Programiranje u Haskelu

Knjiga o funkcionalnom programiranju

Nikola Ubavić

07.09.2026

Sadržaj

1. Predgovor

Ova knjiga je nastala sasvim neplanirano. Prvobitno sam napisao par stranica o Haskelu za potrebe predmeta *Programske paradigme* za učenike Zemunske gimnazije. Ti prvobitni podsetnici su neprestano dopunjavani, tako da su prerasli u skriptu od nekoliko desetina strana dostupnu na *GitHub*-u, a zatim i u digitalnu knjigu na domenu `haskel.ubavic.rs`. Kako se obim teksta širio, širila se i ciljna publika, tako da sada ova knjiga može poslužiti ne samo učenicima srednje škole, već i studentima koji na nekim od kurseva izučavaju programski jezik Haskel, ali i svima onima koji su zainteresovani za funkcionalno programiranje.

Knjiga je napisana za totalne početnike u funkcionalnom programiranju i od čitaoca se ne očekuje da poseduje bilo kakvo znanje o funkcionalnom programiranju. Ipak, podrazumeva se određena računarska pismenost (npr. poznavanje osnove rada u terminalu ili poznavanje osnova računarske arhitekture). Neophodno računarsko znanje je malo, i lako se može nadoknaditi istovremeno sa čitanjem knjige (npr. uz pomoć materijala dostupnih na internetu ili uz pomoć nastavnika ili prijatelja). Takođe, na nekim mestima u knjizi neophodno je znanje matematike (na nivou gimnazije) - ovakvi "matematički" delovi knjige se mogu preskočiti ako nisu interesantni čitaocu. Poznavanje drugog programskog jezika nije potrebno, ali je veoma korisno jer se u knjizi često ističu sličnosti i razlike između Haskela i nekog od "popularnijih" jezika poput *C*-a, Jave, *Python*-a, itd.

Ako bi se literatura o Haskelu mogla podeliti na osnovni, srednji i napredni nivo, onda bi ova knjiga pokrivala osnovni i deo srednjeg nivoa. Postoji mnogo Haskel knjiga koje pokrivaju sličan opseg, ali su načini izlaganja značajno različiti. Prilikom pisanja ove knjige, trudio sam se da svaki koncept objasnim detaljno (koliko je potrebno) pre prelaženja na sledeći koncept. Takođe, teme su izlagane redosledom koji je omogućio da se nijedno objašnjenje ne odlaže za kasnije (uz par izuzetaka). Na ovaj način, čitalac ima potpuno razumevanje svega što mu se na određenom mestu izlaže. Ovakav pristup zahteva nešto veće strpljenje od čitaoca, ali daje potpuniju sliku o Haskelu. Sama knjiga je podeljena na celine koje se nazivaju *lekcije* i u kojima se obrađuje po jedna tema.

Iako je Haskel jezik originalno zamišljen kao jezik za nastavu i istraživanja, Haskel je odavno prevazišao akademske namene i danas se uspešno koristi u industriji za realizaciju najrazličitijih projekata (kompajleri, web serveri, finansijski softver, desktop okruženja i desktop aplikacije, igrice, itd...) Ipak, broj Haskel projekata u industriji je i dalje neuporedivo manji od broja projekata koji su realizovani kroz bilo koji od gore spomenutih jezika. Stoga, postavlja se pitanje, vredi li učiti Haskel? Moj odgovor (kao i odgovor mnogih, rekao bi i svih, koji su savladali Haskel), je potvrdan. Haskel jezik, fundamentalno različit od proceduralnih i objektno orijentisanih jezika koji danas dominiraju, zahteva od nas da iz korena promenimo način na koji razmišljamo o programiranju. Ovakva promena tačke gledišta, iako prvobitno teška, daje nam širu sliku o programiranju i uči nas mnogim idejama koje možemo kasnije primeniti u drugim jezicima. Mnoge napredne konstrukcije u drugim jezicima imaju pandan u Haskelu koji je sasvim jednostavan i elegantan. Savladavanjem Haskela gotovo uvek savlađujemo i druge jezike.

Uzimajući u obzir rečeno, tokom pisanja ove knjige trudio sam se da dam što više konkretnih primera, i da pravim poređenja sa drugim jezicima. Duži primeri su numerisani i izdvojeni u posebne celine da ne bi remetili glavni tok lekcije. Pored primera, u knjizi su izdvojeni i numerisani zadaci. Neki od zadataka poseduju rešenja: rešenja takvih zadataka mogu se prikazati (i sakriti) pritiskom na trougao koji se nalazi levo od oznake zadatka. Neki zadaci su trivijalni, dok ima i zadataka koji predstavljaju prave male projekte. *Ako bih čitaocu mogao da dam jedan jedini savet, to je da što više zadataka reši koliko god neki zadaci delovali teško ili lako. Programiranje se ne može savladati prostim čitanjem knjige, već je neophodna upotreba naučenog.*

U kodu koji je dat u knjizi, skoro uvek se vrednosti i funkcije imenuju na srpskom jeziku (sve sa dijakriticima). Iako smatram da je ovo loša praksa za projekte (bez obzira da li su lični ili profesionalni), isto smatram da ovakvo imenovanje može pomoći početniku da lakše uoči koja imena su sistemska a koja je definisao korisnik.

Za kraj, naveo bih literaturu o Haskelu koja je korišćena pri izradi ove knjige kao i resurse koji mogu koristiti čitaocu za naredna istraživanja.

1. *Learn You a Haskell for Great Good!*, Miran Lipovača - besplatno dostupna, i verovatno najpoznatija knjiga o Haskelu. Neozbiljan, humoristički pristup čini je privlačnom za mnoge početnike (ali i predmetom mnogih kritika).
2. *Haskell Programming from First Principles*, Julie Moronuki, Chris Allen - opširna knjiga koja postepeno uvodi početnike u funkcionalno programiranje i detaljno opisuje Haskel jezik. Knjiga koju trenutno čitate, verovatno je najsličnija ovoj knjizi i pokriva nešto više od polovine tema obrađenih u *Haskell Programming from First Principles*.
3. *Haskell in Depth*, Vitaly Bragilevsky - predstavlja uvod u naprednije teme Haskel programiranja - rad na realnim projektima (*web* aplikacije) i korišćenje poznatih biblioteka.
4. *Functional Design and Architecture*, Alexander Granin - knjiga koja se bavi arhitekturom Haskel programa.

Postoje mnoge kvalitetne knjige o Haskelu koje nisu više pogodne za učenje jer je značajan deo biblioteka koje se koriste u njima promenjen. To se posebno odnosi na knjige koje su fokusirane na rešavanje praktičnih problema poput poznate i besplatno dostupne knjige *Real World Haskell* (Bryan O'Sullivan, Don Stewart, John Goerzen).

Osim knjiga postoje i mnogi drugi kvalitetni izvori informacija o Haskelu, kao što je *The Haskell Unfolder* serijal, *The Haskell Interlude* podkast, i *Haskellweekly* nedeljni pregled.

2. Prvi koraci

Naredna lekcija sadrži informacije neophodne za razumevanje svih ostalih lekcija. Možda će neke stvari biti nejasne (ili dosadne) prilikom prvog čitanja, ali važno je da budu poznate.

2.1. Haskel kompajler

Iako je tokom istorije Haskela razvijeno nekoliko kompajlera, danas se koristi samo *Glasgow Haskell Compiler* (GHC). Osim kompajlera, za ozbiljnije projekte potrebno je koristiti i jedan od menadžera paketa, *Stack* ili *Cabal*.

GHC instalacija korisniku pruža dva programa. Prvi program je sam kompajler `ghc` koji Haskel kôd kompajlira u izvršnu datoteku. Drugi program je `ghci` koji se koristi kao interaktivno okruženje za interpretaciju Haskel koda. Značajan deo ove knjige biće prikazan kroz rad u *GHCi* okruženju (detalji će biti objašnjeni uskoro).

Specijalizovana desktop razvojna okruženja za Haskel ne postoje, ali postoji *Haskell Language Server* (HLS) koji može da nadomesti ovaj "nedostatak"[1]. Tekstualni editori (poput VS Code-a ili vim-a) prošireni sa HLS-om mogu se koristiti kao razvojno okruženje.

Instalacija navedenih programa može se razlikovati u zavisnosti od sistema. Trenutno se za sve operativne sisteme (Linux, Windows, Mac) preporučuje GHCup kao alat za automatsku instalaciju svih potrebnih programa. Na sajtu *GHCup* projekta možete pronaći instrukcije za instalaciju[2]. Preporučljivo je da prilikom instalacije instalirate sve programe (*Stack*, *Cabal*, *HLS*).

Uz instaliran *GHC* potreban Vam je još samo i editor koda. Naravno, svaki editor koda može se koristiti u svrhu pisanja Haskel koda. Moja preporuka je VS Code sa instaliranim ekstenzijama Haskell i Haskell Syntax Highlighting.

Ipak, za prve korake u Haskelu nije čak ni potrebno instalirati kompajler lokalno, već je sasvim dovoljno koristiti neko od onlajn razvojnih okruženja, kao što je Replit.

2.2. Struktura Haskel koda

Haskel kôd se zapisuje u tekstualnim datotekama sa ekstenzijom `.hs`. Svaka `.hs` datoteka sastoji se od **definicija** kojima se **imenima**[3] pridružuju vrednosti nekog izraza. Na primer, sledeći blok koda je validan primer `.hs` datoteke:

```
dana_u_nedelji = 7

sati_u_danu = 24
```

U ovoj datoteci, imenu `dana_u_nedelji` odnosno `sati_u_danu` je pridružena vrednost `7` odnosno `24`.

Važno je znati da imena *moraju* početi malim slovom[4]. Na ostalim mestima u imenu je dozvoljeno koristiti velika i mala slova[5], brojeve kao i karaktere `_` i `'`.

Ime ne sme biti jedna od ključnih reči[6] `as`, `case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `hiding`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`, `qualified`.

Vrednost koja je pridružena jednom imenu ne može se više menjati u toku izvršavanja programa, te imena predstavljaju konstante. Samim tim, poredak definicija nije bitan.

---

[1] Ovo i nije toliki nedostatak, jer je danas sve zastupljenije da se razvoj programa vrši kroz tekstualni editor sa raznim proširenjima.

[2] Korisnici *Windows* operativnog sistema moraju da instaliraju Msys2 pre pokretanja *GHCup* instalacije.

[3] Koriste se još i izrazi *labele* i *identifikatori*

[4] Barem kada govorimo o imenima vrednosti (što uključuje i funkcije).

[5] Ovo uključuje i slova srpske latinice i ćirilice.

[6] Ovo su sve ključne reči Haskela, i u ovoj knjizi ćemo ih sve pojasniti.

Tip neke vrednosti predstavlja kolekciju kojoj ta vrednost pripada. Svaka vrednost pripada jednom i samo jednom tipu. Ako neka vrednost `v` pripada tipu `T` tada za tu vrednost kažemo da *poseduje tip* `T` ili da još *da je tipa* `T` i to označavamo sa `v :: T`. Za razliku od imena funkcije, ime tipa uvek počinje velikim slovom.

Na primer, vrednosti koje smo dodelili imenima `sati_u_danu` i `dana_u_nedelji` poseduju tip `Int`. Tip `Int` možemo da shvatimo kao kolekciju vrednosti koja sadrži sve cele brojeve koji se mogu predstaviti kroz jednu procesorsku reč[7].

U Haskel kodovima osim samih definicija, moguće je (i poželjno) navesti i tipove vrednosti. Deklaracija tipa ima oblik `ime :: Tip`. Tako bi prethodni primer sa punim deklaracijama tipova bio:

```
dana_u_nedelji :: Int
dana_u_nedelji = 7

sati_u_danu :: Int
sati_u_danu = 24
```

Iako nije neophodno da se deklaracije tipa nalaze neposredno pre definicije, u praksi se to pravilo uvek poštuje.

2.3. Izvršavanje Haskel koda

Prilikom definisanja imena u Haskel kodu, dozvoljeno je koristiti operatore i funkcije. Na primer, vrednosti tipa `Int` možemo sabirati, oduzimati i množiti koristeći operatore `+`, `-` i `*`. Koristeći aritmetičke operatore, vrednosti tipa `Int` i zagrade `(` i `)`, možemo graditi *aritmetičke izraze*:

```
dana_u_nedelji :: Int
dana_u_nedelji = 7

sati_u_danu :: Int
sati_u_danu = 24

dana_u_godini :: Int
dana_u_godini = 28 + 4 * 30 + 7 * 31

sati_u_nedelji :: Int
sati_u_nedelji = dana_u_nedelji * sati_u_danu
```

Definicija `sati_u_nedelji` koristi u sebi definicije `dana_u_nedelji` i `sati_u_danu`. Imena uvek predstavljaju vrednosti koje su im dodeljene prilikom definicije, i mogu se koristiti umesto tih konkretnih vrednosti.

Lista definicija nije sama po sebi mnogo korisna ako ne možemo utvrditi vrednosti izraza. Tako je u prethodnom primeru zgodno saznati koliko to sati ima u nedelji, a da bismo saznali tu vrednost potrebno je da izvršimo Haskel kôd. Postoje dva načina na koje je moguće izvršiti Haskel kôd:

1. *Kompilacijom koda.* Uz pomoć *GHC* kompajlera, moguće je kompajlirati kôd u izvršni fajl. U ovom slučaju u kodu mora biti definisano ime `main :: IO ()` koje služi kao početna tačka izvršavanja programa.
2. *Interpretacijom koda.* Uz pomoć programa *GHCi* moguće je učitati definicije u interaktivno okruženje i evaluirati te definicije.

Iz određenih razloga koji će biti pojašnjeni kasnije, definisanje imena `main` zahteva poznavanje nešto naprednijeg programiranja u Haskelu. Stoga će demonstracija kompilacije programa biti odložena za neku narednu lekciju, a do tada ćemo koristiti isključivo *GHCi* program.

[7] Na većini današnjih uređaja poput računara, telefona, i sl. procesorska reč je dužine 64 bita.

Rad u programu *GHCi* zahteva da prvo sačuvamo prethodni primer u datoteku sa ekstenzijom `.hs`. Na primer, neka je to datoteka `PrviProgram.hs`. Zatim je potrebno da otvorimo terminal i promenimo aktivni direktorijum tog terminala u direktorijum u kom se nalazi datoteka `PrviProgram.hs`. Nakon toga možemo pokrenuti `ghci` program[8]. Nakon pokretanja komande `ghci` pojaviće se tekst poput sledećeg:

```
C:\Users\User\Desktop>ghci
GHCi, version 9.0.2: https://www.haskell.org/ghc/  :? for help
ghci>
```

Izgled terminala nakon pokretanja *GHCi* programa na *Windows* operativnom sistemu.

Pokretanjem `ghci` programa otvara se novi prompt[9] u koji možemo da upišemo Haskel kôd ili *GHCi* naredbe. *GHCi* naredbe je moguće koristiti samo unutar *GHCi* okruženja, i imena tih naredbi počinju sa `:`. Jedna od tih naredbi je i `:load` uz pomoć koje možemo da učitamo sadržaj neke `.hs` datoteke.

Na primer, učitavanje datoteke `PrviProgram.hs` izgleda ovako:

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

Ako niste pokrenuli `ghci` iz direktorijuma u kom se nalazi datoteka `PrviProgram.hs` dobićete grešku `error: can't find file: PrviProgram.hs`. U tom slučaju, možete u naredni prompt navesti putanju do željene datoteke `:l putanja\do\PrviProgram.hs`

Nakon što je datoteka `PrviProgram.hs` učitana, možemo koristiti definisane vrednosti. Ako upišemo `dana_u_nedelji` i pritisnemo `Enter`, vrednost dodeljena ovom imenu će biti ispisana, i nakon toga će se pojaviti prompt spreman za novi unos:

```
ghci> dana_u_nedelji
7
ghci>
```

Mnogo bitnije je ono što dobijamo nakon unosa imena `sati_u_nedelji`. U ovom slučaju, *GHCi* će interpretirati Haskel kôd `dana_u_nedelji * sati_u_nedelji` odnosno iskoristiće definicije navedenih vrednosti a zatim i izvršiti naznačeno množenje. Rezultat će biti ispisan:

```
ghci> sati_u_nedelji
168
```

Možda opisani postupak deluje trivijalno, ali predstavlja suštinu rada u interaktivnom okruženju[10].

Ako želimo, u datoteku `PrviProgram.hs` možemo dodati još definicija poput `sekundi_u_danu = 24 * 60 * 60` itd... Da bismo mogli da koristimo nove definicije, neophodno je da sačuvamo izmene datoteke `PrviProgram.hs` a zatim da ponovo učitamo datoteku u *GHCi*. To možemo da učinimo ponovo sa naredbom `:load PrviProgram.hs` ili sa naredbom `:reload` koja ponovo učitava sve datoteke koje su do tada bile učitane:

[8] Ako koristite *repl.it* web okruženje, dovoljno je da samo pokrenete komandu `ghci` u terminalu koji se nalazi sa desne strane editora koda. Editor koda koji se nalazi na levoj strani prikazuje kôd koji se nalazi u datoteci `main.hs`.

[9] Prompt je jedan ili više karaktera kojim se označava mesto korisničkog unosa. *GHCi* prompt je `ghci>`.

[10] *GHCi* je primer *REPL* (Read-evaluate-print loop) programa čiji se rad sastoji od uzimanja korisničkih komandi, izvršavanja tih komandi i ispisivanja rezultata nazad u komandnu liniju, nakon čega program ponovo čeka korisnički unos.

```
ghci> :reload
Ok, one module loaded.
ghci> sekundi_u_danu
86400
```

Naredba `:reload` je pogotovo korisna kada se u *GHCi* učita više datoteka odjednom. O tome nismo govorili, ali je moguće i to učiniti uz opreznost da se imena ne ponavljaju.

U *GHCi* prompt je moguće direktno unositi i Haskel izraze koji će se odmah evaluirati:

```
ghci> 10
10
ghci> 10 * 10 + 1
101
```

Prvi korisnički unos nije mogao biti dodatno evaluiran, te je stoga ispisan isti nazad. Drugi unos je sveden na jedinstven broj i taj broj je ispisan nazad.

Zapravo u *GHCi* prompt je moguće uneti i definicije. Ove definicije će važiti samo dok je trenutna sesija aktivna i neće biti upisane ni u jednu datoteku. Važno je znati da je u *GHCi* okruženju dozvoljeno predefinisati već postojeće definicije. Tom prilikom, ime će predstavljati onaj izraz koji mu je dodeljen poslednjom definicijom, a sve prethodne definicija će biti izgubljena. Komandom `:reload` sve definicije napisane u *GHCi* okruženju se odbacuju.

```
ghci> a = 100 + 10
ghci> a
110
ghci> a = 200 + 10
ghci> a
210
ghci> :reload
Ok, one module loaded.
ghci> a
<interactive>:1:1: error: Variable not in scope: a
```

Prilikom definisanja nekog imena, vrednost dodeljenog izraza se neće ispisati u terminal. Tek unosom imena u prompt ispisujemo vrednost.

Za izlazak iz *GHCi* programa, koristi se naredba `:quit`.

2.4. *Let ... in*

Prilikom pisanja izraza neretko se javlja potreba da neke vrednosti privremeno dodelimo novom imenu. Upravo nam to omogućuje *let in* konstrukcija.

Na primer, želeli bismo da napišemo izraz kojim računamo površinu kvadra visine `10` širine `5` i dužine `3`. Koristeći formulu $P = 2(ab + bc + ca)$ dobijamo kôd:

```
površina :: Int
površina = 2*(10*5 + 5*3 + 3*10)
```

Navedena definicija je malo nepregledna. Takođe, ako želimo da izračunamo zapreminu nekog drugog kvadra, svaku od dimenzija moramo da promenimo na dva mesta što predstavlja potencijalni izvor greške. Zbog toga ćemo iskoristiti *let in* sintaksu da bismo lokalno definisali imena `a`, `b` i `c`[11]:

[11] Naravno, moguće je i globalno definisati imena `a`, `b` i `c`, ali time nepotrebno rezervišemo ta imena.

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

Dakle, sa *let … in* konstrukcijom moguće je lokalno definisati jedno ili više imena koja će biti dostupna samo u nekom izrazu. Opšti oblik *let … in* izraza sastoji se od niza definicija koje su razdvojene znakom `;`, i nakon čega sledi i izraz u kom želimo da te definicije budu dostupne:

```
let ime1 = izraz1; …; imeN = izrazN in izraz
```

Naravno, imena definisana u *let … in* konstrukciji dostupna su samo unutar te *let … in* konstrukcije.

Napominjemo da *let … in* predstavlja jedan *izraz*, i da *let … in* sintaksa nije nužno vezana za definicije. Zbog toga *GHCi* uspešno pronalazi vrednost izraza `let a = 2 in a + 3`:

```
ghci> let a = 2 in a + 3
5
```

Samim tim, moguće je ugnježdavati *let … in* izraze:

```
ghci> let a = 2 in (let b = 3 in a + b)
5
```

Navedeni Haskel izraz je ekvivalentan izrazu `let a = 2; b = 3 in a + b`.

2.5. *Where*

Sa *where* sintaksom takođe je moguće uvesti lokalne definicije kao i sa `let … in` sintaksom. Za razliku od *let in* konstrukcije koja predstavlja izraz, *where* sintaksa se isključivo koristi uz definicije. Opšti oblik *where* sintakse je:

```
ime = izraz where ime1 = izraz1; …; imeN = izrazN
```

Primer sa površinom kvadra iz prethodne sekcije bi mogao i ovako da se napiše:

```
površina :: Int
površina = 2*(a*b + b*c + c*a) where a = 10; b = 5; c = 3
```

Za razliku *let in* sintakse, sintaksa *where* se ne može koristiti van definicija:

```
ghci> broj = a + 3 where a = 3
ghci> broj
5
ghci> a + 3 where a = 3
<interactive>:1:7: error: parse error on input ‘where’
```

Kôd `a + 3 where a = 3` nije validan izraz, zbog čega je došlo do sintaksne greške.

Važno je znati da je u okviru *let in* ili *where* sintakse moguće lokalno predefinisati neka već definisana imena. Zbog toga, u narednom primeru, ime `b` predstavlja vrednost `11` a ne `21`:

```
a :: Int
a = 20
```

```
b :: Int
b = a + 1 where a = 10
```

```
ghci> b
11
```

2.6. Prelom i nazubljivanje

Često nije zgodno pisati ceo izraz u jednoj liniji. U tom slučaju, možemo prelomiti linije poštujući pravila o nazubljivanju[12] Haskel koda. Kako su opisi tih pravila dosta tehnički, ovde navodimo neformalna pravila koja dobro sažimaju ta tehnička pravila:

1. Nazubljivanje Haskel koda se vrši isključivo sa razmacima.
2. Početak definicije ne sme da bude nazubljen.
3. Svaki izraz se može prelomiti na proizvoljnom razmaku. Ostatak prelomljenog izraza mora da bude nazubljen više nego linija u kojoj počinje izraz koji sadrži taj kod.
4. Niz definicija u *let … in* i *where* sintaksi može se prelomiti u više linija. Tada, početak svake definicije mora da bude postavljen tačno ispod početka prve definicije.

Što se tiče prvog pravila, ono se odnosi na polemiku između razmaka i tabova. Iako će kompajler često iskompajlirati kôd koji sadrži tabove, pri kompajliranju će se javiti upozorenja.

Drugo pravilo je takođe jednostavno, i njim se zabranjuje nazubljivanje samog početka neke definicije:

```
dobraDefinicija = 1

  lošaDefinicija = 1
```

Treće pravilo je pomalo neobično za jezike koji su osetljivi na nazubljivanje[13]. Po ovom pravilu svaki izraz je moguće prelomiti. Na primer, sledeća tri koda su primeri validnog preloma istog izraza:

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

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

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

Broj razmaka nije bitan pri nazubljivanju. Ako neka linija sadrži barem jedan više razmak na početku nego druga, onda ćemo smatrati da je prva linija više nazubljena od druge.

Naredni kôd *nije* pravilno formatiran, jer linija `+ 1` nije dodatno nazubljena u odnosu na početak `a =`:

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

Kako i *let … in* sintaksa predstavlja izraz, i taj izraz možemo prelomiti:

[12] Nazubljivanje je dodavanje praznog prostora na početak linije.
[13] *Python* je primer jednog takvog jezika. Sa druge strane *C* jezik je u potpunosti neosetljiv na nazubljivanje koda.

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

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

Oba primera su pravilno nazubljena

Takođe, i *where* sintaksu je moguće prelomiti:

```
površina :: Int
površina = 2*(a*b + b*c + c*a)
    where a = 10; b = 5; c = 3
```

```
površina :: Int
površina = 2*(a*b + b*c + c*a)
    where
        a = 10; b = 5; c = 3
```

Oba primera su dobro nazubljena

Četvrto pravilo se odnosi na niz definicija u *let ... in* i *where* sintaksama. Po ovom pravilu, niz je moguće prelomiti u više linija, pri čemu se znaci `;` mogu izostaviti. Pri prelomu sve definicije se moraju postaviti tačno jedna ispod druge. Naredni kodovi su dobro nazubljeni:

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

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

```
površina :: Int
površina = 2*(a*b + b*c + c*a)
    where
        a = 10
        b = 5
        c = 3
```

```
površina :: Int
površina = 2*(a*b + b*c + c*a)
    where a = 10
```

```
        b = 5
        c = 3
```

Zapravo, ovakvi blokovi definicija se mogu shvatiti kao male `.hs` datoteke. I kao što u `.hs` datoteci svaka definicija mora da počne od leve ivice, tako u bloku definicija sve definicije moraju da počnu od iste kolone[14] (a ta kolona je uspostavljena početkom prve definicije).

```
w :: Int
w = 2 * v
    where
        v :: Int
        v = 10 * 3
```

Osim definicija, u bloku je moguće navesti i deklaracije tipova, ali se to retko radi u praksi.

2.7. Komentari

Kao i svi drugi programski jezici, Haskel podržava pisanje komentara odnosno delova koda koji ne utiču na izvršavanje. U Haskelu postoje dve vrste komentara:

1. *Linijski komentari* jesu komentari koji se nalaze samo u jednoj liniji koda. Linijski komentari počinju sa `--` i obuhvataju (desni) ostatak linije. Linijski komentari se mogu postaviti i u linijama u kojima se nalazi kôd.
2. *Višelinijski komentari* jesu komentari koji se protežu kroz jednu ili više linija koda. Ovi komentari se nalaze između `{-` i `-}`

```
-- ovo je komentar

a = 2 * 10 + 1 -- i ovo je komentar

{-
Ovo
je
takođe
komentar
-}
```

Primer komentara u `.hs` datoteci.

[14] Ovo objašnjava četvrto pravilo.

3. Tipovi

Tip predstavlja kolekciju *vrednosti*. Svaka vrednost u Haskel jeziku pripada jednom i samo jednom tipu. Činjenicu da neka vrednost `v` pripada tipu `T` označavamo sa

```
v :: T.
```

Izraz `v :: T` često čitamo kao "`v` poseduje tip `T`".

Haskel poseduje **statički sistem tipova** što znači da su tipovi svih izraza poznati prilikom kompilacije (ili interpretacije). Time se otklanjaju mnoge greške koje bi mogle zaustaviti program prilikom izvršavanja ili dovele do pogrešnog rezultata[15]. Jezici poput *Python*-a ili *JavaScript*-a nemaju statički sistem tipova, već *dinamički* sistem tipova. U takvim jezicima, tipovi izraza se određuju prilikom izvršavanja programa što ostavlja veliku mogućnost za pojavljivanje *bagova*.

Još jedna karakteristika koja odlikuje sistem tipova Haskela je **zaključivanje tipova**[16]. Dok je u jezicima poput C-a ili Jave neophodno navesti tip svake promenljive ili funkcije, Haskel kompajler (skoro) uvek može zaključiti tip izraza bez potrebe da programer eksplicitno navede tip[17]. Po mogućnosti zaključivanja tipova Haskel se dugo razlikovao od većine ostalih jezika sa statičkim sistemom tipova[18].

Tipovi čine značajan deo Haskel jezika, i neophodno je dobro razumeti ovaj sistem. U nastavku ove lekcije upoznaćemo se sa nekim osnovnim tipovima u Haskelu, kao i operacijama i funkcijama koje se mogu koristiti sa ovim tipovima.

3.1. Tip *Bool*

Tip `Bool` predstavlja kolekciju koja sadrži dve logičke vrednosti: `True` i `False`[19].

Šematski prikaz tipa `Bool`.

Nad logičkim vrednostima moguće je vršiti naredne *logičke operatore* (operacije):

1. *Konjunkcija* (*i*) `p && q` dve logičke vrednosti `p` i `q` je logička vrednost koja ima vrednost `True` ako logičke vrednosti `p` i `q` imaju vrednost `True`.
2. *Disjunkcija* (*ili*) `p || q` dve logičke vrednosti `p` i `q` je logička vrednost koja ima vrednost `True` ako barem jedna od vrednosti `p` i `q` ima vrednost `True`.

Koristeći navedene operatore i zagrade možemo kreirati logičke izraze proizvoljne složenosti.

[15] Na primer, *GHC* neće iskompajlirati kôd u kom se broj oduzima od niske, jer takva operacija nema smisla.

[16] eng. *type inference*

[17] Zaključivanje tipova u Haskelu nam je omogućilo da u prethodnoj sekciji napišemo Haskel izraze bez navođenja tipova.

[18] Mnogi statički tipizirani jezici, poput C++ i Jave, su u novijim verzijama dobili neke mogućnosti zaključivanje tipova. Jezik C je primer statički tipiziranog jezika koji nema mogućnost zaključivanja tipova.

[19] Obratite pažnju da imena logičkih vrednosti počinju velikim slovom.

```
ghci> True || False
True
ghci> ((True && False) || False) && (True || True)
False
```

Kao i aritmetičke izraze, *GHCi* će i logičke izraze pokušati da *svede* na jednu vrednost.

Uz navedene operatore koristimo i *funkciju* `not` koja negira logičku vrednost[20]. Ovde je dobro mesto da kažemo par reči o funkcijama u Haskelu, jer funkcije predstavljaju osnovni koncept ovog jezika.

Funkcije su vrednosti koje od vrednosti "prave" nove vrednosti. Na primer, imenom `not` označena je funkcija koja negira logičku vrednost koja joj je prosleđena, tj. od `True` "pravi" u `False` a od `False` "pravi" `True`. Da bismo "pozvali" funkciju `not` potrebno je da nakon imena funkcije napišemo logičku vrednost koju želimo da negiramo. Za razliku od drugih jezika, u Haskelu ime funkcije i argument razdvajamo razmakom, a ne zagradom:

```
ghci> not True
False
ghci> a = True && False
ghci> not a
True
```

Funkciju `not` možemo pozivati nad logičkim vrednostima ili nad imenima kojima su dodeljene logičke vrednosti (u ovom primeru to je ime `a`).

Funkcija `not` vraća novu logičku vrednost, te stoga tu vrednost možemo kombinovati ponovo u složenijim logičkim izrazima:

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

Zagrade postavljamo oko izraza da bismo uspostavili poredak kojim je potrebno evaluirati izraz. U prethodnom primeru postavili smo zagrade oko logičkog izraza `not True`, da bismo bili sigurni da se celokupan izraz neće protumačiti kao `not (True || False)`[21]. Ali kao što u aritmetici imamo konvenciju o prioritetu računskih operacija, tako i u Haskelu imamo niz pravila koja se tiču prioriteta raznih operatora i poziva funkcija. Za sada je važno znati da primena funkcije na vrednost ima *najveći* prioritet. Prema tome, izraz `not True || False` se tumači kao `(not True) || False`, baš kao što se aritmetički izraz $5 \cdot 4 + 7$ tumači kao $(5 \cdot 4) + 7$ a ne kao $5 \cdot (4 + 7)$.

Logičke vrednosti možemo da poredimo sa operatorima `==` i `/=`, koji utvrđuju da li su vrednosti iste odnosno različite. Rezultati ovih poređenja su takođe logičke vrednosti:

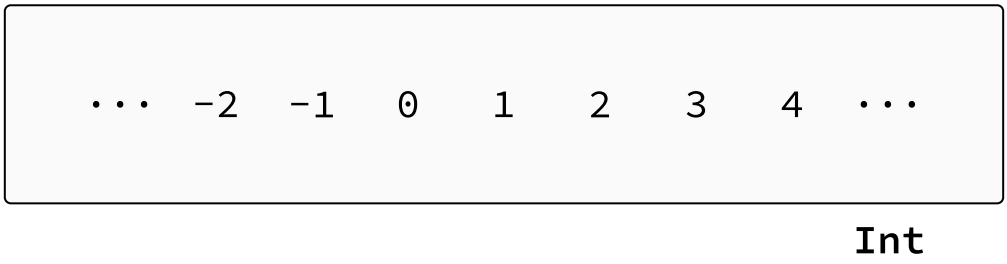
```
ghci> p = True
ghci> q = False
ghci> p == p
True
ghci> p == q
False
ghci> p /= q
True
```

[20] Zapravo, i operatori poput `&&` i `||` su funkcije zapisane u takozvanom *infiksnom zapisu*. Više o tome kasnije.

[21] Na prvi pogled izraz `not True || False` bi mogao da se shvati i kao `(not True) || False` ali i kao `not (True || False)`.

3.2. Tipovi *Int* i *Integer*

Tip `Int` predstavlja kolekciju celobrojnih vrednosti koje se mogu predstaviti u procesorskoj reči, odnosno u 32 ili 64 bita[22].



Nad vrednostima tipa `Int` moguće je vršiti naredne operacije i funkcije:

3.2.1. `+` i `*`

Sa sabiranjem i množenjem brojeva smo se već upoznali u prethodnoj lekciji. Ovi operatori se ponašaju isto kao i u svim drugim programskim jezicima ili u svakodnevnoj aritmetici.

3.2.2. `-`

Oduzimanje brojeva je takođe poznato iz svakodnevnog života i ostalih programskih jezika. Ali potrebno je naglasiti da se sa znakom `-` može negirati broj. Znak `-` ima nizak prioritet, te je često potrebno koristiti zagrade oko izraza koji se negira:

```
ghci> 3 * -2
<interactive>:1:1: error:
precedence parsing error
cannot mix ‘*’ [infixl 7] and prefix `-' [infixl 6] in the same infix expression
ghci> 3 * (-2)
-6
```

Izraz `3 * -2` dovodi do greške u parsiranju, te je potrebno staviti zagrade oko broja koji se negira.

3.2.3. `div`

Za deljenje celobrojnih vrednosti, koristi se funkcija `div` sa dva parametra. Prvi parametar je deljenik (broj koji se deli) a drugi je delilac (broj kojim se deli). Rezultat deljenja je *celobrojna* vrednost odnosno vrednost tipa `Int`.

I pri radu sa funkcijama više parametra, argumenti se razdvajaju razmacima[23]:

```
ghci> div 7 2
3
```

Rezultat je 3 jer je $3 \cdot 2 + 1 = 7$.

Ponovo napominjemo da primena funkcije na argument ima najveći prioritet. Stoga se kôd `div 7 2 * 2` interpretira kao `(div 7 2) * 2` a ne kao `div 7 (2 * 2)`.

3.2.4. `mod`

Funkcija `mod` vraća ostatak pri celobrojnom deljenju prvog argumenta sa drugim.

[22] Specifikacija Haskel jezika garantuje da je vrednost tipa `Int` predstavljena sa barem 30 bitova. Tačan broj bitova koji se koristi za reprezentovanje vrednosti tipa `Int` zavisi od arhitekture na kojoj se izvršava program.

[23] Česta početnička greška je da se sintaksa pozivanja funkcija iz drugih jezika prenese u Haskel, odnosno da se argumenti funkcije postave u zagrade i razdvoje zarezom. Zato početnici često pokušavaju da podele dva broja sa `div (5, 7)` što dovodi do tipske greške. Uverite se i sami.

```
ghci> mod 7 3
1
```

Za funkcije `div` i `mod` i proizvoljne cele brojeve `x` i `y` različite od nule važi veza

$$(\texttt{div x y})\texttt{*y} + (\texttt{mod x y}) = \texttt{x}.$$

3.2.5. ^

Operatorom ^ je moguće stepenovati celobrojnu vrednost na prirodan stepen. Na primer `7^2` se evaluira u `49`.

Ako pokušamo da sa ^ stepenujemo broj na negativan stepen dobićemo izuzetak[24]:

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

Rezultat dizanja prirodnog broja na negativan stepen pozitivan racionalan broj manji od 1, a takav broj se ne može predstaviti tipom `Int`

Izuzetak je posebna vrednost koja označava da je došlo do neke greške prilikom izvršavanja programa. Pojava izuzetka dovodi do prekida izvršavanja (interpretacije) programa. Mnogo više o izuzecima biće rečeno kasnije.

3.2.6. Poređenja

Vrednosti tipa `Int` takođe možemo porediti sa operatorima `==` i `/=`. Ali možemo koristiti i operatore `<`, `<=`, `>` i `>=` koji upoređuju brojeve po veličini. Rezultat svih navedenih operacija je logička vrednost:

```
ghci> 6 == 5
False
ghci> 6 == 3 * 2
True
ghci> 2 < 3
True
ghci> 2 >= 3
False
```

U Haskelu nije dozvoljeno porediti tipove različitog tipa sa operatorima `==`, `/=`, `<`, `<=`, `>` i `>=`. U jezicima poput *JavaScript*-a, poređenje poput `True == 1` bi bilo validno (i tačno), dok u Haskelu takvo poređenje dovodi do greške:

```
ghci> True == 1
<interactive>:1:9: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the second argument of ‘(==)’, namely ‘1’
      In the expression: True == 1
      In an equation for ‘it’: it = True == 1
```

3.2.7. Tip *Integer*

`Integer` je takođe tip koji predstavlja celobrojne vrednosti. I nad ovim vrednostima ovog tipa mogu se koristiti funkcije `+`, `-`, `*`, `div`, `mod`, `rem`, `^`, `==`, `/=`, `<`, `<=`, `>` i `>=`. Razlika između tipova `Int` i `Integer` je u tome

[24] Pokušajte da iz neke `.hs` datoteke učitate definiciju `a = 7^(-2)`. Primetićete da će se datoteka uspešno učitati. Tek kada kroz *GHCi* prompt potražite vrednost od `a` doći će do izuzetka. Ovo nam govori da se Haskel kôd ne izvršava pri učitavanju datoteke već prilikom evaluacije izraza.

što su vrednosti tipa `Integer` neograničene[25], dok su vrednosti tipa `Int` ograničene dužinom procesorske reči.

Da bi demonstrirali razliku možemo napraviti mali eksperiment. U jednoj `.hs` datoteci definišimo jednu `Int` i jednu `Integer` konstantu:

```
a :: Int
a = 7

b :: Integer
b = 7
```

Obe vrednosti predstavljaju broj 7, ali poseduju različit tip.

Nakon učitavanja gorenavedenog koda, možemo stepenovati definisane vrednosti na velike stepene i tako prekoračiti ograničenje koje nameće `Int` tip. Uvidećemo da su rezultati različiti:

```
ghci> a^70
254007274765394321
ghci> b^70
1435036016098684342856030763566710717400773837392460666639249
```

Broj 254007274765394321 je velik ali nije jednak 7^{70}. Evaluacija koda `b^70` daje tačnu vrednost.

Pri radu sa tipom `Integer` trebalo bi imati na umu da su aritmetičke operacije nad ovim tipom značajno sporije nego nad `Int` tipom. Razlog tome je što aritmetičke operacije sa vrednostima tipa `Int` odgovaraju procesorskim instrukcijama, što nije slučaj i sa vrednostima tipa `Integer`.

3.3. Tipovi *Float* i *Double*

Za reprezentovanje realnih brojeva koriste se tipovi `Float` i `Double` koji realan broj predstavljaju u skladu sa *IEEE 754* standardom. Jedina razlika između ova dva tipa je u tome što `Float` koristi jednostruku tačnost (32 bita) a `Double` dvostruku tačnost (64 bita). Kako moderne arhitekture računara podjednako dobro podržavaju računske operacije u jednostrukoj i dvostrukoj tačnosti, preporučeno je da se uvek koristi `Double`.

Sa tipovima `Float` i `Double` moguće je koristiti operatore `+`, `-`, `*`, `^`, `==`, `/=`, `<`, `<=`, `>` i `>=`. Za deljenje realnih brojeva koristi se operator `/`. Uz operator `^`, za stepenovanje se može koristiti i operator `**` koji dozvoljava da stepen bude proizvoljan realan broj[26]

Takođe, Haskel ima ugrađene funkcije poput `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `sqrt` itd... Dostupna je takođe i Arhimedova konstanta π pod imenom `pi`.

```
ghci> r = 10
ghci> pi * r**2
314.1592653589793
```

3.4. Liste

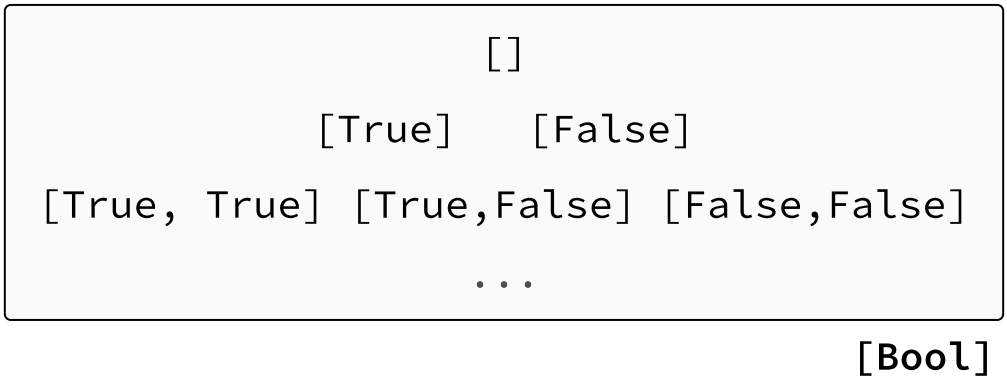
U Haskelu **liste** su strukture podataka koje predstavljaju uređenu kolekciju elemenata nekog tipa. U jednoj listi se ne mogu naći dve vrednosti različitog tipa.[27]

[25] Naravno, ništa na računaru nije neograničeno pa tako ni tip `Integer` ne može da predstavi proizvoljno veliki broj. Ograničenja ovog tipa određena su raspoloživom radnom memorijom računara.

[26] Iako pri korišćenju operatora `**` neće nikad doći do izuzetka, rezultat može biti `NaN`.

[27] Zbog toga za liste kažemo da su *homogene* strukture podataka.

U kodu, lista se konstruiše navođenjem vrednosti između uglastih zagrada, međusobno razdvojenih zarezima. Na primer, lista od prva četiri prirodna broja može da se definiše kao `[0,1,2,3]`. Razmaci oko elemenata i zagrada nisu od bilo kakvog značaja.[28] **Prazna lista** se zapisuje kao `[]`.



Tip `[Bool]` sadrži beskonačno mnogo vrednosti, čak i ako tip `Bool` sadrži samo dve.

Za označavanje tipa liste koristi se specijalna sintaksa. Ako je data lista elemenata tipa `A` tada ta lista ima tip koji se označava sa `[A]`[29]. Na primer, lista elemenata tipa `Int` poseduje tip `[Int]`.

```
maliProstiBrojevi :: [Int]
maliProstiBrojevi = [2,3,5,7,11,13,17,19]

logičkeVrednosti :: [Bool]
logičkeVrednosti = [True,False]

važneKonstante :: [Double]
važneKonstante = [3.14159,2.71828,0.57721]
```

Definicije tri liste različitog tipa.

U specijalnom slučaju, možemo definisati i listu lista, i listu lista lista, itd...

```
a :: [[Int]]
a = [[1], [2,3], [4,5,6]]

b :: [[[Int]]]
b = [a, [[100], [200]], []]
```

Lista `a` je lista lista celih brojeva. Tip ove liste je `[[Int]]`. Primetimo, da se u listi tipa `[[Int]]` ne može naći element `5` (jer je to vrednost tipa `Int`), ali može element `[5]` koji je tipa `[Int]`.

> **Zadatak 1.** Koliko elemenata ima lista `[[]]`?
>
> *Rešenje.* Lista `[[]]` ima jedan element. Taj element je prazna lista `[]`.

