čvora `2` čvor `3` se preskače jer je već posećen, pa se nastavlja ka čvoru `4`, a zatim ka čvoru `5`.

Ako pretragu pokrenemo iz drugog čvora, dobijamo samo čvorove koji su iz njega dostižni:

```
ghci> dfs graph1 2
[2,3,4,5]

ghci> dfs graph1 3
[3]
```

Kod pretrage u dubinu svaki čvor se posećuje najviše jednom, a svaka grana se razmatra najviše jednom prilikom obilaska liste suseda. U našoj implementaciji za posećene čvorove koristimo skup, pa su operacije provere i dodavanja dovoljno efikasne za ovu namenu.

16.2 Pretraga u širinu

Pretraga u širinu je algoritam za obilazak grafa koji čvorove posećuje po nivoima. Polazimo od zadatog čvora, zatim obilazimo sve njegove susede, potom sve još neposećene čvorove do kojih se stiže u dva koraka, zatim one do kojih se stiže u tri koraka, i tako dalje.

Za razliku od pretrage u dubinu, koja ide što dalje niz jednu granu pre nego što se vrati unazad, pretraga u širinu najpre obrađuje čvorove koji su bliži početnom čvoru. Zbog toga se pretraga u širinu prirodno implementira pomoću reda.

Red je struktura podataka kod koje se elementi dodaju na jedan kraj, a uklanjaju sa drugog kraja. Element koji je prvi dodat biće prvi i obrađen. Takvo ponašanje odgovara ideji pretrage u širinu, gde se čvorovi obrađuju redom kojim su otkriveni.

Definisaćemo jednostavan red pomoću para listi:

```
type Queue a = ([a], [a])
```

Prva lista sadrži elemente koji se skidaju sa reda. Druga lista sadrži elemente koji su dodati na kraj reda, ali u obrnutom redosledu. Ovakva implementacija omogućava da dodavanje novog elementa bude jednostavno, jer ga dodajemo na početak druge liste.

Za početak obilaska biće nam potreban red koji sadrži samo početni čvor. Zato definišemo funkciju koja od jednog elementa pravi takav red:

```
singletonQueue :: a -> Queue a
singletonQueue x = ([x], [])
```

Element `x` stavljamo u prvu listu, jer je on odmah spreman za skidanje sa reda. Dodavanje novog elementa u red vrši se dodavanjem u drugu listu:

```
enqueue :: a -> Queue a -> Queue a
enqueue x (front, back) = (front, x : back)
```

Funkcija za skidanje elementa sa reda vraća `Maybe` vrednost, jer red može biti prazan:

```
dequeue :: Queue a -> Maybe (a, Queue a)
dequeue (x:xs, back) = Just (x, (xs, back))
dequeue ([], back) =
  case reverse back of
    []     -> Nothing
    x:xs   -> Just (x, (xs, []))
```

Ako je prva lista neprazna, skidamo njen prvi element. Ako je prva lista prazna, tada okrenemo drugu listu i od nje napravimo novu prvu listu. Ako su obe liste prazne, red je prazan i rezultat je `Nothing`.

Sada možemo definisati pretragu u širinu. Koristićemo isti tip grafa kao u prethodnoj podsekciji:

```
bfs :: Graph -> Int -> [Int]
bfs graph start = search (Set.singleton start) [] (singletonQueue start)
  where
    search visited order queue =
      case dequeue queue of
        Nothing -> reverse order
        Just (v, queue') ->
          let
            (visited', queue'') =
              foldl addNeighbour (visited, queue') (graph ! v)
          in
            search visited' (v : order) queue''

    addNeighbour (visited, queue) u
      | Set.member u visited = (visited, queue)
```

```
      | otherwise =
          (Set.insert u visited, enqueue u queue)
```

Funkcija `bfs` prima graf i početni čvor, a vraća listu čvorova redom kojim su obrađeni. Na početku je posećen samo početni čvor, pa je skup posećenih čvorova `Set.singleton start`. U redu se na početku nalazi samo početni čvor.

Pomoćna funkcija `search` ima tri argumenta. Prvi argument, `visited`, jeste skup već otkrivenih čvorova. Drugi argument, `order`, jeste lista obrađenih čvorova. Treći argument, `queue`, jeste red čvorova koje tek treba obraditi.

Ako je red prazan, obilazak je završen i vraćamo listu `order`. Pošto smo čvorove dodavali na početak te liste, na kraju primenjujemo `reverse`.

Ako red nije prazan, funkcija `dequeue` skida prvi čvor iz reda. Neka je to čvor `v`. Tada redom razmatramo sve njegove susede, koje dobijamo izrazom `graph ! v`. Za svakog suseda proveravamo da li je već otkriven. Ako jeste, preskačemo ga. Ako nije, dodajemo ga u skup posećenih čvorova i stavljamo ga na kraj reda.

To radi pomoćna funkcija `addNeighbour`. Ona prima trenutno stanje oblika `(visited, queue)` i jednog suseda `u`. Ako je `u` već u skupu `visited`, stanje se ne menja. Ako nije, čvor `u` se dodaje u skup i u red.

Na primer, za graf `graph1` iz prethodne podsekcije dobijamo:

```
ghci> bfs graph1 0
[0,1,2,3,4,5]
```

Obilazak počinje u čvoru `0`. Njegovi susedi su čvorovi `1` i `2`, pa oni dolaze sledeći. Zatim se obrađuju njihovi susedi. Iz čvora `1` dolazimo do čvora `3`, a iz čvora `2` do čvorova `3` i `4`. Čvor `3` se ne dodaje drugi put, jer je već otkriven. Na kraju se iz čvora `4` dolazi do čvora `5`.

Ako pretragu pokrenemo iz čvora `2`, dobijamo samo čvorove koji su iz njega dostižni:

```
ghci> bfs graph1 2
[2,3,4,5]

ghci> bfs graph1 3
[3]
```

Slično kao kod pretrage u dubinu, i kod pretrage u širinu svaki čvor se obrađuje najviše jednom, a svaka grana se razmatra najviše jednom prilikom obilaska liste suseda.