Liste možemo spajati uz pomoć operatora `++`. Nadovezivanjem prazne liste `[]` na početak ili kraj liste, ta lista se ne menja. Naravno, moguće je spajati samo liste istog tipa.

```
ghci> [1,2,3,4] ++ [10,11,12]
[1,2,3,4,10,11,12]
```

[28] Prema tome, navedeni primer može da se zapiše i kao `[0, 1,2 ,3 ]` itd..
[29] Kad god govorimo o nekom opštem tipu, a ne nekom konkretnom tipu, koristićemo ime poput `A`, `B`, `T`, itd...

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

```
ghci> [1,2,3] ++ [True]
<interactive>:1:2: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the expression: 1
      In the first argument of ‘(++)’, namely ‘[1,2,3]’
      In the expression: [1,2,3] ++ [True]
```

Prilikom spajanja lista različitog tipa dolazi do tipske greške.

Za nadovezivanje elemenata na početak liste može se koristi i operator `:`. Sa leve strane operatora `:` potrebno je navesti vrednost tipa `A`, a sa desne strane je potrebno navesti listu elemenata tipa `A`.

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

Dva načina nadovezivanja jednog elementa na početak liste.

```
ghci> [1] : [2,3,4,5]
<interactive>:1:1: error:
    • Non type-variable argument in the constraint: Num [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Num a, Num [a]) => [[a]]

ghci> 1 ++ [2,3,4,5]
<interactive>:2:1: error:
    • Non type-variable argument in the constraint: Num [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Num a, Num [a]) => [a]
```

Česta greška je da se element nadovezuje na početak sa operatorom `++` ili da se liste spajaju pomoću operatora `:`.

Liste predstavljaju strukturu podataka koja se najčešće koristi u Haskelu. Zbog toga postoje mnoge funkcije koje uzimaju ili vraćaju liste. Sa mnogim takvim funkcijama upoznaćemo se u lekciji *Liste*. Za sada navodimo samo naredne funkcije:

1. `length` vraća dužinu liste.
2. `head` vraća prvi element liste ako taj element postoji. Pozivanjem ove funkcije nad praznom listom dolazi do izuzetka.
3. `last` vraća poslednji element liste ako taj element postoji. Pozivanjem ove funkcije nad praznom listom dolazi do izuzetka.
4. `init` vraća sve elemente liste osim poslednjeg. Pozivanjem ove funkcije nad praznom listom dolazi do izuzetka.
5. `tail` vraća sve elemente liste osim prvog. Pozivanjem ove funkcije nad praznom listom dolazi do izuzetka.
6. `reverse` vraća prosleđenu listu u obrnutom smeru.
7. `elem` je funkcija od dva parametra koja proverava da li se prvi argument nalazi u listi prosleđenoj kao drugi argument. Rezultat provere je logička vrednost.

8. `null` proverava da li je lista prazna. Rezultat provere je logička vrednost.

```
ghci> mojaLista = [10,20,30,40,50,60]
ghci> length mojaLista
6
ghci> head mojaLista
10
ghci> last mojaLista
60
ghci> init mojaLista
[10,20,30,40,50]
ghci> tail mojaLista
[20,30,40,50,60]
ghci> reverse mojaLista
[60,50,40,30,20,10]
ghci> elem 10 mojaLista
True
ghci> elem 100 mojaLista
False
ghci> null mojaLista
False
ghci> null []
True
```

Sa operatorom `!!` možemo pristupiti m-tom elementu liste. Sa leve strane operatora se navodi lista, a sa desne strane vrednost tipa `Int`. Elementi su indeksirani od nule. Pokušaj pristupanja indeksu koji je veći ili jednak dužini liste dovodi do izuzetka.

```
ghci> [True, False, True, False, True] !! 2
True
```

```
ghci> [True, False, True, False, True] !! 7
*** Exception: Prelude.!!: index too large
```

Kad god imamo tip čije se vrednosti mogu upoređivati sa `==`, `/=`, `<`, `<=`, `>` i `>=` tada se i liste elemenata tog tipa mogu porediti sa istim operatorima. Ova poređenja se vrše leksikografski. Specijalno, liste su jednake ako i samo ako sadrže iste vrednosti poređane istim redosledom.

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

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

Liste se porede član po član. Pošto su prvi elementi obe liste jednaki, poređenje prelazi na druge elemente u listi. Kako je `2 < 3` zaključujemo da je leva lista strogo manja.

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

Pri ovom poređenju, prvo su upoređeni prvi elementi obe liste. Kako su oni jednaki, upoređeni su drugi elementi druge liste, pa je poređenje nastavljeno po trećim elementima. Međutim, jedna od lista nema treći element, te se ona smatra strogo manjom.

```
ghci> [1, 2, 3, 100] < [1, 2, 4, 5]
True
```

I pri ovom poređenju, prvo su upoređeni prvi elementi a zatim i drugi elementi. Kako su i drugi elementi jednaki, poređenje je nastavljeno po trećim elementima. Kako je `3 < 4`, leva lista se smatra manjom. Primetimo da je leva lista manja od desne čak iako je četvrti član leve liste značajno veći od četvrtog člana desne liste. Dakle, prilikom poređenja, razmatra se samo prvo mesto na kom se liste razlikuju, dok naredne vrednosti ne utiču na poredak.

```
ghci>[] < [1]
True
```

Prazna lista je strogo manja od bilo koje druge liste.

3.5. Karakteri i niske

Vrednosti tipa `Char` su karakteri kodirani *Unicode* standardom. U samom kodu, karakteri se navode između jednostrukih navodnika:

```
ghci> mojKarakter = 'ш'
ghci> mojKarakter
'ш'
```

Funkcija `fromEnum` konvertuje karakter u vrednost tipa `Int` koja predstavlja poziciju tog karaktera u *Unicode* kodnoj tabeli

```
ghci> fromEnum 'A'
65
```

Nekome ko ne poznaje *Unicode* ili *ASCII* može delovati neobično da se `'A'` nalazi tek na 65 mestu. To je zato što se na početku kodne tabele nalaze specijalni karakteri, zatim cifre, pa tek onda velika i mala slova latinice.

Karaktere, kao i brojeve, možemo porediti. To poređenje je takođe ustanovljeno pozicijom u *Unicode* kodnoj tabeli. Stoga imamo

```
ghci> 'a' < 'b'
True
ghci> 'a' < 'B'
False
ghci> 'a' < 'ш'
True
```

Terminom **niske** nazivamo liste karaktera. Niske se mogu koristiti za prezentovanje tekstualnih vrednosti[30]. Na primer, pozdrav *Hello!* se u Haskelu predstavlja kao lista `['H', 'e', 'l', 'l', 'o', '!']`. Srećom, niske se mogu zapisati i prostim navođenjem karaktera između dvostrukih navodnika potpuno

[30] Postoje još neki tipovi koji mogu prezentovati tekstualne vrednosti, ali ih nećemo spominjati u ovoj knjizi.

analogno sintaksi u ostalim programskim jezicima. Tako se niska *Hello!* može konstruisati i kao literal[31] `"Hello!"`:

```
ghci> "Hello" == ['H', 'e', 'l', 'l', 'o']
True
```

Poređenjem uviđamo da dva navedena načina konstrukcije niski daju istu vrednost.

Kako su niske u suštini liste, sve funkcije nad listama možemo iskoristiti i za niske.

```
ghci> niska = "Hello" ++ "!"
ghci> niska
"Hello!"
ghci> length niska
6
ghci> head niska
'H'
ghci> last niska
'!'
ghci> init niska
"Hello"
ghci> tail niska
"ello!"
ghci> reverse niska
"!olleH"
ghci> null ""
True
```

Obratite pažnju na razliku između niske i pojedinačnog karaktera. Na primer funkcija `head` vraća prvi element liste, a u ovom primeru to je karakter `'H'` (a ne niska `"H"`). Karakteri se uvek navode između jednostrukih navodnika, a niske između dvostrukih.

Mnogi tipovi u Haskelu (ali svakako ne svi), dozvoljavaju da se nad njima pozove funkcija `show`. Ova funkcija uzima vrednost i vraća nisku koja prezentuje tu vrednost. Na primer:

```
ghci> show 2
"2"
ghci> show True
"True"
```

Niska `"2"` i vrednost `2` nisu isto, kao što ni logička vrednost `True` i niska `"True"` nisu isto, itd...

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

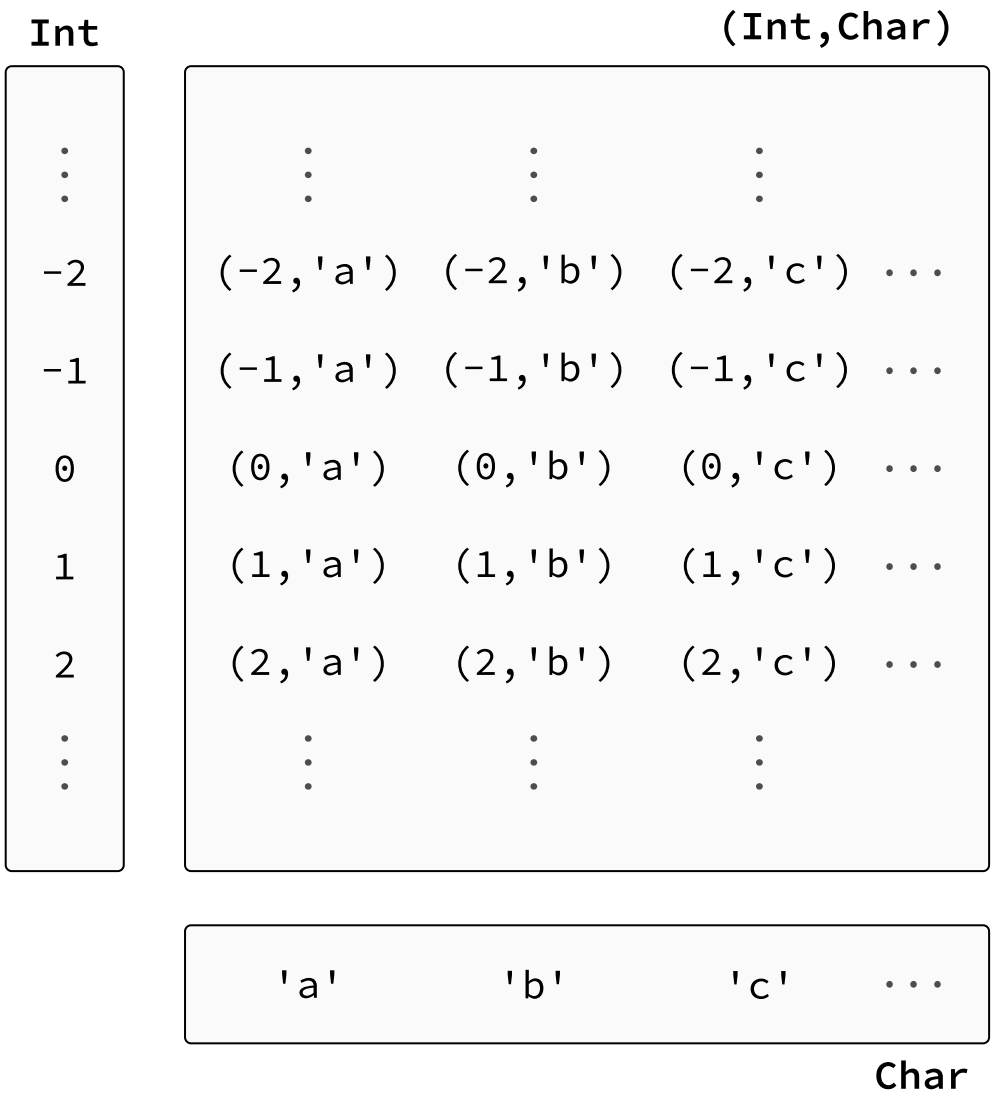
Ako je neke vrednosti moguće "prikazati" sa `show` funkcijom, tada je i listu tih vrednosti moguće "prikazati" sa `show` funkcijom. Lista će uvek biti prikazana na kanonski način (bez razmaka između elemenata), bez obzira kako smo tu listu konstruisali u kodu.

3.6. Uređene n-torke

Liste sadrže proizvoljan broj elemenata istog tipa. Suprotno tome, uređene n-torke sadrže fiksiran broj elemenata ne nužno istog tipa. Uređene n-torke se konstruišu navođenjem vrednosti odvojenih zarezima između zagrada `( )`. Tip uređene n-torke je uređena n-torka odgovarajućih tipova.

[31] *Literal* je deo koda koji predstavlja neku određenu vrednost. Na primer literal `42` predstavlja određenu celobrojnu vrednost, `'A'` određen karakter, a `"Hello!"` određenu nisku.

Primer 1. Vrednost `("Pozdrav!", True, 100.0)` je jedna uređena trojka koja sadrži nisku, logičku vrednost i `Float`. Ova uređena trojka ima tip `([Char], Bool, Float)`.



Vrednosti tipa `(A, B)` možemo elegantno predstaviti u pravougaonom rasporedu u kom vrste odgovaraju tipu `A` a kolone tipu `B`. U ovom slučaju broj kolona (a i vrsta) je izuzetno veliki, pa ne možemo predstaviti sve vrednosti na ilustraciji.

Najznačajnije uređene n-torke su svakako uređene dvojke koje češće nazivamo **uređeni parovi**. Nad svakim uređenim parom možemo pozvati funkciju `fst` i `snd` koje redom vraćaju prvu i drugu koordinatu uređenog para.

```
ghci> fst (5, 'C')
5
ghci> snd (5, 'C')
'C'
```

Funkcije `fst` i `snd` se mogu koristiti za "izvlačenje" prve i druge koordinate uređenog para.

Uređene trojke, četvorke, itd. se ređe koriste, te za njih ne postoje funkcije poput `fst` i `snd`. U narednim sekcijama ćemo videti kako možemo sami konstruisati odgovarajuće funkcije.

Poredak tipova u tipu uređene n-torke je važan: tip `(Int, Char)` nije isti kao tip `(Char, Int)`. Takođe broj pojavljivanja nekog tipa u uređenoj n-torci je bitan: tip `(Int, Int, Char)` nije isti kao tip `(Int, Char)`.

Zadatak 2. Koliko vrednosti sadrži tip `(Bool, Bool)`?

Rešenje. Navedeni tip sadrži četiri vrednosti: `(True, True)`, `(True, False)`, `(False, True)` i `(False, False)`.

Zadatak 3. Ako tipovi $\texttt{A}_1$, $\texttt{A}_2$ ... $\texttt{A}_n$ sadrže redom $k_1, k_2 \ldots k_n$ vrednosti, koliko vrednosti sadrži tip $(\texttt{A}_1, \texttt{A}_2, \ldots, \texttt{A}_n)$?

4. Funkcije

Pojam funkcije u Haskel jeziku ima izvesne sličnosti sa pojmom funkcije u imperativnim jezicima poput *C*-a ili Jave, ali izvesne razlike. Dok se u navedenim jezicima funkcija shvata kao niz instrukcija koje je potrebno izvršiti, u Haskelu funkcija predstavlja pravilnost po kojoj se od neke vrednosti dobija druga vrednost. Drugim rečima u Haskelu funkcije predstavljaju transformacije vrednosti. Ovo shvatanje se podudara sa matematičkom definicijom funkcije.

Takođe, dok je u imperativnim jezicima pojam funkcije razdvojen od pojma podatka (te je teško ili nemoguće vršiti sa funkcijama ono što je moguće vršiti sa podacima), u Haskelu je granica između ova dva pojma mnogo tanja. U Haskelu, svaka funkcija je samo jedna vrednost koja je sadržana u odgovarajućem tipu. I kao što vrednosti tipa `Int` mogu transformisati sa odgovarajućim funkcijama, ili vrednosti tipa `Bool` sa logičkim operacijama i funkcijama, tako se i funkcije mogu transformisati sa odgovarajućim funkcijama...

4.1. Pojam funkcije i tip funkcije

U matematici, funkcija je pravilnost po kojoj se *svakom* elementu nekog skupa pridružuje element nekog drugog skupa. Činjenicu da neka funkcija f elementima skupa A pridružuje elemente skupa B označavamo sa $f : A \to B$ i kažemo da f ima (poseduje) tip $A \to B$. Skup A nazivamo **domen** a skup B **kodomen** funkcije f[32]. Ako funkcija f elementu $x \in A$ pridružuje vrednost $y \in B$ to označavamo sa $f(x) = y$.

Za funkciju tipa $A \to B$ često kažemo da *preslikava* (*slika*, *transformiše*) skup A u skup B. Za izraz $f(x)$ kažemo da predstavlja *primenu funkcije* f *na vrednost* x.

Dve funkcije f_1 i f_2 smatramo za jednake ako imaju isti domen, isti kodomen, i ako je $f_1(x) = f_2(x)$ za svako x iz domena funkcije.

> **Primer 1.** Neka je $\mathbb{N} = \{0, 1, 2, ...\}$ skup prirodnih brojeva. Funkcija $s : \mathbb{N} \to \mathbb{N}$ definisana sa izrazom $s(x) = x + 1$ svakom prirodnom broju pridružuje naredni prirodni broj tj. važi $s(0) = 1$, $s(1) = 2$, $s(2) = 3$ itd... Domen i kodomen ove funkcije je skup prirodnih brojeva $\mathbb{N}$.
>
> Funkciju s možemo da shvatimo kao transformaciju: s transformiše broj 0 u broj 1, broj 1 u broj 2, itd...

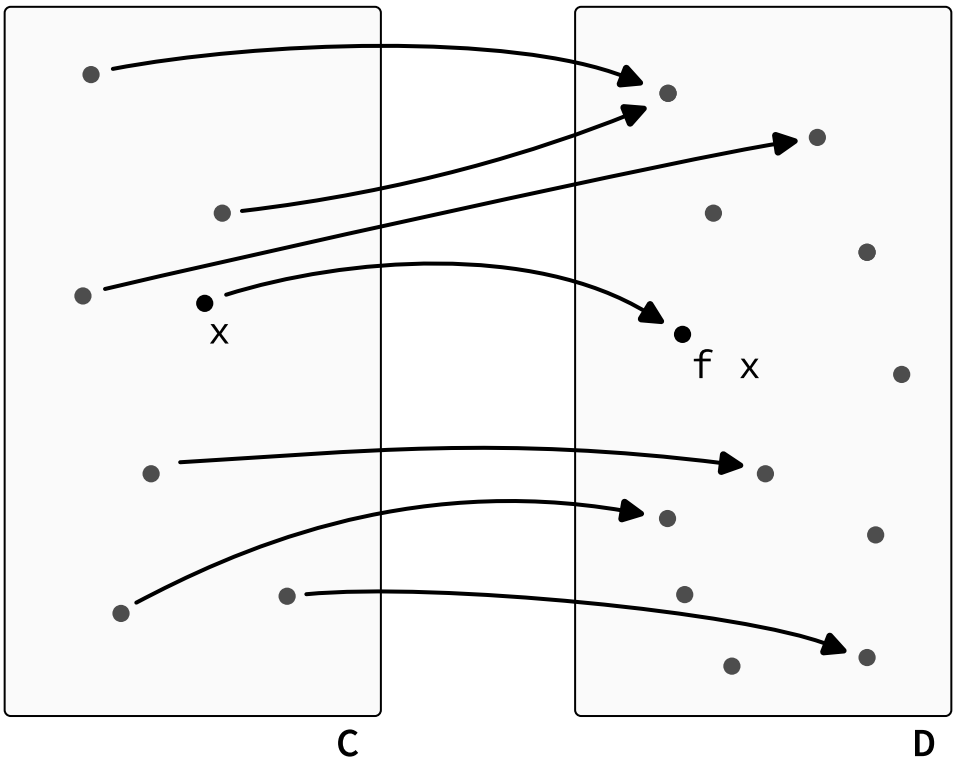
> **Primer 2.** Funkcija $c : \mathbb{N} \to \mathbb{N}$ definisana sa $c(x) = 1$ svakom prirodnom broju pridružuje isti broj - broj 1. Za funkciju c kažemo da je **konstantna funkcija** jer je njena vrednost konstantna bez obzira na vrednost x.
>
> Zapravo, za svaki prirodan broj n može se konstruisati funkcija tipa $\mathbb{N} \to \mathbb{N}$ koja sve prirodne brojeve preslikava u n.

> **Primer 3.** Neka je P skup svih tačaka jedne ravni, a T skup svih duži te ravni. Definišimo funkciju s koja svakoj duži iz T pridružuje središte te duži. Domen funkcije s je T, kodomen je P, a tip je $T \to P$.
>
> Primetimo da ovu funkciju nismo zadali sa formulom, već sa opisom. Ali jasno je da se radi o pravilnosti kojom se svakoj duži pridružuje tačka.

> **Primer 4.** Neka je X proizvoljan skup. Tada možemo definisati funkciju $i : X \to X$ sa $i(x) = x$. Funkcija i preslikava svaku vrednost u samu sebe, tj. svakom elementu x pridružuje njega samog. Funkciju i nazivamo **identička funkcija** na skupu X.

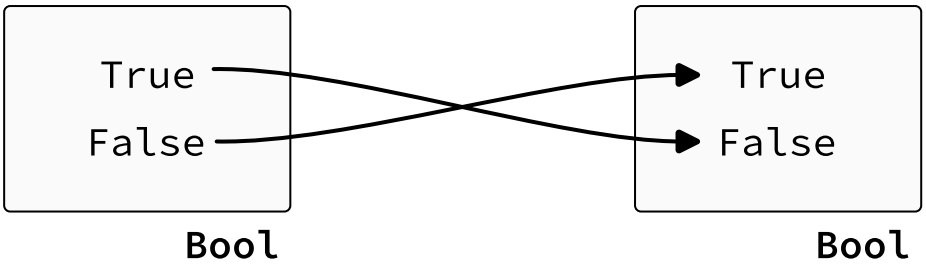
[32] Primetimo da ako znamo domen i kodomen funkcije, tada znamo i tip te funkcije. Takođe važi obrnuto, ako znamo tip funkcije onda znamo i domen i kodomen te funkcije.

U Haskelu, analogno matematici, funkcije predstavljaju pravilnosti po kojima se vrednostima nekog tipa (domena) pridružuju vrednosti drugog tipa (kodomena). Tip funkcije koja vrednostima tipa `A` pridružuje vrednosti tipa `B` označava se sa `A -> B`. Tip funkcije, kao i svaki drugi tip, predstavlja kolekciju nekih vrednosti. Tip `A -> B` predstavlja kolekciju svih funkcija koje tip `A` preslikavaju u tip `B`.



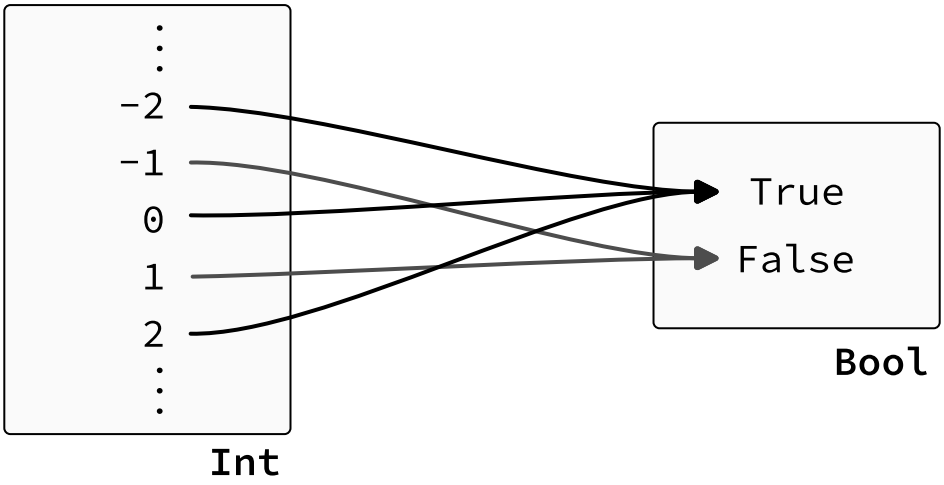
Na ilustraciji vidimo dva tipa `C` i `D`, čije vrednosti su predstavljene kao tačkice. Neku funkciju `f` tipa `C -> D`, možemo zamisliti kao strelice koje *sve* vrednosti tipa `C` "povezuju" sa vrednostima tipa `D`. Na ilustraciji je radi preglednosti istaknuta samo jedna takva strelica, ona koja vrednost `x` "povezuje" sa `f x`. Primetimo da svaka vrednost iz `C` mora biti povezana sa nekom vrednošću iz `D`, ali da ne moraju sve vrednosti iz `D` imati strelice ka sebi.

Primer 5. Funkcija `not` svakoj logičkoj vrednosti dodeljuje njenu negaciju koja je takođe logička vrednost. Prema tome, funkcija `not` ima tip `Bool -> Bool`.



Primer 6. Funkcija `even`, predefinisana u *GHCi*-ju, svakom celom broju dodeljuje logičku vrednost u zavisnosti da li je taj broj paran. Prema tome, tip funkcije `even` je[33] `Int -> Bool`. Isto važi i za funkciju `odd` koja proverava da li je broj neparan.

[33] Ovo nije u potpunosti tačno jer navedena funkcija može da se primeni na još neke tipove. Detalje ćemo ostaviti za kasnije.



Za razliku od imperativnih jezika, funkcije u Haskelu uvek moraju da poseduju kodomen. Drugim rečima, primena funkcije na vrednost mora i sama da predstavlja vrednost nekog tipa. U imperativnim jezicima neretko koristimo `void` funkcije koje nemaju povratnu vrednost. Takve funkcije se ne koriste u Haskelu.

Zadatak 1. Neka je B skup koji sadrži dve vrednosti. Koliko postoji funkcija tipa $B \to B$? Koje su to funkcije?

Zadatak 2. Neka je U skup koji sadrži jednu vrednost. Koliko postoji funkcija tipa $\mathbb{N} \to U$? Koliko postoji funkcija tipa $U \to \mathbb{N}$?

4.2. Lambda izrazi

Haskel je zasnovan na matematičkom formalizmu zvanom **lambda račun**. Lambda račun je formalni sistem koji opisuje pojam izračunljivosti pomoću *definicija* i *primene* funkcija. Ovde neće biti prezentovan lambda račun, ali mnoge stvari koje slede važe i za sam lambda račun.

U Haskelu, funkcije su *vrednosti* koje se definišu pravilom **apstrakcije**[34]. U Haskelu sintaksa za apstrakciju je

```
\x -> v.
```

Lambda apstrakcija se sastoji od znaka[35] `\`, zatim imena parametra (u ovom primeru to je `x` ali može biti bilo koja validna labela), zatim strelice `->` nakon koje sledi povratna vrednost funkcije odnosno neki izraz koji potencijalno zavisi od parametra (tj. od `x`). Izraz `\x -> v` nazivamo **lambda funkcija** a izraz koji sledi nakon strelice (tj. `v`), nazivamo **telo lambda funkcije**.

Primer 7. Funkcija koja vraća dvostruki argument se definiše pravilom apstrakcije kao:

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

Matematičkom notacijom zapisano, navedeni izraz predstavlja funkciju $g(x) = 2x$. Ime parametra je u potpunosti proizvoljno uzeto. Ista funkcija se može definisati i kao `\y -> (2 * y)` ili kao `\broj -> (2 * broj)`, itd... Telo prve navedene lambda funkcije je `(2 * x)`. Primetimo da navedenim kodom nismo

[34] *Apstrakcija* je samo naziv za sintaksu oblika `\x -> v`. U lambda računu koristi se nešto malo drugačija sintaksa $\lambda x.v$. Račun nosi naziv baš po slovu koje se nalazi na početku svake apstrakcije. Ovo slovo je proizvoljno odabrano i nema nikakvo dodatno značenje.

[35] koji je odabran tako da podseća na slovo λ

definisali i ime funkcije[36]. Da bismo mogli ovu funkciju da primenjujemo na vrednosti, dodelićemo je nekom imenu:

```
g = (\x -> (2 * x))
```

Primer validne definicije funkcije u `.hs` datoteci.

Nakon učitavanja navedenog koda u *GHCi*, možemo koristiti funkciju `g` i primenjivati[37] je na neke vrednosti. Da bi funkciju primenili na vrednost potrebno je tu vrednost navesti nakon imena funkcije, razdvajajući ih razmakom:

```
ghci> g 10
20
```

Izraz `g 10` shvatamo kao $g(10)$. Nismo ranije napomenuli, ali dužina razmaka (dokle god je taj razmak u jednoj liniji), ne igra nikakvu ulogu.

Svaki izraz oblika `m n`, gde su `m` i `n` neki izrazi, nazivamo **aplikacija**[38]. U Haskelu, svaka aplikacija predstavlja pokušaj primene izraza `m` na izraz `n`. Međutim, `m n` nema smisla ako `m` nije funkcija. Na primer, "primena" `'a' 'b'` daje grešku

```
ghci> 'a' 'b'
<interactive>:15:1: error:
    • Couldn't match expected type ‘Char -> t’ with actual type ‘Char’
    • The function ‘'a'’ is applied to one value argument,
        but its type ‘Char’ has none
      In the expression: 'a' 'b'
      In an equation for ‘it’: it = 'a' 'b'
    • Relevant bindings include it :: t (bound at <interactive>:15:1)
```

Ova *GHCi* greška nam govori da se izraz `'a'` ne može protumačiti kao funkcija i nema smisla primeniti `'a'` na `'b'`.

U prethodnom primeru, kada smo u *GHCi* prompt uneli `g 10` i pritisnuli `Enter`, *GHCi* je zamenio ime `g` sa odgovarajućim lambda izrazom. Tom zamenom se dobija izraz `(\x -> (2 * x)) 10`. Štaviše, i sami smo mogli da unesemo izraz u prompt i dobili bismo isti rezultat:

```
ghci> (\x -> (2 * x)) 10
20
```

Primena funkcije na vrednost bez prethodnog imenovanja te funkcije. Ovo ilustruje zašto se lambda funkcije nazivaju i *anonimne funkcije*

Izraz oblika `(\x -> a) b`, gde su `a` i `b` proizvoljni lambda izrazi, naziva se **redeks**. Redeksi zapravo predstavljaju smislene primene funkcije na vrednost.

Primer redeksa je upravo `(\x -> (2 * x)) 10` ali i `g 10` jer je `g` po definiciji `(\x -> (2 * x))`. Za razliku od izraza `'a' 'b'`, svaki redeks predstavlja smislenu aplikaciju funkcije na vrednost.

[36] Lambda funkcije se često nazivaju i anonimne funkcije. Čist lambda račun ne podrazumeva način imenovanja funkcija, pa su lambda funkcije *anonimne*.

[37] U imperativnim jezicima, češće se koristi izraz *pozivanje funkcije*. U funkcionalnim jezicima pravilnije je reći *primena funkcije na vrednost*, ali ćemo mi koristiti oba izraza.

[38] eng. *application* - primena

Zadatak 3. Napisati lambda izraz koji za dati broj r daje površinu kruga čiji je poluprečnik `r`. Dodeliti ovaj izraz imenu `površinaKruga`. Izračunati površinu krugova čiji su poluprečnici 2 i 5. Konstanta π je dostupna pod imenom `pi`.

Rešenje. Površina kruga sa poluprečnikom r je πr^2, te stoga možemo definisati traženi lambda izraz sa `\r -> (pi * r^2)`. Ovaj izraz možemo imenovati, odnosno dodeliti nekom imenu, i sačuvati u `.hs` datoteci:

```
površinaKruga = (\r -> (pi * r^2))
```

Nakon učitavanja datoteke u *ghci*, možemo naći površine različitih krugova:

```
ghci> površinaKruga 2
12.566370614359172
ghci> površinaKruga 5
78.53981633974483
```

Zadatak 4. Temperatura od x stepeni Farenhajta je jednaka temperaturi od $\frac{5}{9}(x - 32)$ stepeni Celzijusa. Napisati funkciju `uCelzijuse` koja stepene Farenhajta pretvara u stepene Celzijusa. Izraziti i stepene Farenhajta preko stepena Celzijusa, i napisati funkciju `uFarenhajte`. Potvrditi sa obe funkcije da je 30°C jednako 86°F

4.3. Beta redukcija

Ako je data matematička funkcija $f(x) = 2*x$, tada se postupak nalaženja vrednosti $f(10)$ sastoji u tome da se svako pojavljivanje promenljive x u izrazu $2*x$ zameni sa argumentom 10. Time se dobija izraz 2*10 koji može direktno da se izračuna. Analognim postupkom redeksi se mogu svesti na neku vrednost koja u sebi ne sadrži redeks.

Primer 8. Da bi evaluirao redeks `(\x -> (2 * x)) 10`, *GHCi* u telu lambda funkcije zamenjuje `x` sa `10`, čime se dobija `2 * 10`. Izraz `2 * 10` se zatim sračunava na nivou hardvera.

Opisani postupak nazivamo **beta redukcija**. Tačnije beta redukcija je zamena redeksa `(\x -> a) b` sa izrazom koji se dobije kada se izrazu `a` sva pojavljivanja imena `x` zamene sa izrazom `b`. Pritom se početak lambda izraza, `\x ->`, uklanja.

Primer 9. Neka je dat redeks

`(\x -> (((\y -> (y + 1)) x) * x)) 10`.

Ovaj redeks je zanimljiv, zato što se u telu lambda funkcije nalazi druga lambda funkcija. Beta redukcijom vrednost `10` zamenjujemo umesto `x` u telu "veće" lambda funkcije

`(((\y -> (y + 1)) 10) * 10)`

U rezultatu beta redukcije imamo redeks `(\y -> (y+1)) 10` koji takođe možemo da redukujemo, pri čemu ostatak izraza ostaje nepromenjen:

`((((10 + 1))) * 10)`.

Višestruke zagrade ne igraju ulogu, te smo došli do izraza `((10 + 1) * 10)` koji se direktno računa do vrednosti `110`.

Postavlja se pitanje, da li je poredak evaluacije redeksa važan? Da li bismo isti rezultat dobili i da smo prvo izvršili beta redukciju unutrašnjeg redeksa a zatim i spoljašnjeg?

Primer 10. *Nastavak prethodnog primera.* Beta redukcijom unutrašnjeg redeksa izraza u

```
(\x -> (((\y -> (y + 1)) x) * x)) 10
```

dobijamo izraz

```
(\x -> ((x + 1) * x)) 10.
```

Vršenjem još jedne beta redukcije dobijamo isti izraz kao malopre

```
((10 + 1) * 10).
```

Kao što vidimo, poredak vršenja lambda redukcije nije uticao na krajnju vrednost!

Osobina da poredak vršenja beta redukcije ne utiče na krajnu vrednost je poznata kao **konfluentnost**. Neće za svaki izraz važiti konfluentnost, a i u slučajevima kada važi, različit poredak beta redukcije može zahtevati drastično različit broj koraka (pa samim tim i vremena potrebnog kompletnu beta redukciju). Više o tome ćemo reći kasnije...

4.4. Tip funkcije

Do sada smo objasnili kako se definišu funkcije, ali nismo naveli mnogo o tipu funkcije. Razlog tome je što je Haskel automatski zaključivao tip definisanih funkcija. Ipak, kao i sa drugim vrednostima, programer može (i poželjno je) da eksplicitno navede tip funkcije.

Primer 11. Tip funkcije `g` iz prethodne sekcije možemo eksplicitno da navedemo u kodu:

```
g :: Int -> Int
g = (\x -> (2 * x))
```

Iz tipa `Int -> Int` vidimo da funkcija uzima jednu vrednost tipa `Int` i vraća takođe vrednost tipa `Int`.

Primer 12. Nešto složeniji primer bi bila funkcija koja uzima listu tipa `[Int]`, i vraća prvi član te liste uvećan za jedan:

```
h :: [Int] -> Int
h = (\x -> (head x + 1))
```

Aplikacija ima najveći prioritet te se izraz `head x + 1` tumači kao `(head x) + 1`.

Zanimljivo je primetiti šta se dešava sa tipom funkcije kada se funkcija primeni na vrednost. Pre svega, Haskel će proveriti da li se tip te vrednosti i domen funkcije podudaraju. Ako to nije slučaj, tada će doći do **tipske greške**.

Primer 13. Ako primenimo funkciju `h :: [Int] -> Int` na broj, dobićemo grešku:

```
ghci> h 2
<interactive>:3:3: error:
    • No instance for (Num [Int]) arising from the literal ‘2’
    • In the first argument of ‘h’, namely ‘2’
      In the expression: h 2
      In an equation for ‘it’: it = h 2
```

> Naravno, razlog ove greške je taj što funkcija `h` "očekuje" vrednost tipa `[Int]` a ne `Int`. Domen funkcije `h` je `[Int]`, i stoga se ona može primenjivati samo na vrednosti koje poseduju ovaj tip.

Ako nije došlo do tipske greške, tada primena funkcija na neku vrednost predstavlja **dobro tipiziran izraz**. Pritom, ako je funkcija `f` tipa `A -> B`, a vrednost `v` tipa `A`, tada je `f v` predstavlja vrednost tipa `B`. Beta redukcijom moguće je odrediti tačnu vrednost izraza `f v`, ali i bez redukcije znamo da ta vrednost poseduje tip `B`. Prosto rečeno, primenom funkcije tipa `A -> B` na vrednosti tipa `A` "brišemo" `A ->` iz tipa[39].

> **Zadatak 5.** Deklarisati tipove funkcija iz prethodnih zadataka ove lekcije.

4.5. Funkcije više promenljivih

Do sada je prikazano kako se mogu definisati funkcije samo sa jednim parametrom, ali su u prethodnoj lekciji korišćene funkcije sa dva parametra (kao što je funkcija `mod` koja određuje ostatak pri deljenju ili funkcija `elem` koja ispituje da li se element nalazi u listi). Postavlja se pitanje kako se ovakve funkcije definišu?

Pažljivi čitalac koji se seća sadržaja prethodne lekcije mogao je sam da konstruiše funkcije sa dva parametra. Naime, u prethodnoj sekciji smo videli kako se uređeni parovi mogu konstruisati. Prema tome, ako želimo funkciji da prosledimo dve vrednosti, dovoljno je da napišemo funkciju koja uzima jedan uređen par[40].

```
ljubav :: ([Char], [Char]) -> [Char]
ljubav = (\x -> (fst x ++ " voli " ++ snd x ++ "!"))
```

Funkcija `ljubav` uzima uređen par niski i vraća nisku.

```
ghci> ljubav ("Ana", "Milovana")
"Ana voli Milovana!"
ghci> ljubav ("Marijana", "Marijana")
"Marijana voli Marijana!"
```

> **Zadatak 6.** Napisati funkciju
>
> ```
> zapreminaValjka :: (Double, Double) -> Double
> ```
>
> koja na osnovu uređenog para visine i poluprečnika baze računa zapreminu valjka.

Pozivanje ovako definisane funkcije podseća na pozivanje funkcija u imperativnim jezicima. Ali primetimo da smo mi u prethodnoj lekciji funkcije poput `mod` pozivali sa `mod 7 3` a ne sa `mod (7, 3)`. Takođe možemo da primetimo da iako funkciji prosleđujemo dva podatka, funkcija `ljubav` suštinski uzima samo jednu vrednost tipa `([Char], [Char])`.

Lambda račun, koji je osmišljen pola veka pre Haskela, je zamišljen kao minimalan formalni sistem i kao takav ne poseduje pojam uređene n-torke. Umesto toga, funkcije sa više parametra u lambda računu se realizuju uz pomoć 'trika'. Naime, da bi dobili funkciju od više parametra, definisaćemo funkciju `f` od jednog parametra čija povratna vrednost je druga funkcija. Na taj način, izraz `(f x)` predstavlja novu funkciju koju možemo da primenimo na neku drugu vrednost. Ako je rezultat i ove primene funkcija, tada taj rezultat možemo da primenimo na sledeću vrednost i tako dalje.

---

[39] Kao što u matematici f predstavlja funkciju a $f(10)$ konkretan broj, tako i primena funkcije `f :: A -> B` na vrednost `v :: A` predstavlja rezultat tipa `B`.

[40] Kada bismo znali kako da pristupimo članovima uređene trojke, četvorke itd... tada bismo na sličan način mogli da definišemo i "funkcije sa više parametra". To će biti prikazano tek u narednoj lekciji.

Primer 14. Funkcija koja uzima dve vrednosti tipa `Double` i vraća njihovu aritmetičku sredinu može se definisati sa:

```
aritm = (\x -> (\y -> ((x+y)/2))).
```

U *GHCi* okruženju možemo da se uverimo da funkcija ispravno računa aritmetičku sredinu:

```
ghci> (aritm 8) 10
9
```

Zašto je funkcija pozvana na ovaj način? Pogledajmo samu definiciju funkcije `aritm`. Izraz `aritm 8` predstavlja jedan redeks. Beta redukcijom ovaj redeks se svodi na

```
(\y -> ((8+y)/2)).
```

Ali ovaj rezultat je ponovo jedna funkcija, stoga je možemo primeniti na vrednost `10`. Aplikacija

```
(\y -> ((8+y)/2)) 10
```

se beta-redukuje u izraz `(8 + 10)/2` koji se zatim sračunava u vrednost `9`.

Ipak, aplikacija `(aritm 8) 10` ne izgleda baš u potpunosti isto kao aplikacija `mod 7 2` koju smo videli u prethodnoj lekciji. Razlog tome je što u Haskelu aplikacija (primena funkcije na argument) je levo asocijativna operacija što znači da se izraz `(f x) y` može jednostavnije zapisati kao `f x y`. I zaista, ako se vratimo na prethodni primer, možemo se uveriti u to:

```
ghci> aritm 8 10
9
ghci> aritm 20 100
60
```

Kog tipa je funkcija `aritm`? Za trenutak pretpostavimo da su parametri `x` i `y` tipa `Double`[41]. Funkcija `aritm` uzima vrednost tipa `Double` i vraća neku funkciju, nazovimo je `g`. Funkcija `g` uzima i vraća vrednost tipa `Double` te je njen tip `Double -> Double`. Sada možemo da zaključimo da funkcija `aritm` poseduje tip `Double -> (Double -> Double)` jer uzima vrednost tipa `Double` i vraća vrednost tipa `Double -> Double`.

Koristeći opisan postupak, možemo konstruisati funkcije koje poseduju tri ili više parametra[42]. Tom prilikom konstruišemo funkcije tipa

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } (\ \ldots \texttt{ -> } (T_n \texttt{ -> } B)))$$

gde tipovi T_1, ..., T_n predstavljaju tipove parametra funkcije, a `B` povratnu vrednost takve funkcije. Primenom funkcije `f` tipa

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } (\ \ldots \texttt{ -> } (T_n \texttt{ -> } B)))$$

na vrednost tipa T_1 dobija se funkcija tipa

$$T_2 \texttt{ -> } (T_3 \texttt{ -> } \ldots \texttt{ -> } (T_n \texttt{ -> } B)).$$

Novodobijena funkcija može se primeniti na vrednost tipa T_2 čime se dobija funkcija tipa

$$T_3 \texttt{ -> } \ldots \texttt{ -> } (T_n \texttt{ -> } B).$$

Postupak se može ponavljati sve dok se ne stigne do vrednosti tipa `B`.

[41] Zapravo, za ovako napisano definiciju kompajler će zaključiti nešto drugačiji tip, ali to ne treba da nas brine trenutno.

[42] Primetimo da mi zapravo radimo samo sa funkcijama koje imaju jedan parametar. Funkcija `aritm` zaista ima samo jedan parametar! Ipak često ćemo za ovakve funkcije reći da imaju `k` parametra, znajući iz konteksta o čemu se radi.

Funkcija koja "uzima" ili "vraća" neku drugu funkciju naziva se **funkcija višeg reda**. Mnogi programski jezici nemaju podršku za funkcije višeg reda, ali kao što vidimo, u Haskelu je ovaj pojam sasvim prirodan.

Zadatak 7. Napisati funkciju

```
zapremina :: Double -> (Double -> (Double -> Double))
```

koja na osnovu tri argumenta računa zapreminu kvadra čije su dužine stranica zadate tim argumentima.

Rešenje.

```
zapremina :: Double -> (Double -> (Double -> Double))
zapremina = (\a -> (\b -> (\c -> (a * b * c))))
```

Zadatak 8. Napisati funkciju

```
zapreminaValjka :: Double -> (Double -> Double)
```

koja na osnovu prosleđene visine i poluprečnika baze računa zapreminu valjka.

4.6. *Prelude*

U Haskelu, **Prelid** (eng. *prelude*) predstavlja kôd koji je dostupan svakom programu[43]. Ovaj kod je deo standardne biblioteke Haskela.

Sve predefinisane funkcije koje smo do sada predstavili, poput `not`, `mod`, `div`, `length`, `head`, `fst`, i tako dalje, zapravo su definisane u Haskel Prelidu. Takođe, konstante poput `pi` su definisane u Prelidu. U narednim lekcijama videćemo kako su tačno navedene funkcije definisane, i upoznaćemo još mnoge druge definicije iz Prelida.

4.7. Zadaci

Zadatak 9. Rešenja x_1 i x_2 kvadratne jednačine $ax^2 + bx + c = 0$ su data sa *kvadratnom formulom*

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Implementirati funkciju

```
rešenja :: Double -> (Double -> (Double -> (Double, Double)))
```

koja za tri prosleđena argumenta a, b i c vraća uređeni par rešenja koja su dobijena putem navedene formule. Pretpostaviti da su koeficijenti tako zadati da su sve računske operacije definisane (u realnom domenu).

Zadatak 10. Napisati funkciju

```
primeni :: (Int -> Bool) -> ((Int, Int) -> (Bool, Bool))
```

koja uzima jednu funkciju `f :: Int -> Bool`, jedan uređen par tipa `(Int, Int)`, i vraća uređen par koji se dobija primenom `f` na svaku koordinatu uređenog para.

[43] U muzici prelid je kratka muzička forma improvizacijskog karaktera, koja se koristi kao uvod u veća muzička dela. Tako i definicije iz Haskel Prelida mogu da se shvate kao uvod u naš program.

Rešenje. Kada se `primeni` primeni na neku funkciju `f`, potrebno je da dobijemo lambda funkciju koja uzima par i vraća par dobijen primenom `f` na svaku od koordinata. Ova lambda funkcija se može zapisati kao

```
\par -> (f (fst par), f (snd par)).
```

Jedino što je potrebno je da uvedemo i `f` kao jedan parametar, tj. da napravimo apstrakciju po `f`. Time dobijamo:

```
primeni :: (Int -> Bool) -> ((Int, Int) -> (Bool, Bool))
primeni = (\f -> (\par -> (f (fst par), f (snd par))))
```

Da bi testirali našu funkciju, definisaćemo i funkciju `paran :: Int -> Bool`

```
paran = (\n -> (0 == mod n 2))
```

Nakon učitavanja u *GHCi*, uveravamo se da funkcija zaista radi:

```
ghci> primeni paran (2, 7)
(True,False)
```

Za naredne zadatke neophodno je podsetiti se funkcija za rad sa listama iz prethodne lekcije: `length`, `head`, `last`, `init`, `tail`, `reverse`, `elem`.

Zadatak 11. Napisati funkciju

```
inicijali :: [Char] -> ([Char] -> [Char])
```

koja od dve niske koje predstavljaju ime i prezime pravi inicijale. Na primer `inicijali "Nikola" "Tesla"` daje `"N.T."`. Pretpostaviti da su obe niske neprazne.

Rešenje. Korišćenjem `head` funkcije, "uzećemo" prva slova imena i prezimena. Primena funkcije `head` na nisku nam daje vrednost tipa `Char`. Stoga dobijenu vrednost moramo postaviti u novu nisku, da bismo mogli koristiti operator `++` koji nadovezuje niske.

```
inicijali :: [Char] -> ([Char] -> [Char])
inicijali = (\x1 -> (\x2 -> (
    let y1 = head x1; y2 = head x2
    in [y1] ++ "." ++ [y2] ++ "."
  )))
```

Zadatak 12. Napisati funkciju

```
ukloni3 :: [Char] -> [Char]
```

koja uklanja prva tri elementa iz niske. Pretpostaviti da će funkcija uvek biti primenjena na niske dužine veće od tri.

Rešenje. Funkcija `tail` nam vraća rep niske (sve osim prvog elementa). Primenom ove funkcije tri puta na neku listu, efektivno uklanjamo prva tri elementa iz liste

```
ukloni3 :: [Char] -> [Char]
ukloni3 = (\x -> (tail (tail (tail x))))
```

Zadatak 13. Za nisku kažemo da je *palindrom* ako se čita isto i sa levo i sa desno, npr. `"anavolimilovana"` jeste palindrom. Napisati funkciju

```
palindrom :: [Char] -> Bool
```

koja proverava da li je niska palindrom.

Rešenje. Niske, odnosno vrednosti tipa `[Char]`, pripadaju klasi `Eq`. Zbog toga je moguće koristiti operator `==` za poređenje dve niske. Niska `s` je palindrom ako je jednaka nisci `reverse s`. Ovo zapažanje nas direktno dovodi do rešenja:

```
palindrom :: [Char] -> Bool
palindrom = (\s -> (s == reverse s))
```

Zadatak 14. Sa tipom `(Double, Double)` moguće je predstaviti vektor dvodimenzionalne ravni. Napisati funkciju

```
zbir :: (Double, Double) -> ((Double, Double) -> (Double, Double))
```

koja sabira vektore.

Rešenje.

```
zbir :: (Double, Double) -> ((Double, Double) -> (Double, Double))
zbir = (\u -> (\v -> (fst u + fst v, snd u + snd v)))
```

5. Sintaksa u funkcijama

5.1. Brisanje zagrada

U prethodnim primerima videli smo da lambda izrazi mogu biti veoma nepregledni zbog višestrukih zagrada. Zbog toga su u lambda račun, pa i Haskel, uvedene konvencije koje omogućavaju brisanje suvišnih zagrada u lambda izrazima:

1. *Zagrade oko lambda izraza ne zapisujemo.* Kada unosimo neki izraz u *GHCi*, tada nije neophodno da postavljamo zagradu koja obuhvata celokupan izraz. Isto važi i kada u kodu definišemo neki lambda izraz. Na primer, umesto `g = (f 20)` pišemo samo `g = f 20`.
2. *Aplikacija ima veći prioritet u odnosu na apstrakciju.* Praktično to znači da telo lambda funkcije nije neophodno postavljati u posebne zagrade. Na primer umesto `\x -> (f x)` pišemo `\x -> f x`.
3. *Aplikacija lambda izraza je levoasocijativna.* To znači da se izraz oblika `((M N) P)` može zapisati kao `M N P`. Induktivnim argumentom sledi da se i izraz `(((M N) P) Q)` može zapisati kao `M N P Q`, itd... Na primer, umesto `(zbir 22) 33` pišemo samo `zbir 22 33`. Sa druge strane, iz izraza `f 10 (g 20)` ne možemo obrisati zagrade, jer bismo time dobili izraz koji se interpretira kao `((f 10) g) 20`.
4. *Apstrakcija je desnoasocijativna.* Npr. umesto `\x -> (\y -> x + y)` pišemo `\x -> \y -> x + y`.
5. *Višestruke apstrakcije se mogu svesti pod jednu lambdu.* Na primer umesto `\x -> \y -> x + y` pišemo samo `\x y -> x + y`. Obratite pažnju da parametri funkcije moraju međusobno biti razdvojeni razmakom.

> **Primer 1.** Pojednostavimo lambda izraz
>
> ```
> (\x -> (\y -> ((f x) (g y)))).
> ```
>
> Pre svega, zagrade oko celog lambda izraza mogu se obrisati, čime se dobija
>
> ```
> \x -> (\y -> ((f x) (g y))).
> ```
>
> Višestruka apstrakcija može se svesti pod jednu apstrakciju, čime se dobija
>
> ```
> \x y -> ((f x) (g y)).
> ```
>
> Zagrade u telu lambda funkcije nisu neophodne, jer se u tim zagradama nalazi jedna aplikacija (a po drugom pravilu, aplikacija ima veći prioritet u odnosu na apstrakciju)
>
> ```
> \x y -> (f x) (g y).
> ```
>
> Kako je aplikacija levo asocijativna, levi par zagrada se može obrisati
>
> ```
> \x y -> f x (g y)
> ```

Osim pravila o zagradama u lambda izrazima, koristi se i jedno pravilo koje se odnosi na zagrade u tipovima. Ovo pravilo kaže da je `->` desno-asocijativan operator, odnosno da `a -> (b -> c)` može pisati kao `a -> b -> c`.

> **Primer 2.** Pogledajmo tipove `Int -> (Char -> Bool)` i `(Int -> Char) -> Bool`. Gledano kao na nizove simbola, razlika između ova dva tipa je samo raspored zagrada. Ali tipovi nam govore mnogo o ovim funkcijama!
>
> Neka je `f :: Int -> (Char -> Bool)`. Tada primenom `f` na neku vrednost `n :: Int` dobijamo funkciju `f n :: Char -> Bool`. Funkciju `f n` možemo zatim da primenimo vrednost `c :: Char`, i dobijemo vrednost tipa `Bool`. Dakle, `f` možemo da shvatimo kao binarnu funkciju koja uzima jednu `Int` i jednu `Char` vrednost. Tip ove funkcije, po gorepomenutom pravilu, možemo da napišemo i kao `Int -> Char -> Bool`

Neka je `g :: (Int -> Char) -> Bool`. Funkciju `g` možemo da primenimo samo na neku funkciju tipa `Int -> Char` i time dobijemo vrednost tipa `Bool`. Dakle `g` je funkcija "jednog argumenta"[44].

Primer 3. U prethodnoj lekciji, u zadatku 7 se tražilo da se napiše funkcija

```
primeni :: (Int -> Bool) -> ((Int, Int) -> (Bool, Bool))
```

koja uzima jednu funkciju `f :: Int -> Bool`, jedan uređen par tipa `(Int, Int)` i vraća uređen par koji se dobija primenom `f` na svaku koordinatu uređenog para. Rešenje do kog smo došli je

```
primeni :: (Int -> Bool) -> ((Int, Int) -> (Bool, Bool))
primeni = (\f -> (\par -> (f (fst par), f (snd par))))
```

Navedena lambda je oblika

```
(\f -> (\par -> …)).
```

Pre svega, možemo se osloboditi zagrada oko celog izraza, a zatim svesti dvostruku apstrakciju pod jednu lambdu.

Tip funkcije je oblika `a -> (b -> c)`, pa možemo ukloniti jedan par zagrada.

Dobijeni kod je nešto čitljiviji:

```
primeni :: (Int -> Bool) -> (Int, Int) -> (Bool, Bool)
primeni = \f par -> (f (fst par), f (snd par))
```

Iz ovog koda ne možemo ukloniti više zagrada

Zadatak 1. Uraditi ponovo zadatke iz prethodne lekcije koristeći što manje zagrada u kodu.

5.2. *If then else*

Jedna od osnovnih mogućnosti koju svaki programski jezik mora posedovati je sposobnost promene toka izvršavanja programa u zavisnosti od stanja u kom se nalazi. Ovakvu promenu toka nazivamo **grananje**.

Izraz *if then else* je najjednostavniji primer grananja u Haskelu[45]. Sintaksa za ovu konstrukciju ima oblik

```
if t then a else b,
```

gde je `t` neki logički izraz kog nazivamo *uslov*, a `a` i `b` izrazi istog tipa. Izraz `if t then a else b` se evaluira u `a` ako se `t` evaluira u `True`, a u suprotnom u `b`. Sledeći primer ilustruje korišćenje *if then else* konstrukcije:

```
ghci> if 5 > 0 then "5 je pozitivan broj" else "5 nije pozitivan broj"
"5 je pozitivan broj"
```

U prikazanom primeru, uslov je konstantan, te će navedeni *if then else* izraz uvek imati istu vrednost. Mnogo korisnije je kada *if then else* postavimo u telo funkcije:

```
daLiJepozitivan = \x -> if x > 0 then "jeste" else "nije"
```

[44] Svaka funkcija u Haskelu je funkcija jednog argumenta. Ali zbog karijevanja, funkcije čiji tip je oblika `a1 -> (a2 -> b)` možemo da nazivamo funkcijama dva argumenta, itd...

[45] Izraz u potpunosti odgovara trenarnom operatoru `? :` koji je prisutan u mnogim imperativnim jezicima.

```
ghci> daLiJepozitivan 3
"jeste"
```

Važno je da poštujemo tri pravila:

1. Uslov (izraz između `if` i `then` klauze) mora biti tipa `Bool`.
2. Moraju se navesti povratne vrednosti za oba slučaja (u suprotnom izraz bi mogao biti nedefinisan u zavisnosti od vrednosti uslova).
3. Povratne vrednosti za oba slučaja moraju biti istog tipa (u suprotnom izraz bi imao različit tip u zavisnosti od vrednosti uslova).

Zadatak 2. Bez korišćenja ugrađene funkcije `max`, implementirati funkciju `max' :: Int -> Int -> Int` koja nalazi maksimum dva broja.

Rešenje.

```
max' :: Int -> Int -> Int
max' = \x y -> if x > y then x else y
```

Zadatak 3. Implementirati funkciju `prestupna :: Int -> Bool` koja na osnovu pravila Gregorijanskog kalendara određuje da li je data godina prestupna. Godina se smatra prestupnom ako je deljiva sa 4, osim u slučaju kada je deljiva i sa 100. Međutim, ako je deljiva i sa 400 tada se ipak smatra prestupnom. Na primer, godina 2000. je bila prestupna jer je deljiva sa 400. Takve su bile i 2004, 2008 i 2012. jer su sve deljive sa 4. Ali 2100. godina neće biti prestupna jer je deljiva sa 100 ali ne i sa 400.

5.3. Ograđene definicije

Ograđena definicija[46] je poseban oblik definicije funkcije koja omogućava pregledno testiranje više uslova. Svaki od uslova je logički izraz koji se navodi nakon vertikalne crte `|`, a nakon uslova postavlja se povratna vrednost funkcije.

Ograđena definicija ne predstavlja izraz[47]. Opšti oblik ograđenih definicija bi bio[48]:

```
f x1 x2 x3
  | uslov1 = vrednost1
  | uslov2 = vrednost2
  ...
  | uslovN = vrednostN
```

U ovom zapisu, `f` je ime funkcije koja se definiše, a `x1`, `x2`, `x3` su parametri funkcije `f`. Uslovi, `uslov1` ... `uslovN` su izrazi koji se evaluiraju u logičku vrednost.

Kao i kod *if then else* konstrukcije, i kod ograđenih definicija sve povratne vrednosti moraju biti istog tipa.

[46] eng. *guards*
[47] Za razliku od *let in* ili *if then else* sintakse.
[48] Ovakva definicija funkcije podseća na narednu matematičku sintaksu

$$f(x) = \begin{cases} \text{vrednost1}, & \text{uslov1} \\ \text{vrednost2}, & \text{uslov2} \\ \vdots & \\ \text{vrednostN}, & \text{uslovN} \end{cases}$$

Primer 4. Funkcija `pozitivan :: Int -> [Char]` koja vraća nisku `"Pozitivan"` ako je broj veći ili jednak nuli a u suprotnom `"Negativan"`, može se ovako definisati

```
pozitivan :: Int -> [Char]
pozitivan x
 | x >= 0 = "Pozitivan"
 | x < 0 = "Negativan"
```

Prilikom "poziva" `pozitivan (-4)`, prvo će se proveriti prvi navedeni slučaj, odnosno `x >= 0`. Kako se `-4 >= 0` evaluira u `False`, proveriće se naredni slučaj `x < 0`. Kako se `-4 < 0` redukuje u `True`, vrednost `pozitivan (-4)` je `"Negativan"`.

Dakle, sa ograđenim definicijama moguće je proveriti niz uslova. Funkcija će imati vrednost koja odgovara prvom tačnom uslovu. Ako ni jedan od uslova nije tačan, tada će doći do izuzetka.

```
pozitivan' :: Int -> [Char]
pozitivan' x
 | x > 0 = "Pozitivan"
 | x < 0 = "Negativan"
```

U ovoj, *lošoj*, definiciji uslov `x == 0` nikad nije proveren, zbog čega će poziv `pozitivan' 0` dovesti do izuzetka.

Primer 5. Pogledajmo nešto složeniji primer. Funkcija koja vraća opis jačine zemljotresa na osnovu njegove jačine u Rihterima može biti definisana sa narednim kodom:

```
opisZemljotresa :: Double -> [Char]
opisZemljotresa r
  | (r >= 0.0) && (r < 2.0) = "Mikro"
  | (r >= 2.0) && (r < 4.0) = "Manji"
  | (r >= 4.0) && (r < 5.0) = "Lakši"
  | (r >= 5.0) && (r < 6.0) = "Srednji"
  | (r >= 6.0) && (r < 7.0) = "Jak"
  | (r >= 7.0) && (r < 8.0) = "Velik"
  | (r >= 8.0) && (r < 10.0) = "Razarajući"
  | r >= 10 = "Epski"
```

Ako pozovemo `opisZemljotresa 5.6`, tada će uslov `(r >= 5.0) && (r < 6.0)` biti zadovoljen zbog čega će se `opisZemljotresa 5.6` evaluirati `"Srednji"`. Primetimo da funkcija nije definisana za negativne argumente.

U prethodnom primeru možemo primetiti da uslove možemo pojednostaviti. Naime kako su uslovi poređani redom po jačini zemljotresa, možemo se osloboditi dela uslova kojim se proverava da je jačina zemljotresa jača od nekog intenziteta[49]. Stoga funkciju `opisZemljotresa` možemo i ovako definisati

[49] Jer je povratna vrednost funkcije definisane sa *guards* sintaksom ona koja odgovara prvom zadovoljenom uslovu.

```
opisZemljotresa :: Double -> [Char]
opisZemljotresa r
  | r < 2.0 = "Mikro"
  | r < 4.0 = "Manji"
  | r < 5.0 = "Lakši"
  | r < 6.0 = "Srednji"
  | r < 7.0 = "Jak"
  | r < 8.0 = "Velik"
  | r < 10.0 = "Razarajući"
  | r >= 10 = "Epski"
```

Pozivanjem `opisZemljotresa 5.6` ponovo dobijamo `"Srednji"`. Iako su i uslovi `r < 7.0`, `r < 8.0`, `r < 10.0` tačni, uslov `r < 6.0` je prvi naveden.

Primetimo da je i u ovom (kao i u prethodnom primeru) na poslednjem mestu postavljen slučaj koji je zadovoljen kada svi ostali slučajevi nisu. Zbog toga, za poslednji uslov ne mora se stavljati nikakav izraz, već je dovoljno postaviti konstantu `True`, i tada će poslednji uslov "uhvatiti" sve što nije uhvaćeno sa prethodnim uslovima. Uobičajeno je da se na kraju ograđene definicije postavi konstanta `otherwise` koja je ime za vrednost `True`[50]:

```
opisZemljotresa r
  | r < 2.0 = "Mikro"
  | r < 4.0 = "Manji"
  | r < 5.0 = "Lakši"
  | r < 6.0 = "Srednji"
  | r < 7.0 = "Jak"
  | r < 8.0 = "Velik"
  | r < 10.0 = "Razarajući"
  | otherwise = "Epski"
```

Zadatak 4. Definisati funkciju `max3 :: Int -> Int -> Int -> Int` sa ograđenom definicijom, koja vraća najveću od tri celobrojne vrednosti.

Rešenje. Jedno od mogućih rešenja bi bilo:

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

5.4. Podudaranje oblika

Podudaranje oblika[51] je sintaksa koja dozvoljava da se vrednosti funkcije definišu u zavisnosti od "oblika" argumenta. Da bismo definisali funkciju pomoću podudaranja oblika, potrebno je da odmah nakon imena funkcije navedemo oblik argumenta koji želimo da *uhvatimo* a zatim i vrednost funkcije za takav argument. Na primer, funkciju tipa `dupliraj :: Int -> Int` koja duplira argument možemo definisati sa:

[50] Zaista, u *GHCi*-u proverite da je `otherwise == True`.
[51] Eng. *pattern matching*

```
dupliraj :: Int -> Int
dupliraj x = 2 * x
```

Ovo je primer trivijalnog podudaranja oblika. Ovde x predstavlja oblik proizvoljnog broja i ne vrši se nikakvo grananje.

Kao što vidimo, navedenom sintaksom oslobodili smo se definicije funkcija preko lambda izraza. I zaista, gotovo uvek ćemo umesto sintakse za lambda apstrakciju koristiti podudaranje oblika, pa čak i kada u funkciji nema grananja.

Ipak, sintaksa podudaranja oblika je veoma zgodna za grananje. U najjednostavnijem slučaju, to grananje se sastoji u tome da se "uhvati" određena vrednost argumenta. Na primer funkcija `f` koja slika `0` u `1` a sve ostale brojeve u `0`, može biti ovako definisana:

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

Naravno, funkciju `f` smo mogli definisati lako i uz pomoć *guards* ili *if then else* sintakse.

Dakle sa kodom `f 0` "uhvaćena je" posebna vrednost argumenta. Kao i kod *guards* sintakse, pri podudaranju oblika slučajevi se proveravaju redom, odozgo ka dole. Zato je naredna funkcija konstantna:

```
f' :: Int -> Int
f' x = 0
f' 0 = 1
```

U ovom slučaju, prva linija "hvata" oblik svakog argumenta, i nikad neće doći do provere druge linije. Prema tome, `f' 0` će se evaluirati u `0`.

Naravno, moguće je navesti proizvoljno mnogo slučajeva:

```
daLiJeCifra :: Char -> Bool
daLiJeCifra '0' = True
daLiJeCifra '1' = True
daLiJeCifra '2' = True
daLiJeCifra '3' = True
daLiJeCifra '4' = True
daLiJeCifra '5' = True
daLiJeCifra '6' = True
daLiJeCifra '7' = True
daLiJeCifra '8' = True
daLiJeCifra '9' = True
daLiJeCifra x = False
```

Funkcija koja proverava da li dati karakter reprezentuje cifru.

U navedenom primeru, poslednja linija koda, odnosno `daLiJeCifra x = False` "hvata" svaki argument. Ali kao što vidimo, povratna vrednost funkcije, ne zavisi od vrednosti uhvaćenog argumenta. U ovakvim situacijama, može se koristiti znak _ koji predstavlja argument koji ne želimo da vezujemo za ime. U ovom kontekstu, znak _ nazivamo **džoker**[52].

Koristeći džoker, poslednju liniju prethodnog primera možemo da zamenimo sa `daLiJeCifra _ = False`. Ako se vratimo na pretposlednji primer (funkcija `f` koja slika `0` u `1` a sve ostale brojeve u `0`), ta funkcija može biti definisana i sa

[52] eng *wildcard*

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

Sa podudaranjem oblika, možemo lako definisati i funkcije više parametra. Tako na primer naredna dva koda definišu istu funkciju:

```
g :: Int -> Int -> Int
g = \x y -> x + 2*y
```

```
g :: Int -> Int -> Int
g x y = x + 2*y
```

Možemo i da "pomešamo" sintaksu lambda apstrakcije i sintaksu podudaranja oblika i da napišemo sledeći kod

```
g :: Int -> Int -> Int
g x = \y -> x + 2*y
```

Sintaksu podudaranja oblika možemo da shvatimo na sledeći način: kada `g` primenimo na neku vrednost `x`, tada dobijamo funkciju `\y -> x + 2*y`.

5.4.1. Podudaranje oblika liste

Tehnika podudaranja oblika je veoma korisna pri radu sa listama. Tada lako možemo da uhvatimo praznu, jednočlanu, dvočlanu listu, itd... ili da rastavimo listu na glavu i rep.

> **Primer 6.** Funkciju koja proverava da li je lista celih brojeva prazna[53], možemo definisati na sledeći način:
>
> ```
> prazan :: [Int] -> Bool
> prazan [] = True
> prazan x = False
> ```
>
> Slučaj prazne liste je uhvaćen sa oblikom `[]`, dok su svi ostali slučajevi obuhvaćeni generičkim promenljivom `x`.
>
> Neophodno je obuhvatiti sve moguće oblike promenljive. Ako se neki od oblika izostavi, definisana funkcija neće biti totalna, i za neke od argumenata će doći do izuzetka:
>
> ```
> prazan' :: [Int] -> Bool
> prazan' [] = True
> ```
>
> ```
> ghci> prazan' [1, 2, 3]
> *** Exception: main.hs:3:1-15: Non-exhaustive patterns in function prazan'
> ```

Na primer, ako želimo da napišemo funkciju `s :: [Int] -> Int` koja praznoj listi dodeljuje 0, jednočlanoj listi dodeljuje element te liste, dvočlanoj listi dodeljuje zbir dva elementa, a svim ostalim listama dodeljuje 1, korišćenjem *guards* sintakse dobijamo naredni kod:

[53] Ovo je naravno funkcija `null`

```
s :: [Int] -> Int
s xs
  | xs == [] = 0
  | length xs == 1 = xs !! 0
  | length xs == 2 = xs !! 0 + xs !! 1
  | otherwise = 1
```

Korišćenjem podudaranja oblika dobijamo elegantniji kod:

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

Kao što vidimo, podudaranjem oblika liste *dekonstruisali* smo neke moguće oblike argumenta funkcije. U trećoj liniji definisali smo funkciju u slučaju kada je njen argument jednočlana lista `[x]`. Imenom `x` ovde smo "vezali" član te jednočlane liste. Slično, u narednoj liniji koda, dekonstruisali smo dvočlane liste. U ovom slučaju, prvi element je predstavljen imenom `x` a drugi element imenom `y`.

Moguće je takođe koristi činjenicu da se lista poput `[x, y, z]` može predstaviti kao `x:[y,z]` ili `x:y:[z]` ili `x:y:z:[]`. Tako je prethodni primer moguće napisati i kao

```
s :: [Int] -> Int
s [] = 0
s (x:[]) = x
s (x:y:[]) = x + y
s _ = 1
```

Primer 7. Funkcija koja vraća rep liste brojeva (sve element osim prvog), može se definisati na sledeći način:

```
rep :: [Int] -> [Int]
rep (x:xs) = xs
```

Operator `:` nadovezuje element na početak *liste*. Prema tome `x` je prvi element, a `xs` predstavlja ostatak liste. Primetimo da ova funkcija nije definisana za slučaj prazne liste.

Zapravo, pošto nas u navedenom kodu vrednost `x` ne interesuje, funkciju možemo definisati i sa džokerom:

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

Džokere bi trebalo koristiti kad god je to moguće, jer poboljšavaju čitljivost koda tako što umanjuju broj vezanih imena.

Dobro je naglasiti da pri podudaranju oblika nije moguće navesti oblik koji sadrži operator `++` (na primer ne možemo uhvatiti dvočlanu listu sa `[x] ++ [y]`). Razlog zašto je operator `:` moguće koristiti, a `++` ne, postaće jasan u jednoj od narednih lekcija.

5.4.2. Podudaranje oblika n-torke

Uređene n-torke se takođe mogu dekonstruisati podudaranjem oblika.

Primer 8. Funkciju `saberi :: (Double, Double) -> Double` koja sabira koordinate uređenog para možemo definisati sa narednim kodom

```
saberi :: (Double, Double) -> Double
saberi (x, y) = x + y
```

Primer 9. Funkciju `saberi3` koja sabira koordinate uređene trojke možemo definisati na sledeći način

```
saberi3 :: (Double, Double, Double) -> Double
saberi3 (x, y, z) = x + y + z
```

Iz dva navedena primera, vidimo da nam funkcije poput `fst` ili `snd` nisu neophodne.

Zadatak 5. Sa uređenom trojkom `(Double, Double, Double)` može se prezentovati vektor trodimenzionalnog prostora. Napisati funkcije za zbir, skalarni i vektorski proizvod dva vektora, kao i funkciju koja skalira vektor za dati koeficijent i funkciju koja računa dužinu vektora.

5.4.3. Podudaranje oblika unutar lambda izraza

U lambda izrazima moguće je upotrebiti podudaranje oblika za dekonstrukciju argumenta. Zbog same forme lambda izraza, moguće je navesti samo jedan oblik po kom se vrši podudaranje. Taj oblik postavljamo između "lambde" i strelice.

Primer 10. Funkciju `head` možemo elegantno da definišemo kao

```
\(x:xs) -> x.
```

Naravno, ova funkcija neće biti definisana za sve vrednosti, jer slučaj prazne liste nije obrađen (niti može biti obrađen ako je funkcija definisana sa lambda izrazom).

Podudaranje oblika u lambda izrazima retko se koristi sa listama, baš iz razloga zato što možemo proveriti samo jedan oblik argumenta (što je retko kad dovoljno u slučaju lista). Ali, kako je oblik uređenih n-torki u potpunosti određen tipom, sa takvim tipovima se često koristi podudaranje oblika u lambda izrazima.

Primer 11. Umesto funkcija `fst` i `snd` možemo koristiti lambda funkcije `\(x, y) -> x` i `\(x, y) -> y`.

6. Rekurzija

Rekurzija je način rešavanja nekog problema korišćenjem rešenja istog problema manje složenosti (dimenzije, veličine). To u praksi znači da rekurzivna implementacija algoritma podrazumeva (direktno ili indirektno) samopozivanje funkcije sa promenjenim argumentima. Moć rekurzije leži u tome što se rešenja teških problema mogu iskazati kroz jednostavne veze.

U nastavku ćemo kroz neke klasične primere videti kako možemo implementirati rekurziju u Haskelu. Primeri su poređani od jednostavnijih ka složenijim, i demonstriraju različite oblike rekurzije. Neke druge rekurzivne tehnike ćemo upoznati kasnije kroz Haskel.

6.1. Aritmetička suma

> **Zadatak 1.** Napisati funkciju `zbir :: Int -> Int` koja vraća zbir prvih `n` prirodnih brojeva.

Funkciju `zbir` je lako implementirati u nekom imperativnom jeziku pomoću jedne petlje. Međutim, u Haskelu ne postoji pojam petlje, te moramo iskoristiti tehniku rekurzije. To podrazumeva da pri nalaženju rešenja koristimo rešenje istog problema manje složenosti. Konkretno u ovom slučaju prilikom nalaženja zbira brojeva koristimo zbir manje brojeva. Ako napišemo izraz za zbir, rešenje će nam se samo ukazati:

```
zbir n = 1 + 2 + … + (n - 1) + n
```

Kao što vidimo, u zbiru prvih `n` prirodnih brojeva, pojavljuje se zbir prvih `n - 1` brojeva. Stoga se rešenje u kodu samo nameće:

```
zbir n = zbir (n - 1) + n
```

Dakle da bismo izračunali, na primer, zbir prvih 5 prirodnih brojeva, izračunaćemo zbir prva 4 prirodna broja i dodati `5` na taj zbir. Da bismo izračunali zbir prva 4 prirodna broja, izračunaćemo zbir prva tri prirodna broja i dodati `4` na taj zbir, itd.. U jednom trenutku ćemo stići do toga da nam je potrebna vrednost `zbir 0` (zbir nula prirodnih brojeva - prazan zbir). U tom slučaju, nećemo više koristiti vezu koju smo opisali, već ćemo prosto vratiti `0`. Takav slučaj nazivamo *baznim slučajem*, jer ne koristimo rekurzivnu definiciju.. Prema tome, funkcija za računanje zbira prvih `n` brojeva može se ovako zapisati:

```
zbir :: Int -> Int
zbir n
    | n == 0    = 0
    | otherwise = zbir (n - 1) + n
```

Ovde možete i koristiti *if then else* notaciju da biste definisali `zbir` u jednoj liniji.

Pogledajmo na primeru kako napisani program radi. Svaki korak u narednom nizu predstavlja ili izračunavanje ili poziv funkcije `zbir`:

$$\begin{aligned}
\texttt{zbir}\ 3 &= \texttt{zbir}\ (3-1)+3 \\
&= \texttt{zbir}\ 2+3 \\
&= (\texttt{zbir}\ (2-1)+2)+3 \\
&= (\texttt{zbir}\ 1+2)+3 \\
&= ((\texttt{zbir}\ (1-1)+1)+2)+3 \\
&= ((\texttt{zbir}\ 0+1)+2)+3 \\
&= ((0+1)+2)+3 = 6
\end{aligned}$$

Ipak, postoji jedan mali problem sa konstruisanom funkcijom: ako bismo potražili npr. `zbir (-3)`, funkcija bi ušla u *beskonačnu rekurziju*. Za računanje vrednosti `zbir (-3)` bilo bi potrebno izračunati `zbir (-4)`, a za `zbir (-4)` bi bilo potrebno izračunati `zbir (-5)`, i tako u nedogled. Da bismo sprečili beskonačnu

rekurziju, za sve brojeve `<= 0`, vratićemo `0` kao rezultat. Stoga definišemo funkciju `zbir` tako da vraća `0` za negativne argumente:

```
zbir :: Int -> Int
zbir n
    | n <= 0 = 0
    | otherwise = zbir (n - 1) + n
```

Zadatak 2. U matematici, faktorijel prirodnog broja $n!$ predstavlja proizvod svih prirodnih brojeva manjih ili jednakih sa n tj. $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$. Napisati funkciju `faktorijel` koja vraća faktorijel $n!$ prirodnog broja.

Rešenje. Možemo primetiti da se $n!$ može izračunati kao proizvod $(n-1)!$ sa n. Postupajući kao u prethodnom primeru dobijamo sledeći Haskel kod:

```
faktorijel' n
    | n <= 0 = 1
    | otherwise = faktorijel' (n - 1) * n
```

Zadatak 3. Napisati rekurzivnu funkciju koja računa sumu prvih $k+1$ članova aritmetičkog niza čiji je prvi član a, a korak (razlika između dva susedna člana niza) d. Tačnije napisati rekurzivnu funkciju koja računa sumu

$$a + (a+d) + (a+2d) + \cdots + (a+dk).$$

6.2. Par-nepar

Zadatak 4. Implementirati funkciju `paran :: Int -> Bool` koja proverava da li je prirodan broj paran. Pretpostaviti da će argument funkcije biti pozitivan broj.

Korišćenjem funkcije `mod`, možemo dati odmah sledeće rešenje

```
paran :: Int -> Bool
paran n = 0 == mod n 2
```

Izraz `0 == mod n 2` ima vrednost `True` ako je `n` paran broj, prema tome ovde nema potrebe da koristimo *if then else*.

Problem možemo rešiti i rekurzivno. Znamo da je broj `0` paran, i ta činjenica predstavlja bazu rekurzije. Za svaki drugi prirodan broj `n` znamo da je paran ako i samo ako `n - 1` nije paran. Stoga možemo dati ovakvu definiciju:

```
paran :: Int -> Bool
paran 0 = True
paran n = not (paran (n - 1))
```

Demonstrirajmo izvršavanje koda na jednom primeru:

$$\begin{aligned}\texttt{paran}\ 3 &= \texttt{not}\ (\texttt{paran}\ 2)\\ &= \texttt{not}\ (\texttt{not}\ (\texttt{paran}\ 1))\\ &= \texttt{not}\ (\texttt{not}\ (\texttt{not}\ (\texttt{paran}\ 0)))\\ &= \texttt{not}\ (\texttt{not}\ (\texttt{not}\ \texttt{True})) = \texttt{False}\end{aligned}$$

Primetimo da funkcija `paran` nije definisana za negativne brojeve.

> **Zadatak 5.** Izmeniti definiciju funkcije `paran` tako da i za negativne brojeve vraća ispravne rezultate.

> **Zadatak 6.** Definisati rekurzivnu funkciju `ostatak3 :: Int -> Int` koja određuje ostatak broja pri deljenju sa 3.

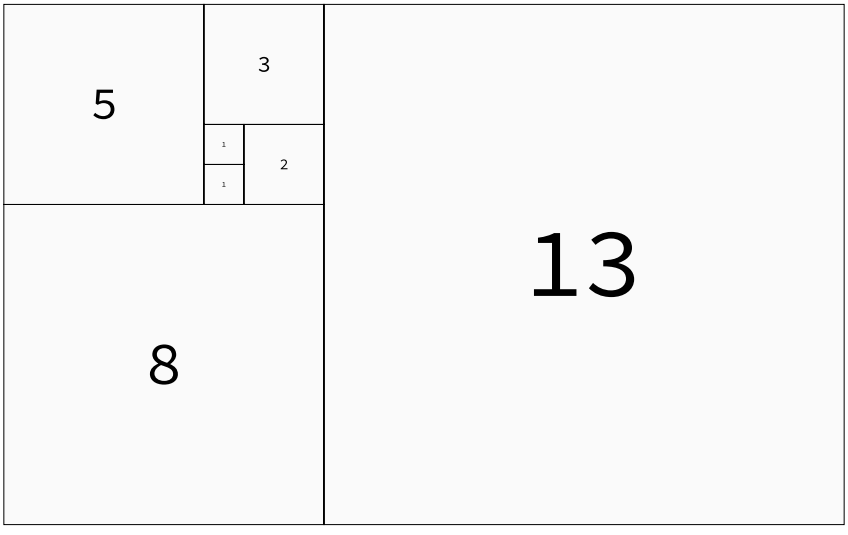
> **Zadatak 7.** Definisati rekurzivnu funkciju `ostatak :: Int -> Int -> Int` koja određuje ostatak prilikom deljenja prvog argumenta sa drugim argumentom.

6.3. Fibonačijev niz

Italijanski srednjovekovni matematičar Leonardo iz Pize je objavio knjigu *Liber Abaci*[54], priručnik za trgovce. U jednom od mnogobrojnih primera Fibonači je prezentovao račun koji se bavi rastom populacije zečeva na nekoj lokaciji koja do tada nije imala zečeve. Fibonači je ustanovio da je broj zečeva na kraju svakog meseca jednak broju zečeva na kraju prošlog meseca uvećanog za broj zečeva sa kraja pretposlednjeg meseca.

Danas definicija *Fibonačijevog niza* nema mnogo veze sa zečevima, a niz se konstruiše po sledećem principu: prva dva člana Fibonačijevog niza su 1 i 1. Svaki naredni član Fibonačijevog niza je zbir prethodna dva člana.

Po navedenom principu, treći član Fibonačijevog niza je 2 (jer je $2 = 1 + 1$), četvrti član je 3 (jer je $3 = 1 + 2$), peti član je 5 (jer je $5 = 2 + 3$), šesti član je 8 (jer je $8 = 3 + 5$), i tako dalje...



Niz kvadrata čije stranice odgovaraju članovima Fibonačijevog niza se može "upakovati" u spiralnu konfiguraciju kako je prikazano na ovoj slici. Ovakva konfiguracija se može neograničeno nastavljati.

[54] U istoriji matematike ova knjiga je ostala upamćena po tome što je evropljanima prezentovala arapski sistem brojeva. Do tada su u Evropi korišćeni rimski brojevi.

Konstruisanje Haskel funkcije `fib` koja za prosleđeni argument `n` vraća n-ti Fibonačijev broj (n-ti element Fibonačijevog niza) je neznatno složenije nego što su bili prethodni primeri. U ovom slučaju vrednost `fib n` zavisi od `fib (n - 1)` i `fib (n - 2)` te je stoga potrebno navesti dva bazna slučaja[55].

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

Aplikacija argumenta ima veći prioritet od aritmetičkih operacija, te se `fib (n - 1) + fib (n - 2)` interpretira kao `(fib (n - 1)) + (fib (n - 2))`. Iz istog razloga ne možemo pisati samo `fib n - 1 + fib n - 2`, jer se to interpretira kao `(fib n) - 1 + (fib n) - 2`.

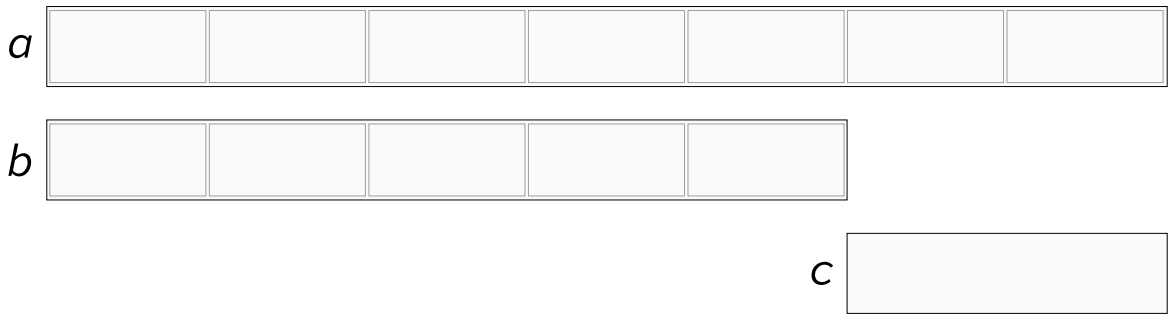
> **Zadatak 8.** Takozvani *Tribonačijev niz* se konstruiše na sledeći način: prva tri člana Tribonačijevog niza su 1. Svaki naredni član Tribonačijevog niza je zbir prethodna tri člana. Implementirati funkciju koja računa n-ti element Tribonačijevog niza.

6.4. Euklidov algoritam

Rekurzija može delovati komplikovano kada se prvi susretnemo sa njom, ali je ideja o rekurziji veoma prirodna i veoma stara. Još je Euklid u trećem veku pre nove ere u knjizi *Elementi* izneo nekoliko rekurzivnih postupaka. Najpoznatiji je svakako postupak za traženje najvećeg zajedničkog delioca, koji danas nosi naziv **Euklidov algoritam**.

Kao što samo ime kaže, *najveći zajednički delilac* (*NZD*) prirodnih brojeva a i b je prirodni broj d takav da deli a i b, i pritom je d najveći broj sa takvom osobinom.

> **Primer 1.** Najveći zajednički delilac brojeva 15 i 9 je 3. Najveći zajednički delilac brojeva 8 i 32 je 8.



Pretpostavimo da želimo da popločamo dve staze dužine a i b metara. Pritom, širina pločica sa kojima raspolažemo je jednaka širini naših staza a dužinu pločice možemo da biramo. Staze želimo da popločamo tako da obe staze budu popločane sa celim pločama, ali tako da koristimo što duže pločice. Takvo popločavanje je prikazano na slici (razmak između pločica je ilustrativnog karaktera - zbir dužina pločica je jednak dužini staze u oba slučaja). Upravo će *NZD* od a i b biti dužina tražene pločice. Primetimo da pločica koja ima traženu osobinu takođe će u potpunosti popločavati i stazu koja je "razlika" staza dužina a i b (to je staza dužine c na slici). Ovo trivijalno zapažanje je osnova Euklidovog algoritma.

Euklidov algoritam se zasniva na veoma jednostavnoj činjenici: Ako je d NZD brojeva a i b, pri čemu je $a > b$, tada je d NZD brojeva b i $a - b$. Zaista, d deli razliku $a - b$ jer deli i umanjenik i umanjilac, pa d

[55] Ako bi bio definisan samo jedan bazni slučaj, neka je to na primer $n = 1$, tada bi funkcija `fib` ušla u beskonačnu petlju. Zaista, za računanje `fib 2` bilo bi potrebno izračunati `fib 1` i `fib 0`. Međutim za računanje `fib 0` bilo bi potrebno izračunati `fib (-1)` i `fib (-2)` i tako dalje...

jeste zajednički delilac brojeva b i $a - b$. Broj d je najveći broj koji deli i b i $a - b$, jer ako bi postojao još veći broj d' koji deli i b i $a - b$ tada bi d' delio i brojeve b i $a = (a - b) + b$, a već znamo da je d najveći takav broj.

Dakle, ako je $a > b$ tada je dovoljno naći *NZD* od b i $a - b$. Ako je $a < b$ tada je po istom argumentu dovoljno naći *NZD* od a i $b - a$. A ako je $a = b$ tada je *NZD* od a i b jednak baš broju a, jer je svaki broj deljiv samim sobom i ni jedan veći broj ga ne deli. Rekurzivna implementacija je sada u potpunosti jasna:

```
nzd :: Int -> Int -> Int
nzd a b
    | a == b  = a
    | a >  b  = nzd (a - b) b
    | a <  b  = nzd a (b - a)
```

Primer 2. Nađimo *NZD* brojeva 15 i 9. Izraz `nzd 15 9` se svodi

```
nzd (15 - 9) 9 = nzd 6 9
```

koji se dalje svodi na

```
nzd 6 (9 - 6) = nzd 3 3 = 3
```

Primer 3. Nađimo *NZD* brojeva 8 i 32. Izraz `nzd 8 32` se svodi

```
nzd 8 (32 - 8) = nzd 8 24
```

koji se svodi na

```
nzd 8 (24 - 8) = nzd 8 16
```

koji se svodi na

```
nzd 8 (16 - 8) = nzd 8 8 = 8.
```

Primetimo da smo više puta za redom oduzimali isti broj (8) od 32.