16.3 Povezane komponente

Kod neusmerenog grafa, povezana komponenta je najveći skup čvorova takav da se između svaka dva čvora iz tog skupa može doći nekim putem. Drugim rečima, povezana komponenta je jedan povezan deo grafa koji je odvojen od ostalih takvih delova.

Na primer, ako u grafu postoje grane između čvorova `0`, `1` i `2`, zatim posebna grana između čvorova `3` i `4`, dok je čvor `5` izolovan, tada graf ima tri povezane komponente: `[0,1,2]`, `[3,4]` i `[5]`.

Povezane komponente možemo pronaći pomoću pretrage u dubinu. Ideja je da izaberemo jedan još neobrađen čvor i iz njega pokrenemo pretragu. Svi čvorovi koje tada obiđemo pripadaju istoj povezanoj komponenti. Zatim te čvorove uklonimo iz skupa neobrađenih čvorova i postupak ponavljamo dok ne obradimo ceo graf.

Koristićemo isti tip `Graph` i istu funkciju `dfs` kao ranije. Funkcija `dfs` samo prati liste susedstva, pa se može koristiti i za usmerene i za neusmerene grafove. U ovoj podsekciji radimo sa neusmerenim grafovima. Zato svaku granu unosimo u oba smera: ako postoji grana između čvorova `u` i `v`, onda se `v` nalazi u listi suseda čvora `u`, a `u` u listi suseda čvora `v`.

Na primer:

```
graph2 :: Graph
graph2 = listArray (0, 5)
  [ [1]     -- susedi cvora 0
  , [0, 2]  -- susedi cvora 1
  , [1]     -- susedi cvora 2
  , [4]     -- susedi cvora 3
  , [3]     -- susedi cvora 4
  , []      -- susedi cvora 5
  ]
```

Ovaj graf ima tri povezane komponente. Prvu čine čvorovi `0`, `1` i `2`, drugu čine čvorovi `3` i `4`, a treću čini izolovani čvor `5`.

Za određivanje svih čvorova grafa biće nam potrebna funkcija `bounds` iz modula `Data.Array`, pa uvoz možemo proširiti ovako:

```
import Data.Array (Array, listArray, bounds, (!))
```

Sada možemo definisati funkciju koja određuje povezane komponente:

```
connectedComponents :: Graph -> [[Int]]
connectedComponents graph = components allVertices
  where
    (lo, hi) = bounds graph
    allVertices = Set.fromList [lo .. hi]

    components remaining
      | Set.null remaining = []
      | otherwise          = component : components remaining'
      where
        v = Set.findMin remaining
        component = dfs graph v
        remaining' =
          Set.difference remaining (Set.fromList component)
```

Funkcija `connectedComponents` vraća listu komponenti, pri čemu je svaka komponenta predstavljena listom čvorova.

Najpre pomoću `bounds graph` dobijamo najmanji i najveći indeks grafa. Pošto su čvorovi grafa upravo indeksi niza, lista `[lo .. hi]` sadrži sve čvorove. Od te liste pravimo skup `allVertices`, koji predstavlja skup čvorova koje još treba obraditi.

Pomoćna funkcija `components` prima skup preostalih neobrađenih čvorova. Ako je taj skup prazan, sve komponente su pronađene. U suprotnom biramo jedan čvor iz tog skupa:

```
v = Set.findMin remaining
```

Funkcija `Set.findMin` vraća najmanji element nepraznog skupa. Iz tog čvora pokrećemo

ranije definisanu pretragu u dubinu:

```
component = dfs graph v
```

Pošto je graf neusmeren, svi čvorovi koje DFS obiđe iz čvora `v` čine celu povezanu komponentu kojoj `v` pripada. Te čvorove zatim uklanjamo iz skupa preostalih čvorova:

```
remaining' =
  Set.difference remaining (Set.fromList component)
```

Funkcija `Set.difference` računa razliku skupova. Dakle, iz skupa `remaining` uklanjamo sve čvorove pronađene komponente. Zatim se isti postupak nastavlja nad skupom `remaining'`.

Na primer:

```
ghci> connectedComponents graph2
[[0,1,2],[3,4],[5]]
```

Prvo se bira čvor `0`. Pretraga u dubinu iz njega obilazi čvorove `0`, `1` i `2`, pa dobijamo prvu komponentu. Zatim se ti čvorovi uklanjaju iz skupa neobrađenih čvorova. Sledeći najmanji neobrađeni čvor je `3`, iz koga se obilaze čvorovi `3` i `4`. Na kraju ostaje još izolovani čvor `5`, koji sam čini poslednju komponentu.

Ako je graf povezan, rezultat će imati samo jednu komponentu. Posmatrajmo, na primer, sledeći neusmereni povezani graf:

```
connectedGraph :: Graph
connectedGraph = listArray (0, 4)
  [ [1, 2]
  , [0, 3]
  , [0, 4]
  , [1]
  , [2]
  ]
```

Za njega dobijamo:

```
ghci> connectedComponents connectedGraph
[[0,1,3,2,4]]
```

Umesto pretrage u dubinu mogli bismo koristiti i pretragu u širinu. Za određivanje jedne povezane komponente nije važno kojim redosledom obilazimo dostižne čvorove, već samo da iz izabranog početnog čvora obiđemo sve čvorove do kojih se iz njega može doći. Zato bi se u prethodnoj implementaciji poziv `dfs graph v` mogao zameniti pozivom `bfs graph v`, a rezultat bi i dalje bile iste povezane komponente, sa eventualno drugačijim redosledom čvorova unutar pojedinačnih komponenti.