Poslednji primer nam ukazuje da ponekad se može desiti da jedan argument umanjujemo više puta za istu vrednost. Broj takvih oduzimanja nije bitan, već samo ostatak koji se dobija na kraju. Da ne bismo neefikasno rekurzivno pozivali funkciju, možemo da podelimo argumente po modulu. Na primer, ako je $a = nb + r$ za neke prirodne brojeve n i r, pri čemu je $n \geq 1$ i $0 \leq r < b$, tada se oduzimanjem n puta vrednosti b od a dobija ostatak r. Stoga, najveći zajednički delilac brojeva a i b je najveći zajednički delilac brojeva b i r (jer je[56] $r = a - nb$). Ovo nas dovodi do malo drugačije i efikasnije implementacije.

Neka je r ostatak deljenja a sa b. Ako je r nula, to znači da je a deljivo sa b, pa sledi da je *NZD* ova dva broja baš b. U suprotnom, dovoljno je potražiti *NZD* brojeva b i r. Ovo nas dovodi do narednog koda:

```
nzd' :: Int -> Int -> Int
nzd' a b = if r == 0 then b else nzd' b r
    where r = mod a b
```

Primer 4. Nađimo još jednom *NZD* brojeva 8 i 32. Kako je `mod 8 32 = 8`, izraz `nzd' 8 32` se svodi na izraz

[56] Ustanovili smo da je *NZD* brojeva a i b isti kao *NZD* brojeva $a - b$ i b. Ali *NZD* brojeva $a - b$ i b je isti kao *NZD* brojeva $(a - b) - b$ i b koji je isti kao *NZD* brojeva $a - 3b$ i b, itd. sve do *NZD* brojeva $a - nb$ i b.

```
nzd' 8 8
```

koji se direktno svodi na vrednost 8.

Zadatak 9. Skakavac A se nalazi u koordinatnom početku brojevne prave, a skakavac B se nalazi u tački p, gde je p prirodan broj. Kretanje oba skakavca je ograničeno samo na pravu, i pritom A uvek skače m jedinica (levo ili desno), a B uvek skače n jedinica (takođe, levo ili desno). Brojevi m i n su takođe prirodni. Napisati funkciju

```
susret :: Int -> Int -> Int -> Bool
```

koja na osnovu tri prirodna broja p, m i n određuje da li je moguće da se skakavci ikad susretnu u jednoj tački.

Rešenje. Oba skakavca se mogu naći samo u celobrojnim tačkama. Pretpostavimo da su se nakon μ skokova skakavaca A i ν skokova skakavaca B, dva skakavca susrela u celom broju z. Tada važi $z = \mu m = p + \nu n$, odnosno

$$p = \mu m - \nu n.$$

Neka je d *NZD* brojeva m i n. Ako važi navedena jednakost, tada d mora deliti broj p. Obratno, ako d deli p tada Bezuov stav garantuje gornju jednakost za neke μ i ν. Dakle, skakavci se mogu susresti, ako i samo ako d deli p.

Neophodno je razmotriti još specijalne slučajeve. Ako su i m i n jednaki nuli, tada se skakavci mogu susresti samo ako su već u istoj tački. Ako je samo m jednak nuli, tada se skakavci mogu susresti samo ako n deli početnu razdaljinu p. I slično važi ako je samo n jednak nuli.

```
susret :: Int -> Int -> Int -> Bool
susret p 0 0 = p == 0
susret p m 0 = mod p m == 0
susret p 0 n = mod p n == 0
susret p m n = mod p (gcd n m) == 0
```

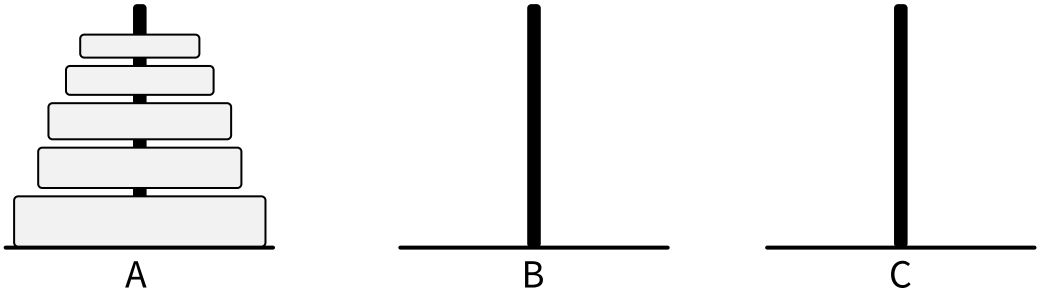
6.5. Hanojske kule

Francuski matematičar Edvard Lukas[57] je krajem devetnaestog veka smislio igru koja je danas poznata kao *Hanojske kule*[58].

Igru igra jedan igrač na sledeći način: Postavljena su tri vertikalna štapa koja označavamo sa A, B i C, i dato je n diskova različite veličine. Diskovi su probušeni u centru tako da se štap može provući kroz rupu u disku. Na početku igre, diskovi su složeni po veličini na štapu A, od najmanjeg na vrhu do najvećeg na dnu (videti narednu ilustraciju). Igrač u svakom potezu može napraviti jedan potez koji se sastoji samo od pomeranja jednog diska sa jednog štapa na drugi štap. Pritom, disk koji se pomera uvek mora da bude gornji disk na nekom štapu i ne sme se postaviti na manji disk. Cilj je pomeriti diskove tako da se svi diskovi nalaze na štapu C.

[57] Inače, upravo je Lukas dao ime Fibonačijevom nizu kog smo obradili gore.

[58] Iako su od samog početka igru pratile mistične priče o poreklu u budističkim hramovima Dalekog istoka, za sada nema nikakvog dokaza o takvom poreklu.



Za osobu nije teško da sama dođe do rešenja nakon malo početnog igranja. Ali je na prvi pogled veoma teško napisati program koji formira rešenje za proizvoljan broj diskova.

Prvo ćemo utvrditi kako želimo da prezentujemo rešenje, i koji ulazni parametri su potrebni. Rešenje možemo predstaviti kao listu poteza, od kojih je svaki potez uređen par karaktera koji označavaju štap sa kog i štap na koji prebacujemo disk. Na primer, uređen par `('A', 'B')` predstavlja potez u kom se sa prvi disk sa štapa A prebacuje na štap B. Sa druge strane, rešenje zadatka zavisi samo od broja diskova n. Stoga, naš zadatak je da napišemo funkciju

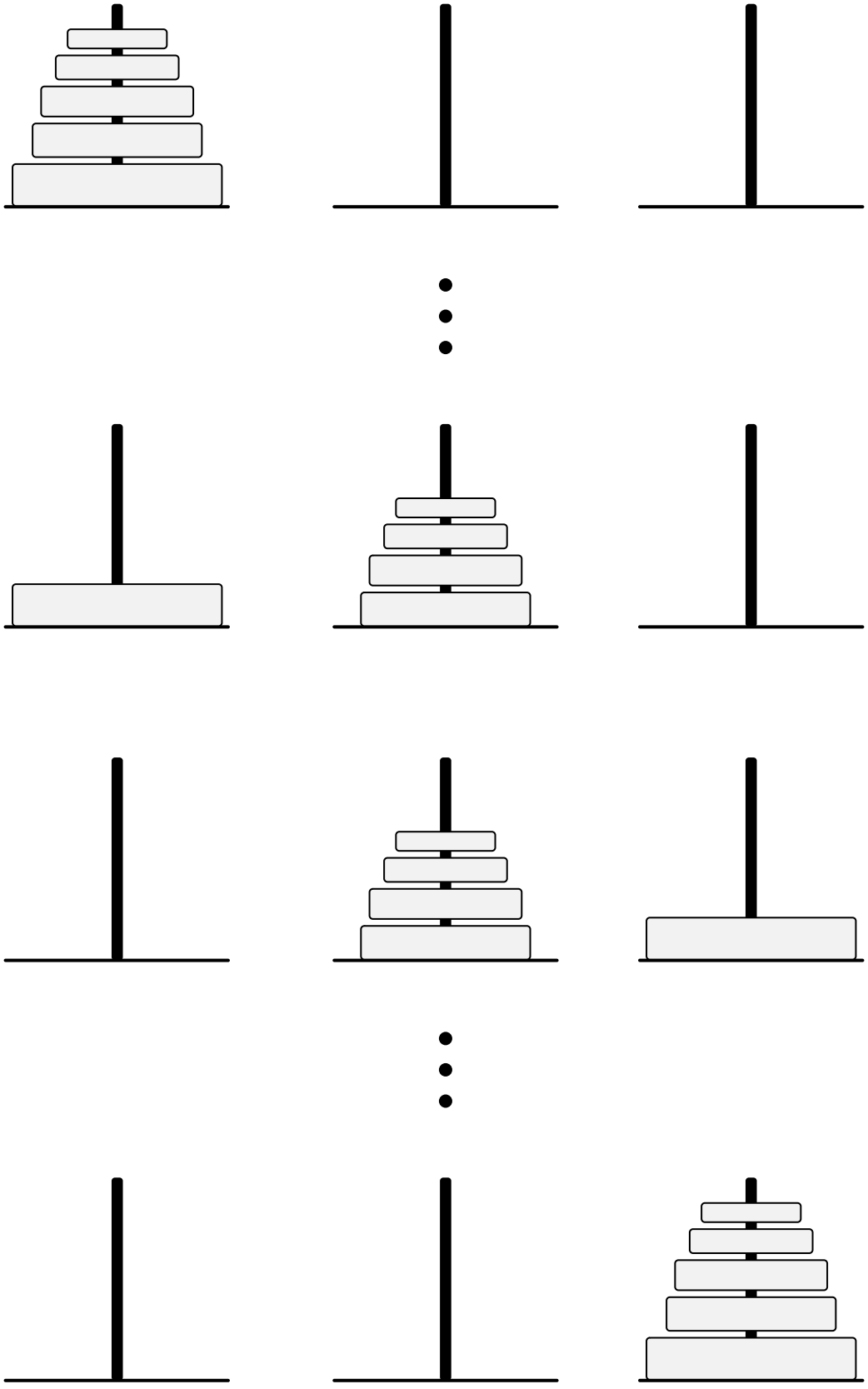
```
kule :: Int -> [(Char, Char)]
```

koja pronalazi jedno rešenje[59].

> **Primer 5.** Lista
>
> ```
> [('A', 'B'), ('A', 'C'), ('B', 'C')]
> ```
>
> predstavlja rešenje slučaja kada za $n = 2$.

Naravno, problem ćemo rešiti rekurzivno. Kako problem suštinski zavisi samo od broja diskova, rekurziju ćemo vršiti po n. U najjednostavnijem slučaju, kad je $n = 1$, jasno je da je rešenje jednočlana lista `[('A', 'C')]`, tj. par koji se sastoji od početnog i krajnjeg štapa. Pretpostavimo zato da je $n > 1$ i da znamo da rešimo slučajeve za manje diskova. Tada možemo postupiti na sledeći način: prvih $n - 1$ diskova ćemo prebaciti sa početnog na pomoćni (B) štap, zatim ćemo prebaciti najveći disk sa početnog na krajnji, i na kraju ćemo pomeriti $n - 1$ diskova sa pomoćnog na krajnji štap. Ova strategija je prikazana na narednoj ilustraciji.

[59] Primetimo da rešenje nije jedinstveno. Igru možemo odigrati na mnogo načina.



Kao što vidimo u rekurzivnom koraku $n-1$ diskova prebacujemo prvo sa štapa A na štap B. Stoga za rekurzivni poziv, osim broja diskova, mora biti navedeno i sa kog štapa i na koji štap se diskovi prebacuju. Naravno, rešenje ne može koristiti samo dva štapa[60], već se mora iskoristiti i onaj treći štap kao pomoćni.

[60] Osim u trivijalnom slučaju kada se prebacuje jedan disk.

Stoga funkcija za rekurzivno pronalaženje rešenja uzima četiri parametra: broj diskova, početni štap, krajnji štap i pomoćni štap. Ova rekurzivna funkcija će imati potpis

```
pomeri :: Int -> Char -> Char -> Char -> [(Char, Char)].
```

Ako je funkcija `pomeri` implementirana, tada "glavnu" funkciju `kule` možemo jednostavno implementirati:

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

Rešenje početnog problema su potezi koji pomeraju n diskova sa A na C.

Potrebno je još implementirati funkciju `pomeri`, ali to je sasvim jednostavno ako uzmemo u obzir opis rešenja koje smo izložili:

```
pomeri :: Int -> Char -> Char -> Char -> [(Char, Char)]
pomeri 1 pocetak kraj _       = [(pocetak, kraj)]
pomeri n pocetak kraj pomocni = p1 ++ [(pocetak, kraj)] ++ p2
    where
        p1 = pomeri (n - 1) pocetak pomocni kraj
        p2 = pomeri (n - 1) pomocni kraj pocetak
```

Zadatak 10. Od koliko poteza se sastoji rešenje koje daje ovaj algoritam? Možete li matematičkom indukcijom da dokažete pravilnost?

6.6. Složenost rekurzivnih algoritama

Lepota rekurzivnih algoritama leži u tome što programer na jednostavan način može implementirati komplikovane funkcionalnosti. Programiranje baznog slučaja i rekurzivnog koraka su dovoljni da bi se i naizgled nemogući problemi jednostavno rešili. Primeri i zadaci koji su navedeni u ovoj lekciji bi trebalo da demonstriraju tu elegantnost.

Ipak, lepota rekurzivnih funkcija dolazi po ceni da su one često manje efikasne od odgovarajućih nerekurzivnih funkcija.

Primer 6. Neka je $S_n = 1 + \cdots + n$, suma koju smo posmatrali na početku ove lekcije. Ako članove sume raspišemo u dva reda, pri čemu ćemo u drugom redu obrnuti poredak sabiraka, dobijamo šemu:

$$\begin{array}{cccccc} 1 & 2 & \cdots & n-1 & n \\ n & n-1 & \cdots & 2 & 1 \end{array}$$

Odavde vidimo da u ovoj sumi imamo n parova brojeva koji odgovaraju kolonama, pri čemu je zbir svakog para $n+1$. Stoga je $2S_n = n(n+1)$, pa je

$$S_n = \frac{n(n+1)}{2}.$$

Dobijeni rezultat možemo iskoristiti za alternativnu implementaciju funkcije `zbir`:

```
zbir' :: Int -> Int
zbir' n = div (n * (n+1)) 2
```

Možemo ignorisati vrednosti funkcije za negativne argumente, pošto ionako traženje zbira negativno mnogo brojeva nema mnogo smisla.

Navedena alternativna definicija nije mnogo kraća od rekurzivne, ali je značajno brža. Vreme potrebno za računanje vrednosti `zbir n` je proporcionalno sa `n`: za `zbir n` potrebno je izračunati `zbir (n-1)`, a za računanje te vrednosti potrebno je izračunati `zbir (n-2)`, i tako dalje sve do `zbir 1`. Sa druge strane, vreme potrebno za izračunavanje `zbir' n` ne zavisi od `n`, jer je funkcija sačinjena od konstantnog broja računskih operacija koje se izvršavaju u nekoliko procesorskih taktova.

Primer 7. Gore su navedene dve definicije funkcije koja računa parnost broja. Prva funkcija, funkcija koja koristi `mod` funkciju, se izvršava u konstantnom vremenu, jer procesor u jednom ciklusu može da podeli dva broja sa ostatkom. Naredna, rekurzivna, definicija funkcije zahteva `n` rekurzivnih poziva za izračunavanje izraza `paran n`.

Primer 8. Metodama linearne algebre moguće je dokazati da je n-ti član F_n Fibonačijevog niza dat sa

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

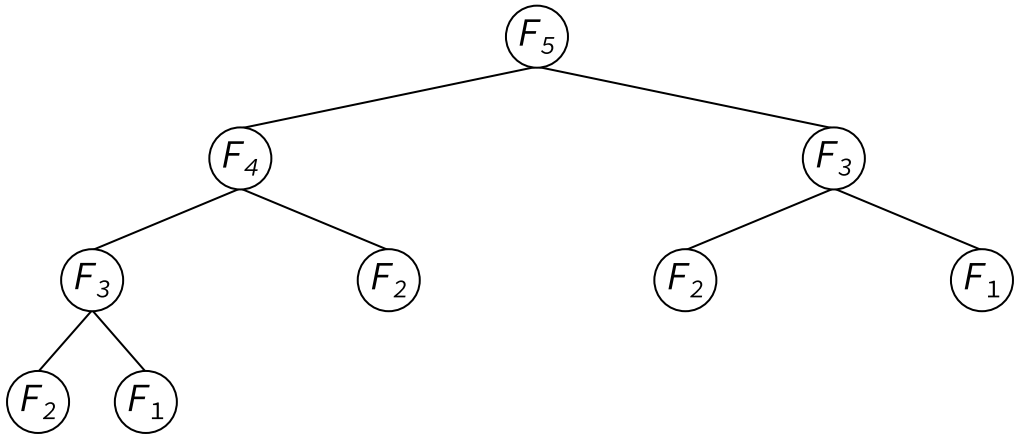
Ova formula se može iskoristiti za nerekurzivnu definiciju funkcije `fib`:

```
fib' :: Int -> Int
fib' n = round ((a^n - b^n)/t)
    where t = sqrt 5
          a = (1 + t)/2
          b = (1 - t)/2
```

Ugrađena funkcija `round` zaokružuje `Float` broj u `Int`.

Navedena definicija omogućava da se `n`-ti član Fibonačijevog niza izračuna u konstantnom broju koraka (broj koraka ne zavisi od veličine parametra `n`).

Sa druge strane, definicija koju smo dali ranije zahteva značajno više koraka pri izvršavanju. Za izračunavanje `fib n` potrebno je izračunati `fib (n-1)` i `fib (n-2)`. Za svaku od navedenih vrednosti potrebno je izračunati po još dve vrednosti i tako dalje. Odavde sledi da je broj koraka potreban za izračunavanje `fib n` proporcionalan sa 2^n. Primetimo da se vrednosti računaju više puta: na primer `fib (n-2)` je potrebno izračunati za `fib n` ali i za `fib (n-1)`.



Šematski prikaz rekurzivnog računanja vrednosti F_5.

Kao što vidimo u prethodnim primerima, implementacija nerekurzivnih funkcija je ponekad značajno složenija nego implementacija rekurzivnih jer su neophodna dodatna znanja iz matematike i teorije

algoritama. Sa druge strane, nerekurzivne funkcije mogu posedovati znatno bolju vremensku i prostornu složenost. Prema tome, na programeru je da odluči na koji način će pristupiti problemu: da li je prihvatljivo žrtvovati performanse zarad jednostavnije definicije? Odgovor na ovo pitanje je nemoguće dati jer zavisi od konteksta. U svakom slučaju, tehnika rekurzije je odličan izbor za prvi korak pri rešavanju problema koji deluje značajno teško. Često kada se algoritam savlada rekurzijom, nerekurzivno rešenje postane mnogo jasnije.

6.7. Totalne i parcijalne funkcije

Po matematičkoj definiciji funkcije, vrednost $f(x)$ mora biti definisana za svako x koje pripada domenu funkcije f. Tako na primer, funkcija $f(x) = 1/x$ ne može za domen imati ceo skup realnih brojeva $\mathbb{R}$, već u najboljem slučaju[61] $\mathbb{R} \setminus \{0\}$. Slično, funkcija korenovanja $s(x) = \sqrt{x}$ može za domen imati samo podskup skupa pozitivnih brojeva[62].

U računarstvu je malo drugačije. Funkcija tipa `A -> B` u Haskelu ne mora biti definisana za sve vrednosti tipa `A`. To smo videli na primeru prve definicije funkcije koja sabira aritmetičku sumu $1 + \cdots + n$:

```
zbir :: Int -> Int
zbir n
    | n == 0    = 0
    | otherwise = zbir (n - 1) + n
```

Za negativne argumente funkcija `zbir` ulazi u beskonačnu petlju, pa samim tim izraz `zbir x` nije definisan za svaku vrednost `x :: Int`.

Funkciju koja nije definisana za svaku vrednost domena, nazivamo **parcijalna funkcija**, dok funkciju koja je definisana za svaku vrednost domena nazivamo **totalna funkcija**. Prema tome, pojam funkcija odgovara pojmu totalnih funkcija u računarstvu.

Rekurzivne funkcije su čest primer funkcija koje nisu totalne, ali ne i jedini[63]. Najjednostavniji način za definisanje nerekurzivne parcijalne funkcije je upotrebom podudaranja oblika (ili ograđenih izraza), pri čemu nisu obrađeni svi mogući oblici argumenta:

```
p :: Int -> Int
p 0 = 1
```

Funkcija `p` je definisana samo za argument `0`.

```
ghci> p 10
*** Exception: <interactive>:3:1-16: Non-exhaustive patterns in function p
```

Pokretanje `p x` za bilo koju vrednost `x` različitu od `0` dovodi do izuzetka.

Postavlja se pitanje, zašto jezik poput Haskela dozvoljava da radimo sa parcijalnim funkcijama? Zašto se umesto toga ne ograničimo na totalne funkcije kao u matematici? Razlog tome je što je nemoguće kreirati programski jezik koji ne bi sadržao parcijalne funkcije a koji bi i dalje bio dovoljno ekspresivan za izražavanje svakog algoritma.

Iako sistem tipova u Haskelu može značajno olakšati rad sa parcijalnim funkcijama (videćemo ovo kroz kasnije lekcije), programer uvek mora biti svestan do kakvih problema može doći. Što se tiče rekurzivnih

[61] Naravno mi možemo definisati ovu funkciju na bilo kom podskupu od $\mathbb{R}$ koji ne sadrži nulu. Upravo zbog toga smatramo da je funkcija u potpunosti određena kada su navedeni domen, kodomen, i pravilo pridruživanja.

[62] Ovde u potpunosti ignorišemo kompleksne brojeve.

[63] Ranije smo videli funkcije koje nisu totalne, a to su funkcije za rad sa listama `head`, `tail`, `init` i `last`. Ako se prazna lista prosledi bilo kojoj od navedenih funkcija doći će do izuzetka. U narednoj lekciji uverićemo se da funkcije `head` i `tail` nisu rekurzivne

algoritama, bitno je uočiti slučajeve kada rekurzivni algoritam može ući u beskonačnu petlju, a zatim izmenom tog algoritma obezbediti da do beskonačne petlje nikad ne dođe.

6.8. Zadaci

Zadatak 11. Napisati funkciju koja određuje n-ti stepen broja.

Zadatak 12. Napisati funkciju koja određuje zbir neparnih cifara prirodnog broja.

Zadatak 13. Napisati funkciju koja određuje količnik dva prirodna broja na k decimala.

Zadatak 14. Napisati funkciju koja određuje binomni koeficijent $\binom{n}{k}$.

Zadatak 15. Spratovi jedne zgrade se kreče u zeleno, plavo ili crveno. Svaki sprat se kreči u jednu od te tri boje. Na koliko načina je moguće okrečiti zgradu, ako važi pravilo da se dva susedna sprata ne smeju okrečiti istom bojom.

Zadatak 16. Za proizvoljan prirodan broj n definišemo njegov Kolacov niz (eng. *Collatz sequence*) na sledeći način: $x_0 = n$, $x_{m+1} = x_m/2$ ako je x_m parno, odnosno $x_{m+1} = 3x_m + 1$ ako je x_m neparno. Pritom ako je $x_{m+1} = 1$ niz se prekida. Na primer Kolacov niz za $n = 12$ je $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$. Pretpostavlja se da će se Kolacov niz svakog prirodnog broja eventualno prekinuti, tj. da će se pojaviti 1 u njemu. Naći prirodan broj manji od 10000 koji ima najduži Kolacov niz.

Zadatak 17. *Metoda polovljenja intervala* je numerička metoda određivanja korena neprekidne funkcija tipa $\mathbb{R} \to \mathbb{R}$. Posmatrajmo interval $[a, b]$. Ako za funkciju f važi da je $f(a) < 0$ i $f(b) > 0$ (ili da je $f(a) > 0$ i $f(b) < 0$) za neke brojeve $a < b$, tada zbog neprekidnosti znamo da postoji x, $a < x < b$, takvo da je $f(x) = 0$. Uzmimo $c = (a + b)/2$. Ako je $f(c) \neq 0$ postupak možemo da ponovimo za jedan od intervala $[a, c]$ ili $[c, b]$ (naravno, ako je $f(c) = 0$, postupak prekidamo). Na ovaj način dobijamo niz intervala čija dužina se prepolovljava, i za koje znamo da sadrže barem jedno rešenje jednačine $f(x) = 0$. Stoga, u n koraka možemo odrediti koren jednačine sa greškom manjom od $(b - a)/2^n$. Koristeći ovaj metod, naći koren polinoma $p(x) = 2x^3 - 4x^2 - 6x + 2$ na intervalu $[1, 4]$ sa greškom ne većom od 0.001.

7. Polimorfnost

Svaki programski jezik je osmišljen sa ciljem da olakša proces programiranja računara, koliko god ovaj iskaz delovao očigledno. Zaista, iako se svaki program može napisati u mašinskom jeziku, programeri biraju programske jezike zbog osobina tih jezika koje omogućuju efikasnije programiranje. Osobine koje olakšavaju programiranja su zapravo *apstrakcije* kojima se nebitne pojedinosti skrivaju i samim tim obezbeđuje lakše izražavanje suštinskih ideja o programu[64].

Jednoj vrsti apstrakcija posvetićemo ovu lekciju, a na samom početku ćemo pokušati da motivišemo razloge za uvođenje takvih apstrakcija.

U prethodnim lekcijama naučili smo kako da definišemo funkcije, i upoznali smo se sa pojmom tipa funkcije. Programi koje smo pisali su bili mali i sačinjeni samo od nekoliko definicija funkcija. Ako bismo radili na velikim programima (kao što su *web* serveri, desktop aplikacije, itd.) uvideli bismo jedan jednostavan ali naporan problem. Naime, često bismo pisali gotovo iste funkcije koje imaju malo različite tipove.

Primer 1. Zamislimo da je potrebno implementirati funkciju koja izražava određenu numeričku zakonitost (nebitno kakvu) po formuli koja je eksperimentalno dobijena

$$y = f(x) = x \cdot x + x.$$

Ako očekujemo da će vrednosti x i y biti tipa `Int`, tada možemo napisati sledeću definiciju

```
f :: Int -> Int
f x = x*x + x
```

Sa druge strane, ako očekujemo da će x i y biti tipa `Float`, tada možemo napisati sledeću definiciju

```
f :: Float -> Float
f x = x*x + x
```

Kako ne možemo imati dve definicije istog imena (a pogotovo dve definicije istog imena sa različitim tipovima), problem nastaje ako želimo da u kodu definišemo obe. Tada, moramo odabrati novo ime za jednu od funkcija (npr. `f` i `fFloat`). Ako želimo da dodamo odgovarajuću funkciju i za tip `Double`, i toj funkciji moramo dati novo ime.

Opisano rešenje navedenog problema je jednostavno ali dovodi do drugih problema:

1. Imena funkcija postaju duža, teža za pamćenje i za pisanje. Problem je posebno izražen kod funkcija više promenljivih.
2. Ako se definicija promeni (npr. ako se gorenavedena formula promeni zbog preciznijih eksperimentalnih rezultata), tada je neophodno svaku od funkcija posebno izmeniti, što predstavlja veliki prostor za greške (i to one greške koje sistem tipova ne može otkriti).

Prethodni primer je trivijalan, ali demonstrira probleme koji se sreću na iole većim projektima bez obzira na programski jezik. Srećom, mnogi jezici su našli rešenja za navedene probleme, pa tako i Haskel. U Haskelu, moguće je definisati vrednosti (posebno, funkcije) za koje deluje da ne pripadaju jednom određenom tipu, već čitavoj kolekciji tipova. Ovakvi tipovi se nazivaju **polimorfni tipovi**[65].

[64] Na primer, u mašinskom kodu koriste se različite instrukcije za sabiranje različitih vrsta brojeva (celobrojnih vrednosti, brojeva u pokretnom zarezu, itd.). Sa druge strane, u gotovo svakom programskom jeziku sabiranje svih tipova brojeva se izražava sa operatorom `+`. Na taj način, programer se može posvetiti zapisivanju matematičkih izraza, dok kompajler obezbeđuje da se ti matematički izrazi prevedu u ispravan niz instrukcija.

[65] Naziv potiče od grčkih reči *πολύς* (mnogo) i *μορφή* (oblik). Prema tome polimorfni tipovi su oni tipovi koji imaju više oblika.

> **Primer 2.** Do sada smo već imali prilike da se susretnemo sa polimorfnom vrednošću (vrednošću koja ima polimorfni tip). To je bila prazna lista `[]`. Za ovu vrednost možemo smatrati da pripada svakom tipu lista. U nastavku ćemo videti kog je tačno tipa `[]`.

> **Primer 3.** Numerički literal (poput npr. `2`) može predstavljati vrednost tipa `Int`, `Integer`, `Float` ili `Double`.

Polimorfnost se u Haskelu obezbeđuje na dva načina: kroz parametarski polimorfizam i kroz *ad-hoc* polimorfizam.

7.1. Parametarski polimorfizam

Tipska promenljiva (ili još **tipski parametar**), je ime koje predstavlja proizvoljan tip (eventualno iz neke kolekcije tipova). Za razliku od tipova, tipske promenljive zapisuju se sa početnim malim slovom, a najčešće se ime tipske promenljive sastoji samo od jednog slova. Ako tipska promenljiva predstavlja svaki mogući tip (a ne neku specifičnu kolekciju tipova), tada za tu tipsku promenljivu kažemo da je **slobodna**. Kasnije ćemo objasniti promenljive koje nisu slobodne.

> **Primer 4.** U tipu `a -> Int`, ime `a` predstavlja slobodnu tipsku promenljivu. Prema tome, navedeni tip može da predstavlja bilo koji tip oblika `A -> Int`, gde je `A` neki određen tip. Na primer: `Bool -> Int`, `Int -> Int`, `[Char] -> Int`, `(Bool -> Char) -> Int`, i beskonačno mnogo drugih mogućnosti...

Parametarski polimorfni tipovi su oni tipovi koji sadrže *slobodne tipske promenljive*.

Najjednostavniji primer vrednosti koja poseduje parametarski polimorfan tip, je konstantna funkcija poput naredne:

```
konstantnoTačno :: a -> Bool
konstantnoTačno = \x -> True
```

Navedenom definicijom, kao[66] da je definisano beskonačno mnogo funkcija sa različitim domenima: za svaki tip `T` po jedna funkcija tipa `T -> Bool`. Sve te funkcije nazivamo istim imenom `konstantnoTačno`. Kada u Haskel programu primenimo funkciju `konstantnoTačno` na neku određenu vrednost, tada tipska promenljiva `a` u tipu funkcije preuzima tip vrednosti na koju primenjujemo funkciju. Na primer, ako napisemo `konstantnoTačno 'a'` tada možemo smatrati da ime `konstantnoTačno` u navedenom izrazu predstavlja funkciju tipa `Char -> Bool` (jer je samo funkciju tog tipa moguće primeniti na vrednost tipa `Char`).

Nešto složeniji primer funkcije sa polimorfnim tipom je identička funkcija, definisana u Prelidu kao `id`:

```
id :: a -> a
id = \x -> x
```

Tip identičke funkcije sadrži tipsku promenljivu i u domenu i u kodomenu. Kad god se u tipu na više mesta pojavljuje promenljiva sa istim imenom, tada sva pojavljivanja te promenljive određuju isti tip. Stoga, funkcija `id` predstavlja funkciju tipa `Int -> Int` ili funkciju tipa `[Char] -> [Char]` i tako dalje, ali ne može predstavljati funkciju tipa `Bool -> Float`. Sam tip `a -> a` ukazuje na to da domen i kodomen funkcije moraju biti isti tipovi, prema tome `id True` ili `id False` mogu predstavljati samo vrednosti tipa `Bool`.

[66] Ističemo reč *kao*. Pomenuta definicija definiše samo jednu funkciju čiji je tip polimorfan. Taj tip se može konkretizovati na beskonačno mnogo načina, ali je funkcija samo jedna.

Prethodno navedene funkcije su bile trivijalne. Ipak, polimorfne funkcije su izuzetno korisne, i mi smo ih zapravo već koristili.

Funkciju `head` (koja vraća prvi element prosleđene liste), ima smisla primenjivati na liste tipa `[Int]`, `[Char]`, `[Bool]` itd... Prema tome, umesto da se za svaki tip liste definise posebna funkcija, definisana je jedinstvena funkcija `head` sa polimorfnim tipom `[a] -> a`:

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

Podudaranjem oblika liste, možemo rastaviti listu na glavu i rep.

U ovoj funkciji, vidimo da se tipska promenljiva pojavljuje unutar zagrada `[]`. Prema tome `[a]` ne predstavlja svaki tip, već predstavlja svaki tip lista. Npr. tip `Int` nije oblika `[a]`, i stoga `head 3` nije dobro tipiziran izraz. Sa druge strane, ako `head` primenimo na `[True, True, False]` tada Haskel "poklapa" domen `[a]` sa tipom `[Bool]`, odakle sledi da `a` mora biti `Bool`. Prema tome, `head [True, True, False]` je vrednost tipa `Bool`.

I ostale funkcije koje smo do sada koristili za rad sa listama (poput `tail`, `init`, `last`, `null`) imaju polimorfne tipove. Samim tim te funkcije se mogu koristiti za listom bilo kog tipa.

> **Zadatak 1.** Kog tipa je funkcija `null` koja određuje da li je lista prazna?
>
> *Rešenje.* Funkcija `null` se može primeniti na listu bilo kog tipa, te je njen domen `[a]`. Rezultat provere je logička vrednost. Stoga je
>
> `null :: [a] -> Bool`.

Nije nužno da vrednost polimorfnog tipa bude funkcija.

> **Primer 5.** prazna lista `[]` je sama za sebe vrednost tipa `[a]`, i samim tim predstavlja praznu listu bilo kog tipa. Ali to ne znači da, na primer, prazne liste u izrazima `[] ++ [True]` i `[] ++ ['a']`, predstavljaju iste vrednosti. Jedna od navedenih praznih lista je tipa `[Bool]` a druga `[Char]`. Ove dve vrednosti nisu iste, ali poseduju isto ime[67].

Iz prethodnih redova vidimo da polimorfne vrednosti poprimaju različite tipove u zavisnosti od izraza u kojima se nalaze (otuda i naziv *poli-morfne*).

Ponekad nije dovoljno da tip poseduje samo jednu promenljivu. Ako se u tipu nalaze više različitih promenljivih, tada su te promenljive nezavisne.

> **Primer 6.** Funkcija `fst` uzima proizvoljan uređen par i vraća prvu koordinatu. Funkcija `fst` mora da nam vrati prvu koordinatu para `(True, 2) :: (Bool, Int)` ili `("Pi", 3.141) :: ([Char], Float)` ili bilo kog drugog uređenog para. Stoga domen funkcije `fst` moramo zapisati kao `(a, b)`, a samim tim je kodomen `a`, odnosno važi `fst :: (a, b) -> a`[68]. Ilustrovano na konkretnom slučaju, ako `fst` primenimo na `(2, 'a') :: (Int, Char)` tada se `fst` ponaša kao funkcija tipa `(Int, Char) -> Int`.
>
> Naravno, tip funkcije `snd` je `(a, b) -> b`. U ovom tipu, tipska promenljiva `b` vezuje tip druge koordinate sa tipom povratne vrednosti (tj. kodomenom).

> **Zadatak 2.** Kog tipa je funkcija `reverse` koja obrće listu?

[67] Kao što dve različite osobe mogu da dele isto ime

[68] Imena tipskih promenljiva su proizvoljna, te smo mogli i da napišemo `fst :: (p, q) -> p`.

Rešenje. Funkcija `reverse` uzima listu bilo kog tipa i vraća listu *istog* tipa. Prema tome tip ove funkcije je `[a] -> [a]`. Tip funkcije `reverse` nije `[a] -> [b]` jer tipske promenljive `a` i `b` mogu predstavljati različite tipove.

Zadatak 3. Implementirati funkciju koja vraća prvu koordinatu proizvoljne uređene trojke.

Rešenje. Proizvoljna uređena trojka ima tip oblika `(A, B, C)` gde su `A`, `B`, `C` neki proizvoljni tipovi. Stoga polimorfni tip koji predstavlja svaku uređenu trojku je `(a, b, c)`. Podudaranjem oblika možemo konstruisati narednu funkciju:

```
fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x
```

Pošto ne koristimo vrednosti druge i treće koordinate, te vrednosti ne moramo da vezujemo novim imenom, već možemo iskoristi džoker.

Zadatak 4. Implementirati funkciju

```
zameni :: (a, b) -> (b, a)
```

koja menja mesta koordinatama uređenog para.

Zadatak 5. Implementirati funkciju `razmeni :: (a, b) -> (c, d) -> ((a, d), (c, b))` koja razmenjuje koordinate dva uređena para.

Iz prethodnih primera , vidimo da su tipovi sa slobodnim tipskim promenljivama izuzetno korisni. Mnoge funkcije iz prelida koje smo već koristili su parametarski polimorfne. Ipak, tehnika parametarskog polimorfizma i dalje ne rešava konkretan problem koji smo opisali na početku lekcije. Tada smo hteli da implementiramo funkciju koja bi prihvatala razne numeričke tipove. Tu funkciju ne možemo definisati sa sledećim kodom, koliko god to bilo privlačno:

```
f :: a -> a
f x = 253 * x + 17
```

Prethodna definicija nije dobra, iz prostog razloga zato što tipska promenljiva `a` može predstaviti svaki tip. Za tipove poput `Char` ili `Bool`, navedena definicija nema smisla jer se vrednosti tog tipa ne mogu sabirati niti množiti brojevima. Stoga bi bilo korisno kada bismo ograničili tipsku promenljivu `a` samo na one tipove koji dozvoljavaju sabiranje i množenje. Upravo ćemo to prikazati u nastavku.

7.2. *Ad-hoc* polimorfizam

Slobodne tipske promenljive mogu predstavljati bilo koji tip. Ponekad je poželjno ograničiti tipsku promenljivu na neku specifičnu kolekciju tipova. Upravo takva ograničenja se izražavaju kroz klase tipova.

Tipska klasa (ili skraćeno klasa[69]) je kolekcija tipova koji dozvoljavaju pozivanje određenih funkcija nad tipovima te klase. Često te funkcije nazivamo **metode klase**. Ime konkretne klase uvek počinje velikim slovom. Za svaki tip koji pripada nekoj klasi kažemo da je **instanca** te klase.

[69] Kad god govorimo o klasama u Haskelu uvek mislimo na tipske klase. Pojam tipske klase nema nikakve veze sa pojmom *klase* koji se koristi u objektno orijentisanim jezicima.

Primer 7. U Haskelu je definisana klasa `Show` koja poseduje metodu `show`. Tipovi poput `Int` ili `Bool` pripadaju ovoj klasi, i nad njima se moze pozivati funkcija `show` koja vrednost nekog tipa predstavlja kao nisku.

```
ghci> show True
"True"
ghci> show 2
"2"
```

Sa druge strane, bilo koji tip funkcija (npr. tip `Int -> Bool`) ne pripada ovoj klasi, te izraz poput `show (\x -> x == 0)` nije dobro tipiziran.

Primer 8. Ugrađena klasa `Num` predstavlja sve tipove čije vrednosti se mogu sabirati, oduzimati i množiti. Između ostalog, tipovi `Int`, `Integer`, `Float` i `Double` pripadaju ovoj klasi.

Za razliku od vrednosti koja pripada samo jednom tipu, tipovi se mogu istovremeno nalaziti u više[70] klasa (ili nijednoj). Činjenicu da neki tip `T` pripada klasi `C`, označavamo sa `C T`.

Korišćenjem klasa, moguće je ograničiti tipsku promenljivu na neku određenu klasu. **Klasno ograničenje** kojim se tipska promenljiva `a` ograničava na tipove klase `C` se zapisuje kao `C a`. Ovakva ograničenja se razdvajaju od samog tipa sa strelicom `=>`. U slučaju da postoji vise ograničenja, tada se ta ograničenja navode u obliku uređene n-torke.

Primer 9. Tip `Show a => a` označava sve tipove koji pripadaju klasi `Show`.

Primer 10. Tip `Show a => a -> [Char]` je tip polimorfne funkcije čiji domen je neki tip klase `Show`, a kodomen tip niska.

Funkcija `show` koja vrednost preslikava u nisku ima upravo ovaj tip.

Primer 11. Tip `(Num a, Show b) => a -> b` označava tip polimorfne funkcije čiji domen je neki tip iz klase `Num`, a kodomen neki tip iz klase `Show`.

Klase su korisne jer omogućavaju korišćenje funkcija sa istim imenom nad različitim tipovima što značajno pojednostavljuje programiranje.

U Haskel programima često se koriste naredne klase:

1. Klasa `Show` je klasa svih tipova čije vrednosti mogu da se reprezentuju kao niska. Nad vrednostima instance klase `Show` dozvoljeno je pozivati funkciju `show` koja vrednost pretvara u nisku.
2. Klasa `Eq` je klasa svih tipova čije vrednosti se mogu porediti sa operatorima `==` i `/=`.
3. Klasa `Ord` je klasa svih tipova čije vrednosti se mogu porediti pomoću operatora `<`, `<=`, `>` i `>=`.
4. Klasa `Num` je klasa svih tipova čije vrednosti se mogu sračunavati sa operatorima `+`, `-`, *.
5. Klasa `Fractional` je *podklasa* klase `Num` koja sadrži tipove nad kojima se može pozivati operacija `/`.

Tipovi poput `Bool`, `Int`, `Integer`, `Float`, `Double`, `Char` pripadaju klasama `Eq`, `Ord` i `Show`. To znači da vrednosti ovih tipova možemo da poredimo sa `==`, `/=`, `<`, `<=`, `>`, `>=` kao i da pretvaramo u niske sa funkcijom `show`.

[70] Upravo smo u prethodnim primerima videli da `Int` pripada i klasi `Show` i klasi `Num`.

Tipovi `Int`, `Integer`, `Float`, `Double` svi pripadaju klasi `Num`. Vrednosti ovog tipa možemo da sabiramo, oduzimamo, i množimo sa operatorima `+`, `-`, `*`. Tipovi `Float` i `Double` pripadaju klasi `Fractional`[71].

Primer 12. Sa tipskim klasama moguće je rešiti probleme o kojima smo govorili na početku ove lekcije. Funkciju iz prvog primera, $y = f(x) = x \cdot x + x$ možemo implementirati tako da sa *jednom* definicijom obuhvatimo sve numeričke tipove:

```
f :: Num a => a -> a
f x = x*x + x
```

Gornji kôd je dobro tipiziran iz narednog razloga: za svaku vrednost `x :: N` gde je `N` neki tip iz klase `Num`, važi da je `x*x` dobro tipizaran izraz tipa `N`. Kako su izrazi `x*x` i `x` takođe tipa `N`, sledi da je `x*x + x :: N`. Prema tome, primenom funkcije `f` na neku vrednost `x :: N` dobijamo takođe vrednost `N`.

Ako definišemo naredne numeričke konstante različitih tipova, uverićemo se da je `f` moguće primeniti na svaku od njih:

```
x1 :: Int
x1 = 5

x2 :: Integer
x2 = 5

x3 :: Float
x3 = 5

x4 :: Double
x4 = 5
```

```
ghci> f x1
30
ghci> f x2
30
ghci> f x3
30
ghci> f x4
30
```

Iz primera vidimo da klase zaista omogućavaju polimorfizam: vrednost `f` kao da istovremeno poseduje različite tipove `Int -> Int`, `Integer -> Integer`, `Float -> Float` i `Double -> Double`.

Važno je imati na umu da metode neke određene klase su različite funkcije (što možda nije očigledno u prethodnom primeru). Na primer, operacija `+` u smislu tipa `Int` nije isto što i operacija `+` u smislu tipa `Integer` iako na prvi pogled ne deluje tako.

Primer 13. Definišimo:

```
y1 :: Int
y1 = 12345678987654321
```

[71] Klasa `Fractional` predstavlja kolekciju onih tipova koji mogu da predstave racionalne brojeve. Zato `Int` i `Integer` ne pripadaju ovoj klasi.

```
y2 :: Integer
y2 = 12345678987654321

y3 :: Float
y3 = 12345678987654321

y4 :: Double
y4 = 12345678987654321
```

```
ghci> f y1
-1878358338631772398
ghci> f y2
152415789666209432556012777625362
ghci> f y3
1.5241581e32
ghci> f y4
1.5241578966620942e32
```

Prilikom računanja `f y1` došlo je do prekoračenja zbog čega je krajnji rezultat besmislen. Tip `Integer` omogućeje predstavljanje proizvoljne velikih celih brojeva, te `f y2` daje tačan rezultat. Tipovi `Float` i `Double` mogu predstaviti mnogo veće brojeve nego `Int` ali po cenu tačnosti: vidimo da poslednja dva rezultata imaju samo 8 odnosno 18 tačnih cifara.