17 Geometrijski algoritmi

17.1 Presek dve duži

Posmatrajmo dve duži AB i CD. Želimo da proverimo da li one imaju bar jednu zajedničku tačku. Tačku u ravni predstavljaćemo parom realnih brojeva:

```
type Point = (Double, Double)
```

Pošto se vrednosti tipa `Double` predstavljaju približno, poređenje sa nulom nećemo raditi neposredno. Umesto toga uvodimo malu toleranciju:

```
eps :: Double
eps = 1e-9
```

Zatim definišemo funkciju koja određuje znak broja uz tu toleranciju:

```
sign :: Double -> Int
sign x
  | x > eps   = 1
  | x < -eps  = -1
  | otherwise = 0
```

Ako je broj veći od `eps`, smatraćemo ga pozitivnim. Ako je manji od `-eps`, smatraćemo ga negativnim. Vrednosti između `-eps` i `eps` smatraćemo nulom.

Razmotrimo sada orijentaciju tri tačke. Neka su date tačke $A(x_1, y_1)$, $B(x_2, y_2)$ i $P(x_3, y_3)$. Orijentaciju određujemo znakom determinante

$$\begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix} = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

Ako je ova vrednost pozitivna, tačka P nalazi se sa jedne strane usmerene prave AB. Ako je negativna, nalazi se sa druge strane. Ako je jednaka nuli, tačke A, B i P su kolinearne.

U Haskellu ovu vrednost računamo funkcijom:

```
orientation :: Point -> Point -> Point -> Double
orientation (x1, y1) (x2, y2) (x3, y3) =
  (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1)
```

Na primer:

```
ghci> orientation (0,0) (4,0) (2,3)
12.0

ghci> orientation (0,0) (4,0) (2,-3)
-12.0

ghci> orientation (0,0) (4,0) (2,0)
0.0
```

U prvom primeru tačka $(2, 3)$ nalazi se sa jedne strane prave određene tačkama $(0, 0)$ i $(4, 0)$. U drugom primeru tačka $(2, -3)$ nalazi se sa druge strane te prave. U trećem primeru sve tri tačke su kolinearne.

Sada se možemo vratiti problemu preseka dve duži. Duži AB i CD se, u opštem slučaju, seku ako su krajevi jedne duži sa različitih strana prave određene drugom duži. Preciznije, tačke C i D treba da budu sa različitih strana prave AB, a tačke A i B sa različitih strana prave CD.

To proveravamo pomoću četiri orijentacije:

```
orientation a b c
orientation a b d
orientation c d a
orientation c d b
```

Ako prve dve orijentacije imaju suprotne znakove, onda su tačke C i D sa različitih strana prave AB. Ako druge dve orijentacije imaju suprotne znakove, onda su tačke A i B sa različitih strana prave CD. Kada su oba uslova ispunjena, duži AB i CD se seku u unutrašnjim tačkama.

Međutim, time nisu obuhvaćeni svi slučajevi. Moguće je da neka krajnja tačka jedne duži leži na drugoj duži. Posebno, duži mogu biti kolinearne i tada se mogu preklapati, dodirivati u jednoj tački ili biti razdvojene. Zato nam je potrebna pomoćna funkcija koja proverava da li se broj nalazi između dva data broja, uz toleranciju `eps`:

```
between :: Double -> Double -> Double -> Bool
between a b x = min a b - eps <= x && x <= max a b + eps
```

Ako su tačke A, B i P kolinearne, onda tačka P pripada duži AB ako se njena x-koordinata nalazi između x-koordinata tačaka A i B, i ako se njena y-koordinata nalazi između njihovih y-koordinata:

```
onSegment :: Point -> Point -> Point -> Bool
onSegment (x1, y1) (x2, y2) (x, y) =
  between x1 x2 x && between y1 y2 y
```

Funkciju `onSegment` koristićemo samo onda kada već znamo da su posmatrane tri tačke kolinearne. Sama provera pomoću koordinata tada je dovoljna da odluči da li tačka leži između krajeva duži.

Sada možemo definisati funkciju koja proverava da li se dve duži seku:

```
intersect :: Point -> Point -> Point -> Point -> Bool
intersect a b c d =
  let
    o1 = sign (orientation a b c)
    o2 = sign (orientation a b d)
    o3 = sign (orientation c d a)
    o4 = sign (orientation c d b)
  in
    (o1 == 0 && onSegment a b c) ||
    (o2 == 0 && onSegment a b d) ||
    (o3 == 0 && onSegment c d a) ||
    (o4 == 0 && onSegment c d b) ||
    (o1 * o2 < 0 && o3 * o4 < 0)
```

Prva četiri uslova proveravaju slučajeve kada neka krajnja tačka jedne duži leži na drugoj duži. Na primer, uslov

```
o1 == 0 && onSegment a b c
```

proverava da li je tačka C kolinearna sa tačkama A i B, i da li pripada duži AB. Analogno se tumače i naredna tri uslova.

Poslednji uslov

```
o1 * o2 < 0 && o3 * o4 < 0
```

proverava pravi presek, kada se duži seku u unutrašnjim tačkama. Uslov `o1 * o2 < 0` znači da su tačke C i D sa različitih strana prave AB, a uslov `o3 * o4 < 0` znači da su tačke A i B sa različitih strana prave CD.

Na primer, sledeće dve duži se seku u unutrašnjoj tački:

```
ghci> intersect (0,0) (4,4) (0,4) (4,0)
True
```

Sledeće dve duži su paralelne i nemaju zajedničkih tačaka:

```
ghci> intersect (0,0) (4,0) (0,1) (4,1)
False
```

Sledeće dve duži leže na istoj pravoj, ali su razdvojene:

```
ghci> intersect (0,0) (2,0) (3,0) (5,0)
False
```

Ako se kolinearne duži delimično preklapaju, presek je neprazan:

```
ghci> intersect (0,0) (4,0) (2,0) (6,0)
True
```

Duži koje imaju zajedničku krajnju tačku takođe se seku:

```
ghci> intersect (0,0) (2,2) (2,2) (4,0)
True
```

Moguće je i da se odgovarajuće prave seku, ali da se same duži ne seku:

```
ghci> intersect (0,0) (1,1) (2,0) (2,3)
False
```

Prava određena prvom duži seče pravu određenu drugom duži, ali se tačka preseka ne nalazi na prvoj duži, pa je rezultat `False`.