Ako smo u pretposlednjem primeru demonstrirali kako klase omogućuju polimorfizam, u poslednjem primeru smo uvideli zašto se taj polimorfizam naziva *ad hoc*: konkretna implementacija metoda razlikuje se od instance do instance!

Naravno, i dalje nismo razjasnili mnoge stvari oko klasa, kao na primer kako se klase definišu ili kako se tipovi uvode u klase. Više o tipskim klasama, njihovom definisanju i instanciranju, biće rečeno u lekciji *Tipske klase*. Za sada je bitno samo da razumemo tipske potpise koji sadrže klasna ograničenja.

7.2.1. Klase i složeni tipovi

Videli smo u lekciji o tipovima, da je od postojećih tipova moguće dobiti nove tipove lista i proizvode (npr. od `Int` dobijamo `[Int]`). Deluje prirodno da se neke osobine tipova prenose pri ovom procesu. Tačnije, ako tip `T` pripada nekoj klasi `C`, tada je nekad smisleno da i `[T]` ili `(T,T)` pripadaju istoj toj klasi. Srećom, ovakva "nasleđivanja" su moguća i prisutna u Haskelu.

Ako neki tip `A` pripada klasi `Eq`, tada i tip `[A]` pripada klasi `Eq`. To praktično znači da ako se vrednosti nekog tipa mogu porediti sa operatorima `==` i `/=`, tada se i liste tog tipa mogu porediti sa istim operatorima. Poređenja lista se vrše član po član. Analogno važi i za klase `Ord` i `Show`: ako tip `A` pripada nekoj od ovih klasa, tada i `[A]` pripada istoj klasi. Prema tome, tipovi poput `[Bool]`, `[Int]`, `[Char]` ali i `[[Bool]]`, `[[[Int]]]` itd... pripadaju klasama `Eq`, `Ord` i `Show`.

Primer 14. Kako tip `Int` pripada klasi `Show`, sledi da i tipovi `[Int]` i `[[Int]]` pripadaju istoj klasi.

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

Za uređene n-torke važi sličan princip. Ako tipovi `A`, `B`, ... `T` pripadaju klasi `Eq`, tada i tip `(A, B, … T)` pripada klasi `Eq`. Ovo takođe važi i za klase `Ord` i `Show`.

Primer 15. Već znamo da vrednosti tipa `Int` i `Char` možemo da poredimo sa `==`:

```
ghci> 8 == 2 * (3 + 1)
True
ghci> 'a' == 'b'
False
```

Prema gore napisanom, i vrednosti tipa `(Int, Char)` možemo da poredimo:

```
ghci> (2, 'a') == (3, 'b')
False
ghci> (2, 'a') == (2, 'a')
True
```

Dva uređena para će biti jednaka samo ako su im jednake prve i druge koordinate.

Što se tiče poređenja sa `>` i `<`, ono se kao i kod lista vrši leksikografski, koordinate se porede par po par.

```
ghci> (2, 'a') < (3, 'b')
True
ghci> (100, 'W') < (100, 'A')
False
```

Prvo poređenje je tačno jer je `2 < 3`, i pri ovom poređenju nije ni došlo do provere `'a' < 'b'`. Rezultat drugog poređenja sledi iz činjenice da je `'A' < 'W'` a ne `'W' < 'A'`.

Zadatak 6. Da li tip `[(Int, Char)]` pripada klasi `Eq`?

Rešenje. Da. Oba tipa `Int` i `Char` pripadaju klasi `Eq`. Prema tome, i tip `(Int, Char)` pripada klasi `Eq`, jer uređen par dva tipa iz klase `Eq` takođe pripada klasi `Eq`. Sada, iz toga što `(Int, Char)` pripada klasi `Eq` sledi da i tip `[(Int, Char)]` pripada klasi `Eq`.

I zaista, vrednosti tipa `[(Int, Char)]` možemo da poredimo sa `==` i `/=`:

```
ghci> [(2,'a')] == [(3,'b'), (2,'a')]
False
```

Zadatak 7. Funkcija

```
max :: Ord a => a -> (a -> a)
```

uzima dve vrednosti i vraća veću od vrednosti. Napisati funkciju `max3 :: Ord a => a -> (a -> (a -> a))` koja vraća maksimum tri vrednosti. Naravno `Ord a =>` je klasno ograničenje koja nam govori da tipska promenljiva `a` pripada klasi `Ord`, te se vrednosti ovog tipa mogu porediti sa `<`, `>`, `<=` i `>=`.

Rešenje. Ako su nam date vrednosti `x`, `y` i `z` tada ćemo prvo naći veću od vrednosti `x` i `y` a zatim ćemo tu veću vrednost da uporedimo sa `z`:

```
max3 :: Ord a => a -> a -> a -> a
max3 x y z = max (max x y) z
```

7.3. Zaključivanje tipova

GHC kompajler (skoro) uvek može da zaključi tip izraza. Zbog toga često nije neophodno navoditi tipske deklaracije u izvornim datotekama.[72]

Zaključivanje tipova nam takođe omogućava da u *GHCi* okruženju proverimo tipove izraza. To možemo uraditi pomoću naredbe `:type` nakon koje navodimo izraz čiji tip nas interesuje:

```
ghci> :type [True, False]
[True, False] :: [Bool]
ghci> :type [True, False] !! 1
[True, False] !! 1 :: Bool
ghci> :type length [True, False]
length [True, False] :: Int
ghci> :type (True, 'a')
(True, 'a') :: (Bool, Char)
```

Primetimo da se `:type` komanda odnosi na ceo ostatak linije. Stoga nije neophodno pisati `:type ([True] !! 0)`, već samo `:type [True] !! 0`, itd...

Zaključivanje tipova je složen proces, ali možemo demonstrirati osnove ovog procesa.

1. Pri zaključivanju tipa izraza `[True, False]`, kompajler prvo uviđa da se radi o listi. Prema tome, prvo se zaključuje da je tip ovog izraza `[a]` za neki neodređeni tip `a`. Zatim, kompajler uvidom u prvi element ove liste zaključuje da je tip `a` baš `Bool`, pa je samim tim tip celog izraza `[Bool]`. Naravno pri ovom procesu se proverava da svi ostali elementi liste imaju isti tip.
2. Operacijom `!!` se pristupa n-tom elementu liste. Kako lista u izrazu `[True, False] !! 1` ima tip `[Bool]` sledi da izraz `[True, False] !! 1` ima tip `Bool`.
3. Izraz `length [True, False]` je nešto složeniji. Funkcija `length` poseduje polimorfan tip `[a] -> Int`. U ovom slučaju jasno je da će povratna vrednost funkcije imati tip `Int`. Ali svakako, pri proveri dobre tipiziranosti celog izraza kompajler prvo mora da utvrdi da argument poseduje tip koji se može poklopiti sa `[a]`.

Rezultat `:type` naredbe će nas iznenaditi u slučaju numeričkih literala:

```
ghci> :type 2
2 :: Num a => a
```

Za izraz `2` *GHCi* nije mogao da zaključi konkretan tip. Literal `2` može predstavljati vrednost tipa različitih numeričkih tipova. Zbog toga *GHCi* zaključuje da je `2` nekog tipa `a` koji pripada klasi `Num`, što se naravno označava sa `Num a => a`.

Ako potražimo tip izraza `3.14` uvidećemo da je u pitanju tip `Fractional a => a`. Svakako izraz `3.14` predstavlja jedan racionalan broj, te ne može biti tipa `Int` ili `Integer`. Ali racionalan broj može biti predstavljen i kao `Float` i kao `Double` vrednost. Zbog toga ni ovde nemamo konkretan odgovor već klasno ograničenje.

7.4. Karijevanje

U lekciji o funkcijama prikazana su dva načina definisanja funkcije sa više parametra. Prvi, i jednostavniji, način podrazumeva konstrukciju funkcije koja uzima uređen par, dok drugi način podrazumeva konstrukciju funkcije višeg reda. Prva tehnika daje funkciju tipa `(A, B) -> C` dok druga tehnika daje funkciju tipa `A -> (B -> C)`, za neke tipove `A`, `B`, `C`.

[72] Ipak, uvek je dobro navesti tipove svih definicija, jer je takav kod znatno čitljiviji.

Intuitivno je jasno da su ove dve tehnike podjednako dobre. Svaka funkcija koja se može definisati na jedan od pomenutih načina može se definisati i na drugi. Stvar je ukusa za koju tehniku ćemo se opredeliti[73]. Ono što je zanimljivo je da je Haskel toliko ekspresivan jezik da se lako može definisati funkcija višeg reda koja transformiše funkciju tipa `(A, B) -> C` u funkciju tipa `A -> (B -> C)` (gde su `A`, `B` i `C` neki tipovi). Ovakva transformacija se naziva **karijevanje**[74]. Označimo funkciju koja vrši ovu transformaciju sa `k`.

Kog tipa je funkcija `k`? Funkcija `k` uzima funkciju tipa `(A, B) -> C` i vraća funkciju tipa `A -> (B -> C)`. Stoga je tip ove funkcije `((A, B) -> C) -> (A -> (B -> C))`.

Kada smo utvrdili tip, možemo da implementiramo `k`. Pretpostavimo da je `f :: (A, B) -> C` funkcija koju želimo da transformišemo. Pošto primena `k f` predstavlja vrednost tipa `A -> (B -> C)`, podudaranjem oblika možemo definisati da `k f` predstavlja neki lambda izraz `\x y -> ...`. U telu ovog lambda izraza mora se nalaziti rezultat primene `f` na uređen par `(x, y)`. Prema tome, `k` možemo da definišemo sa narednim kodom

```
k :: ((A, B) -> C) -> (A -> (B -> C))
k f = \x y -> f (x, y)
```

U definiciji funkcije `k` nigde nismo koristili činjenicu da radimo sa vrednostima tipova `A`, `B` ili `C`, tj nismo iskoristili osobine ovih tipova. Prema tome, funkciju `k` možemo učiniti u potpunosti parametarski polimorfnom:

```
k :: ((a, b) -> c) -> (a -> (b -> c))
k f = \x y -> f (x, y)
```

Umesto na funkcije nekog konkretnog tipa, sada se `k` može primeniti na sve funkcije čiji je domen uređen par.

Na sličan način možemo da definišemo i inverznu funkciju `k'` koja funkcije tipa `a -> (b -> c)` transformiše u funkcije tipa `(a, b) -> c`. Jasno, tip funkcije `k'` je `(a -> (b -> c)) -> ((a, b) -> c)`. Ako je `g` funkcija tipa `a -> (b -> c)` tada je `k' g` funkcija koja uzima uređen par, te ćemo stoga takvu funkciju konstruisati sa lambda apstrakcijom `\(x, y) -> ...`. U telu lambda funkcije je potrebno da se nađe vrednost `(g x) y`. Stoga, `k'` možemo da definišemo sa narednim kodom

```
k' :: (a -> (b -> c)) -> ((a, b) -> c)
k' g = \(x, y) -> g x y
```

Uverimo se da navedene funkcije zaista dobro rade. Uzmimo primer iz lekcije o funkcijama:

```
ljubav :: (String, String) -> String
ljubav (x, y) = x ++ " voli " ++ y ++ "!"
```

```
ghci> ljubav' = k ljubav
ghci> ljubav' "Ana" "Milovana"
"Ana voli Milovana!"
```

Transformacija zaista funkcioniše: funkcija `ljubav'` ne uzima uređen par već je u pitanju funkcija višeg reda.

Sa druge strane, od ugrađene funkcije `mod` možemo dobiti našu verziju funkcije koja uzima jedan uređen par brojeva:

[73] Ipak, dobro je naglasiti da je u duhu Haskel jezika koristiti funkcije višeg reda. Kasnije ćemo videti da funkcije višeg reda dovode do značajno elegantnijeg koda.

[74] Eng. *currying*. po Haskel Kariju (Haskell Curry)

```
ghci> mod' = k' mod
ghci> mod' (7, 2)
1
```

Funkcije koje smo konstruisali u prethodnim redovima dostupne su pod imenima `curry` i `uncurry` i poseduju sledeće tipove

```
curry :: ((a, b) -> c) -> (a -> (b -> c))
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

> **Zadatak 8.** Konstruisati funkciju `curry3` koja uzima funkciju tipa `(a, b, c) -> d` i daje funkciju tipa `a -> (b -> (c -> d))`.

> **Zadatak 9.** Konstruisati funkciju `curry2x2` koja uzima funkciju tipa `(a, b) -> (c, d) -> e` i daje funkciju tipa `a -> (b -> (c -> (d -> e)))`.

> **Zadatak 10.** Kog tipa je izraz `uncurry curry`?

8. Liste

Liste u Haskelu su strukture podataka koje predstavljaju linearno uređenu kolekciju proizvoljno mnogo vrednosti istog tipa. Liste se konstruišu navođenjem vrednosti unutar uglastih zagrada. Na primer, zapis

```
[5, 8, 13, 21, 34]
```

predstavlja listu od pet elemenata. Međutim, zapis poput navedenog, predstavlja samo skraćeni oblik zapisa

```
5:(8:(13:(21:(34:[])))).
```

Operacija `:` nadovezuje element na početak neke postojeće liste i predstavlja jedan od *konstruktora vrednosti liste*, odnosno funkcije kojom se konstruiše neka lista. Drugi konstruktor vrednosti liste je konstruktor prazne liste koji se označava sa `[]`. Važno je napomenuti da su ovo jedini konstruktori vrednosti liste, i da se svaka druga funkcija koja vraća neku listu svodi na neki način na upotrebu ova dva konstruktora. Takođe, ova činjenica povlači i da je svaka lista prazna (dobijena konstruktorom prazne liste), ili poseduje prvi član (odnosno, nastala je nadovezivanjem neke vrednosti na postojeću listu).

8.1. Funkcije za rad sa listama

U prethodnim lekcijama već smo se upoznali sa nekim funkcijama za rad sa listama:

1. `(++) :: [a] -> [a] -> [a]` spaja dve liste
2. `length :: [a] -> Int` vraća dužinu liste
3. `elem :: Eq a => a -> [a] -> Bool` ispituje da li se element nalazi u listi
4. `reverse :: [a] -> [a]` obrće poredak elemenata u listi
5. `head :: [a] -> a` vraća prvi element liste. Ako je lista prazna dolazi do izuzetka
6. `init :: [a] -> [a]` vraća sve elemente osim poslednjeg. Ako je lista prazna dolazi do izuzetka
7. `tail :: [a] -> [a]` vraća sve elemente osim prvog. Ako je lista prazna dolazi do izuzetka
8. `last :: [a] -> a` vraća poslednji element. Ako je lista prazna dolazi do izuzetka

Primetimo da se sve navedene funkcije, osim funkcije `elem`, mogu koristiti sa listom proizvoljnog tipa jer poseduju parametarski polimorfan tip. Funkcija `elem` može se koristiti samo sa instancama `Eq` klase[75].

Prelid obezbeđuje neke korisne funkcije za rad sa listama karaktera:

1. `words :: [Char] -> [[Char]]` deli nisku na listu niski pri čemu se prelom vrši na belinama. Na primer: `words "a bb ccc "` daje `["a", "bb", "ccc"]`. Samo ime funkcije ukazuje da se od niske dobijaju "reči".
2. `unwords :: [[Char]] -> [Char]` spaja listu niski u jedinstvenu nisku dodavajući razmak između. Na primer: `unwords ["boža", "zvani", "pub"]` nam daje `"boža zvani pub"`.
3. `lines :: [Char] -> [[Char]]` rastavlja nisku na linije, vršeći prelom na karakteru za novi red `'\n'`.
4. `unlines :: [[Char]] -> [Char]` spaja listu niski u jedinstvenu nisku, dodavajući `'\n'` nakon svake niske.

Ovde je zgodno napomenuti kako se u Haskel kodu mogu navesti niske u više redova. Neophodno je na kraju i početku svake linije koja pripada nisci dodati znak `\`, osim u prvoj i poslednjoj na mestima gde je `"`:

```
stih :: [Char]
stih = "Ti, međutim, rasteš, uz zornjaču jasnu,\
\sa Avalom plavom, u daljini, kao breg.\
\Ti treperiš, i kad ovde zvezde gasnu,\
\i topiš, ko Sunce, i led suza, i lanjski sneg."
```

Ipak, ovako definisana niska *neće* sadržati karaktere za novi red! Neophodno ih je dodati na svakom mestu:

[75] Podsećamo, instance `Eq` klase su tipovi čije vrednosti se mogu porediti sa `==` i `/=`. Prema tome, sasvim je jasno zašto je neophodno da `elem` ima ovakvo klasno ograničenje.

```
stih :: [Char]
stih = "Ti, međutim, rasteš, uz zornjaču jasnu,\n\
\sa Avalom plavom, u daljini, kao breg.\n\
\Ti treperiš, i kad ovde zvezde gasnu,\n\
\i topiš, ko Sunce, i led suza, i lanjski sneg.\n"
```

Ako u *GHCi* interpreteru evaluiramo vrednost `stih`, niska će se ispisati unutar dvostrukih navodnika u jednoj (ali dugačkoj) liniji terminala[76]. Probajte i uverite se sami. Zbog toga je zgodno iskoristiti `putStr` koji sadržaj ispisuje u terminal onako kao bismo i očekivali:

```
ghci> putStr stih
Ti, međutim, rasteš, uz zornjaču jasnu,
sa Avalom plavom, u daljini, kao breg.
Ti treperiš, i kad ovde zvezde gasnu,
i topiš, ko Sunce, i led suza, i lanjski sneg.
```

Zadatak 1. Napisati funkciju

```
brojReči :: [Char] -> Int
```

koja broji reči u nisci.

Rešenje.

```
brojReči :: [Char] -> Int
brojReči x = length (words x)
```

8.2. Rasponi

Neretko se javlja potreba da se konstruiše lista poput liste prirodnih brojeva od 1 do n ili liste brojeva od m do n itd... Iako nije teško implementirati rekurzivne funkcije koje vraćaju ovakve liste, Haskel jezik poseduje sintaksu za **raspone**[77] koja značajno olakšava konstrukciju ovakvih lista. Raspon se konstruiše navođenjem prvog i poslednjeg elementa liste, između kojih se postavljaju dve (spojene) tačke[78]. Naredni primeri ilustruju sintaksu raspona:

```
ghci> a = [1 .. 10]
ghci> a
[1,2,3,4,5,6,7,8,9,10]
ghci> [-3 .. 3]
[-3,-2,-1,0,1,2,3]
ghci> [10 .. 1]
[]
```

Ako je poslednji element manji od prvog, tada će biti konstruisana prazna lista.

Sa rasponima je moguće konstruisati i liste u kojima je korak različit od 1. U tom slučaju, potrebno je navesti i drugi član liste na osnovu koga će korak biti izračunat.

[76] Takođe, u zavisnosti od verzije *GHCi* programa, može se desiti da slova koja ne pripadaju engleskom alfabetu (*đ*, *ž*, *ć* i *č*) budu zamenjena odgovarajućim kodovima.

[77] eng. *ranges*

[78] Napomenimo da se u ovoj knjizi na nekim mestima koristi trotačka ... da bi se skratio neki Haskel izraz. Izraz za trotačkom nije validan Haskel izraz. Rasponi se isključivo definišu sa dve, spojene, tačke.

```
ghci> [1, 3 .. 11]
[1,3,5,7,9,11]
ghci> [0, 10 .. 100]
[0,10,20,30,40,50,60,70,80,90,100]
ghci> [1, 5 .. 11]
[1,5,9]
```

Primetimo u poslednjem primeru da lista uključuje sve brojeve aritmetičke progresije manje od `11`. Dakle, gornja granica raspona ne mora biti uključena u listu.

Navođenjem koraka progresije moguće je lako konstruisati liste i koje su opadajuće:

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

Rasponi se takođe mogu koristiti i sa racionalnim brojevima:

```
ghci> [1.0 .. 5.0]
[1.0,2.0,3.0,4.0,5.0]
ghci> [1.0, 1.5 .. 5.0]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

Ipak, treba biti pažljiv pri korišćenju raspona racionalnih brojeva, jer se računska greška može lako ukazati[79]

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

Rasponi dozvoljavaju samo definisanje aritmetičkih progresija. Za definisanje geometrijske (ili neke druge progresije), korisnik će morati sam da implementira odgovarajuće funkcije.

Međutim, rasponi mogu da se koriste i sa tipovima koji nisu numerički. Na primer, rasponi karaktera su često korisni

```
ghci> ['A' .. 'W']
"ABCDEFGHIJKLMNOPQRSTUVW"
ghci> ['a' .. 'd']
"abcd"
ghci> ['a', 'c' .. 'w']
"acegikmoqsuw"
```

Lista karaktera je naravno niska te se `['a', 'b', 'c', 'd']` prikazuje kao `"abcd"` u terminalu.

Da bi neki tip mogao da se koristi sa rasponima, mora pripadati tipskoj klasi `Enum`. Ova klasa sadrži sve tipove čije vrednosti se mogu enumerisati. Svaki tip `a` iz `Enum` klase mora da implementira funkcije `fromEnum :: a -> Int` i `toEnum :: Int -> a` koje redom kodiraju i dekodiraju vrednosti tog tipa kroz cele brojeve. Vrednosti dobijene ovim kodiranjem koriste se za konstrukciju raspona.

> **Primer 1.** Tip `Char` pripada klasi `Enum`. Funkcija `fromEnum` primenjena na karakter vraća poziciju tog karaktera u *Unicode* tabeli.

[79] Kao što u decimalnom sistemu ne možemo sa konačno simbola zapisati vrednost $1/3$, tako u binarnom sistemu ne možemo tačno zapisati vrednost $1/10$ koristeći konačno bitova. Racionalni brojevi koji se tačno mogu zapisati u IEEE 754 zapisu (koji je osnova za `Float` i `Double` tipove) su oblika $k/2^i$ gde $i, k \in \mathbb{Z}$.

```
ghci> fromEnum 'a'
97
ghci> toEnum 100 :: Char
'd'
```

Kako je `toEnum` tipa `Int -> a` kompajler ne može zaključiti tip izraza `toEnum 100` (na koju instancu `Enum` klase se ovaj izraz odnosi?). Zbog toga je neophodno eksplicitno deklarisati tip izraza. Više o ovome kasnije...

Kada napišemo raspon poput `['a', 'c' .. 'g']`, taj raspon se tumači kao raspon

```
[fromEnum 'a', fromEnum 'c' .. fromEnum 'g'],
```

odnosno kao `[97, 99 .. 103]`. Time se dobija lista `[97,99,101,103]` koja se primenom funkcije `toEnum` na svaki od elemenata transformiše u listu `"aceg"`.

Zadatak 2. Napisati funkciju `velikoSlovo :: Char -> Bool` koja proverava da li je uneto slovo veliko latinično.

Rešenje. Možemo iskoristi funkciju `elem` koja proverava da li element pripada listi.

```
velikoSlovo :: Char -> Bool
velikoSlovo c = elem c ['A' .. 'Z']
```

Zadatak 3. Implementirati funkciju

```
daLiJeCifra :: Char -> Bool
```

bez korišćenja bilo koje sintakse za grananje.

8.3. Rekurzija nad listama

Liste imaju prirodnu rekurzivnu strukturu. Svaka lista osim prazne je nastala nadovezivanjem nekog elementa na početak neke kraće liste. Mnogi algoritmi za rad sa listama se mogu implementirati rekurzijom po dužini liste. Baza rekurzije čini slučaj prazne liste, dok rekurzivni korak predstavlja svođenje problema na isti problem za kraću listu.

Primer 2. Funkciju `zbir :: [Int] -> Int` koja sabira sve elemente liste možemo napisati ovako:

```
zbir :: [Int] -> Int
zbir xs =
    if null xs
    then 0
    else head xs + zbir (tail xs)
```

Uslovom `null xs` se proverava da li je lista prazna. Ako jeste, tada vraćamo `0`. U suprotnom, vrednost prvog elementa dodajemo na zbir repa liste.

Tehnika podudaranja oblika liste je posebno korisna za implementaciju rekurzivnih algoritama. Sa ovom tehnikom, kôd algoritma često postaje elegantniji:

Primer 3. Funkciju `zbir` možemo napisati i ovako:

```
zbir :: [Int] -> Int
zbir [] = 0
zbir (x:xs) = x + zbir xs
```

Zadatak 4. Implementirati funkciju `proizvod :: [Int] -> Int` koja pronalazi proizvod elemenata liste.

Rešenje. Po ugledu na prethodne primere, lako dolazimo do rešenja

```
proizvod :: [Int] -> Int
proizvod [] = 1
proizvod (x:xs) = x * proizvod xs
```

Osim promene same operacije, promenili smo i povratnu vrednost za praznu listu. Za svaki drugi izbor vrednosti `proizvod []`, dobio bi se pogrešan rezultat. Štaviše, u slučaju da smo definisali `proizvod [] = 0`, tada bi proizvod svake liste bio `0`.

Primer 4. Do sada smo se upoznali sa funkcijom `last :: [a] -> a` koja vraća poslednji element neprazne liste. Tehnikom rekurzije lako možemo da sami implementiramo ovu funkciju[80]. Baza rekurzije je sada jednočlana lista, jer nema smisla vraćati poslednji element prazne liste. Ako prosleđena lista nije prazna tada je njen poslednji element baš poslednji element repa te liste. Na ovaj način, sveli smo problem na rekurzivni poziv.

```
last' :: [a] -> a
last' [x] = x
last' (_:xs) = last' xs
```

Pošto nas u drugom slučaju ne interesuje glava liste, iskoristili smo džoker _.

Zadatak 5. Reimplementirati funkcije za rad sa listama poput `tail`, `reverse`.

Ponekad je potrebno implementirati funkcije koje su rekurzivne po više parametra.

Primer 5. Često je potrebno odrediti da li je neka niska podniska druge. Tačnije, potrebna je funkcija

```
daLiJePodniska :: [Char] -> [Char] -> Bool
```

koja određuje da li se prva prosleđena niska sadrži u drugoj prosleđenoj nisci. Ideja za implementaciju funkcije `daLiJePodniska` je naravno tehnika rekurzije.

Pretpostavimo da su nam date dve neprazne niske `igla` i `seno`[81], i da se pitamo da li se `igla` sadrži u nisci `seno`. Tada postoje dve mogućnosti:

1. Niska `igla` predstavlja početak niske `seno`. U ovom slučaju, odgovor je na pitanje da li se igla nalazi u senu je potvrdan.
2. Niska `igla` nije početak niske `seno`. U ovom slučaju, ako se niska `igla` nalazi u nisci `seno`, tada se mora nalaziti u repu te niske. Stoga pretragu možemo da ograničimo na rep niske `seno`.

[80] Uvek kada govorimo o reimplementaciji neke funkcije `f`, tu reimplementaciju ćemo označavati sa `f'`. Znak `'` je validan deo imena u Haskelu i često se koristi na kraju imena baš u ovom kontekstu (tj. kada se funkcije reimplementiraju).

[81] U literaturi je uobičajeno da se niske imenuju *igla* i *seno*, jer to jasno sugeriše u kojoj od niska očekujemo da se nalazi druga niska.

Ako je niska `seno` prazna, tada se niska `igla` nalazi u njoj ako i samo ako je `igla` prazna niska. Da bi proverili da li je neka lista (specijalno niska) prazna, možemo iskoristiti funkciju `null :: [a] -> Bool` iz prelida.

Na osnovu ove analize, implementacija `daLiJePodniska` je jednostavna:

```
daLiJePodniska :: [Char] -> [Char] -> Bool
daLiJePodniska igla "" = null igla
daLiJePodniska igla seno =
    if daLiJePočetak igla seno
        then True
        else daLiJePodniska igla (tail seno)
```

Ostaje nam još da implementiramo funkciju

```
daLiJePočetak :: [Char] -> [Char] -> Bool
```

koja proverava da li prva prosleđena niska predstavlja početak druge prosleđene niske. Ponovo ćemo koristiti rekurziju, ali ćemo sada vršiti rekurziju po oba parametra.

Ako su glave obe prosleđene niske iste, tada možemo da nastavimo da poredimo repove. U suprotnom možemo da budemo sigurni da druga niska ne počinje sa prvom niskom. Baza indukcije su slučajevi u kojima je jedna od niski prazna. Ako je prva niska prazna, onda druga niska započinje prvom niskom. Ako je druga niska prazna, a prva niska nije prazna, tada znamo da se prva niska ne sadrži u drugoj nisci

```
daLiJePočetak :: [Char] -> [Char] -> Bool
daLiJePočetak "" _ = True
daLiJePočetak _ "" = False
daLiJePočetak (x:xs) (y:ys) =
    if x == y
        then daLiJePočetak xs ys
        else False
```

Primetimo da se funkcija

```
daLiJePodlista :: Eq a => [a] -> [a] -> Bool
```

može implementirati na gotovo isti način.

Zadatak 6. Definisati funkciju `dupliraj :: [Int] -> [Int]` koja duplira svaki element liste brojeva. Redosled dupliranih vrednosti treba da odgovara redosledu vrednosti u originalnoj listi. Na primer

```
dupliraj [2, 3, 4]
```

je `[4, 6, 8]`, a `dupliraj []` je `[]`.

Zadatak 7. Definisati funkciju `dužine :: [[Char]] -> [Int]` koja listu niski preslikava u listu brojeva, pri čemu svaki broj predstavlja dužinu niske. Pritom, redosled brojeva je isti kao redosled odgovarajućih niski. Na primer,

```
dužine ["Pop","Ćira","i","pop","Spira"]
```

je `[3,4,1,3,5]`.

Zadatak 8. Definisati funkciju `parni :: [Int] -> [Int]` koja listu brojeva preslikava u listu brojeva koja je dobijana uklanjanjem svih neparnih brojeva iz početne liste. Na primer, `parni [1,2,3,4]` je `[2,4]`.

Zadatak 9. Definisati funkciju `zbirCifara :: Int -> Int` koja određuje zbir cifara broja zapisanog u dekadnom zapisu. Koristeći ovu funkciju, naći sve prirodne brojeve n manje od 100, takve da je zbir cifara broja n^2 deljiv sa 5.

Zadatak 10. Za nisku `a` kažemo da deli nisku `s` ako je `s` oblika `a ++ a ++ … ++ a`. Na primer, niska `"Abc"` deli nisku `"AbcAbcAbc"`. Takođe, smatra se da prazna niska `""` deli svaku nisku. Napisati funkciju

```
nzd :: [Char] -> [Char] -> [Char]
```

koja određuje *najveći zajednički delilac* dve niske, odnosno najdužu nisku koja deli obe niske.

8.4. Funkcije višeg reda

Implementacijom mnogih rekurzivnih algoritama nad listama uvidećemo da se neki šabloni konstrukcije algoritama ponavljaju. Zbog toga su definisane funkcije višeg reda za rad sa listama. Korišćenjem ovih funkcija možemo da izbegnemo eksplicitnu rekurziju u mnogim slučajevima.

8.4.1. *Filter*

Funkcija višeg reda `filter :: (a -> Bool) -> [a] -> [a]` iz liste izdvaja samo elemente koji zadovoljavaju neki uslov.

Prvi parametar funkcije je funkcija tipa `a -> Bool` koju nazivamo **predikat**. Predikat treba da vrati vrednost `True` ako element treba da se nađe u povratnoj listi. Drugi parametar funkcije je lista koju filtriramo. Rezultat je lista iz koje su uklonjeni elementi koji ne zadovoljavaju predikat, pri čemu poredak elemenata ostaje isti.

Primer 6. Ako želimo da "uzmemo" parne prirodne brojeve manje ili jednake sa `10` možemo napisati:

```
ghci> filter (\x -> mod x 2 == 0) [1 .. 10]
[2, 4, 6, 8, 10]
```

Zadatak 11. Data je lista `visine :: [Float]` koja predstavlja visine svih učenika jedne škole. Odrediti broj učenika viših od 180cm.

Rešenje.

```
brojVisokihUčenika = length visokiUčenici
    where visokiUčenici = filter (\x -> x >= 180) visine
```

Zadatak 12. Svaka osoba je predstavljena sa uređenim parom tipa `([Char], Int)` koji sadrži ime osobe i njene godine. Implementirati funkciju

```
punoletne :: [([Char], Int)] -> [([Char], Int)]
```

koja iz liste osoba eliminiše maloletne osobe. Dati rešenja sa i bez funkcije `filter`.

Zadatak 13. Reimplementirati funkciju `filter`.

Rešenje. Iskoristićemo tehniku rekurzije. Ako funkciju `filter' p` primenimo na praznu listu, tada je logično da dobijemo praznu listu bez obzira na predikat `p`. Ovaj slučaj predstavlja bazu rekurzije.

Ako lista nije prazna, tada je možemo rastaviti na glavu `x` i rep `xs`. Ako je `p x` tačno, tada se `x` nalazi na početku liste koju vraćamo, a u suprotnom ne. Za ostatak liste `xs` iskoristićemo rekurzivni poziv, da bismo isfiltrirali `xs`.

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs) = if p x then x : filter' p xs else filter' p xs
```

Pošto nas u baznom slučaju ne interesuje predikat, iskoristili smo džoker _.

8.4.2. *Map*

Još jedna funkcija višeg reda koja se često koristi sa listama je

$$\texttt{map :: (a -> b) -> [a] -> [b]}.$$

Funkcija `map` primenjuje datu funkciju na svaki element liste i vraća listu dobijenih vrednosti tj. važi[82]

$$\texttt{map f [}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{] = [f }x_1\texttt{, f }x_2\texttt{, … f }x_n\texttt{]}.$$

Prvi parametar funkcije je funkcija tipa `a -> b`, a drugi parametar je lista tipa `[a]`. Rezultat je lista tipa `[b]` koja je nastala od prosleđene liste primenom prosleđene funkcije na svaki element liste.

Primer 7. Ako želimo da kvadriramo svaki element liste `[1, 2, 3, 4]`, možemo napisati

```
ghci> map (\x -> x^2) [1, 2, 3, 4]
[1, 4, 9, 16]
```

Zadatak 14. Reimplementirati funkciju `map`.

Rešenje. Iskoristićemo ponovo tehniku rekurzije. Funkcija `map'` treba da primeni neku funkciju `f` na svaki element liste `l`. Ako je lista `l` prazna, tada je i `map f l` prazna lista. Ovaj slučaj predstavlja bazu rekurzije.

Ako lista nije prazna, tada ćemo na glavu liste `x` primeniti funkciju `f`, čime dobijamo prvi element tražene liste. Za dobijanje ostatka liste, rekurzivno ćemo pozvati `map' f` nad repom liste `xs`.

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

8.4.3. *Fold*

Funkcije `foldl` i `foldr` koriste neku binarnu funkciju da bi "presavile" složenu vrednost (poput liste) u jednu vrednost. Ove funkcije se mogu koristiti sa tipovima koji pripadaju `Foldable` klasi, a liste su osnovni primer takvih tipova. Zapravo, upravo je implementacija funkcija `foldr` neophodna za instanciranje `Foldable` klase[83]. Nadalje, možemo zaboraviti na `Foldable` klasu, i posmatrati funkcije `foldl` i `foldr` samo kroz kontekst lista.

[82] Primetimo da navedena jednakost nije validan Haskel kôd, već samo zapis koji ilustruje funkciju `map`.
[83] Kao što je na primer funkcija `show` neophodna za instanciranje `Show` klase.

Razlika između `foldl` i `foldr` funkcije je tome što `foldl` počinje od početka liste (levog kraja), dok `foldr` počinje od desnog kraja.

Pogledajmo prvo funkciju `foldr`. Tip funkcije `foldr` je[84]

```
foldr :: (a -> b -> b) -> b -> [a] -> b.
```

Prvi parametar funkcije `foldr` je binarna funkcija tipa `a -> b -> b`. Drugi parametar je **početna vrednost**, a treći parametar je lista koju želimo da "presavijemo". Funkcija `foldr` "prolazi" kroz datu listu s desna na levo pozivajući nad svakim elementom datu binarnu funkciju. Za svaki taj poziv, rezultat poziva nad prethodnim elementom se koristi kao drugi argument binarne funkcije. Inicijalna vrednost se koristi kao drugi argument pri prvom pozivu (a to je poziv nad poslednjim elementom u listi). Komplikovano objašnjenje će postati jasnije kroz naredne primere.

Primer 8. Uz pomoć `foldr` možemo sabrati sve elemente jedne liste. Binarna funkcija koju ćemo koristiti je funkcija koja sabira argumente `saberi = \a b -> a + b`.

Ako `foldr` pozovemo nad praznom listom, dobićemo početnu vrednost[85]:

```
ghci> foldr saberi 0 []
0
```

Ako `foldr saberi 0` pozovemo nad jednočlanom listom, dobićemo rezultat primene funkcije `saberi` nad jedinim elementom liste i inicijalnom vrednošću (tj. `saberi 1 0`):

```
ghci> foldr saberi 0 [1]
1
```

Navedeni izraz se svodi na `saberi 1 0`, tj. na primenu prosleđene funkcije na jedini element liste, pri čemu je inicijalna vrednost iskorišćena kao drugi argument.

U slučaju dvočlane liste, prosleđena funkcija biće dva puta pozvana. Pri drugom pozivu biće iskorišćena povratna vrednost prvog poziva.

```
ghci> foldr saberi 0 [2, 1]
3
```

Navedeni izraz se svodi na `saberi 2 (saberi 1 0)`

Za listu od više elemenata imamo:

```
ghci> foldr saberi 0 [3, 2, 1]
6
```

Prethodih izraz je ekvivalentan izrazu

```
saberi 3 (saberi 2 (saberi 1 0))
```

koji je naravno ekvivalentan izrazu

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

[84] Zapravo, tip navedene funkcije je `Foldable t => (a -> b -> b) -> b -> t a -> b`, ali se za sada možemo ograničiti samo na liste.

[85] Ovo je uvek tačno za obe *fold* funkcije, pozivanjem nad praznom listom dobijamo početnu vrednost, jer u listi nije bilo elementa na koje je prosleđena binarna funkcija mogla biti primenjena.

Isti rezultat se dobija i sa `foldl` funkcijom, koja primenjuje funkciju sa leva na desno. Izraz `foldl saberi 0 [3, 2, 1]` svodi se na izraz

$$((0 + 3) + 2) + 1.$$

U prethodnim primerima, obe funkcije su dale isti rezultat jer smo koristili funkciju `saberi` koja je simetrična po parametrima (tj. `saberi a b` daje isti rezultat kao `saberi b a`). Ali nije neophodno da rezultati primene `foldl` i `foldr` funkcija budu isti:

```
ghci> g = \a b -> a / b
ghci> foldl g 1 [2, 3]
0.16666666666
ghci> foldr g 1 [2, 3]
0.66666666666
```

Prvi izraz se svodi na `(1 / 2) / 3`, dok se drugi izraz svodi na `2 / (3 / 1)`.

Često je zgodno da umesto posebne inicijalne vrednosti, iskoristimo prvi (ili poslednji) element liste. Tada nam mogu pomoći funkcije

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

i

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

koje se razlikuju od `foldl` i `foldr` funkcija po tome što nemaju i inicijalnu vrednost kao jedan od parametra.

Primer 9. Binarna funkcija `min :: Ord a => a -> a -> a` vraća manju od dve vrednosti[86]. Ako želimo da odredimo najmanju vrednost u tročlanoj listi, dovoljno je da prvo odredimo najmanju vrednost od prva dva elementa, a zatim tu vrednost uporedimo sa trećim elementom. Slično važi i za liste veće dužine. Prema tome, funkcija `min` nam je dovoljna za implementaciju funkcije `najmanji :: Ord a => [a] -> a`:

```
najmanji :: Ord a => [a] -> a
najmanji xs = foldl1 min xs
```

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

Poziv `najmanji [3, 4, 1, 2]`, svodi se na izraz `min (min (min 3 4) 1) 2`, koji se naravno redukuje u `1`

Treba napomenuti da funkcije `foldl1` i `foldr1` očekuju nepraznu listu. Ako se ovim funkcijama prosledi prazna lista, doći će do izuzetka (jer ove funkcije jednostano "nemaju" vrednost da vrate). Sa druge strane, kao što smo napomenuli, funkcije `foldl` i `foldr` mogu se primeniti i na prazne liste.

Da bi ste shvatili prednost `foldl1` i `foldr1` funkcija, pokušajte da implementirate funkciju iz prethodnog primera koristeći `foldl` ili `foldr`. Uvidećete da izbor inicijalne vrednosti nije uvek trivijalna stvar. Obratite pažnju da lista može da sadrži i negativne vrednosti.

[86] Setimo se, klasa `Ord` sadrži sve one tipove čiji se vrednosti mogu porediti sa `<`, `==`, `>`, itd... Prema tome za dve vrednosti nekog tipa iz `Ord`, uvek možemo odrediti koja je manja.

Zadatak 15. Implementirati funkciju `najveći :: Ord a => [a] -> a` koja vraća najveći element liste. Pretpostaviti da lista neće biti prazna.

Zadatak 16. Implementirati funkciju `svi :: [Bool] -> Bool`, koja vraća `True` samo ako su svi elementi u prosleđenoj listi tačni. Navesti implementaciju i sa `foldl` i sa `foldl1` funkcijom.

Zadatak 17. Implementirati funkciju `baremJedan :: [Bool] -> Bool` koja vraća `True` ako je barem jedan element u prosleđenoj listi tačan.

Zadatak 18. Naći n-tu parcijalnu sumu harmonijskog reda. Tačnije, napisati funkciju tipa `Int -> Float` koja u zavisnosti od parametra n računa sumu

$$\sum_{i=1}^{n} \frac{1}{i}.$$

Zadatak 19. Definisati funkciju `uCifru :: Char -> Int`, koja karaktere pretvara u cifre. Iskoristiti ovu funkciju za definiciju funkcije `uBroj :: [Char] -> Int` koja zapis prirodnog broja pretvara u celobrojnu vrednost. Na primer, `uBroj "123"` treba da se evaluira u vrednost `123`, itd... Pretpostaviti da će data niska predstavljati validan zapis prirodnog broja (tj. neće biti prazna i sadržaće samo cifre).

Rešenje. Funkcija `uCifru :: Char -> Int` se može lako definisati podudaranje oblika:

```
uCifru :: Char -> Int
uCifru '1' = 1
uCifru '2' = 2
uCifru '3' = 3
uCifru '4' = 4
uCifru '5' = 5
uCifru '6' = 6
uCifru '7' = 7
uCifru '8' = 8
uCifru '9' = 9
uCifru _ = 0
```

Alternativno, mogli smo da iskoristimo funkciju `fromEnum`.

Koristeći funkcije `map` i `uCifru` nisku možemo prevesti u listu cifara tj. vrednost tipa `[Int]`. Na primer `map uCifru "123"` nam daje `[1,2,3]`. Sada je potrebno rekurzivno proći kroz ovakvu listu cifara i formirati broj. Za to ćemo napisati pomoćnu funkciju `cifreUBroj :: [Int] -> Int`

Algoritam je najlakše sprovesti rekurzijom po poslednjem elementu liste (pokušajte i da implementirate algoritam rastavljajući listu na glavu i rep). Bazu rekurzije čini slučaj jednočlane liste. U tom slučaju kao povratnu vrednost vraćamo jedini element te liste (jer je vrednost jednocifrenog broja jednaka vrednosti jedine cifre tog broja). U suprotnom, rastavljamo listu na početak i na poslednji element, i računamo vrednost koju predstavlja početak. Tu vrednost početka pomeramo za jedno decimalno mesto (množeći sa `10`) i na to dodajemo vrednost poslednjeg elementa. Konkretno, ako nam je data lista `[1, 2, 3]`, rekurzivno nalazimo vrednost `cifreUBroj [1, 2]` što bi trebalo da nam da `12`. Množenjem sa `10` dobijamo vrednost `120`, koja kad je saberemo sa poslednjim elementom liste daje traženu vrednost funkcije. Kada razumemo postupak, kod nije teško napisati:

```
cifreUBroj :: [Int] -> Int
cifreUBroj [] = 0
cifreUBroj [x] = x
cifreUBroj xs = (cifreUBroj (init xs)) * 10 + (last xs)
```

Slučaj prazne liste je dodat da ne bi došlo do izuzetka prilikom prosleđivanja prazne liste.

Koristeći navedenu pomoćnu funkciju, zadatak je lako rešiti

```
uBroj :: [Char] -> Int
uBroj s = cifreUBroj (map uCifru s)
```

Zadatak 20. Napisati funkciju `daLiJeBroj :: [Char] -> Bool` koja proverava da li je data niska validan zapis prirodnog broja (tj. sastoji se samo od cifara).

Zadatak 21. Napisati funkciju `daLiJeCeoBroj :: [Char] -> Bool` koja proverava da li je data niska validan zapis celog broja (tj. sastoji se od cifara sa eventualnim znakom `-` na početku).

Zadatak 22. Reimplementirati funkciju `map` koristeći funkciju `foldl`.

Zadatak 23. Reimplementirati funkciju `filter` koristeći funkciju `foldl`.

Zadatak 24. Implementirati identičku funkciju nad listama `idList :: [a] -> [a]`, `idList xs = xs`, koristeći funkciju `foldl`.

8.4.4. Zip

Za uparivanje dve liste član-po-član u listu parova koristi se funkcija

```
zip :: [a] -> [b] -> [(a, b)].
```

Pritom, dužina liste parova koja se dobija kao rezultat je jednaka dužini kraćoj od prosleđenih lista.

Primer 10. Funkcija `zip` se često koristi za "numerisanje" članova neke liste. Tako možemo lako numerisati listu dana u nedelji:

```
ghci> dani = ["Pon", "Uto", "Sre", "Čet", "Pet", "Sub", "Ned"]
ghci> dani2 = zip [1 .. 7] dani
ghci> dani2
[(1,"Pon"),(2,"Uto"),(3,"Sre"),(4,"Čet"),(5,"Pet"),(6,"Sub"),(7,"Ned")]
```

Kada na rezultat `zip` funkcije primenjujemo `map` funkciju, tada je zgodnije koristiti funkciju

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

Funkcija `zipWith` kombinuje dve liste element-po-element koristeći prosleđenu funkciju.

Primer 11. (*Nastavak prethodnog primera*) Listu numerisanih dana možemo pretvoriti u listu niski na sledeći način

```
ghci> f (a, b) = show a ++ ". " ++ b
ghci> numerisaniDani = map f dani2
ghci> numerisaniDani
["1. Pon","2. Uto","3. Sre","4. Čet","5. Pet","6. Sub","7. Ned"]
ghci> head numerisaniDani
"1. Pon"
```

Funkcija `f` spaja broj i nisku u nisku. Primetimo da smo ovde koristili podudaranje oblika za definisanje funkcije.

Ali, ako nam je potrebna samo vrednost `numerisaniDani` a ne i `dani2`, možemo preskočiti korak u kom kreiramo vrednost `dani2`, i iskoristiti funkciju `zipWith`:

```
ghci> g a b = show a ++ ". " ++ b
ghci> numerisaniDani = zipWith g [1 .. 7] dani
```

Funkcija koju prosleđujemo funkciji `zipWith`, ne treba da uzima par već da bude funkcija višeg reda. Zbog toga smo definisali `g`.

Zadatak 25. Reimplementirati funkciju `zipWith` koristeći rekurziju.

Zadatak 26. Reimplementirati funkciju `zip` koristeći funkciju `zipWith`.

Zadatak 27. Reimplementirati funkciju `zipWith` koristeći funkcije `zip` i `map`.

8.5. Primer: Cezarova šifra

Po rimskom istoričaru Svetoniju, Julije Cezar je šifrovao vojne poruke tako što bi slovo *A* zamenio slovom *D*, slovo *B* slovom *E*, slovo *C* slovom *F*, i tako dalje, zamenjujući svako slovo latinskog alfabeta sa slovom koje se nalazi tri mesta nakon tog slova. Na primer, reč *CLEOPATRA* bi pri ovom šifrovanju postala *FOHRSDYVD*[87]. Onaj ko bi primio poruku, dešifrovao bi poruku tako što bi uradio sličnu zamenu, ali ovog puta zamenjujući svako slovo sa slovom koje se u alfabetu nalazi tri mesta pre. Iako Cezar nije prvi iskoristio ovakvu šifru, danas šifra nosi ime po njemu. Štaviše, pod istim imenom nazivaju se šifre koje "pomeraju" slova za n mesta, gde n ne mora biti baš 3.

Za implementaciju Cezarove šifre prvo ćemo konstruisati listu parova odgovarajućih karaktera (svaki par će sadržati slovo i šifrovano slovo). To možemo učiniti sa `zip` funkcijom

```
šifra :: [(Char, Char)]
šifra = zip ['A' .. 'Z'] ['D' .. 'Z']
```

Navedenim kodom smo uparili listu `['A' .. 'Z']` sa listom `['D' .. 'Z']`. Time dobijamo listu koja redom sadrži parove `('A','D')`, `('B','E')`, `('C','F')`, itd.. Međutim, kako `['A' .. 'Z']` ima 26, a `['D' .. 'Z']` samo 23 člana, lista `šifra` će imati samo 23 člana, i slova `'X'`, `'Y'` i `'Z'` neće imati para.

Da bismo prevazišli problem, na kraj liste `['D' .. 'Z']` dodaćemo karaktere `['A', 'B', 'C']`. Na ovaj način `'X'` će se šifrovati u (upariti sa) `'A'`, `'Y'` u `'B'` i `'Z'` u `'C'`.