Funkcija ispravno obrađuje i slučaj kada se jedna duž svodi na jednu tačku:

```
ghci> intersect (1,1) (1,1) (0,0) (2,2)
True

ghci> intersect (3,3) (3,3) (0,0) (2,2)
False
```

U prvom primeru tačka $(1, 1)$ leži na duži od $(0, 0)$ do $(2, 2)$, pa je presek neprazan. U

drugom primeru tačka $(3, 3)$ ne pripada toj duži, pa preseka nema.

Prema tome, algoritam se zasniva na orijentaciji trojki tačaka. U opštem slučaju proveravamo da li krajevi svake duži leže sa različitih strana prave određene drugom duži. Posebni slučajevi, uključujući zajedničke krajnje tačke i kolinearne duži, obrađuju se proverom da li odgovarajuća tačka pripada odgovarajućoj duži.

17.2 Površina prostog poligona

Poligon ćemo predstaviti listom njegovih temena, datih redom po obodu. Na primer, pravougaonik sa temenima $(0, 0)$, $(4, 0)$, $(4, 3)$ i $(0, 3)$ možemo predstaviti listom:

```
[(0,0), (4,0), (4,3), (0,3)]
```

Redosled temena je važan. Pretpostavićemo da su temena navedena redom kojim se obilazi granica poligona, bilo u smeru kazaljke na satu, bilo u suprotnom smeru. Poligon ne mora biti konveksan, ali ćemo pretpostaviti da je prost, odnosno da mu se stranice ne seku osim u susednim temenima.

Neka su temena poligona redom $A_1, A_2, \ldots, A_n$. Ako je $A_i = (x_i, y_i)$, onda ćemo koordinate narednog temena označavati sa x_{i+1} i y_{i+1}. Pošto se poslednja stranica poligona završava u početnom temenu, indekse posmatramo ciklično, pa važi $A_{n+1} = A_1$.

Površinu poligona možemo izračunati tako što saberemo orijentisane površine trouglova OA_iA_{i+1}, gde je O koordinatni početak. Za tačke $O(0, 0)$, $A_i(x_i, y_i)$ i $A_{i+1}(x_{i+1}, y_{i+1})$, dvostruka orijentisana površina trougla OA_iA_{i+1} jednaka je determinanti

$$\begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix} = x_i y_{i+1} - y_i x_{i+1}.$$

Zato je orijentisana površina tog trougla jednaka

$$\frac{1}{2}(x_i y_{i+1} - y_i x_{i+1}).$$

Površina celog poligona dobija se uzimanjem apsolutne vrednosti zbira orijentisanih površina svih odgovarajućih trouglova:

$$P = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1})\right|.$$

Apsolutna vrednost je potrebna zato što znak sume zavisi od redosleda obilaska temena. Ako su temena navedena u jednom smeru, suma je pozitivna, a ako su navedena u suprotnom smeru, suma je negativna. Površina, naravno, ne zavisi od izabranog smera obilaska.

Za svaku stranicu A_iA_{i+1} računamo dvostruku orijentisanu površinu trougla OA_iA_{i+1}:

```
doubleSignedArea :: Point -> Point -> Double
doubleSignedArea (x1, y1) (x2, y2) =
  x1 * y2 - y1 * x2
```

Na primer:

```
ghci> doubleSignedArea (4,0) (4,3)
12.0
```

```
ghci> doubleSignedArea (4,3) (0,3)
12.0
```

U prvom primeru posmatramo trougao određen koordinatnim početkom i tačkama $(4, 0)$ i $(4, 3)$. Njegova dvostruka orijentisana površina jednaka je 12. U drugom primeru dobija se ista vrednost, jer se i odgovarajući trougao obilazi u istom smeru u odnosu na koordinatni početak.

Da bismo primenili formulu za površinu poligona, potrebno je da posmatramo sve parove susednih temena, uključujući i par koji čine poslednje i prvo teme. Zato definišemo funkciju koja od liste temena pravi listu odgovarajućih stranica:

```
edges :: [Point] -> [(Point, Point)]
edges [] = []
edges ps = zip ps (tail ps ++ [head ps])
```

U ovoj definiciji prva lista je lista temena poligona, a druga lista je ista ta lista pomerena za jedno mesto ulevo, pri čemu se prvo teme dodaje na kraj. Tako dobijamo sve parove susednih temena.

Na primer:

```
ghci> edges [(0,0), (4,0), (4,3), (0,3)]
[((0.0,0.0),(4.0,0.0)),
 ((4.0,0.0),(4.0,3.0)),
 ((4.0,3.0),(0.0,3.0)),
 ((0.0,3.0),(0.0,0.0))]
```

Sada možemo izračunati orijentisanu površinu poligona. Za svaku stranicu računamo odgovarajuću dvostruku orijentisanu površinu, zatim sve te vrednosti saberemo i rezultat podelimo sa 2:

```
signedArea :: [Point] -> Double
signedArea ps =
  sum [doubleSignedArea p q | (p, q) <- edges ps] / 2
```

Orijentisana površina može biti pozitivna ili negativna, u zavisnosti od smera kojim su temena navedena. Ako želimo običnu površinu, uzimamo apsolutnu vrednost:

```
polygonArea :: [Point] -> Double
polygonArea ps = abs (signedArea ps)
```

Na primer, za pravougaonik stranica 4 i 3 dobijamo površinu 12:

```
ghci> polygonArea [(0,0), (4,0), (4,3), (0,3)]
12.0
```

Ako ista temena navedemo u suprotnom redosledu, orijentisana površina menja znak, ali obična površina ostaje ista:

```
ghci> signedArea [(0,0), (4,0), (4,3), (0,3)]
12.0

ghci> signedArea [(0,0), (0,3), (4,3), (4,0)]
-12.0
```

```
ghci> polygonArea [(0,0), (0,3), (4,3), (4,0)]
12.0
```

Formula važi i za nekonveksne proste poligone. Na primer, posmatrajmo poligon sa temenima:

```
poly :: [Point]
poly = [(0,0), (4,0), (4,2), (2,1), (0,2)]
```

Njegova površina je:

```
ghci> polygonArea poly
6.0
```

U ovom primeru poligon nije konveksan, ali su temena data redom po obodu i stranice se ne seku. Zato se formula može neposredno primeniti.

Ako lista ima manje od tri temena, ne dobijamo pravi poligon. U tom slučaju površina treba da bude nula. Prethodne definicije daju upravo takav rezultat:

```
ghci> polygonArea []
0.0

ghci> polygonArea [(1,1)]
0.0

ghci> polygonArea [(0,0), (3,0)]
0.0
```

Za listu sa jednim temenom funkcija `edges` pravi degenerisanu stranicu od te tačke do nje same, pa je odgovarajuća dvostruka orijentisana površina jednaka nuli. Za listu sa dva temena dobijaju se dve suprotno orijentisane degenerisane stranice, pa je zbir takođe jednak nuli.

Prema tome, površinu poligona možemo izračunati jednim prolaskom kroz listu temena. Za svaku stranicu A_iA_{i+1} računamo vrednost $x_iy_{i+1} - y_ix_{i+1}$, sve te vrednosti saberemo, podelimo sa 2, a zatim uzmemo apsolutnu vrednost. Znak orijentisane površine dodatno nam govori u kom smeru su temena navedena.

17.3 Konveksnost poligona

Poligon je konveksan ako duž koja spaja bilo koje dve njegove tačke cela pripada tom poligonu. Intuitivno, to znači da poligon nema udubljenja. U ovoj podsekciji pretpostavljamo da je poligon prost i da su njegova temena data redom po obodu.

Konveksnost možemo proveriti pomoću orijentacije uzastopnih trojki temena. Ako su temena poligona navedena redom po obodu, onda kod konveksnog poligona svi zaokreti imaju isti smer. Drugim rečima, za svake tri uzastopne tačke A_i, A_{i+1} i A_{i+2}, orijentacija treba da ima isti znak.

Na primer, kod pravougaonika

```
[(0,0), (4,0), (4,3), (0,3)]
```

pri obilasku temena stalno skrećemo u istu stranu. Kod nekonveksnog poligona postoji bar jedno teme u kome se smer skretanja promeni, što odgovara udubljenju.

Koristićemo već definisanu funkciju `orientation`. Podsetimo se, izraz

```
orientation a b c
```

određuje orijentaciju tačaka `a`, `b` i `c`. Pozitivan i negativan znak predstavljaju različite smerove zaokreta, dok nula znači da su tačke kolinearne.

Najpre ćemo definisati funkciju koja od liste temena pravi listu svih uzastopnih trojki. Kao i kod stranica poligona, temena posmatramo ciklično:

```
triples :: [Point] -> [(Point, Point, Point)]
triples ps
  | length ps < 3 = []
  | otherwise     = zip3 ps ps' ps''
  where
    ps'  = tail ps ++ [head ps]
    ps'' = drop 2 ps ++ take 2 ps
```

Funkcija `zip3` spaja tri liste po pozicijama. Prva lista je početna lista temena. Druga lista je ista ta lista pomerena za jedno mesto ulevo, a treća lista je pomerena za dva mesta ulevo. Tako se dobijaju trojke uzastopnih temena.

Na primer:

```
ghci> triples [(0,0), (4,0), (4,3), (0,3)]
[((0.0,0.0),(4.0,0.0),(4.0,3.0)),
 ((4.0,0.0),(4.0,3.0),(0.0,3.0)),
 ((4.0,3.0),(0.0,3.0),(0.0,0.0)),
 ((0.0,3.0),(0.0,0.0),(4.0,0.0))]
```

Sada za svaku takvu trojku računamo znak orijentacije. Kolinearne trojke ne utiču na konveksnost, pa ćemo znakove jednake nuli izostaviti. Na taj način dozvoljavamo da se na jednoj stranici poligona nalazi više uzastopnih temena.

```
turns :: [Point] -> [Int]
turns ps =
  filter (/= 0)
    [sign (orientation a b c) | (a, b, c) <- triples ps]
```

Ako je poligon konveksan, svi elementi liste `turns ps` treba da budu pozitivni ili svi treba da budu negativni. Ako se pojavljuju i pozitivni i negativni znakovi, poligon ima udubljenje i nije konveksan.

Zato funkciju za proveru konveksnosti možemo definisati ovako:

```
isConvex :: [Point] -> Bool
isConvex ps =
  length ps >= 3 &&
  not (null ts) &&
  (all (> 0) ts || all (< 0) ts)
  where
    ts = turns ps
```

Funkcija `all` proverava da li svi elementi liste zadovoljavaju odgovarajući uslov. Uslovom `length ps >= 3` se proverava da li imamo bar tri temena. Uslov `not (null ts)` isključuje

degenerisan slučaj kada su sva temena kolinearna. Poslednji uslov proverava da li su svi nenulti zaokreti istog znaka.

Na primer, pravougaonik je konveksan:

```
ghci> isConvex [(0,0), (4,0), (4,3), (0,3)]
True
```

Ako ista temena navedemo u suprotnom smeru, poligon je i dalje konveksan:

```
ghci> isConvex [(0,0), (0,3), (4,3), (4,0)]
True
```

U prvom slučaju svi nenulti zaokreti imaju jedan znak, a u drugom slučaju svi imaju suprotan znak. Konveksnost ne zavisi od smera obilaska temena.

Posmatrajmo sada nekonveksan poligon:

```
nonConvex :: [Point]
nonConvex = [(0,0), (4,0), (4,2), (2,1), (0,2)]
```

Za njega dobijamo:

```
ghci> isConvex nonConvex
False
```

Razlog je u tome što se u temenu $(2, 1)$ javlja udubljenje. Pri obilasku poligona jedan zaokret ima suprotan smer od ostalih.