```
šifra :: [(Char, Char)]
šifra = zip ['A' .. 'Z'] (['D' .. 'Z'] ++ ['A', 'B', 'C'])
```

[87] Za ovaj primer smo koristili rimski alfabet koji ne sadrži *J*, *U* i *W*. Nadalje koristimo engleski alfabet.

```
ghci> šifra
[('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 listu `šifra` možemo iskoristiti da šifrujemo pojedinačne karaktere. Ideja je sledeća: za dati karakter `k :: Char`, iz liste `šifra` uzećemo par čija prva koordinata je jednaka vrednosti `k`. Šifrovan karakter će biti druga koordinata tog para. Za pronalazak odgovarajućeg para, koristimo funkciju `filter`.

```
šifrujKarakter :: Char -> Char
šifrujKarakter k = snd (head šifrovano)
  where
    šifrovano = filter (\par -> fst par == k) šifra
```

Na primer, ako je `k = 'A'`, tada će `filter` u listi `sifrovano` ostaviti samo uređene parove čija je prva koordinata `'A'`, a to je samo par `('A', 'D')`. Dakle, `šifrovano` će biti *lista* `[('A', 'D')]`. Prvom i jedinom elementu te liste ćemo pristupiti sa funkcijom `head`, a drugoj koordinati tog elementa ćemo pristupiti sa funkcijom `snd`. Time dobijamo `šifrujKarakter 'A' = 'D'`.

Funkcija `šifrujKarakter` šifruje velika slova engleske abecede. Ali, ako pozovemo `šifrujKarakter` nad karakterima `'a'`, `'9'` ili `'ш'`, doći će do katastrofalnih rezultata. Naime, u takvim slučajevima lista `šifrovano` će biti prazna, a pokušaj pristupa prvom članu prazne liste (`head šifrovano`) će dovesti do prekida programa. Zbog toga, pre pristupa elementu liste `šifrovano` moramo proveriti da li je lista prazna. Ako jeste prazna, onda ćemo vratiti prosleđeni karakter, a u suprotnom ćemo vratiti šifrovan karakter.

```
šifrujKarakter :: Char -> Char
šifrujKarakter k
  | null šifrovano = k
  | otherwise      = snd (head šifrovano)
  where
    šifrovano = filter (\par -> fst par == k) šifra
```

Šifrovana niska dobija se primenom funkcije `šifrujKarakter` na svaki karakter te niske:

```
šifruj :: [Char] -> [Char]
šifruj niska = map šifrujKarakter niska
```

```
ghci> šifruj "NAPAD NA GALE VECERAS U OSAM"
"QDSDG QD JDOH YHFHUDV X RVDP"
```

Pošto se razmak `' '` ne nalazi u listi `šifra`, važi `šifrujKarakter ' ' = ' '`. Samim tim, prilikom šifrovanja razmaci se čuvaju.

Funkcija koja dešifruje poruku je potpuno analogna funkciji koja je šifruje, samo što je uloga koordinata obrnuta. Pretragu vršimo po drugoj koordinati, a vraćamo prvu:

```
dešifrujKarakter :: Char -> Char
dešifrujKarakter k
  | null dešifrovano = k
  | otherwise      = fst (head dešifrovano)
  where
```

```
    dešifrovano = filter (\par -> snd par == k) šifra

dešifruj :: [Char] -> [Char]
dešifruj niska = map dešifrujKarakter niska
```

Zadatak 28. Implementacija Cezarove šifre koju smo naveli je dobra za demonstraciju rada sa listama u Haskelu, ali nije mnogo efikasna. Umesto da za svako slovo pretražujemo listu `šifra`, možemo iskoristiti

```
fromEnum :: Enum a => a -> Int
```

i

```
toEnum :: Enum a => Int -> a
```

funkcije da bi konvertovali slova u *ASCII* vrednosti i obrnuto. Koristeći navedene funkcije, kao i funkciju ostataka `mod`, dati alternativnu definiciju funkcije `šifrujKarakter`.

Zadatak 29. Koristeći prethodni zadatak, implementirati Cezarovu šifru sa proizvoljnim pomerajem n.

Zadatak 30. Vižnerova šifra (po francuskom kriptografu Blezu de Vižneru) predstavlja uopštenje Cezarove šifre. Za Vižnerovu šifru, potrebno je odabrati *ključ* koji će se koristiti prilikom šifrovanja i dešifrovanja. Ključ je jedna reč, čija slova određuju pomeraje za Cezarovu šifru. Uzmimo ponovo prethodni primer tajne poruke

```
"NAPAD NA GALE VECERAS U OSAM",
```

a za ključ uzmimo `"RIM"`. Pri Vižnerovom šifrovanju, prvi karakter poruke, `'N'`, će se šifrovati sa Cezarovom šifrom sa pomerajem 18, jer je prvi karakter ključa, `'R'`, osamnaesto slovo u abecedi. Drugi karakter će se šifrovati sa pomerajem 9 (jer je `'I'` deveto slovo abecede), a treći sa pomerajem 13 (jer je `'M'` trinaesto slovo abecede). Pošto smo stigli do kraja ključa, ključ koristimo ispočetka, i četvrto slovo poruke, `'A'` šifrujemo sa pomerajem 18. Ovaj proces nastavljamo, ponavljajući ključ, sve dok ne stignemo do kraja poruke. Implementirati Vižnerovu šifru.

8.6. Zadaci

Zadatak 31. Napisati program koji vraća listu prvih n Fibonačijevih brojeva.

Zadatak 32. Svi prirodni brojevi manji od 10 koji su deljivi sa 3 ili 5 su 3, 5, 6 i 9. Njihov zbir je 23. Naći zbir svih prirodnih brojeva manjih od 100 koji su deljivi sa 3 ili 5.

Zadatak 33. Palindromski broj je prirodan broj koji se čita isto i sa leva i sa desna. Najveći palindromski broj koji je proizvod dva dvocifrena broja je 9009, jer je

$$9009 = 91 \cdot 99.$$

Naći najveći palindromski broj koji je proizvod dva trocifrena broja.

Zadatak 34. Implementirati funkciju `dekartovProizvod` koja vraća "Dekartov proizvod dve liste". Na primer: `dekartovProizvod [1, 2, 3] ['a', 'b']` daje

```
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')].
```

Zadatak 35. Naći sve početke date liste. Na primer `počeci [1, 2, 3]` bi trebalo da nam vrati `[[], [1], [1, 2], [1, 2, 3]]`.

Zadatak 36. Napisati funkciju `kodiraj :: [Char] -> [(Char, Int)]` koja kodira nisku tako što je predstavlja kao listu uređenih parova karaktera i prirodnih brojeva. Broj označava dužinu uzastopnog ponavljanja datog karaktera. Na primer: `kodiraj "aaaabb"` daje

```
[('a', 4), ('b', 2)],
```

a `kodiraj "Google"` daje

```
[('G', 1), ('o', 2), ('g', 1), ('l', 1), ('e', 1)].
```

Napisati i funkciju `dekodiraj :: [(Char, Int)] -> [Char]`.

Zadatak 37. Napisati funkciju `rotiraj :: Int -> [a] -> [a]` koja "rotira" listu tako što svaki element pomera za `n` mesta u levo, pri čemu se elementi sa kraja liste vraćaju na početak: `rotiraj 1 "ABCD"` daje `"BCDA"`, a `rotiraj 2 "ABCD"` daje `"CDAB"`, itd...

Rešenje. Pomoć: Implementirati pomoćnu funkciju koja rotira listu za jedan element. Opštu funkciju implementirati rekurzijom po broju rotiranja koristeći pomoćnu funkciju.

Zadatak 38. Implementirati funkciju

```
srednjaVrednost :: [Float] -> Float
```

koja računa srednju vrednost liste. Implementirati funkciju

```
disperzija :: [Float] -> Float
```

koja računa disperziju vrednosti liste.

Zadatak 39. Data je lista prirodnih brojeva sa $n > 0$ članova. Element na poziciji $k \leq n$ predstavlja cenu neke akcije na berzi u k-tom danu. Potrebno je u jednom od ponuđenih dana kupiti akciju, a zatim u nekom od narednih dana (uključujući i dan kupovine) i prodati akciju. Naravno, cilj je ostvariti što veći profit (profit računamo kao razliku prodajne i kupovne cene). Napisati funkciju `profit :: [Int] -> Int` koja uzima cene akcija tokom vremena, i vraća najveći profit koji je moguće ostvariti trgovinom akcija. Akcija se može samo jednom kupiti i prodati, prodaja se mora odviti nakon kupovine ili istog dana. Pretpostaviti da prosleđena lista nije prazna. Na primer, `profit [7,1,5,3,6,4] = 5` jer kupovinom drugog dana i prodajom petog dana ostvarujemo profit $6 - 1 = 5$. Ne možemo ostvariti dobit $7 - 1 = 6$ jer se prodaja mora ostvariti nakon kupovine (a 7 je pre 1).

9. Tipovi i vrste

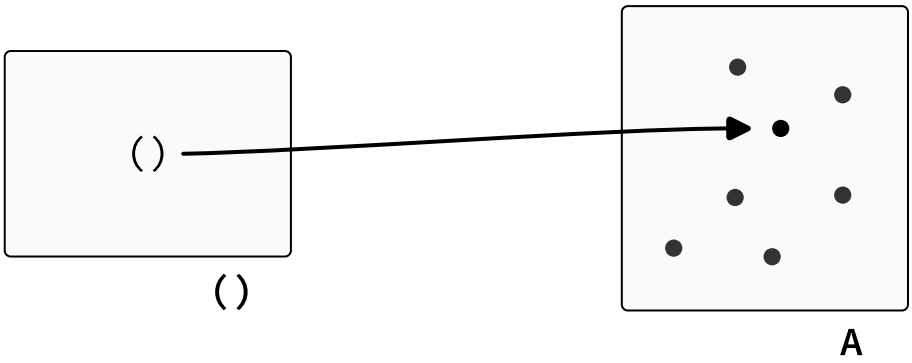
U dosadašnjem izlaganju uvideli smo koliko je sistem tipova bitan za Haskel. Razmotrili smo neke osnovne tipove, navikli smo da govorimo o tipu funkcija i tipskim promenljivama, i ukratko smo opisali klase tipova kao kolekcije tipova.

9.1. Dva posebna tipa

Najjednostavniji tip sa kojim smo se upoznali je `Bool` koji sadrži samo dve vrednosti. Sada ćemo predstaviti tip koji sadrži samo jednu vrednost, kao i tip koji ne sadrži nijednu vrednost.

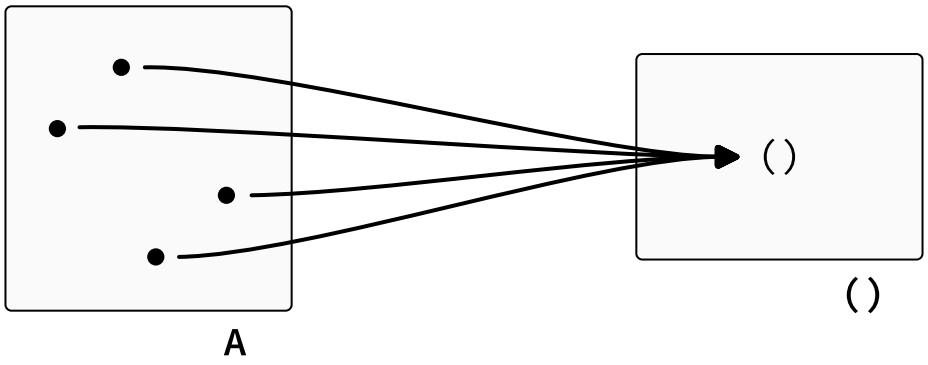
Tip koji sadrži jednu vrednost označavamo sa `()` i nazivamo **jedinični tip**[88]. Jedinu vrednost ovog tipa označavamo takođe sa `()`. Dakle `() :: ()`.

Svaka totalna[89] funkcija je u potpunosti određena sa slikama svih vrednosti domena te funkcije. Ako je data totalna funkcija `f :: () -> A`, za neki tip `A`, tada je ta funkcija u potpunosti određena sa `f ()`. Možemo da kažemo da `f` "bira" jedinstvenu vrednost tipa `A`.



Sa druge strane, ako je data totalna funkcija `g :: A -> ()`, tada za svaku vrednost `x :: A` mora važiti `g x = ()` (jer je `()` jedina vrednost kodomena). Stoga, za svaki tip `A` važi da postoji *samo jedna* totalna funkcija tipa `A -> ()`, i ta funkcija je definisana sa

```
\_ -> ().
```



Tip koji ne sadrži nijednu vrednost označava se sa `Void` i nazivamo ga **prazan tip**[90].

Posmatrajmo sada totalnu funkciju `s :: A -> Void`. Za `x :: A`, `s x` bi morala biti vrednost tipa `Void`. Vrednost tipa `Void` ne postoji, iz čega sledi da funkcija `s` ne može postojati ako je tip `A` neprazan. Prema tome, jedina totalna funkcija čiji kodomen je `Void` je jedinstvena funkcija tipa `Void -> Void`.

[88] *unit type*

[89] U nastavku isključivo govorimo o totalnim funkcijama, odnosno o funkcijama koje su definisane za svaku vrednost domena.

[90] *void type*

Sa druge strane, za svaki tip `A`, postoji jedinstvena totalna funkcija[91] tipa `Void -> A`. Ali kako je nemoguće konstruisati vrednost tipa `Void`, takve funkcije ne možemo nikad "pozvati".

Primetimo da tip `Void` ne odgovara pojmu `void` koji se može naći u drugim programskim jezicima poput C-a, C++-a, Jave. U takvim programskim jezicima, `void` se koristi za funkcije čija povratna vrednost nas ne interesuje. U Haskelu, tip `()` se koristi kad nas ne interesuje povratna vrednost.

> **Zadatak 1.** Koliko postoji totalnih funkcija tipa `() -> Bool`?
>
> *Rešenje.* Postoji onoliko funkcija tipa `() -> T` koliko postoji vrednosti tipa `T`. Prema tome, postoje dve funkcije tipa `() -> Bool`, koje su definisane sa
>
> ```
> f1 () = True
>
> f2 () = False
> ```
>
> Kako tip `()` sadrži samo jednu vrednost, podudaranje oblika je trivijalno. Zapravo, kako su obe navedene funkcije konstantne, mogli smo da koristimo i džoker _.

Iako tipovi `()` i `Void` na prvi pogled deluju beskorisno, ti tipovi imaju i teorijsku i praktičnu važnost. Takođe, ova dva tipa će nam poslužiti kao dodatni primeri pri ilustraciju naprednih koncepata.

> **Zadatak 2.** Koliko vrednosti sadrži tip `[Void]`?
>
> *Rešenje.* Tip `[Void]` sadrži samo jednu vrednost: `[]`.

> **Zadatak 3.** Koliko vrednosti sadrži tip `((), ())`?
>
> *Rešenje.* Tip `((), ())` sadrži samo jednu vrednost: `((), ())`.

> **Zadatak 4.** Koliko vrednosti sadrži tip `(Bool, Void)`?

> **Zadatak 5.** Posmatrajmo i parcijalne funkcije (funkcije koje nisu totalne). Koliko postoji funkcija (parcijalnih i totalnih) tipa `() -> Bool`?

9.2. Tipski sinonimi

Do sada smo već upoznali neke načine konstrukcije novih tipova. Na primer, od od nekoliko tipova T_1, T_2, ... T_n možemo napraviti odgovarajući tip uređenih n-torki $(T_1, T_2, \ldots T_n)$. Pri ovakvim konstrukcijama često se dobijaju tipovi koji nisu baš kratki za pisanje, a što je još bitnije, ni za čitanje[92].

Da bismo izbegli navedene probleme, možemo definisati **tipski sinonim**. Sinonim predstavlja novo ime *već postojećeg tipa*, i može se koristiti umesto originalnog imena. Tipski sinonim se definiše sa deklaracijom poput naredne:

```
type Ime = PostojeciTip
```

Novo ime, kao i ime svakog tipa, mora početi velikim slovom. Naravno, nije dozvoljeno koristiti već zauzeta imena tipova.

[91] Takva polimorfna funkcija je poznata pod imenom `absurd`.

[92] Na primer, funkcija koja sabira dva dvodimenzionalna vektora ima tip

```
(Float, Float) -> (Float, Float) -> (Float, Float)
```

Primer 1. Možemo tipu `Bool` dodeliti sinonim `Logicki` na sledeći način

```
type Logicki = Bool
```

Sada na svim mestima na kojima se pojavljuje `Bool`, možemo koristiti ime `Logicki`:

```
a :: Logicki
a = True || False
```

Učitavanjem navedenog koda u *GHC*-i, možemo proveriti tipove:

```
ghci> :t a
Logicki
ghci> a && False
False
ghci> :t (a && False)
Bool
```

Tip `Logicki` je u potpunosti isti kao tip `Bool`. Ali, ako tip neke vrednosti nije eksplicitno definisana kao `Logicki`, tada će "prednost imati" originalno ime.

Naravno, tipski sinonim za već kratko ime nije mnogo koristan. Pogledajmo zato reprezentativniji primer.

Primer 2. Uređenoj trojci `Float` brojeva možemo dodeliti kratko ime `Vektor3`:

```
type Vektor3 = (Float, Float, Float)
```

Sa ovakvim sinonimom, definicije postaju mnogo preglednije

```
zbirVektora :: Vektor3 -> Vektor3 -> Vektor3
zbirVektora (a, b, c) (x, y, z) = (a + x, b + y, c + z)
```

U Prelidu već je definisan naredni tipski sinonim

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

9.3. Novi tipovi

Slično definisanju tipskog sinonima, možemo definisati i potpuno novi tip od postojećeg tipa. Definicija novog tipa ima naredni oblik

```
newtype NoviTip = Konstruktor PostojeciTip
```

Gornjim kodom definisan je tip `NoviTip`, koji se može shvatiti kao kopija tipa `PostojeciTip`. Svaka vrednost u tipu `NoviTip` odgovara nekoj vrednosti u tipu `PostojeciTip`, ali ova dva tipa nemaju zajedničke vrednosti.

U gore navedenoj definiciji, reč `Konstruktor` predstavlja ime *konstruktora*. **Konstruktor** je *funkcija* kojim se konstruišu vrednosti tipa `NoviTip`. Svaka vrednost tipa `NoviTip` je oblika `Konstruktor x` za neko `x :: PostojeciTip`. Ime konstruktora i ime novog tipa, ne smeju biti već zauzeti i moraju počinjati velikim

slovom[93]. Međutim, dozvoljeno je da tip i konstruktor imaju isto ime.

Veoma bitna razlika između tipova definisanih sa `newtype` i tipskih sinonima je ta što sa `newtype` dobijamo u potpunosti novi tip. Stoga nije moguće novi tip koristiti sa funkcijama koje su namenjene za postojeći tip (kao što je to bio slučaj sa tipskim sinonimima).

Primer 3. Od tipa `Bool` možemo kreirati novi tip `MyBool`:

```
newtype MyBool = MkMyBool Bool
```

Često se za ime konstruktora uzima ime tipa ispred kojeg se stavlja *Mk* (skraćeno od *Make*), jer konstruktor predstavlja jedini način da se "napravi" vrednost tipa `MyBool`.

Kreirani tip `MyBool` sadrži samo dve vrednosti. Te vrednosti su `MkMyBool True` i `MkMyBool False`. Ove dve vrednosti nisu isto što i `True` i `False`! Na primer, možemo se uveriti da vrednosti tipa `MyBool` ne mogu se koristiti sa logičkim operatorima:

```
ghci> a = (MkMyBool True) || (MkMyBool False)

<interactive>:1:6: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘MyBool’
    • In the first argument of ‘(||)’, namely ‘(MkMyBool True)’
      In the expression: (MkMyBool True) || (MkMyBool False)
      In an equation for ‘a’: a = (MkMyBool True) || (MkMyBool False)

<interactive>:1:25: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘MyBool’
    • In the second argument of ‘(||)’, namely ‘(MkMyBool False)’
      In the expression: (MkMyBool True) || (MkMyBool False)
      In an equation for ‘a’: a = (MkMyBool True) || (MkMyBool False)
```

Prema tome za vrednosti tipa `MyBool` moramo implementirati posebnu funkciju disjunkcije:

```
ili :: MyBool -> MyBool -> MyBool
ili (MkMyBool False) (MkMyBool False) = MkMyBool False
ili _ _ = MkMyBool True
```

Svaka vrednost tipa `MyBool` je oblika `MkMyBool x`. Haskel dozvoljava da upotrebimo tehniku podudaranja oblika u ovom slučaju. Prva linija definicije odnosi se samo na slučaj kada su obe prosleđene vrednosti baš `MkMyBool False`. Svi ostali slučajevi su obuhvaćeni drugom linijom, u kojoj se koriste džokeri.

Nakon učitavanja navedenog koda, možemo naći disjunkciju novih logičkih vrednosti

```
ghci> a = ili (MkMyBool True) (MkMyBool False)
```

Za sada sve deluje dobro. Kreirali smo novi tip, i napisali smo jednu funkciju čiji je domen i kodomen novi tip. Ali ako u interaktivnom okruženju pokušamo da ispišemo vrednost `a`, dobićemo grešku:

```
ghci> a
```

[93] S obzirom da su konstruktori funkcije, u ovom slučaju nije ispoštovano pravilo da su imena sa velikim početnim slovom rezervisana za tipove, a imena sa malim početnim slovom za vrednosti. Ali kako su konstruktori uvek "vezani" za tipove, ovakvo odstupanje ima smisla.

```
<interactive>:1:1: error:
    • No instance for (Show MyBool) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Razlog nastanka greške je taj što program *GHCi* ne zna kako da ispiše vrednost tipa `MyBool`. Za ispisivanje svih vrednosti, *GHCi* koristi `show` funkciju koja je deklarisana u `Show` klasi tipova. Kako tip `MyBool` nije instanca ove klase, *GHCi* ne može da iskoristi funkciju `show` sa vrednostima tipa `MyBool`.

Navedeni problem se rešava uvođenjem tipa `MyBool` u klasu `Show`, što će biti detaljno objašnjeno u lekciji *Klase tipova*. Srećom, u ovom slučaju instanciranje može biti automatsko. Dovoljno je nakon definicije tipa dopisati `deriving Show`. Automatsko instanciranje poput navedenog nazivamo **izvođenje**.

```
newtype MyBool = MkMyBool Bool deriving Show
```

Za sada nećemo detaljnije objašnjavati `deriving Show`.

Ponovnim učitavanjem koda, možemo bez problema ispisivati vrednosti novog tipa

```
ghci> a = ili (MkMyBool True) (MkMyBool False)
ghci> a
MkMyBool True
```

Zašto je `newtype` konstrukcija korisna ako već imamo tipske sinonime? Deluje da sa `newtype` samo komplikujemo kod. Funkcije koje smo ranije koristili, moramo reimplementirati za novi tip. Dobar razlog korišćenja `newtype`-a je taj, što ponekad namerno želimo da budemo ograničeni sa sistemom tipova, što prikazuje primer koji sledi.

Primer 4. Zamislimo da radimo na bankarskom programu, koji se mora osposobiti za rad sa svotama u evrima i dolarima. U kodu možemo definisati dva tipa, čije vrednosti predstavljaju svote novca:

```
newtype Eur = MkEur Double

newtype Usd = MkUsd Double
```

Vrednosti oba definisana tipa sadrže suštinski iste podatke (jedan *double* broj). Kako su u pitanju različiti tipovi, kompajler neće dozvoliti proizvoljno mešanje ovih tipova (kao što bi bio slučaj sa tipskim sinonimima). Na taj način, na primer, možemo napisati funkciju koju je moguće isključivo primeniti na svote novca u evrima. Pokušaj primene takve funkcije na svote u dolarima dovešće do tipske greške i program neće biti iskompajliran.

Ograničenja poput opisanih ne deluju mnogo korisna kada se posmatraju mali programi. Ali sa programima s velikim brojem linija ovakva ograničenja značajno “umanjuju brigu” pri menjanju koda. Na primer, promena funkcija koje koriste tip `Eur` neće uopšte uticati na funkcije koje koriste tip `Usd`.

Naravno, razdvajanje valuta ima smisla samo na onom delu programa koji različito obrađuje svote u evrima i dolarima (u ovom slučaju to može biti obračunavanje poreza koji je specifičan za valutu, i sl.). Deo logike koji je isti za obe valute, ne treba duplirati. Izbegavanje dupliranja koda se može postići sa tipskim klasama, što će biti prikazano u narednoj lekciji.

Takođe, `newtype` konstrukcija je specijalan slučaj jedne opštije, i mnogo korisnije konstrukcije, kojoj je posvećena lekcija *Algebarski tipovi podataka*.

9.4. Apstraktni tipovi

Ponekad je slučaj da se u kodu konstruiše nekoliko sličnih tipova, koji se samo razlikuju u jednom "delu":

Primer 5. Za predstavljanje trodimenzionalnih vektora možemo konstruisati tipove:

```
newtype VecInt = MkVecInt (Int, Int, Int)
newtype VecFloat = MkVecFloat (Float , Float, Float)
newtype VecDouble = MkVecDouble (Double, Double, Double)
```

Za ovako definisane tipove nije teško konstruisati odgovarajuće funkcije sabiranja, skaliranja, vektorskog i skalarnog množenja, itd..

```
sumVecInt :: VecInt -> VecInt -> VecInt
sumVecInt (MkVecInt (a,b,c)) (MkVecInt (x,y,z)) = MkVecInt (a+x, b+y, c+z)

sumVecFloat :: VecFloat -> VecFloat -> VecFloat
sumVecFloat (MkVecFloat (a,b,c)) (MkVecFloat (x,y,z)) = MkVecFloat (a+x, b+y, c+z)

sumVecDouble :: VecDouble -> VecDouble -> VecDouble
sumVecDouble (MkVecDouble (a,b,c)) (MkVecDouble (x,y,z)) = MkVecDouble (a+x, b+y,
c+z)
```

Kao što vidimo, svaki numerički tip (`Int`, `Float`, `Double`, ...) zahteva posebnu definiciju odgovarajućeg tipa vektora, što dovodi do toga da imamo mnogo međusobno sličnih definicija.

Da bismo pri definiciji tipova izbegli nepotrebno dupliranje koda, možemo koristiti apstraktne tipove. **Apstraktni tip** je tip koji zavisi od nekog drugog tipa. U nekom smislu, apstraktni tipovi predstavljaju funkciju čiji su argumenti tipovi! Apstraktni tip se definiše tako što se nakon imena tipa navede tipska promenljiva, a nakon konstruktora tip koji potencijalno zavisi od te tipske promenljive[94].

Primer 6. Za jednoobraznu konstrukciju trodimenzionalnih vektora, možemo koristiti naredni apstraktni tip

```
newtype Vec a = MkVec (a, a, a)
```

Na opisani način konstruisana je jedna funkcija na nivou tipova.

Tipovi iz prethodnog primera nam nisu više neophodni, i sada možemo da koristimo `Vec Int`, `Vec Float` i `Vec Double`:

```
sumVecInt :: Vec Int -> Vec Int -> Vec Int
sumVecInt (MkVec (a,b,c)) (MkVec (x,y,z)) = MkVec (a+x,b+y,c+z)

sumVecFloat :: Vec Float -> Vec Float -> Vec Float
sumVecFloat (MkVec (a,b,c)) (MkVec (x,y,z)) = MkVec (a+x,b+y,c+z)

sumVecDouble :: Vec Double -> Vec Double -> Vec Double
sumVecDouble (MkVec (a,b,c)) (MkVec (x,y,z)) = MkVec (a+x,b+y,c+z)
```

[94] Ovakva definicija podseća na podudaranje oblika kod "običnih" funkcija.

> I dalje nismo umanjili broj definicija funkcija, ali možemo primetiti da je karakteristično za tipove `Int`, `Float` i `Double` to što se vrednosti ovih tipova mogu sabirati. Sva tri navedena tipa pripadaju `Num` klasi, pa možemo tri definicije zameniti sa jednom:
>
> ```
> sumVec :: Num a => Vec a -> Vec a -> Vec a
> sumVec (MkVec (a,b,c)) (MkVec (x,y,z)) = MkVec (a+x,b+y,c+z)
> ```

Nije nužno da apstraktan tip zavisi samo od jednog parametra, ili da ne sadrži konstantne tipove u sebi. Na primer, sledeći apstraktni tip je u potpunosti validan

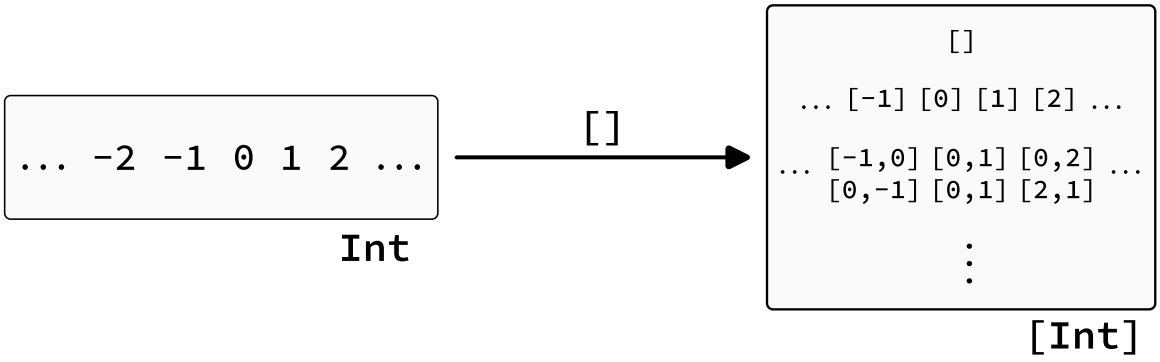
```
newtype A a b c = MkA (a, [b], Int -> c)
```

Navedeni apstraktni tip je zasigurno veoma čudan, ali validan. Sada izraz `A Char Bool Float` predstavlja tip čije vrednosti `MkA x` sadrže vrednosti tipa `(Char, [Bool], Int -> Float)`.

Apstraktni tipovi, nazivaju se još i **konstruktori tipa**, jer kao što ime kaže, oni konstruišu tipove. U tom smislu, konstruktori koji konstruišu vrednosti se nazivaju **konstruktori vrednosti**. U gornjem bloku koda `A` je konstruktor tipa, a `MkA` konstruktor vrednosti.

Mi smo se do sada susreli sa dva apstraktna tipa a da toga nismo bili ni svesni. Oba apstraktna tipa imaju specifičnu sintaksu, koja se razlikuje od onoga što smo gore opisali.

Prvi apstraktni tip koji nam je poznat je konstruktor tipa liste, koji se obeležava sa `[]`. Ovaj konstruktor preslikava tip `A` u tip `[A]`[95]. U lekciji *Rekurzivni tipovi podataka*, videcemo kako je tačno `[]` definisan, i dodatno ćemo razjasniti ovaj tip.



Drugi apstraktni tip koji nam je poznat je konstruktor tipa funkcije `->`, takozvana *strelica*[96]. Konstruktor `->` uzima dva konkretna tipa `A` i `B` i daje tip funkcija iz `A` u `B`. Taj tip se naravno označava sa `A -> B`. Ovde je korisno napomenuti da su lambda izrazi ništa drugo nego konstruktori vrednosti za tip `A -> B`. Za razliku od apstraktnog tipa `[]`, tip `->` je interno implementiran u Haskel jeziku.

9.5. Vrste

Apstraktni tipovi jesu funkcije koje tipove preslikavaju u tipove. Ali kao i sa svim preslikavanjima, potrebno je biti pažljiv pri primeni funkcija na vrednost.

Vratimo se za trenutak korak u nazad, i posmatrajmo vrednosti. Sistem tipova nam govori nema smisla primeniti funkciju tipa `Int -> Int` na vrednost tipa `Char`. Još manje smisla ima primena vrednosti tipa `Bool` na vrednost tipa `Int`, jer prva vrednost nije čak ni funkcija.

[95] Možda će biti jasnije ako `[A]` zapišemo kao `[] A` (što je validan zapis u kodu). Pomalo zbunjujuće, ali i konstruktor *vrednosti* prazne liste se takođe označava sa `[]`.

[96] eng. *arrow*.

Iz istih razloga potrebno je klasifikovati tipove i apstraktne tipove na sličan način kao što su vrednosti klasifikovane u tipove. Drugim rečima, potrebno je uvesti pojam "tipa" za tipove[97]. Ovakav "tip tipa" nazivamo **vrsta**[98].

Ako neki tip ili apstraktni tip `T` poseduje vrstu `v`, tada to označavamo sa `T :: v`, baš kao i tip vrednosti.

U Haskelu, vrsta koja obuhvata sve tipove koji nisu apstraktni označava se sa `*`. Za svaki tip `T` koji nije apstraktan kažemo da je vrste `*`, i pišemo `T :: *`. Za ovakve tipove kažemo još da su **konkretni tipovi** (ili *monotipovi*).

Sa druge strane, apstraktni tipovi poseduju vrstu oblika `P -> Q` gde su `P` i `Q` takođe neke vrste. Vrsta oblika `P -> Q` sugeriše da su apstraktni tipovi funkcije koje tipove preslikavaju tipove.

Primer 7. Tipovi `Bool`, `Int`, `Float`, `[Int]`, `Bool -> Int`, `()`, `(Int, [Bool])`, itd... su svi vrste `*`.

Primer 8. Neka je apstraktni tip `AbsType` definisan sa

```
newtype AbsType a = MkAbsType a.
```

Tip `AbsType` uzima jedan konkretan tip i konstruiše konkretan tip, pa je `AbsType` vrste `* -> *`. Prema tome, izraz `AbsType AbsType` nema smisla, ali izraz `AbsType Int` ima smisla i predstavlja tip.

Primer 9. Uzmimo primer apstraktnog tipa iz prošle sekcije

```
data A a b c = MkA (a, [b], Int -> c)
```

Navedeni apstraktan tip ima vrstu `* -> (* -> (* -> *))` koja se naravno jednostavno zapisuje kao

```
* -> * -> * -> *.
```

I kao što *GHCi* naredba `:type` može odrediti tip neke vrednosti, tako i naredba `:kind` može odrediti vrstu nekog tipa.

Primer 10. Nastavimo sa prethodnim primerom u interaktivnom okruženju:

```
ghci> data A a b c = MkA (a, [b], Int -> c)
ghci> :kind A
A :: * -> * -> * -> *
```

Primer 11. Ako niste do sad verovali, `[]` zaista jeste apstraktan tip:

```
ghci> :kind []
* -> *
```

Primer 12. I strelica jeste apstraktan tip. Strelica uzima dva konkretna tipa da bi dala konkretan tip:

[97] Klase tipova jesu kolekcije tipova, ali su nedovoljne za klasifikaciju koja nam je potrebna. Klase tipova nisu disjunktne (jedan isti tip može pripadati različitim klasama), a nama su ovde potrebne disjunktne kolekcije.

[98] *kind*

```
ghci> :kind (->)
(->) :: * -> * -> *
```

Obratite pažnju da strelica u tipu i strelica u vrsti nema isto značenje.

10. Moduli

Ova knjiga je organizovana u celine (*lekcije*) da bi materija koja se izlaže bila pristupačnija za čitanje. Koncepti koji su povezni izloženi su u okviru iste lekcije, što značajno olakšava i nalaženje informacija. Iz potpuno istih razloga, u svakom programskom jeziku velike kodove želimo da podelimo u posebne datoteke (i direktorijume), grupišući povezane koncepte u celine. Na taj način prvenstveno sami sebi olakšavamo snalaženje u kodu. Haskel, kao i svaki drugi moderan programski jezik, podržava grupisanje koda.

U Haskelu, **modul** je osnovna jedinica podele programa. Jedan modul sastoji se iz definicija funkcija, tipova i klasa tipova. Ime modula mora početi velikim slovom i može sadržati samo alfanumeričke karaktere kao i znak tačke. Modul se definiše, tako što se na početku `.hs` datoteke navede linija

```
module ImeModula where.
```

Primer 1. Modul `GeometrijaKruga` definiše dve funkcije koje su mogu uvesti u druge module.

```
module GeometrijaKruga where

površina :: Float -> Float
površina r = r^2 * pi

obim :: Float -> Float
obim r = 2 * pi * r
```

Svaka Haskel datoteka može sadržati definiciju samo jednog modula, i obrnuto, svaki modul mora biti sadržan u jednoj datoteci. Prema tome, moduli se mogu poistovetiti sa `.hs` datotekama. Neophodno je imenovati `.hs` datoteke imenom modula (sve sa početnim velikim slovom)[99].

Datoteke sa Haskel kodom možemo dalje podeliti u direktorijume. Neophodno je da sama imena modula prate hijerarhiju koja je ustanovljena direktorijumima, tako što ćemo imena direktorijuma postaviti u ime modula i razdvojiti ih tačkom. Pri tome, sve putanje se uzimaju u odnosu na korenski direktorijum projekta, tj. u odnosu na direktorijum u kom pokrećemo *GHCi*. Na primer, pretpostavimo da u direktorijumu projekta imamo poddirektorijum `Životinje`. Ako je datoteka `Mačka.hs` smeštena u direktorijumu `Životinje`, modul u datoteci `Mačka.hs` ćemo imenovati `Životinje.Mačka`. Ako se u direktorijumu `Životinje` nalazi direktorijum `Ribe` i u tom direktorijumu se nalazi datoteka `Tuna.hs`, tada ćemo modul u toj datoteci imenovati `Životinje.Ribe.Tuna`. I analogno za složenije slučajeve. Kao i datoteke, i direktorijume je neophodno imenovati sa početnim velikim slovom, koristeći samo alfanumeričke karaktere.

10.1. Izvoz definicija

Dekleracijom modula sa konstrukcijom `module X where` sve definicije koje se nalaze u istoj datoteci biće dostupne za uvoz u druge datoteke. Tj. na ovaj način sve definicije će biti *javne*.

Često ne želimo da sve definicije iz nekog modula učinimo dostupnim za uvoz u druge module, tj. želimo da neke definicije ostavimo *privatnim*. To se postiže tako što ćemo u dekleraciji modula, odmah nakon imena modula postaviti uređenu n-torku imena koje želimo da budu javni. Sva imena koja se ne nalaze u ovoj n-torci biće privatna. Poredak kojim su navedene imena nije važan. Naravno, ako nakon imena modula navedemo n-torku dužine 0, tj. `()`, sve definicije će biti privatne.

[99] Iako sam Haskel jezik ne uslovljava da naziv datoteke i naziv modula moraju da budu isti, svi alati (uključujući *GHC* i *GHCi*) koriste ovu konvenciju. Ako koristite samo jednu `.hs` datoteku kroz *GHCi*, onda ime datoteke nije mnogo važno. Ali u iole složenijoj postavci, svako odstupanje od konvencija koje su izložene u ovoj lekciji će vam zadati glavobolje.

Primer 2. Modul `GeometrijaValjka` definiše četiri funkcije, ali samo su `površina` i `zapremina` javne definicije:

```
module GeometrijaValjka (površina, zapremina) where

površinaOsnovice :: Float -> Float
površinaOsnovice r = r^2 * pi

površinaOmotača :: Float -> Float -> Float
površinaOmotača r h = 2 * pi * r * h

površina :: Float -> Float -> Float
površina r h = 2 * (površinaOsnovice r) + (površinaOmotača r h)

zapremina :: Float -> Float -> Float
zapremina r h = h * (površinaOsnovice r)
```

Postoji nekoliko dobrih razloga zašto ne želimo sav kod da izvezemo iz modula:

1. Određene funkcije imaju određene pretpostavke, na primer, nisu korektne za sve argumente, ili čak nisu ni definisane za sve argumente. Ako takve funkcije učinimo privatnim, onda ćemo biti sigurni da se koriste samo u okviru modula na način na koji smo zamislili. Ovaj razlog je zapravo dublji od pukih provera argumenata. Pažljivim odabirom funkcija koje izvozimo možemo se osigurati da će funkcije biti pozvane određenim redosledom (što može biti važno ako se bavimo ulazom i izlazom u program, npr. upisivanjem u bazu).
2. Privatne funkcije je lakše menjati jer se koriste na manje mesta. Karakteristična je promena tipa neke funkcije koja zahteva da se izvrše izmene na svim mestima gde se ta funkcija poziva.
3. Smanjenjem broja funkcija koje su javne, olakšavamo korisnicima (kolegama programerima) naših modula razumevanje istih. Umesto da proučavaju desetine funkcija, korisnicima će biti izloženo nekoliko funkcija od suštinskog značaja.

10.1.1. Izvoz tipova i konstruktora

Kao i funkcije, i tipovi i konstruktori se mogu izvoziti iz modula. Konstruktori se izvoze tako što se navedu u zagradama odmah nakon tipa (u okviru iste "koordinate").

Primer 3. Ako iz modula `Vektor` želimo da izvezemo i tip `Vektor` i konstruktor tog tipa, možemo sledeće da napišemo

```
module Vektor (Vektor(MkVektor), dužina) where

newtype Vektor = MkVektor (Float, Float)

dužina :: Vektor -> Float
dužina (MkVektor (x, y)) = sqrt (x^2 + y^2)
```

Izvozom konstruktora, dopuštamo da se u drugim modulima konstruišu vrednosti odgovarajućeg tipa kao i da vrši dekonstrukcija podudaranjem oblika. Ako ne želimo da dozvolimo ove operacije u drugim modulima, umesto samih konstruktora možemo izvoziti funkcije koje imaju analognu ulogu ali sa dodatnom logikom (najčešće vezanu za validaciju podataka). Ova tehnika se naziva *pametni konstruktor* (eng. *smart constructor*).

Primer 4. U nekom programu, u modulu `Ljudi` podatak o osobi definisan je kao tip

```
newtype Osoba = MkOsoba (String, Int),
```

gde prva koordinata predstavlja ime osobe a druga koordinata predstavlja broj godina. Ako bi konstruktor `MkOsoba` bio javan, tada bi se bilo gde u ostatku programa mogla kreirati nova vrednost tipa `Osoba`. Međutim, greške se lako potkradu, i mogu se kreirati besmislene vrednosti poput `Osoba ("Petar", -100)`. Stoga, umesto konstruktora `MkOsoba` svuda u programu ćemo koristiti funkciju koja ima isti tip kao i konstruktor, ali i dodatnu logiku:

```
napraviOsobu :: String -> Int -> Osoba
napraviOsobu ime godine = MkOsoba (ime, godine')
        where
            godine' = min 120 (max godine 0)
```

Na ovaj način, ograničili smo broj godina na interval od 0 do 120. U lekciji algebarski tipovi podataka videćemo kako možemo da elegantno vratimo grešku, umesto da "tiho" zamenimo vrednost.

Ovu funkciju ćemo postaviti za javnu, i na taj način ćemo se osigurati da u drugim modulima ne možemo kreirati besmislene vrednosti. Takođe, ako želimo da dodamo novu validaciju (na primer za ime), biće dovoljno da dodamo tu validaciju samo na jedno mesto - u definiciji funkcije `napraviOsobu`.

10.2. Uvoz definicija

Dekleracije uvoza se navode odmah nakon deklaracija modula, i imaju oblik

```
import X (...),
```

gde su u zagradama nalazi uređena lista javnih definicija iz `X` koje želimo da uvezemo. Definicije koje smo uvezli možemo koristi pod istim imenom koji im dodeljen u izvornom modulu.

Ako se uvozi više različitih modula, tada se svaki uvoz navodi u posebnom redu, pri čemu poredak uvoza nije bitan. Greška pri kompilaciji (interpretaciji) će se dogoditi ako se pokuša uvoz istih imena iz različitih modula.

Primer 5. Da bismo uvezli funkciju `obim` iz modula `GeometrijaKruga` u modul `GeometrijaValjka` potrebno odmah nakon dekleracije modula a pre svih drugih definicija, napišemo liniju

```
import GeometrijaKruga (obim).
```

Na ovaj način funkcija `obim` postaje dostupna u modulu `GeometrijaValjka`:

```
module GeometrijaValjka (površina, zapremina) where

import GeometrijaKruga (obim)

površinaOsnovice :: Float -> Float
površinaOsnovice r = r^2 * pi

površinaOmotača :: Float -> Float -> Float
površinaOmotača r h = (obim r) * h

površina :: Float -> Float -> Float
površina r h = 2 * (površinaOsnovice r) + (površinaOmotača r h)
```

```
zapremina :: Float -> Float -> Float
zapremina r h = h * (površinaOsnovice r)
```

Primetimo da je funkcija `obim` dostupna pod baš tim imenom, i da smo je iskoristili u funkciji `površinaOmotača`.

Ako se datoteke `GeometrijaValjka.hs` i `GeometrijaKruga.hs` nalaze u istom direktorijumu tada možemo da učitamo `GeometrijaValjka.hs` u *GHCi* okruženje sa `:load` komandom. Oba modula će se učitati:

```
ghci> :load GeometrijaValjka.hs
[1 of 2] Compiling GeometrijaKruga  ( GeometrijaKruga.hs, interpreted )
[2 of 2] Compiling GeometrijaValjka ( GeometrijaValjka.hs, interpreted )
Ok, two modules loaded.
```

Učitavanjem modula `GeometrijaValjka.hs` sve definicije (uključujući i privatne), postaju dostupne u okruženju. Iz modula `GeometrijaKruga.hs` biće dostupne javne definicije.

Kao i kod izvoza, sa praznom n-torkom, `()` nećemo uvesti ni jednu definiciju[100], dok izostavljanjem bilo kakve n-torke uvozimo sve javne definicije iz modula. Takođe, ako želimo da uvezemo sve definicije *osim* par određenih, to možemo da uradimo sa dodajući listu "neželjenih" definicija nakon `hiding` ključne reči. Na primer, da uvezemo sve definicije iz modula `M` osim `f` i `g`, napisaćemo

```
import M hiding (f, g).
```

10.2.1. Kvalifikovani uvoz

Dekleracije uvoza koje smo do sada videli uvoze definicije (funkcije, tipove, konstruktore...) pod istim imenom kako su originalno definisane. Međutim ako se u modul `M` uveze ceo modul `N` koji sadrži i ime `x`, a u samom modulu je ime `x` takođe definisano, tada će doći do greške prilikom kompilacije. Da bi se izbegao ovakav problem, koristi se **kvalifikovani uvoz**.

Pri kvalifikovanom uvozu, sva imena koja se uvoze, dobijaju prefiks. Podrazumevani prefiks je ime modula koji se uvozi. Deklaracija kvalifikovanog uvoza je

```
import qualified N.
```

Da bi se umesto imena modula koristio prefiks X potrebno je koristiti dekleraciju

```
import qualified N as X,
```

pri čemu ime `X` ne sme sadržati tačke.

Primer 6. Modul `GeometrijaKruga` možemo uvesti pod imenom `GK`:

```
module GeometrijaValjka (površina, zapremina) where

import qualified GeometrijaKruga as GK

površinaOsnovice :: Float -> Float
površinaOsnovice = GK.površina
```

[100] Mada ovakav uvoz nije u potpunosti beskoristan, i koristi se zbog nekih efekata...

```
površinaOmotača :: Float -> Float -> Float
površinaOmotača r h = (GK.obim r) * h
```

10.3. Moduli u interaktivnom okruženju

Do sada smo učitavali datoteke u *GHCi* sa `:load` komandom. Kao što smo mogli da vidimo do sada, ako u Haskel datoteci nije deklarisan modul, tada će taj modul biti implicitno učitan pod imenom *Main*. Komanda `:load` takođe dozvoljava učitavanja više *.hs* datoteka od jednom, dovoljno je navesti listu datoteka razdvojenih razmakom:

```
ghci> :load A.hs B.hs
```

Komanda `:load` može da učita datoteke na osnovu putanje. Tako na primer, ako pokrenemo *GHCi* u `home` direktorijumu, a modul `A.hs` se nalazi u `home/haskel-project` direktorijumu, tada možemo da upišemo komandu `:load haskel-project/A.hs` da bismo učitali modul. Međutim, Alternativno, i mnogo bolje[101], je da se *GHCi* pozicioniramo u korenski direktorijum Haskel projekta. To možemo uraditi sa `:cd` komandom, na primer: `:cd haskel-project`.

Ako upišemo komandu `:load` bez argumenata, tada ćemo "odbaciti" sve učitane module do tada.

Sa komandom `:browse` možemo prikazati listu svih definicija u poslednje učitanom modulu. Specijalno, ako nakon komande `:browse` navedemo ime modula, možemo pregledati sve definicije iz tog određenog modula.

10.4. Standardna biblioteka

Haskel kompajler dolazi sa mnogobrojnim modulima. I sami smo do sada koristili predefinisane funkcije za koje smo rekli da dolaze iz Prelida (`Prelude`), koji je zapravo samo jedan od predefinisanih modula. Modul `Prelude` se razlikuje po tome što se implicitno uvozi u svaki drugi modul. Taj implicitni uvoz može se sprečiti sa deklaracijom `import Prelude ()`.

U Prelid modulu se ne nalaze same definicije, već se imena uvoze iz drugih modula i ista takva takva izvoze (takozvani *reeksport*). Definicije koje Prelid reeksportuje su one koje se najčešće koriste. Mnoge druge funkcije nisu dostupne kroz Prelid, već kroz posebne module. Nekih od tih modula su pojašnjeni u nastavku, a lista svih modula sa dokumentacijom se može pregledati na Haskell Hierarchical Libraries stranici.

1. `Data.Char` sadrži mnoge funkcije za proveru karaktera (`isDigit`, `isSpace`, `isUpper`...) kao i funkcije za transformacije karaktera (`toUpper`, `toLower`)
2. `Data.Complex` sadrži definiciju apstraktnog tipa `Complex` kao i funkcije za rad sa kompleksnim brojevima. Tip `Complex` je definisan kao apstraktan, jer time je omogućeno da se naprave posebni tipovi za različite konkretne numeričke tipove, ako na primer `Complex Int` i `Complex Float`. Konstruktor za kompleksne brojeve je `:+` koji se navodi između argumenata, pa tako `2 :+ 3` predstavlja kompleksan broj $2 + 3i$.
3. `Data.List` sadrži mnogobrojne funkcije za rad sa listama: `isPrefixOf`, `isSuffixOf`, `intersect`, `intercalate`, `lines`, `tail`, `take`, `takeWhile`, `words`, `zip`, `zip3`...
4. `Data.Set` sadrži definiciju apstraktnog tipa `Set`. Vrednost tipa `Set T` za neki konkretan tip `T`, predstavlja *skup* elementa tipa `T`. Za razliku od lista, vrednosti u skupovima ne mogu da se ponavljaju, i njihov poredak nije bitan. Naravno, u modulu `Data.Set` su dostupne mnoge funkcije za rad sa skupovima, poput `member`, `union`, `cartesianProduct`, `size`...

[101] Kao što je gore navedeno, u imenima modula koji se uvoze mogu se navesti direktorijumi razdvojeni tačkama. Da bi uvoz funkcionisao ime modula mora u potpunosti da prati relativnu putanju do modula, u odnosu na direktorijum u kom smo trenutno pozicionirani sa `:cd` komandom.

5. `Data.Map` sadrži definiciju apstraktnog tipa `Map :: * -> * -> *`. Za neke tipove `A` i `B`, vrednost tipa `Map A B` predstavlja strukturu koja sadrži vrednosti tipa `B` indeksirane vrednostima tipa `A` (ovakva struktura se naziva i *rečnik*).

Da bi koristili bilo koji od navedenih modula, uglavnom je dovoljno da navedemo dekleraciju uvoza[102].

Zadatak 1. Pregledati definicije `Prelude` modula koristeći `:browse` komandu.

Zadatak 2. U matematici, *Mebijusova transformacija* je kompleksna funkcija oblika

$$m(z) = \frac{\alpha z + \beta}{\gamma z + \delta},$$

gde su α, β, γ, δ kompleksni brojevi za koje važi $\alpha\delta - \beta\gamma \neq 0$. Definisati modul `Mebijus`, i u modulu definisati funkciju `novaT` koja od uređene četvorke tipa

```
(Complex Float, Complex Float, Complex Float, Complex Float)
```

konstruiše funkciju tipa `(Complex Float -> Complex Float)`. Na primer, `novaT (1, 0, 2, 0 :+ 1)` treba da vrati funkciju koja predstavlja transformaciju $z \mapsto z/(2z + i)$. U funkciji `novaT` izvršiti neophodne provere, i u slučaju greške vratiti identičku transformaciju

$$id(z) = z = (z + 0)/(0z + 1).$$

Demonstrirati da transformacija $t(z) = \frac{z-i}{z+i}$ preslikava realnu liniju u jedinični krug (npr. koristeći `map` funkciju i opseg `[-10 .. 10]`, i funkciju `abs` koja računa *modul* kompleksnog broja).

[102] U slučaju `Set` i `Map` modula, na nekim Linux sistemima je neophodno uvesti paket `containers` tako što ćemo u *GHCi* upisati

```
:set -package containers.
```

11. Haskel programi

U dosadašnjim lekcijama prikazali smo isključivo rad u interaktivnom okruženju *GHCi*. Sada ćemo se upoznati sa Haskel kompajlerom i naučićemo kako da kompajliramo Haskel kôd u izvršne datoteke. Takođe ćemo se upoznati sa metodama koje nam omogućavaju ulaz i izlaz iz programa (bilo kroz terminal, bilo kroz datoteke), kao i nekim osnovama organizacije Haskel koda.

11.1. *Hello World!*

Većina knjiga o programskim jezicima započinje sa *Hello World!* programom: jednostavnim programom čija je jedina svrha da ispiše pozdrav u terminal. Sa mnogim Haskel knjigama, pa tako i ovom koju čitate, to nije slučaj. Umesto toga, na početku se prikazuje rad kroz interaktivno okruženje *ghci* da bi se osnovne osobine jezika dobro upoznale. Tek onda se demonstrira pisanje programa koji se samostalno izvršavaju i koji mogu prilikom izvršavanja da interaguju sa korisnikom, datotekama, itd... Razlog ovakvog pristupa je jednostavan: pisanje samostalnih Haskel programa zahteva razumevanje nešto naprednijih koncepata Haskel jezika. Ali čitaoca ne treba da brine ova konstatacija jer ćemo mi te koncepte ovde postupno uvoditi. Znanje iz prethodnih lekcija značajno će olakšati čitanje ove lekcije.

Definišimo datoteku `Hello.hs`, i u nju smestimo naredni kôd:

```
main :: IO ()
main = putStrLn "Hello World!"
```

I kao što je u većini drugih programskih jezika uvek potrebno definisati *main* funkciju koja predstavlja početnu tačku izvršavanja programa, tako i svaki Haskel program koji se kompajlira mora imati definisanu vrednost `main` tipa `IO ()`. Vrednost `main` mora biti definisana u `Main` modulu. Kako *GHCI* implicitno dodaje `Main` modul kad drugi nije definisan, mi nećemo definisati modul u narednim primerima.

Kôd možemo iskompajlirati u izvršni program pomoću *GHC* kompajlera. U sistemskom promptu[103] potrebno je pokrenuti kompajler `ghc` i kao jedini argument proslediti mu ime datoteke koju je potrebno iskompajlirati:

```
$ ghc Hello.hs
[1 of 1] Compiling Main             ( Hello.hs, hello.o )
Linking Hello ...
```

Primetimo da je ime modula koji se kompajlira `Main`

Ako je kompilacija prošla uspešno, u direktorijumu u kom je `Hello.hs` naći će se izvršna datoteka `Hello` (ili `Hello.exe` ako govorimo o *Windows* sistemima). Pokretanjem datoteke dobijamo dugo očekivani pozdrav:

```
$ ./Hello
Hello World!
```

Na *Windows* sistemima, umesto komande `./Hello`, potrebno je uneti `.\Hello.exe`. U nastavku lekcije ćemo ignorisati ovu razliku.

Navedeni kôd nije dugačak, ali jeste drugačiji od koda kojeg smo do sada pisali. Funkcija `putStrLn` primenjena na nisku `"Hello World!"` ne izgleda neobično, ali tip `IO ()` svakako izgleda.

Tip `()` nam je poznat iz jedne od prethodnih lekcija. U pitanju je tip koji sadrži samo jednu vrednost, vrednost koja se označava sa `()`. Tip `IO` je zapravo apstraktni tip vrste `* -> *`. Definiciju tipa `IO` ne možemo navesti, ali možemo objasniti šta ovaj tip predstavlja.

[103] Dakle, ovde ne govorimo o pokretanju *GHCi* programa. Komande koje navodimo se sve pokreću u sistemskom promptu kog u nastavku označavamo sa `$`.

Za neki tip `T`, tip `IO T` predstavlja proceduru (u najširem smislu te reči), koja se može izvršiti, i čiji rezultat je vrednost tipa `T`. Vrednosti koje poseduju tip oblika `IO T` nazivamo **akcije**. Prema tome, `main` je jedna akcija koja će se izvršiti tokom izvršavanja programa i čiji rezultat je vrednost tipa `()`, odnosno baš `()`.

Zapravo `main` je akcija koja ima poseban status: to je akcija koja se uvek prva izvršava pri radu programa, i poziva ostale akcije. U navedenom programu, akcija `main` je pozvala akciju `putStrLn "Hello World!"`. Funkcija `putStrLn` poseduje[104] tip `String -> IO ()`, pa vrednost primene `putStrLn` na neku nisku je akcija tipa `IO ()` koja se može dodeliti imenu `main`. Dakle, `putStrLn` je funkcija koja nisku preslikava u akciju koja će ispisati tu nisku, ali vrednost `putStrLn` sama za sebe nije akcija.

Za sada nismo objasnili zašto tako naizgled jednostavna funkcija poput `main` ima tako neobičan tip, niti smo objasnili kako se funkcija poput `putStrLn` može iskoristiti unutar funkcija koje smo do sada pisali. Tip `IO` je za sada otvara više pitanja nego što daje odgovora, ali ćemo pokušati da kroz ovu lekciju odgovorimo na neka od tih pitanja.

11.2. Interakcija sa konzolom

Programi koji ispisuju jednu nisku nisu mnogo korisni, te ćemo prvo pokazati kako se akcije mogu nadovezivati. U tu svrhu koristi se *do* notacija koja dozvoljava da se akcije navedu u zasebnim redovima jednog bloka. Navedene akcije biće izvršene sekvencijalno, redom kojim su navedene.

```
main :: IO ()
main = do
    putStrLn "Hello World!"
    putStrLn "Don't worry,"
    putStrLn "Be happy."
    putStrLn "Bye!"
```

Obratite pažnju na ključnu reč `do`. Ova reč otvara blok linija, a svaka akcija mora se navesti u posebnoj liniji.

Kompilacijom i izvršavanjem programa dobijamo očekivani rezultat

```
$ ghc Main.hs
[1 of 1] Compiling Main             ( Main.hs, main.o )
Linking main ...
$ ./Main
Hello World!
Don't worry,
Be happy.
Bye!
```

Na linijama unutar *do* bloka moraju se navesti `IO` akcije, ali mi te akcije možemo i definisati i van odgovarajućeg bloka. Na primer, možemo kreirati funkciju `pozdrav` koja uzima ime (nisku) i vraća akciju koja ispisuje pozdrav sa prosleđenim imenom.

[104] U šta se možete uveriti pomoću `:type` naredbe u *GHCi* programu.

```
pozdrav :: String -> IO ()
pozdrav x = do
    putStrLn ("Zdravo " ++ x ++ "!")

main :: IO ()
main = do
    pozdrav "Nikola"
    pozdrav "Nikolina"
```

Kada se u `do` bloku nalazi samo jedna akcija (kao u slučaju `pozdrav` funkcije), tada nije potrebno otvarati `do` blok. Pogledati kako je u gore definisana `main` akcija sa jednom akcijom.

```
$ ./Main
Zdravo Nikola!
Zdravo Nikolina!
```

Osim `putStrLn` funkcije, u Prelidu su dostupne i funkcije `putChar :: Char -> IO ()` i `putStr :: String -> IO ()`. Funkcija `putStrLn` se razlikuje od `putStr` samo po tome što dobijana akcija uz prosleđenu nisku ispisuje i karakter za novi red. Funkcija `putChar` daje akciju koja će ispisati jedan karakter.

Uz ispisivanje sadržaja u terminal, neophodno je i prihvatati korisnički unos. Za "prihvatanje" linije koristi se akcija `getLine :: IO String` dostupna u Prelidu. Tip `IO String` označava da se radi o akciji čiji rezultat je niska - to će biti niska koju je korisnik uneo. Da bi se toj vrednosti pristupilo, potrebno ju je **dodeliti** novom imenu uz pomoć strelice `<-` unutar `do` bloka:

```
main :: IO ()
main = do
    putStr "Vaše ime je: "
    ime <- getLine
    putStrLn ("Zdravo " ++ ime ++ "!")
```

Strelicom `<-` se vrednost tipa `T` "otpakuje" iz vrednosti tipa `IO T`.

```
$ ./Main
Vaše ime je: Marija
Zdravo Marija!
```

Vrednost koju dodeljujemo sa `<-` je "obična" vrednost. Dodeljenu vrednost možemo dalje prosleđivati funkcijama. Akcije `putStrLn` i `getLine` koristimo samo kada želimo da ispišemo ili učitamo podatke.

```
inicijali :: String -> String -> String
inicijali ime prezime = [head ime] ++ "." ++ [head prezime] ++ "."

main :: IO ()
main = do
    putStr "Vaše ime: "
    ime <- getLine
    putStr "Vaše prezime: "
    prezime <- getLine
    putStrLn (inicijali ime prezime)
```

Unutar `do` bloka je dozvoljeno definisati vrednosti koristeći `let` ključnu reč. U `do` blokovima definicije moraju imati smisleni redosled: definicije koje sadrže definisane (ili dodeljene) vrednosti moraju se nalaziti nakon tih definicija. Prethodni primer smo mogli i da napišemo ovako:

```
inicijali :: String -> String -> String
inicijali ime prezime = [head ime] ++ "." ++ [head prezime] ++ "."

main :: IO ()
main = do
    putStr "Vaše ime: "
    ime <- getLine
    putStr "Vaše prezime: "
    prezime <- getLine
    let inic = inicijali ime prezime
    putStrLn inic
```

U poslednjoj liniji `do` bloka mora se nalaziti akcija. Tip celog `do` bloka je tip poslednje navedene akcije.

> **Primer 1.** Definišimo akciju bez deklaracije tipa:
>
> ```
> akcija1 = do
>     putStr "Unesite komandu:"
>     getLine
> ```
>
> Kompajler će zaključiti da je `akcija1` tipa `IO String`, jer je poslednja akcija tipa `IO String`. Da smo u kodu pogrešno deklarisali `akcija1 :: IO ()` tada bi došlo do tipske greške.

Pri radu sa akcijama možemo koristiti funkciju `return :: a -> IO a` koja nam omogućuje da proizvoljnu vrednost "obmotamo" u `IO` tip. To nam sada daje mogućnost da pravimo akcije koje nisu samo tipa `IO ()` ili `IO String`.

> **Primer 2.** Trivijalnu akciju čiji rezultat je uvek vrednost `42` možemo da definišemo sa
>
> ```
> odgovor :: IO Int
> odgovor = return 42
> ```
>
> Primetimo da ova akcija ne vrši nikakav ulaz ili izlaz i programa.

Kroz primere u ovoj sekciji, prošli smo kroz sve pojedinosti `do` bloka: unutar bloka možemo pokretati više akcija, unutar bloka možemo dodeljivati vrednosti iz pokrenutih akcija sa `<-`, unutar bloka možemo definisati vrednosti sa `let`, i tip bloka je tip poslednje navedene akcije u bloku.

> **Zadatak 1.** U jednoj od prošlih lekcija smo konstruisali funkciju `uBroj :: String -> Int` koja prevodi nisku u broj. Koristeći ovu funkciju definisati `main` akciju koja uzima dva broja, a zatim ispisuje njihov zbir. Pretpostaviti da će obe unete niske predstavljati validan zapis broja.
>
> *Rešenje.*
>
> ```
> main :: IO ()
> main = do
>     a <- getLine
>     b <- getLine
>     let zbir = (uBroj a) + (uBroj b)
>     putStrLn (show zbir)
> ```

Zadatak 2. Napisati *Hello World* program u jednoj liniji. Iskoristiti činjenicu da deklaracije tipova nisu neophodne u Haskel kodu.

Rešenje.

```
main = putStrLn "Hello World"
```

Zadatak 3. U jednoj liniji napisati program koji ispisuje *Hello World* program.

Rešenje. Par navodnika označava početak i kraj niske. Međutim problem se javlja ako želimo da napišemo nisku koja sadrži navodnik u sebi. Zbog toga koristimo kosu crtu koja označava da naredni karakter ima specijalno značenje. Konkretno, da bi naveli dupli navodnik unutar niske koristimo `\"`.

```
main = putStrLn "main = putStrLn \"Hello World\""
```

Zadatak 4. Analizirati šta funkcija `show` radi kada se primeni na nisku. Posebno, analizirati razliku između funkcija `putStrLn . show` i `putStrLn`.

U računarstvu, **kvajn**[105] je program koji ispisuje sopstveni kôd. Napisati kvajn u bilo kom jeziku nije lak zadatak. Sledeći zadatak je možda najteži u ovoj knjizi, ali prethodna tri zadatka treba da pomognu pri rešavanju.

Zadatak 5. Napisati program koji ispisuje sopstveni kôd.

Rešenje. Dobra početna tačka je program koji ispisuje neku nisku. Odabraćemo nisku koja je početak koda samog programa:

```
main = putStrLn "main = putStrLn"
```

Kompilacijom i pokretanjem programa dobijamo početak koda.

```
$ ghc Quine.hs
[1 of 1] Compiling Main             ( Quine.hs, quine.o )
Linking Quine ...
$ ./Quine
main = putStrLn
```

Deluje da je dovoljno da ostatak koda smestimo u nisku:

```
main = putStrLn "main = putStrLn \"main = putStrLn\""
```

```
$ ./Quine
main = putStrLn "main = putStrLn"
```

Pokretanjem rešenja uviđamo da je strategija bila naivna. Novim programom jesmo ispisali kôd prethodnog programa, ali nismo uspeli da konstruišemo *quine* - program koji ispisuje sopstveni kôd.

[105] Na engleskom *quine*. Termin je dobio ime po logičaru Willard Van Orman Quine-u.

Kako mi dodajemo tekst u nisku to program postaje duži te moramo da dodajemo još više teksta, i tako dalje... Takav proces se nikad neće zaustaviti. Zbog toga moramo napisati program koji će tokom izvršavanja konstruisati nisku koja će se ispitati. Vratimo se na prvi pokušaj rešenja, tj. program

```
main = putStrLn "main = putStrLn"
```

i dodajmo funkciju `(\s -> s ++ s)` koja "duplira" nisku. Kôd te funkcije ćemo takođe dodati u nisku:

```
main = putStrLn ((\s->s++s) "main = putStrLn (\\s->s++s)")
```

Da bi naveli kosu crtu unutar niske koristimo `\\`

```
$ ./Quine
main = putStrLn (\s->s++s)main = putStrLn (\s->s++s)
```

Poredeći kôd i ispis programa vidimo da nedostaje par zagrada, par navodnika, razmak, i kosa crta u lambda funkciji. Otvorenu zagradu ćemo dodati u sam literal niske, a razmak i zatvorenu zagradu ćemo dodati u telo lambda funkcije:

```
main = putStrLn ((\s->s++" "++s++")") "main = putStrLn ((\\s->s++\" \"++s++\")\")")
```

```
$ ./Quine
main = putStrLn ((\s->s++" "++s++")") main = putStrLn ((\s->s++" "++s++")"))
```

Rešenje sad deluje dosta blizu. Neophodno je sad postaviti drugi deo ispisa u navodnike i dodati kose crte. Tj. potrebno je nisku ispisati u obliku u kom bi bila zapisana kao Haskel literal. Upravo to radi `show` funkcija. Sada je dovoljno umesto

```
\s->s++" "++s++")"
```

napisati

```
\s->s++" "++show s++")".
```

```
main = putStrLn ((\s->s++" "++show s++")") "main = putStrLn ((\\s->s++\" \"++show
s++\")\")")
```

```
$ ./Quine
main = putStrLn ((\s->s++" "++show s++")") "main = putStrLn ((\\s->s++\" \"++show
s++\")\")")
```

11.3. Akcije i funkcije

Haskel početnici često poistovećuju *akcije* (poput `putStrLn "Hello World" :: IO ()` i `getLine :: IO String`) sa *funkcijama* jer pojam funkcije u drugim programskim jezicima obuhvata oba pojma u Haskelu. Na primer, u C jeziku postoji funkcija (funkcija u smislu C jezika) `getline` koja je u potpunosti analogna Haskel akciji `getLine` (čak imaju i skoro identično ime). Ali kao što ćemo uskoro videti, Haskel akcije ne mogu biti Haskel funkcije! Prema tome, pojam *funkcije* u drugim jezicima je širi pojam od pojma funkcije u Haskelu.

U Haskelu funkcije predstavljaju *pravilnosti* po kojima se vrednostima nekog tipa pridružuju vrednosti drugog tipa. Ključna reč u navedenoj definiciji je "pravilnosti", jer ona označava osobinu funkcija da imaju istu povratnu vrednost kada se primene na iste argumente. Tačnije, ako je $x = y$, tada je i $f(x) = f(y)$. U tom smislu, vrednost `getLine` ne može biti funkcija: njena vrednost prilikom svakog "pozivanja" može

biti drugačija (zavisi od toga šta je korisnik konzole uneo). Sličan argument važi za sve ostale procedure čija povratna vrednost zavisi od vrednosti koje se "nalaze" van samog programa. Na primer, to mogu biti procedure koje vraćaju sadržaj neke datoteke, korisnički unos, vreme, sadržaj nekog resursa na internetu, podatke iz baze, i tako dalje. Gledano iz ugla Haskel programa, sve ovakve procedure vraćaju vrednosti iz *spoljnog sveta*. Pod terminom **spoljni svet** smatraju se sve vrednosti koje nisu definisane unutar (Haskel) programa.

Osim što su akcije neophodne za "preuzimanje" informacija iz spoljnog sveta, akcije moramo koristiti i kada "šaljemo" informacije u spoljni svet. Na primer, ako štampamo tekst u terminal, čuvamo podatke na disk, šaljemo zahteve veb serverima i tako dalje. Navedene procedure poseduju **propratne efekte**[106] što znači da menjaju stanje spoljnog sveta. Haskel funkcije nemaju propratne efekte, dok Haskel akcije mogu imati propratne efekte.

Takođe, primetimo još jednu razliku između funkcija i akcija. Funkcije uvek moramo primeniti na vrednost (što takođe nazivamo i *pozivanje sa vrednošću*), tj. moramo napisati kôd poput `f x`. Sa druge strane, akcije ne primenjujemo na vrednosti, zbog čega za akcije kažemo da se *pokreću* unutar `do` bloka.

Apstraktni tip `IO` predstavlja "most" koji spaja spoljni svet sa programom. Sva komunikacija programa sa spoljnim svetom mora se odviti *pokretanjem* akcija sa `IO` tipom, a svaku `IO` akciju je moguće pokrenuti samo iz neke druge `IO` akcije. Zaista, nemoguće je pokrenuti[107] akcije unutar funkcija. Dakle, akcije su uvek pokrenute od strane drugih akcija, i tako dalje, sve do `main` akcije.

Primer 3. Naravno, definisanjem akcije ne znači da će ta akcija biti i pokrenuta. Pokretanjem narednog programa videćemo samo dva pozdrava u konzoli. Obratite pažnju da će akcija `a3` biti pokrenuta iz `a1` akcije a ne iz `main` akcije.

```
main :: IO ()
main = do
    a1

a1 :: IO ()
a1 = do
    putStrLn "Hello"
    a3

a2 :: IO ()
a2 = do
    putStrLn "Ciao"

a3 :: IO ()
a3 = do
    putStrLn "Zdravo"
```

Razliku između definisanja i pokretanja akcije je moguće uvideti i u narednom kodu. Znak jednakosti u prvoj liniji `do` bloka određuje definiciju akcije `a` koja nije pokrenuta! Sa druge strane, strelica `<-` u narednoj liniji pokreće akciju sa desne strane, i rezultat pokretanja te akcije dodeljuje imenu `r`. Kako je `putStr :: String -> IO ()`, to će vrednost `r` imati tip (a i vrednost) `()`.

[106] *side effects*

[107] Pokušajte, ali nećete uspeti. Akcije možete pokrenuti unutar `do` bloka, a `do` blok sam za sebe predstavlja jednu `IO` akciju...

```haskell
main :: IO ()
main = do
    let a = putStr "Hello"
    r <- putStr "Zdravo"
    putStrLn "!"
```

Kako je prva akcija samo definisana, a druga pokrenuta, ispis u konzoli će biti *Zdravo!*

Za korisnike drugih programskih jezika, ovakvo razdvajanje pojma akcija od pojma funkcija deluje kao nepotrebno komplikovanje. Ali zapravo se ovde radi o prednosti Haskel jezika u odnosu na druge jezike. Dok u mnogim drugim jezicima (C/C++, Python, Java, JavaScript, Go, itd..) ne postoji nikakva kontrola interakcije programa sa spoljnim svetom[108], u Haskelu je kroz sistem tipova (tj. `IO` tip) obezbeđeno jasno razdvajanje dela koda koji interaguje sa spoljnim svetom od dela koda koji to ne čini. Na taj način postignuto je da se mnoge potencijalne greške (bagovi), uoče i isprave tokom kompilacije programa.

U navedenim programskim jezicima postoji slična podela koda. Programeri često govore o **čistom**[109] i **nečistom**[110] kodu. Čiste funkcije su one funkcije koje zadovoljavaju dva pravila: za iste parametre vraćaju iste povratne vrednosti i nemaju propratne efekte. To su upravo one dve osobine koje smo naveli za Haskel funkcije. Prema tome možemo reći da su Haskel funkcije uvek čiste dok su akcije nečiste. Primetimo da u navedenim programskim jezicima razlika između čistog i nečistog kod nije uspostavljena kroz kompajler (kao što je to slučaj sa Haskelom), već programer mora sam da zaključi o kakvom kodu se radi.

11.4. Rad sa datotekama

Osim interakcije sa korisnikom kroz konzolu, programi moraju učitavati i zapisivati datoteke. Iako postoji mnogo funkcija dostupnih Haskel programerima za interakciju sa datotekama, ovde ćemo spomenuti samo dve funkcije koje se koriste za čitanje i pisanje iz tekstualnih datoteka. To su funkcije

```haskell
readFile :: FilePath -> IO String
```

i

```haskell
writeFile :: FilePath -> String -> IO ()
```

obe dostupne u Prelidu.

Tip `FilePath` koji se pojavljuje u tipu obe navedene funkcije je tipski sinonim za `[Char]` odnosno `String`. Upotrebom ovog tipa u potpisu funkcije, kroz same tipove smo dokumentovali čemu služe parametri funkcije. Sada je u potpunosti jasno šta predstavljaju parametri funkcije `writeFile`: prvi je putanja do datoteke u koju treba zapisati nisku koja je prosleđena kao drugi argument. Sa druge strane, da nismo konstruisali novi tipski sinonim, iz tipa `String -> String -> IO ()` ne bi bilo jasno koji parametar se odnosi na putanju a koji na sadržaj.

Dakle, funkcija

```haskell
writeFile :: FilePath -> String -> IO ()
```

uzima putanju do datoteke, nisku, i daje akciju koja kada se pokrene upisuje dati sadržaj u datoteku na navedenoj lokaciji. Funkcija

```haskell
readFile :: FilePath -> IO String
```

[108] Na primer, ako radite na projektu koji sadrži hiljade procedura koje pozivaju jedna drugu, tada ne možete lako znati da li će neka funkcija koju pozovete imati neki efekat koji ne želite. Šta ako funkcija koju pozivate, dalje poziva neke funkcije koje štampaju šifre u konzolu ili brišu tabele iz baze? Čak i ako analizirate sav kôd koji pozivate, ne možete biti sigurni da vaš kolega kasnije neće slučajno izmeniti kôd na neželjen način.

[109] *pure*

[110] *impure*

je nešto jednostavnija. Ta funkcija uzima putanju do datoteke a vraća akciju koja kada se pokrene učitava sadržaj datoteke u nisku.

Da bismo demonstrirali primenu navedenih funkcija, kreirajmo datoteku `pesma.txt` i sačuvajmo je u istom direktorijumu u kom se nalazi i kôd programa. Program koji učitava sadržaj pesme i ispisuje u terminal je sledeći:

```
main :: IO ()
main = do
    pesma <- readFile "pesma.txt"
    putStrLn pesma
```

Kao što pokretanje akcije `getLine` daje korisnički unos, tako i pokretanje akcije `readFile "pesma.txt"` daje sadržaj datoteke. U oba slučaja moramo iskoristiti strelicu `<-` da bi pokrenuli akciju i rezultat smestili u novo ime.

```
$ ghc Main.hs
[1 of 1] Compiling Main             ( Main.hs, main.o )
Linking Main ...
$ ./Main
Ono sve što znaš o meni
To je stvarno tako malo
U dv'je riječi sve bi stalo
Kada pričala bi ti
```

Da bi zapisali nisku u datoteku iskoristićemo funkciju `writeFile`. Na primer, pesmu koju smo učitali sa prethodnim programom, možemo zapisati u novu datoteku na sledeći način:

```
main :: IO ()
main = do
    pesma <- readFile "pesma.txt"
    writeFile "novaPesma.txt" pesma
```

Pokretanjem navedenog programa, sadržaj datoteke `pesma.txt` će biti prekopiran u datoteku `novaPesma.txt`, a u terminalu neće biti prikazana bilo kakva poruka. Kao što vidimo, jedina svrha ovog programa je da kopira tekstualnu datoteku sa određenim imenom.

Pri radu sa funkcijama `readFile` i `writeFile` trebalo bi imati na umu da ove funkcije mogu dovesti do izuzetka koji prekidaju rad programa ako se ne obrade. Na primer, ako pokušamo čitanje datoteke koja ne postoji, ili upisavanje u datoteku za koju nemamo odgovarajuće dozvole. Videćemo uskoro kako se izuzeci mogu kontrolisati.

> **Zadatak 6.** Koristeći funkciju `lines :: String -> [String]` naći broj linija u nekoj određenoj datoteci (npr `pesma.txt`).

11.5. Rad sa argumentima komande linije

Komandna linija, ili terminal, predstavlja interfejs u kom korisnik upisuje komande koje računar treba da izvrši. Specijalno, kroz komandnu liniju korisnik može da pokrene programe, što smo i videli u ovoj lekciji. U terminalima lokalni programi (poput programa koje smo sami iskompajlirali) pokreću se navođenjem putanje do tog programa. Prilikom pokretanja programa moguće je proslediti i **argumente komandne linije** koji se navode nakon imena programa. Nulti argument je putanja programa. Svi argumenti prosleđuju se pokrenutom programu i na tom programu je da interpretira značenje tih argumenata.

Primer 4. U terminalima na *Unix* sistemima moguće je pokrenuti program `rm`[111] koji briše datoteke (na *Windows*-u to je `del` program, ali sve isto važi). Ako u terminalu pokrenemo komandu `rm pesma.txt`, tada će terminal pokrenuti program `rm` i proslediće mu listu[112] argumenata `["rm", "pesma.txt"]`.

Koristeći akcije `getProgName :: IO String` i `getArgs :: IO [String]` iz modula `System.Environment` možemo pristupiti argumentima komande linije. Akcija `getProgName` vraća nulti argument tj. ime programa, dok `getArgs` vraća sve ostale argumente. U najjednostavnijem slučaju možemo samo ispisati prosleđene argumente:

```
import System.Environment

main :: IO ()
main = do
    imePrograma <- getProgName
    putStrLn imePrograma
    argumenti <- getArgs
    putStrLn (show argumenti)
```

Sadržaj datoteke `argumenti.hs`

```
$ ghc Argumenti.hs
[1 of 2] Compiling Main             ( Argumenti.hs, argumenti.o )

$ ./Argumenti
Argumenti
[]

$ ./Argumenti a b C DDD 123
Argumenti
["a","b","C","DDD","123"]

$ ./Argumenti argument1 "drugi argument"
Argumenti
["argument1","drugi argument"]
```

Kompilacija navedenog programa i tri primera pokretanja. Svaka reč nakon imena programa se interpretira kao poseban argument. Poslednji primer ilustruje mogućnost prosleđivanja argumenta sačinjenog od više reči korišćenjem navodnika.

Sada kako znamo da pristupimo argumentima komande linije, možemo napraviti jednostavnije verzije nekih poznatih *Unix* alata:

Zadatak 7. Napisati program koji ispisuje sadržaj tekstualne datoteke čije ime je navedeno kao prvi argument komandne linije (ovo je takozvani *cat* program). Ako argument nije prisutan, program treba da ispiše poruku o grešci.

Rešenje. Akcije za učitavanje i ispis datoteke postavićemo u posebnu funkciju `učitajIspiši`. U akciji `main` proverićemo da li lista argumenata sadrži barem jedan element i na osnovu toga pokrenuti sledeću akciju:

[111] skraćeno od *remove*

[112] Ovde smo iskoristili Haskel sintaksu da predstavimo listu (niz) niski, ali bez obzira na jezik u kom je napisan program, princip je isti.