Funkcija dozvoljava i slučaj kada se više uzastopnih temena nalazi na istoj stranici konveksnog poligona:

```
ghci> isConvex [(0,0), (2,0), (4,0), (4,3), (0,3)]
True
```

Ovde su tačke $(0, 0)$, $(2, 0)$ i $(4, 0)$ kolinearne. Takva trojka daje orijentaciju nula, pa se njen znak ne uzima u obzir. Preostali nenulti zaokreti imaju isti smer.

Ako su sva temena kolinearna, ne dobijamo pravi poligon, pa rezultat treba da bude `False`:

```
ghci> isConvex [(0,0), (1,0), (2,0), (3,0)]
False
```

Prema tome, konveksnost prostog poligona možemo proveriti jednim prolaskom kroz njegove uzastopne trojke temena. Za svaku trojku računamo orijentaciju, zanemarujemo kolinearne slučajeve, a zatim proveravamo da li su svi preostali zaokreti istog znaka.

17.4 Pripadnost tačke poligonu

Posmatrajmo prost poligon i tačku P. Želimo da odredimo da li tačka P pripada tom poligonu. Pritom ćemo smatrati da mu pripada i kada se nalazi na nekoj njegovoj stranici.

Koristićemo sledeću ideju. Iz tačke P povučemo polupravu udesno, paralelnu x-osi, i brojimo koliko puta ta poluprava seče stranice poligona. Ako je broj preseka neparan, tačka je unutar poligona, a ako je broj preseka paran, tačka je izvan poligona.

Ovo pravilo može se intuitivno razumeti na sledeći način. Ako se krećemo po polupravoj iz tačke koja je unutar poligona ka beskonačnosti, pri svakom preseku sa granicom poligona

prelazimo iz unutrašnjosti u spoljašnjost ili obrnuto. Pošto na kraju sigurno završavamo izvan poligona, broj takvih prelazaka mora biti neparan ako smo krenuli iz unutrašnjosti, a paran ako smo krenuli iz spoljašnjosti.

Najpre ćemo definisati funkciju koja proverava da li tačka pripada nekoj stranici poligona. Koristićemo već definisane funkcije `orientation`, `sign` i `onSegment`:

```
pointOnSegment :: Point -> Point -> Point -> Bool
pointOnSegment p a b =
  sign (orientation a b p) == 0 && onSegment a b p
```

Funkcija `pointOnSegment p a b` vraća `True` ako tačka `p` leži na duži određene tačkama `a` i `b`. Prvi uslov proverava kolinearnost, a drugi proverava da li se tačka nalazi između krajeva duži.

Na primer:

```
ghci> pointOnSegment (2,0) (0,0) (4,0)
True

ghci> pointOnSegment (5,0) (0,0) (4,0)
False

ghci> pointOnSegment (2,1) (0,0) (4,0)
False
```

Sada definišemo funkciju koja proverava da li poluprava iz tačke P udesno seče jednu stranicu poligona. Neka je $P = (p_x, p_y)$ i neka je posmatrana stranica poligona određena tačkama $A(x_1, y_1)$ i $B(x_2, y_2)$.

Poluprava iz tačke P udesno nalazi se na horizontalnoj pravoj $y = p_y$. Zato stranica AB, gde je $A(x_1, y_1)$ i $B(x_2, y_2)$, može dati presek sa tom polupravom samo ako horizontalna prava $y = p_y$ prolazi između njenih krajeva.

Pri brojanju preseka treba biti pažljiv kada horizontalna poluprava prolazi kroz neko teme poligona. Takvo teme pripada dvema susednim stranicama, pa bi moglo da se desi da isti presek izbrojimo dva puta. Da bismo to izbegli, stranicu brojimo samo kada je tačno jedan njen kraj iznad horizontalne prave kroz P. U Haskellu se to zapisuje ovako:

```
(y1 > py) /= (y2 > py)
```

Operator `/=` proverava da li su dve vrednosti različite. Zato ovaj uslov znači da su izrazi `y1 > py` i `y2 > py` različitih istinitosnih vrednosti. Drugim rečima, jedan kraj stranice je iznad horizontalne prave $y = p_y$, a drugi nije. Na taj način svako teme kroz koje prolazi horizontalna prava pripada tačno jednoj od dve susedne stranice koje se računaju, pa ne dolazi do dvostrukog brojanja.

Ako je ovaj uslov ispunjen, stranica seče horizontalnu pravu $y = p_y$ u tačno jednoj tački. Ostaje da proverimo da li se ta tačka preseka nalazi desno od tačke P.

Tačku na pravoj kroz A i B možemo zapisati u parametarskom obliku:

$$(x, y) = (x_1, y_1) + t(x_2 - x_1, y_2 - y_1).$$

Po koordinatama, to znači da je $x = x_1 + t(x_2 - x_1)$ i $y = y_1 + t(y_2 - y_1)$. Pošto tražimo presek sa horizontalnom pravom $y = p_y$, u drugu jednačinu stavljamo $y = p_y$:

$$p_y = y_1 + t(y_2 - y_1).$$

Odavde dobijamo

$$t = \frac{p_y - y_1}{y_2 - y_1}.$$

Kada ovu vrednost zamenimo u jednačinu za x, dobijamo

$$x = x_1 + \frac{p_y - y_1}{y_2 - y_1}(x_2 - x_1).$$

Ovo x nije x-koordinata tačke P, već x-koordinata preseka stranice AB sa horizontalnom pravom kroz P. Pošto poluprava kreće iz tačke P i ide udesno, presek brojimo samo ako je ta koordinata veća od p_x.

```
rayIntersects :: Point -> (Point, Point) -> Bool
rayIntersects (px, py) ((x1, y1), (x2, y2)) =
  (y1 > py) /= (y2 > py) && x > px
  where
    x = x1 + (py - y1) * (x2 - x1) / (y2 - y1)
```

Deljenje sa `y2 - y1` je bezbedno u ovom izrazu. Ako je stranica horizontalna, onda je `y1 == y2`, pa izrazi `y1 > py` i `y2 > py` imaju istu vrednost. Uslov `(y1 > py) /= (y2 > py)` tada nije ispunjen, pa se horizontalne stranice ne broje u funkciji `rayIntersects`.

To je poželjno ponašanje. Ako tačka P leži na horizontalnoj stranici, taj slučaj je već prethodno obrađen funkcijom `pointOnSegment`. Ako ne leži, horizontalna stranica ne treba posebno da utiče na broj preseka, jer bi mogla da napravi nejasnoću pri brojanju preseka u temenima.

Sada možemo definisati funkciju koja proverava da li tačka pripada poligonu:

```
pointInPolygon :: Point -> [Point] -> Bool
pointInPolygon p poly
  | any (\(a, b) -> pointOnSegment p a b) es = True
  | otherwise = odd crossings
  where
    es = edges poly
    crossings = length [e | e <- es, rayIntersects p e]
```

Funkcija `any` proverava da li bar neki element liste zadovoljava odgovarajući uslov. Ovde najpre proveravamo da li tačka leži na nekoj stranici poligona. Ako leži, odmah vraćamo `True`, jer tačke na granici računamo kao tačke koje pripadaju poligonu. U suprotnom brojimo preseke poluprave sa stranicama poligona. Funkcija `odd` proverava da li je broj preseka neparan.

Na primer, posmatrajmo pravougaonik:

```
rect :: [Point]
rect = [(0,0), (4,0), (4,3), (0,3)]
```

Tačka $(2, 1)$ je unutar pravougaonika:

```
ghci> pointInPolygon (2,1) rect
True
```

Tačka $(5, 1)$ je izvan pravougaonika:

```
ghci> pointInPolygon (5,1) rect
False
```

Tačka na stranici poligona takođe se računa kao tačka koja pripada poligonu:

```
ghci> pointInPolygon (4,2) rect
True

ghci> pointInPolygon (0,0) rect
True
```

Algoritam radi i za nekonveksne proste poligone. Posmatrajmo poligon:

```
nonConvex :: [Point]
nonConvex = [(0,0), (4,0), (4,2), (2,1), (0,2)]
```

Tačka $(1, 1)$ pripada poligonu:

```
ghci> pointInPolygon (1,1) nonConvex
True
```

Tačka $(2, 1.5)$ nalazi se u udubljenju i ne pripada poligonu:

```
ghci> pointInPolygon (2,1.5) nonConvex
False
```

Prema tome, pripadnost tačke prostom poligonu možemo proveriti jednim prolaskom kroz stranice poligona. Najpre posebno proveravamo da li tačka leži na granici. Ako ne leži, brojimo preseke horizontalne poluprave sa stranicama poligona. Neparan broj preseka znači da je tačka unutar poligona, a paran broj preseka znači da je izvan njega.

17.5 Konveksni omotač

Neka je dat konačan skup tačaka u ravni. Konveksni omotač tog skupa je najmanji konveksni poligon koji sadrži sve date tačke. Intuitivno, ako bismo oko svih tačaka zategli gumenu traku, oblik koji bi ona zauzela bio bi konveksni omotač.

U ovoj podsekciji opisaćemo jedan poznat algoritam za određivanje konveksnog omotača, koji se naziva Grejamovo skeniranje. Algoritam najpre bira jednu početnu tačku, zatim sortira ostale tačke po uglu poluprave koja iz početne tačke prolazi kroz njih, a potom jednim prolaskom kroz sortirane tačke izbacuje one koje ne mogu biti temena konveksnog omotača.

Za sortiranje i izbor najmanje tačke koristićemo funkcije iz modula `Data.List`:

```
import Data.List (sortBy, minimumBy, nub)
```

Funkcija `sortBy` sortira listu pomoću zadatog poređenja. Funkcija `minimumBy` pronalazi najmanji element liste po zadatom poređenju. Funkcija `nub` izbacuje ponovljene elemente iz liste.

Prvi korak algoritma jeste izbor početne tačke. Uzećemo tačku sa najmanjom y-koordinatom, a ako takvih ima više, među njima tačku sa najmanjom x-koordinatom. Takva tačka sigurno pripada konveksnom omotaču, jer se nalazi najniže među svim datim tačkama.

```
compareByYThenX :: Point -> Point -> Ordering
compareByYThenX (x1, y1) (x2, y2)
  | y1 < y2   = LT
  | y1 > y2   = GT
```

```
  | x1 < x2   = LT
  | x1 > x2   = GT
  | otherwise = EQ
```

Funkcija `compareByYThenX` vraća vrednost tipa `Ordering`. Taj tip opisuje rezultat poređenja i ima tri moguće vrednosti: `LT`, `EQ` i `GT`. Vrednost `LT` znači da je prvi argument manji od drugog, `GT` da je prvi argument veći od drugog, a `EQ` da su argumenti jednaki po kriterijumu poređenja. U prethodnoj definiciji najpre poredimo y-koordinate. Ako su one jednake, dodatno poredimo x-koordinate. Zato funkcija bira tačku koja je najniža, a u slučaju jednakih y-koordinata onu koja je najviše levo. Na primer, pri izboru početne tačke dobijamo:

```
ghci> minimumBy compareByYThenX [(2,3), (0,0), (1,0), (4,2)]
(0.0,0.0)
```

Između tačaka $(0, 0)$ i $(1, 0)$ bira se tačka $(0, 0)$, jer obe imaju najmanju y-koordinatu, ali tačka $(0, 0)$ ima manju x-koordinatu.

Sledeći korak je sortiranje ostalih tačaka po uglu u odnosu na početnu tačku. Za početnu tačku P, tačka A treba da dođe pre tačke B ako je zaokret od prave PA ka pravoj PB pozitivan. To ponovo možemo proveriti pomoću orijentacije.

Ako su tačke P, A i B kolinearne, onda ih sortiramo po udaljenosti od P. Bliža tačka dolazi pre dalje. Kasnije, tokom skeniranja, bliža kolinearna tačka biće izbačena, a ostaće najudaljenija, što je upravo ono što želimo za konveksni omotač.

Najpre definišemo kvadrat rastojanja dve tačke. Nije potrebno računati koren, jer je za poređenje udaljenosti dovoljno porediti kvadrate rastojanja:

```
dist2 :: Point -> Point -> Double
dist2 (x1, y1) (x2, y2) =
  (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)
```

Zatim definišemo poređenje po uglu u odnosu na zadatu početnu tačku:

```
compareByAngle :: Point -> Point -> Point -> Ordering
compareByAngle p a b
  | s > 0     = LT
  | s < 0     = GT
  | otherwise = compare (dist2 p a) (dist2 p b)
  where
    s = sign (orientation p a b)
```

Ako je orijentacija tačaka `p`, `a` i `b` pozitivna, tačka `a` dolazi pre tačke `b`. Ako je orijentacija negativna, tačka `b` dolazi pre tačke `a`. Ako su tačke kolinearne, imaju isti ugao u odnosu na početnu tačku `p`, pa ih dodatno uređujemo po udaljenosti od `p`.

U poslednjoj grani zato koristimo ugrađenu funkciju `compare`. Ona poredi dva argumenta i vraća vrednost tipa `Ordering`: `LT` ako je prvi argument manji, `GT` ako je prvi argument veći, a `EQ` ako su jednaki.

Na primer:

```
ghci> sortBy (compareByAngle (0,0)) [(1,1), (2,0), (0,2)]
[(2.0,0.0),(1.0,1.0),(0.0,2.0)]
```

Tačke su sortirane po uglu koji poluprave iz koordinatnog početka ka tim tačkama zaklapaju

sa pozitivnim smerom x-ose.

Kada su tačke sortirane, prolazimo kroz njih redom i održavamo trenutno izgrađen deo omotača. Za svaku novu tačku proveravamo poslednje dve tačke koje su trenutno na omotaču. Ako bi dodavanje nove tačke napravilo pogrešan zaokret, poslednju tačku izbacujemo, jer ona ne može biti teme konveksnog omotača.

U sledećoj funkciji lista koja predstavlja omotač čuva se obrnutim redosledom. Zbog toga se u obrascu `q:p:rest` tačka `q` nalazi poslednja na trenutnom omotaču, a tačka `p` pretposlednja.

```
addPoint :: [Point] -> Point -> [Point]
addPoint (q:p:rest) r
  | sign (orientation p q r) <= 0 = addPoint (p:rest) r
  | otherwise                     = r:q:p:rest
addPoint hull r = r:hull
```

Uslov

```
sign (orientation p q r) <= 0
```

znači da tačke `p`, `q` i `r` ne prave pozitivan zaokret. Tada tačka `q` ne može ostati na konveksnom omotaču, pa je izbacujemo i ponavljamo proveru. Ako je zaokret pozitivan, novu tačku dodajemo na omotač.

Kolinearne tačke se u ovoj definiciji takođe izbacuju iz unutrašnjosti stranice omotača. To znači da će rezultat sadržati samo krajnja temena konveksnog omotača, a ne i tačke koje leže na njegovim stranicama.

Sada možemo definisati ceo algoritam:

```
convexHull :: [Point] -> [Point]
convexHull ps
  | length points <= 1 = points
  | otherwise          =
      let
        start  = minimumBy compareByYThenX points
        others = filter (/= start) points
        sorted = start : sortBy (compareByAngle start) others
      in
        reverse (foldl addPoint [] sorted)
  where
    points = nub ps
```

Najpre funkcijom `nub` uklanjamo ponovljene tačke. Ako posle toga imamo najviše jednu tačku, konveksni omotač je upravo ta lista. U suprotnom biramo početnu tačku `start`, sortiramo ostale tačke po uglu u odnosu na nju, a zatim funkcijom `foldl` redom dodajemo tačke na omotač. Pošto funkcija `addPoint` čuva omotač obrnutim redosledom, na kraju primenjujemo `reverse`.

Na primer, posmatrajmo skup tačaka koji sadrži temena pravougaonika i nekoliko unutrašnjih tačaka:

```
points1 :: [Point]
points1 = [(0,0), (4,0), (4,3), (0,3), (2,1), (2,2), (1,1)]
```

Konveksni omotač čine temena pravougaonika:

```
ghci> convexHull points1
[(0.0,0.0),(4.0,0.0),(4.0,3.0),(0.0,3.0)]
```

Ako se neke tačke nalaze na stranicama omotača, one se ne zadržavaju u rezultatu:

```
points2 :: [Point]
points2 = [(0,0), (2,0), (4,0), (4,3), (2,3), (0,3)]
```

Za ovaj skup dobijamo:

```
ghci> convexHull points2
[(0.0,0.0),(4.0,0.0),(4.0,3.0),(0.0,3.0)]
```

Tačke $(2, 0)$ i $(2, 3)$ leže na stranicama konveksnog omotača, ali nisu njegova krajnja temena, pa se izbacuju.

Algoritam ispravno obrađuje i slučaj kada su sve tačke kolinearne:

```
ghci> convexHull [(0,0), (1,0), (2,0), (3,0)]
[(0.0,0.0),(3.0,0.0)]
```

U tom slučaju konveksni omotač nema površinu i svodi se na duž između dve najudaljenije krajnje tačke.

Ako su sve tačke jednake, posle uklanjanja duplikata ostaje samo jedna tačka:

```
ghci> convexHull [(1,1), (1,1), (1,1)]
[(1.0,1.0)]
```

Ako je ulazna lista prazna, rezultat je prazna lista:

```
ghci> convexHull []
[]
```

Vremenska složenost algoritma određena je sortiranjem tačaka po uglu. Ako je broj tačaka n, sortiranje zahteva $O(n \log n)$ vremena, dok završno skeniranje tačaka zahteva $O(n)$ vremena. Ukupna složenost algoritma je zato $O(n \log n)$.