```haskell
import System.Environment

učitajIspiši :: String -> IO ()
učitajIspiši imeDatoteke = do
    sadržaj <- readFile imeDatoteke
    putStrLn sadržaj


main :: IO ()
main = do
    argumenti <- getArgs
    if null argumenti
        then putStrLn "Navedite barem jedan argument!"
        else učitajIspiši (head argumenti)
```

Zadatak 8. Napisati program koji ispisuje broj reči u tekstualnoj datoteci čije ime je navedeno kao prvi argument komandne linije (ovo je takozvani *wc* program). Ako argument nije prisutan, program treba da ispiše poruku o grešci.

Zadatak 9. Napisati program koji sadržaj datoteke čije ime je navedeno kao prvi argument kopira u datoteku čije ime je navedeno kao drugi argument komandne linije (ovo je takozvani *cp* program). Ako svi argumenti nisu prisutni, program treba da ispiše poruku o grešci.

Zadatak 10. Napisati program koji ispisuje prvih pet linija tekstualne datoteke čije ime je navedeno kao prvi argument komandne linije (ovo je takozvani *head* program). Ako argument nije prisutan, program treba da ispiše poruku o grešci. Ako je prisutan drugi argument, proveriti da li taj argument predstavlja prirodan broj, i u potvrdnom slučaju ispisati broj linija u skladu sa tim argumentom. Iskoristiti funkcije `uBroj` i `daLiJeBroj` iz lekcije o listama.

12. Operatori

Iako možda nismo ni svesni, funkcije koristimo još od najranijih dana matematičkog obrazovanja. Operacije sabiranja, oduzimanja, množenja i deljenja su zapravo funkcije od dve promenljive. Tako na primer, operacija sabiranja koju označavamo simbolom $+$ je funkcija koja svakom paru brojeva pridružuje njihov zbir. Za razliku od većine ostalih funkcija od dve promenljive, primena funkcije $+$ na vrednosti x i y, ne označava se sa $+(x, y)$, već sa $x + y$. Glavni razlog je naravno istorijski: simbol $+$ je ušao u upotrebu pre nego što se pojam funkcije pojavio. Ali postoji i praktičan razlog: zapis $x + y$ je jednostavniji (pa samim tim i čitljiviji) od zapisa $+(x, y)$. Slično važi i za simbole $-$, $\times$, : koji takođe predstavljaju funkcije dve promenljive.

Notaciju pri kojoj ime funkcije navodimo između argumenata funkcije, nazivamo *infiksna notacija*. Za razliku od toga, notacija u kojoj se ime funkcije navodi pre argumenta, naziva se *prefiksna notacija*[113]. Do sada, sve funkcije koje smo definisali u Haskelu smo koristili isključivo u prefiksnoj notaciji (npr. `f x y`).

Pojam funkcije je fundamentalan u Haskel jeziku. Zbog toga je Haskel jezik dizajniran tako da omogući programerima da sami definišu funkcije koje će moći da se koriste u infiksnoj notaciji. Ovakve funkcije nazivamo **operatori**.

12.1. Osnovni operatori

Do sada smo već imali prilike da koristimo neke operatore. To su bili

1. *Aritmetički operatori* `+`, `-`, `*`, `/`, `^`
2. *Logički operatori* `&&`, `||`
3. *Operatori poređenja* `==`, `/=`, `<`, `<=`, `>`, `>=`
4. *Operatori nad listama* `:`, `++`, `!!`

Koristeći `:type` naredbu u *GHCi* okruženju, možemo se uveriti da su i navedeni operatori funkcije. Na primer, tip operatora `&&` nam govori da se radi o binarnoj funkciji

```
ghci> :type (&&)
(&&) :: Bool -> Bool -> Bool
```

Da bi smo iskoristili naredbu `:type` sa operatorom (kao u gornjem primeru), neophodno je da ime tog operatora postavimo u zagrade.

Zapravo, postavljanjem imena proizvoljnog operatora u zagrade dobijamo prefiksnu funkciju:

```
ghci> (&&) True False
False
ghci> (+) 4 3
7
ghci> (-) 5 2
3
```

Iz poslednjeg primera vidimo da je "prvi" argument operatora u prefiksnoj notaciji zapravo levi argument, a "drugi" argument pri prefiksnoj notaciji je desni argument u infiksnoj notaciji.

> **Zadatak 1.** Bez korišćenja `:type` naredbe, odrediti tipove operatora nadovezivanja i spajanja lista `:` i `++`.

[113] Postoji takođe i *postfiksna notacija* u kojoj se ime funkcije navodi nakon argumenta. Postfiksna notacija se znatno ređe koristi. Notacija za faktorijel $n!$ je u postfiksnom obliku.

Rešenje. Operator `++` poseduje tip `[a] -> [a] -> [a]` jer ovaj operator spaja dve liste istog tipa. Operator `:` poseduje tip `a -> [a] -> [a]` jer ovaj operator dodaje element (levi argument) na početak liste (desni argument).

Zadatak 2. Kog je tipa operator `==`?

Rešenje. Operator poređenja poredi dve vrednosti istog tipa i vraća logičku vrednost kao rezultat tog poređenja. Da bi se dve vrednosti nekog tipa mogle porediti, neophodno je da taj tip pripada klasi `Eq`. Prema tome, operator `==` ima tip `Eq a => a -> a -> Bool`.

12.2. Prioritet i asocijativnost

Korišćenjem operatora dobijamo nove vrednosti na koje ponovo možemo delovati s operatorima. Na primer, rezultat zbira $1 + 2$ možemo pomnožiti sa 5, a izraz koji odgovara ovom postupku je $(1 + 2) \cdot 5$. U datom izrazu, kao i uvek, zagrade označavaju *podizraz* kog je potrebno prvog izračunati.

Zagrade su neophodne da bi se tačno preneo smisao izraza. Ako bismo iz prethodnog primera obrisali zagrade, dobili bismo izraz $1 + 2 \cdot 5$. U zavisnosti od toga kojim redom vršimo računske operacije, od ovog izraza možemo da dobijemo vrednost 11 ili 15. Naravno ovde grešku nećemo napraviti jer znamo za konvenciju o prioritetu računskih operacija koja govori da je množenje potrebno izvršiti pre sabiranja kada zagradama drugačije nije naznačeno. Pomenutom konvencijom je ustanovljen **prioritet** operatora. Za operator $\cdot$ kažemo da ima **veći** (ili **viši**) prioritet od operatora $+$, odnosno za $+$ kažemo da ima **manji** (ili **niži**) prioritet od operatora $\cdot$.

Dakle, uspostavljenjem prioriteta operatora, možemo se osloboditi nekih parova zagrada bez brige da ćemo izgubiti namereno značenje izraza. Na primer, iz izraza $1 + (2 \cdot 5)$ možemo obrisati zagrade, ali ne i iz izraza $(1 + 2) \cdot 5$ jer bismo time dobili sasvim drugi izraz.

Šta se dešava kada koristimo operatore istog prioriteta? Štaviše, šta se dešava kada koristimo samo jedan operator? Da li možemo iz izraza $(2 + 4) + 6$ obrisati zagrade? U navedenom primeru možemo, jer izrazi $(2 + 4) + 6$ i $2 + (4 + 6)$ daju potpuno isti rezultat.

Za neki operator $\star$ kažemo da je **asocijativan** ako važi

$$(a \star b) \star c = a \star (b \star c)$$

za sve vrednosti a, b, c na koje operator može da deluje.

Primer 1. Operator sabiranja $+$ je asocijativan operator, a takav je i operator množenja $\cdot$.

Primer 2. Nije svaki operator asocijativan. Operator deljenja $/$ nije, jer izraz $1/(2/3)$ ima drugačiju vrednost od izraza $(1/2)/3$. Prema tome, izraz $1/2/3$ je potrebno pažljivo interpretirati[114].

Korišćenjem osobine asocijativnosti i konvencije o prioritetu možemo se osloboditi zagrada u izrazima. Zbog toga su autori Haskel jezika omogućili izražavanje ove dve osobine u jeziku.

U Haskelu, svaki operator ima prioritet koji je označen sa jednim prirodnim brojem. Operatori sa najnižim prioritetom imaju prioritet 0, a oni sa najvišim imaju prioritet 9. Prioritet 10 je rezervisan za aplikaciju funkcija na vrednost.

[114] U matematici, konvencija je da se izračunavanje izraza vrši s leva na desno, ako je izraz sačinjen od operatora istog prioriteta. U matematici izraz $1/2/3$ označava $(1/2)/3$

Primer 3. Operator `++` koji nadovezuje liste je asocijativan operator jer očigledno važi

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

za sve liste `xs`, `ys`, `zs`. Prema tome, uvek kada nadovezujemo više lista, možemo pisati izraz poput

```
xs ++ ys ++ zs
```

jer je sve jedno kako se taj izraz interpretira.

Primer 4. Operator `:` koji nadovezuje element na početak liste nije asocijativan. Zaista iz izraza `(x:xs):ys` možemo zaključiti da ako je `x :: T` tada mora biti `xs :: [T]` a samim tim i `ys :: [[T]]` (jer `xs :: [T]` nadovezujemo na početak `ys`). Sa druge strane, ako u izrazu `x:(xs:ys)` važi `x :: T`, tada mora biti `(xs:ys) :: [T]`, a samim tim i `xs :: T` i `ys :: [T]`.

Dakle, `(x:xs):ys` i `x:(xs:ys)` predstavljaju potpuno različite izraze (štaviše ako je jedan dobro tipiziran, tada onaj drugi nije).

12.3. Definisanje operatora

Operatori se definišu slično ostalim funkcijama. Pri biranju imena operatora, potrebno ispoštovati tri pravila:

1. Ime operatora mora se sastojati samo od znakova `!#$%&*+./<=>?@\^|-~:`
2. Ime operatora ne sme biti: `..`, `:`, `::`, `=`, `\`, `|`, `<-`, `->`, `@`, `~`, `=>`.
3. Ime operatora ne sme počinjati sa `:`.

Operatori se mogu definisati na više načina, kako i pokazuje naredni primer.

Primer 5. Definišimo operator *isključujuće disjunkcije*, odnosno operator tipa `Bool -> Bool -> Bool` koji vraća `True` samo ako argumenti imaju različitu logičku vrednost. Za ime ovog operatora možemo uzeti na primer `|-|`.

Ako definišemo operator dodelom lambda izraza, možemo prosto pisati:

```
(|-|) :: Bool -> Bool -> Bool
(|-|) = \x y -> x /= y
```

Zagrade `()` nisu deo imena operatora, već služe za predstavljanje operatora u prefiksnom obliku.

Moguće je takođe i iskoristiti podudaranje oblika:

```
(|-|) :: Bool -> Bool -> Bool
(|-|) x y = x /= y
```

Prvi argument `x` predstavlja levi argument, a drugi argument `y` predstavlja desni argument.

Podudaranje oblika dozvoljava i narednu definiciju:

```
(|-|) :: Bool -> Bool -> Bool
x |-| y = x /= y
```

Sve tri navedene definicije su u potpunosti iste. Poslednji oblik definicije je najprirodniji pa se i najčešće koristi.

Učitavanjem bilo koje od definicija u *GHCi* možemo koristiti novi logički operator:

```
ghci> True |-| False
True
ghci> True |-| True
False
```

Zadatak 3. Definisati operator logičke implikacije

```
(==>) :: Bool -> Bool -> Bool.
```

Implikacija $p \Rightarrow q$ je netačna ako i samo ako je `p` tačno a `q` netačno.

Moguće je deklarisati prioritet i asocijativnost operatora takozvanom *infix* deklaracijom. Ova deklaracija se sastoji od jedne od ključnih reči `infixl`, `infixr` ili `infix`, i celog broja od `0` od `9` koji označava prioritet operatora. Reč `infixl` označava da se radi o levoasocijativnom operatoru, `infixr` označava da se radi o desnoasocijativnom operatoru, dok `infix` označava da operator nije ni jedno ni drugo.

Ako prioritet i asocijativnost operatora nisu deklarisani, tada će se podrazumevati vrednost `infixl 9`.

Primer 6. Operator nadovezivanja elementa na kraj niza možemo definisati sa

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

Obratite pažnju da je operator definisan tako da levi argument nadovezuje na kraj desnog argumenta! Operator je ovako definisan radi primera.

```
ghci> '!' @: "Hello"
"Hello!"
```

Za gore definisan operator `@:` se podrazumeva da je levoasocijativan sa prioritetom `9`. Zbog leve asocijativnosti, izraz `x @: y @: z` se tumači kao `(x @: y) @: z` što nam ne odgovara:

```
ghci> '?' @: '!' @: "Hello"

<interactive>:16:10: error: [GHC-83865]
    • Couldn't match expected type ‘[Char]’ with actual type ‘Char’
    • In the second argument of ‘(@:)’, namely ‘'!'’
      In the first argument of ‘(@:)’, namely ‘'?' @: '!'’
      In the expression: '?' @: '!' @: "Hello"

<interactive>:16:19: error: [GHC-83865]
    • Couldn't match type ‘Char’ with ‘[Char]’
      Expected: [[Char]]
        Actual: String
    • In the second argument of ‘(@:)’, namely ‘"Hello"’
      In the expression: '?' @: '!' @: "Hello"
      In an equation for ‘it’: it = '?' @: '!' @: "Hello"
```

[115] Sa druge strane, ovako definisan operator nam dozvoljava pisanje izraza

```
'!' @: "world" @: ["Hello"]
```

Izraz `x @: y @: z`, tj `(x @: y) @: z`, zahteva da x bude tipa `T`, y tipa `[T]` a z tipa `[[T]]`. Zbog toga, `@:` nije pogodan za nadovezivanje više elemenata na kraj liste[115] jer moramo pisati zagrade kad povezujemo više elemenata na kraj liste:

```
ghci> '?' @: ('!' @: "Hello")
"Hello!?"
```

Ipak, možemo deklarisati `@:` kao desnoasocijativan operator navođenjem linije:

```
infixr 9 @:
```

Ovu deklaraciju je moguće postaviti bilo gde u kodu.

Ako ponovo učitamo kod u *GHCi*-u, prethodni primer će bez problema funkcionisati:

```
ghci> '?' @: '!' @: "Hello"
"Hello!?"
```

Sada se izraz interpretira kao `'?' @: ('!' @: "Hello")`, što smo i želeli.

Nijedna deklaracija asocijativnosti i prioriteta nije pogrešna. Ipak, pažljivom upotrebom *infix* deklaracija možemo se osloboditi mnogih zagrada u kodu.

Ako je fiksnost operatora deklarisana sa `infix`, tada je *uvek* neophodno koristiti zagrade kada se izraz sastoji od više primena operatora.

Primer 7. Da smo fiksnost operatora iz prethodnog primera definisali sa npr. `infix 8` tada bismo dobili narednu grešku u *GHCi*-u:

```
ghci> '?' @: '!' @: "Hello"
<interactive>:1:1: error:
    Precedence parsing error
        cannot mix ‘@:’ [infix 8] and ‘@:’ [infix 8] in the same infix expression
```

Ova greška nam sugeriše da je neophodno da postavimo zagrade na odgovarajuće mesto. Primetimo da je u ovom slučaju samo jedan poredak zagrada validan (iz razloga koji su opisani ranije).

Operator poređenja vrednosti `==` je primer `infix` operatora. Primetimo da izraz[116]

```
True == False == True
```

izgleda kao validan, ali će zapravo dovesti do greške.

Zadatak 4. Definisati operator `!+ :: (Float, Float) -> (Float, Float) -> (Float, Float)` koji sabira vektore (odnosno sabira odgovarajuće koordinate uređenog para).

12.3.1. Funkcije kao operatori

U Haskelu je moguće *ad-hoc* napraviti operator od funkcije. Potrebno je samo postaviti ime te funkcije između kosih navodnika ``. Navedeno ime se tada može koristiti kao operator, pri čemu se prvi argument postavlja sa leve strane, a drugi argument sa desne strane.

---

koji se evaluira u

```
["Hello", "world!"]
```

[116] Ili bilo koji drugi izraz sačinjen od logičkih vrednosti i operatora `==`.

Primer 8. Funkcija `elem :: Eq a => a -> [a] -> Bool` proverava da li se vrednost nalazi u listi. Ova funkcija se često koristi u infiksnom obliku, pa se umesto `elem x xs` piše `x `elem` xs`:

```
ghci> 2 `elem` [1,2,3]
True
ghci> 'm' `elem` "Beograd"
False
```

Primetimo da izraz `x `elem` xs` podseća na matematičku notaciju $x \in XS$.

Podsetimo se da smo gore opisali da je moguće uraditi i obrnuto, od operatora dobiti prefiksnu funkciju, postavljanjem imena tog operatora u zagrade.

12.3.2. Sekcije

I operatore kao i 'obične funkcije', možemo parcijalno aplicirati na vrednosti. Ovakve parcijalne aplikacije nazivamo **sekcije**. Sekcija se zapisuje navođenjem operatora unutar zagrada zajedno sa još jednim od argumenata, i predstavlja *prefiksnu* funkciju.

Primer 9. Sa sekcijama možemo da apliciramo jedan operand, levi ili desni, na operator `/`:

1. Sekcija `(/ 2)` predstavlja funkciju `\x -> x / 2`.
2. Sekcija `(2 /)` predstavlja funkciju `\x -> 2 / x`.

Kao što smo naveli, sekcije su prefiksne funkcije. Tako da nedostajući argument navodimo nakon sekcije, bez obzira da li je to levi ili desni operand:

```
ghci> (/ 2) 10
5.0
ghci> (2 /) 10
0.2
```

Primer 10. Sekcije su posebno zgodne pri radu sa funkcijama višeg reda. Ako želimo da svaki element liste `xs` dupliramo, možemo da napišemo

```
map (* 2) xs
```

umesto dosadašnjeg

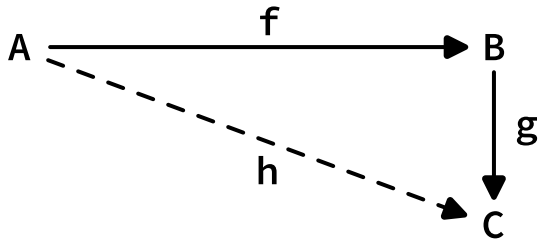
```
map (\x -> x * 2) xs.
```

12.4. Neki važni operatori

Osim aritmetičkih, logičkih i operatora nad listama, Haskel programi sadrže obilje drugih operatora (od kojih mnogi nisu ni prisutni u drugim jezicima). Haskel programeri vole da koriste razne operatore jer time skraćuju programe. Neke od tih operatora ćemo predstaviti u nastavku ove glave, dok ćemo ostale obraditi u narednim lekcijama.

12.4.1. Operator kompozicije

Kompozicija funkcija `f :: A -> B` i `g :: B -> C` jeste funkcija `h :: A -> C` za koju važi `h x = g (f x)` za svako `x :: A`. Drugim rečima, kompozicija funkcija `f` i `g` je funkcija čija vrednost za argument `x` se dobija primenom funkcije `g` na vrednost `f x`.



Kompozicija funkcija je operacija koja se često koristi u matematici. U matematičkoj notaciji, kompozicija funkcija f i g se označava se $g \circ f$. Primetimo da se levo od operatora $\circ$ navodi funkcija koja se primenjuje druga.

> **Primer 11.** Neka su date funkcije $f(x) = 2x$ i $g(x) = x^2$ koje imaju tip $\mathbb{N} \to \mathbb{N}$. Tada je funkcija $g \circ f$ data sa
>
> $$(g \circ f)(x) = g(f(x)) = g(2x) = 4x^2.$$
>
> Kako obe funkcije imaju kodomen i domen $\mathbb{N}$ možemo naći i kompoziciju $f \circ g$:
>
> $$(f \circ g)(x) = f(g(x)) = f(x^2) = 2x^2.$$
>
> Dakle vidimo da funkcije $g \circ f$ i $f \circ g$ ne moraju biti iste.

U Haskelu, operator kompozicije se označava sa `.`, i u potpunosti odgovara matematičkom operatoru kompozicije $\circ$. Operator `.` poseduje parametarski polimorfan tip

```
(b -> c) -> (a -> b) -> a -> c.
```

Funkcije se ne mogu proizvoljno komponovati. Da bismo mogli funkciju `g` da primenimo na vrednost `f x`, neophodno je da domen funkcije `g` bude jednak kodomenu funkcije `f`. Zato su argumenti operatora `.` redom funkcije tipa `b -> c` i `a -> b`.

> **Primer 12.** Prethodni primer možemo prevesti u Haskel. Funkcije `f` i `g` i kompozicije se lako definišu:
>
> ```
> f :: Int -> Int
> f x = 2 * x
>
> g :: Int -> Int
> g x = x ^ 2
>
> h1 :: Int -> Int
> h1 = g . f
>
> h2 :: Int -> Int
> h2 = f . g
> ```
>
> Učitavanjem u *GHCi*, možemo proveriti kompozicije nad nekoliko vrednosti
>
> ```
> ghci> h1 3
> 36
> ghci> h2 3
> 18
> ```

Primer 13. Neka je dat kod

```
paran :: Int -> Bool
paran x = mod x 2 == 0

dužina :: [a] -> Int
dužina xs = length xs
```

Dve navedene funkcije možemo da komponujemo samo na jedan način. Pokušaj konstrukcije kompozicije `dužina . paran` dovešće do tipske greške.

```
ghci> k1 = paran . dužina
ghci> k2 = dužina . paran

<interactive>:7:15: error:
    • Couldn't match type ‘Bool’ with ‘t0 a0’
        Expected: a -> t0 a0
          Actual: a -> Bool
    • In the second argument of ‘(.)’, namely ‘paran’
      In the expression: dužina . paran
      In an equation for ‘k2’: k2 = dužina . paran
```

Zadatak 5. Neka je `f :: A -> B` i `g :: B -> C -> D`. Da li je `g . f` dobro tipiziran izraz? Ako jeste, koji je njegov tip?

Zadatak 6. Neka je `f :: A -> B -> C` i `g :: C -> D`. Da li je `g . f` dobro tipiziran izraz? Ako jeste, koji je njegov tip?

Zadatak 7. Neka je `f :: (A -> B) -> C` i `g :: C -> D`. Da li je `g . f` dobro tipiziran izraz? Ako jeste, koji je njegov tip?

Zadatak 8. Neka su `p` i `q` dve funkcije tipa `a -> Bool`. Zašto su

```
filter p . filter q
```

i

```
filter (\x -> p x && q x)
```

iste funkcije?

Rešenje. Kako je

```
(filter p . filter q) xs = filter p (filter q xs)
```

to se primenom funkcije `filter p . filter q` na listu `xs` odstranjuju svi elementi koji ne zadovoljavaju predikat `q`, a zatim se odstranjuju sve elementi koji ne zadovoljavaju predikat `p`. Prema tome, u listi

```
(filter p . filter q) xs
```

se nalaze svi elementi koji zadovoljavaju predikate `p` i `q`. A upravo se takva lista dobija i kao

```
filter (\x -> p x && q x) xs.
```

Zadatak 9. Implementirati funkciju `ks :: [a -> a] -> a -> a` koja uzima listu funkcija i vraća njihovu kompoziciju. Tačnije, za funkciju `ks` mora važiti

$$\text{ks}\,[f_1, f_2, ..., f_n] = f_1 \circ f_2 \circ \cdots \circ f_n$$

Rešenje. Zadatak možemo da rešimo rekurzivno. Funkcija `ks` primenjena na praznu listu treba da nam da identičku funkciju `\x -> x` koja je već definisana u Haskelu pod imenom `id`. Ako lista funkcija nije prazna, tada je možemo rastaviti na glavu i na rep, rekurzivno naći kompoziciju repa, a zatim komponovati glavu sa tom kompozicijom:

```
ks :: [a -> a] -> a -> a
ks [] = id
ks (f:fs) = f . ks fs
```

Navedeni rekurzivni šablon može se zameniti sa `foldr` funkcijom:

```
ks :: [a -> a] -> a -> a
ks fs = foldr (.) id fs
```

Zadatak 10. Dati primer tipa X i funkcije $f : X \to X$ takve da važi

$$f = f \circ f = f \circ f \circ f = \cdots$$

Zadatak 11. Dati primer tipa X i funkcije $f : X \to X$ takve da važi

$$f \neq f \circ f, \qquad f = f \circ f \circ f.$$

Zadatak 12. Kog je tipa izraz `. (.)`?

Zadatak 13. Kog je tipa izraz `(.) (.)`?

12.4.2. Operator aplikacije

Operator aplikacije je operator koji na prvi pogled ne radi ništa korisno. Definicija ovog operatora je

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ a = f a
```

Dakle, operator `$` primenjuje (aplicira) funkciju na vrednost. Razlog zašto je `$` koristan, je taj što je `$` definisan sa najnižim prioritetom, dok aplikacija funkcije na vrednost (`f x`) ima najviši prioritet. Pošto `$` ima najniži prioritet, često se može iskoristiti za uklanjanje zagrada iz izraza.

Primer 14. Ako želimo funkciju `f :: Int -> Int` da primenimo na vrednost `32 * 81`, tada je neispravno napisati `f 32 * 81`. Aplikacija ima veći prioritet od množenja pa se izraz `f 32 * 81` tumači kao `(f 32) * 81` što ne želimo. Ali ako iskoristimo operator aplikacije `$`, tada jednostavno možemo napisati `f $ 32 * 81`. Kako `$` ima manji prioritet od *, navedeni izraz će se protumačiti kao `f (32 * 81)` što smo i želeli.

Naravno, ušteda od jednog karaktera koju smo ostvarili na prethodnom primeru nije vredna uvođenja posebnog operatora. Ali operator `$` se može iskoristiti u mnogim drugim situacijama.

> **Primer 15.** Ako bismo želeli da niz funkcija `f1`, `f2`, `f3`, `f4` primenimo jednu za drugom, prvi pokušaj bi verovatno bio
>
> ```
> f4 (f3 (f2 (f1 x)))
> ```
>
> Primetimo da ne možemo izostaviti zagrade iz navedenog izraza, jer je aplikacija levo asocijativna te se izraz `f4 f3 f2 f1 x` tumači kao `(((f4 f3) f2) f1) x` (što u ovom slučaju nema smisla).
>
> I ovde ćemo iskoristiti `$` da pojednostavimo zapis. Korišćenjem činjenice da je `$` definisan kao desnoasocijativan operator, možemo pisati
>
> ```
> f4 $ f3 $ f2 $ f1 x
> ```
>
> Naravno, navedeni kod možemo napisati i korišćenjem kompozicije. Time dobijamo
>
> ```
> (f4 . f3 . f2 . f1) x
> ```
>
> Međutim, možemo da iskoristimo operator `$` da bismo aplicirali kompoziciju na vrednost `x`. Kako `$` ima manji prioritet od `.`, možemo pisati
>
> ```
> f4 . f3 . f2 . f1 $ x
> ```

Osim što se često koristi da pojednostavi zapis, `$` je koristan kad god je potrebno eksplicitno navesti primenu funkcije na argument. Na primer, ako su nam date liste `[1, 2, 3] :: [Int]` i `[f, g, h] :: [Int -> Int]`, tada uparivanje funkcija sa brojevima možemo jednostavno napisati kao

```
zipWith ($) [f, g, h] [1, 2, 3]
```

Ovim se naravno dobija lista `[f 1, g 2, h 3]`

Slično, ako je data lista funkcija `[f1, f2, f3]` i *jedna* vrednost `x`, tada možemo iskoristiti parcijalnu aplikaciju i napisati `map ($ x) [f1, f2, f3]` da bi smo dobili listu `[f1 x, f2 x, f3 x]`[117].

12.4.3. Operator podizanja

Za funkciju `map` se kaže da *podiže* funkcije tipa `A -> B` do funkcija tipa `[A] -> [B]`. Zaista, kada `map` apliciramo na `f :: A -> B`, dobijamo vrednost tipa `[A] -> [B]`. Zbog toga što se `map` izuzetno često koristi u kodu, ova funkcija poseduje i operatorski oblik `<$>`[118]. Sa leve strane operatora se navodi funkcija, a sa desne strane lista.

> **Primer 16.** Ako smo do sada koristili `map`, ne bi trebalo da imamo problema sa `<$>`.
>
> ```
> ghci> (*2) <$> [1 .. 10]
> [2,4,6,8,10,12,14,16,18,20]
> ```

[117] Da ne znamo za `$`, verovatno bi smo sa `map` funkcijom iskoristili lambdu `\f -> f x`.

[118] Zapravo `<$>` je funkcija `fmap :: Functor φ => (a -> b) -> φ a -> φ b` koja se može koristiti za podizanje u kontekstu mnogih struktura podataka (a ne samo lista). Funkcija `map` je samo specijalan slučaj funkcije `fmap`. Više o tome kasnije...

Zadatak 14. Neka je `f :: A -> B` i `g :: B -> C`. Zašto su `map g . map f` i `map (g . f)` iste funkcije? Koji su odgovarajući izrazi sa `<$>` operatorom?

Rešenje. Funkcija `map g . map f` je kompozicija funkcija `map f` i `map g`. Funkcija `map f` transformiše neku listu `[x₁, x₂, … xₙ]` u listu

$$\texttt{[f x}_1\texttt{, f x}_2\texttt{ … f x}_n\texttt{]},$$

a zatim funkcija `map g` transformiše tu listu u

$$\texttt{[g (f x}_1\texttt{), g (f x}_2\texttt{), … g (f x}_n\texttt{)]}.$$

Sa druge strane, funkcija `map (g . f)` primenjuje kompoziciju `g . f` na svaki element liste, odnosno `[x₁, x₂ … xₙ]` transformiše u

$$\texttt{[(g . f) x}_1\texttt{, (g . f) x}_2\texttt{, … (g . f) x}_n\texttt{]}.$$

Međutim, po definiciji kompozicije funkcije, jasno je da su liste

$$\texttt{[g (f x}_1\texttt{), g (f x}_2\texttt{), … g(f x}_n\texttt{)]}$$

i

$$\texttt{[(g . f) x}_1\texttt{, (g . f) x}_2\texttt{, … (g . f) x}_n\texttt{]}$$

iste vrednosti.

Kako se `map f` može zapisati kao `(f <$>)`, i analogno za `g`, to se jednakost `map g . map f = map (g . f)` može zapisati kao

```
(g <$>) . (f <$>) = ((g . f) <$>).
```

Ako obe strane ove jednakost apliciramo na vrednost `xs`, dobijamo

```
((g <$>) . (f <$>)) xs = (g . f) <$> xs.
```

Raspisivanjem kompozicije sa leve strane, dobijamo

```
g <$> (f <$> xs) = (g . f) <$> xs.
```

12.5. Primer: Određen integral

U nastavku je dat jedan primer koji nije neophodan za razumevanje ostatka knjige. Primer zahteva određena predznanja iz matematike.

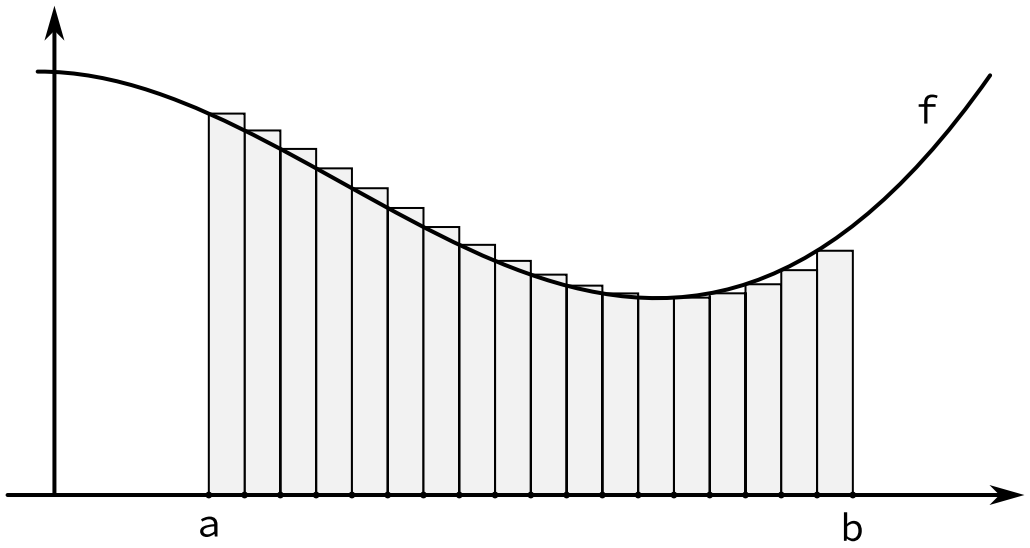
Otkriće diferencijalnog i integralnog računa je jedno od najznačajnijih trenutaka u istoriji nauke. Navedeni matematički aparati su neophodni za formulaciju mnogih prirodnih fenomena: od kretanja elektrona do kretanja galaksija. Svakako, ova tematika je van opsega ove knjige, ali ovde možemo da prikažemo kako se elegantno može implementirati formula za aproksimaciju određenih integrala.

Određeni integral funkcije $f : \mathbb{R} \to \mathbb{R}$ na intervalu $[a, b]$, označava se sa

$$\int_a^b f(x)\,\mathrm{d}x$$

i može se shvatiti kao površina između grafika funkcije f i x ose[119]. Integral, tj. površina, može se aproksimirati *Rimanovom sumom*, odnosno sumom površina malih pravougaonika koji aproksimiraju površinu ispod grafika.

[119] Pri čemu se ta površina uzima sa negativnim predznakom na oblastima gde je $f(x) < 0$.



Aproksimacija površine ispod grafika funkcije f nizom pravougaonika. Iako nije neophodno da svi pravougaonici budu iste širine, mi ćemo algoritam implementirati tako da su svi iste širine, baš kao na ovoj ilustraciji.

Rimanova suma se može zapisati kao

$$I = \sum_{i=0}^{n} f(x_i)(x_{i+1} - x_i),$$

gde je

$$a = x_0 < x_1 < \cdots < x_n < x_{n+1} = b$$

niz tačaka koje dele interval $[a, b]$. Navedene podeone tačke se mogu uzeti proizvoljno, ali što ih više ima to će kvadratići biti uži, a samim tim će aproksimacija površine biti tačnija.

Lako možemo implementirati funkciju

```
int :: (Double -> Double) -> Double -> Double -> Double
```

koja računa određen integral neke funkcije `f :: Double -> Double`. Prvi parametar funkcije `int` je funkcija `f`, drugi početak intervala a treći kraj intervala. Povratna vrednost je aproksimacija određenog integrala Rimanovom sumom.

Listu podeonih tačaka možemo formirati korišćenjem izlistavanja. Ako su `a` i `b` krajevi intervala, a `d` neki dovoljno mali broj, tada lista `[a, a + d .. b]` predstavlja podelu intervala $[a, b]$. Primenom funkcije `f` na svaki element date liste, dobijamo listu visina pravougaonika:

```
map f [a, a + d .. b].
```

Množenjem svake od ovih visina sa širinom intervala, odnosno sa `d`, dobijamo listu površina pravougaonika:

```
map (\h -> h * d) (map f [a, a + d .. b]).
```

Na kraju, dovoljno je da površine saberemo:

```
sum (map (\h -> h * d) (map f [a, a + d .. b])).
```

Kompletna implementacija funkcije `int` je

```
int :: (Double -> Double) -> Double -> Double -> Double
int f a b = sum povrsine
    where
        povrsine = map (\h -> h * d) (map f [a, a + d .. b])
        d = (b - a) / 200
```

U ovoj implementaciji uzeto je da je `d` jednak 200-tom delu intervala $[a, b]$. Za veću preciznost, dovoljno je uvećati konstantu `200`.

Primer 17. U *GHCi* okruženju je dostupna funkcija `sin` kao i konstanta `pi`. Ako učitamo implementaciju funkcije `int` možemo da izračunamo integral sinusa na intervalu $[0, \pi]$:

```
ghci> int sin 0 pi
1.9999588764792144
```

Koristeći operatore i sekcije, prethodnu definiciju možemo elegantnije da napišemo. Umesto `map` funkcije možemo iskoristiti operator podizanja `<$>`. Umesto lambda izraza `\h -> h * d`, možemo koristiti sekciju `(*d)`. A umesto `sum (map g ys)`, možemo iskoristiti `$` i osloboditi se zagrada: `sum $ map g ys`. Sve navedeno nas dovodi do naredne definicije:

```
int :: (Double -> Double) -> Double -> Double -> Double
int f a b = sum $ (* d) <$> (f <$> [a, a + d .. b])
    where d = (b - a) / 200
```

Zadatak 15. Koristeći činjenicu da je

$$\sum_{i=0}^{n}(f(x_i) \cdot d) = d \cdot \sum_{i=0}^{n} f(x_i),$$

dodatno pojednostaviti kôd funkcije `int`.

Rešenje.

```
int :: (Double -> Double) -> Double -> Double -> Double
int f a b = d * (sum $ f <$> [a, a + d .. b])
    where d = (b - a) / 200
```

12.6. Zaključak

Kreiranje operatora je moćna tehnika koju malo programskih jezika poseduje. Korišćenje operatora kao svih ostalih funkcija, kao i *ad hoc* transformisanje operatora u prefiksne funkcije i obrnuto, su tehnike koje još manje jezika poseduje. Nesumnjivo je da navedene osobine dozvoljavaju Haskel programeru veliku ekspresivnost.

Ipak, potrebno je biti veoma pažljiv sa kreiranjem operatora. Uvođenjem novih operatora možemo značajno otežati razumevanje koda. Čak i sami sebi možemo otežati programiranje kada posle kratke pauze u radu na nekom projektu zaboravimo semantiku operatora koje smo ranije definisali.

Iako nećemo u narednim lekcijama često definisati operatore, upoznaćemo sa mnogim, za nas novim, operatorima već definisanim u Haskel jeziku. Kao što ćemo videti, takvi operatori će biti korisni u rešavanju mnogih problema.

13. Tipske klase

Tipska klasa jeste kolekcija tipova koji imaju neke zajedničke osobine. Te zajedničke osobine izražavaju se kroz metode[120] koji se mogu pozivati nad tipovima te klase. Svaki tip koji pripada nekoj klasi naziva se **instanca** te klase.

U lekciji o polimorfizmu videli smo da nam tipske klase omogućuju da pišemo polimorfne funkcije koje koriste neke određene metode koje nisu dostupne svakom tipu[121]. U ovoj lekciji ćemo prikazati kako i sami možemo definisati naše klase, i kroz primere predefinisanih klase videćemo razne tehnike koje je moguće koristiti pri radu sa klasama.

Klasa se definiše deklaracijom klase nakon koje sledi niz deklaracija tipova metoda koje instanca mora da implementira:

```
class C a where
  f1 :: T1
  f2 :: T2
  ⋮
```

Ovde `a` je tipska promenljiva koja predstavlja neku instancu klase `C`. Tipovi `T1`, `T2`... mogu (i ne moraju) da sadrže tipsku promenljivu `a`.

Da bi neki tip `T` pridružili klasi tipova `C`, koristimo narednu konstrukciju nakon koje slede definicije metoda koje propisuje klasa:

```
instance C T where
  f1 = …
  f2 = …
  ⋮
```

U okviru definicije instance ne navode se tipovi metoda.

13.1. Jedan primer

Gotovo da nema uvodne knjige o programiranju u nekom jeziku koja ne sadrži primer sa geometrijskim figurama poput pravougaonika, trougla i kruga. Implementirati strukture koje predstavljaju figure i funkcije koje računaju obim i površinu figura je jednostavno u svakom popularnom jeziku, ali može da posluži za odličnu ilustraciju mnogobrojnih pojedinosti nekog programskog jezika. Nama će poslužiti da ilustrujemo zašto su klase tipova korisne.

Prvo ćemo definisati tri tipa podatka koji predstavljaju figure u Dekartovoj ravni:

1. Svaki pravougaonik se može odrediti sa jednim temenom, širinom i visinom (jednostavnosti radi, pretpostavljamo da su stranice pravougaonika paralelne sa koordinatnim osama). Kako teme možemo predstaviti sa dve koordinate, ceo pravougaonik možemo predstaviti sa četiri `Double` vrednosti (dve za koordinate jednog temena, dve za dužine stranica).
2. Trougao je u potpunosti određen sa tri temena. Prema tome, za predstavljanje trougla nam je potrebna struktura podataka koja sadrži šest `Double` vrednosti.
3. Krug je određen centrom i poluprečnikom. Stoga struktura koja predstavlja krug sadrži tri `Double` vrednosti.

Kada znamo šta strukture sadrže, lako ih je implementirati kao uređene n-torke:

[120] *Metod* je samo naziv za funkciju (specijalno operator) koju klasa deklariše.

[121] Na primer, koristeći klasu numeričkih tipova `Num`, moguće sa jednom definicijom implementirati numeričku formulu za svaki tip koji pripada klasi `Num`.

```
newtype Pravougaonik = Pravougaonik (Double, Double, Double, Double)
  deriving Show

newtype Trougao = Trougao (Double, Double, Double, Double, Double, Double)
  deriving Show

newtype Krug = Krug (Double, Double, Double)
  deriving Show
```

Iz navedenog koda nije jasno šta navedene koordinate predstavljaju. U slučaju pravougaonika prve dve koordinate su koordinate "levog donjeg" temena, dok druge dve koordinate predstavljaju visinu i širinu. U slučaju trougla, prve dve koordinate predstavljaju koordinate prvog temena, druge dve drugog, i poslednje dve koordinate predstavljaju koordinate trećeg temena. U slučaju kruga, prve dve koordinate predstavljaju koordinate centra, dok treća koordinata predstavlja poluprečnik.

Implementacija funkcija koje računaju obim navedenih figura je takođe jednostavna:

```
obimPravougaonika :: Pravougaonik -> Double
obimPravougaonika (Pravougaonik (_, _, a, b)) = 2 * (a + b)

obimTrougla :: Trougao -> Double
obimTrougla (Trougao (x1, y1, x2, y2, x3, y3)) = a + b + c
  where
    a = sqrt $ (x1 - x2)^2 + (y1 - y2)^2
    b = sqrt $ (x2 - x3)^2 + (y2 - y3)^2
    c = sqrt $ (x3 - x1)^2 + (y3 - y1)^2

obimKruga :: Krug -> Double
obimKruga (Krug (x, y, r)) = 2 * pi * r
```

Za računanje obima trougla, prvo smo izračunali dužinu stranica korišćenjem Pitagorine teoreme.

Navedeni kôd je u potpunosti tačan. Ali u ovakvom pristupu se krije mali problem: pisanjem koda za rad sa figurama zahteva od nas da koristimo tri različite funkcije (`obimPravougaonika`, `obimTrougla`, `obimKruga`). Bilo bi jednostavnije kada bismo mogli primeniti jedinstvenu funkciju `obim` na svaku figuru[122].

Da bismo koristili jedinstvenu funkciju `obim` sa sva tri tipa, definisaćemo klasu `Figura`. Pošto želimo da sa svakom instancom klase `Figura` koristimo metod `obim` u definiciji klase ćemo navesti samo jednu deklaraciju tipa.

```
class Figura a where
  obim :: a -> Double
```

Linijom `class ...` započinje se blok definicije klase. Sve linije (nakon prve) u ovom bloku moraju biti nazubljene u odnosu na prvu liniju[123]. Blok se prostire sve do linije koja nije nazubljena.

Slobodna tipska promenljiva `a` u liniji `class Figura a where` predstavlja proizvoljnu instancu klase `Figura`. Ova promenljiva, kao i svaka druga, može biti proizvoljno imenovana. Svako pojavljivanje promenljive `a` unutar bloka odnosi se na isti tip.

Navedeni kôd već možemo učitati u *GHCi*. Korišćenjem komande `:type` možemo uvideti da je tip funkcije `obim` dat sa `Figura a => a -> Double`. Ovo nam govori da se funkcija `obim` može primeniti samo na neku

[122] Baš kao što funkciju `show` možemo primeniti i na vrednost tipa `Bool` i na vrednost tipa `Int`, a operator `+` možemo koristiti i sa `Int` i sa `Float` vrednostima, itd... Bilo bi zaista nezgodno kada bi svaki tip zahtevao funkciju sa jedinstvenim imenom.

[123] Broj razmaka sa kojim se vrši nazubljivanje nije bitan, ali je neophodno da taj broj bude jednak za sve linije unutar tog bloka.

vrednost tipa `A`, pri čemu taj tip `A` pripada klasi `Figura`. Za sada takvi tipovi ne postoje: potrebno je tipove `Pravougaonik`, `Trougao` i `Krug` učiniti instancama klase `Figura`.

Pošto klasa `Figura` pripisuje samo jednu funkciju (`obim`), instanciranje podrazumeva implementaciju funkcije `obim` za svaki od navedenih tipova.

```
instance Figura Pravougaonik where
  obim (Pravougaonik (_, _, a, b)) = 2 * (a + b)

instance Figura Trougao where
  obim (Trougao (x1, y1, x2, y2, x3, y3)) = a + b + c
    where
      a = sqrt $ (x1 - x2)^2 + (y1 - y2)^2
      b = sqrt $ (x2 - x3)^2 + (y2 - y3)^2
      c = sqrt $ (x3 - x1)^2 + (y3 - y1)^2

instance Figura Krug where
  obim (Krug (_, _, r)) = 2 * pi * r
```

Definicije funkcija su u potpunosti iste kao definicije funkcija `obimPravougaonika`, `obimTrougla`, `obimKruga`.

Linijom `instance ...` započinje se blok definicija metoda. Kao i kod definicije klase, sve u ovom bloku mora biti nazubljeno u odnosu na početak linije.

U bloku definicija, neophodno je definisati sve metode koje klasa deklariše[124]. U bloku definicija ne navode se deklaracije tipova!

Nakon učitavanja navedenih definicija u *GHCi*, možemo se uveriti da se funkcija `obim` može primeniti na razne figure:

```
ghci> trougao = Trougao (0, 0, 1, 0, 0, 1)
ghci> obim trougao
3.414213562373095
ghci> pravougaonik = Pravougaonik (0, 0, 2, 2)
ghci> obim pravougaonik
8.0
ghci> krug = Krug (0, 0, 10)
ghci> obim krug
62.83185307179586
```

Figure osim što imaju obim, imaju i površinu. Stoga ćemo u definiciju klase `Figura` dodati i deklaraciju funkcije `površina` koja nam daje površinu figure. Nakon dodavanja nove deklaracije u definiciji klase, u svakoj instanci moramo implementirati i funkciju `površina`. Površinu pravougaonika i kruga ćemo lako izračunati, a za površinu trougla ćemo iskoristiti Heronovu formulu. Kompletan kôd sada izgleda ovako:

```
class Figura a where
  obim :: a -> Double
  površina :: a -> Double


instance Figura Pravougaonik where
  obim (Pravougaonik (_, _, a, b)) = 2 * (a + b)
```

[124] Postoji par izuzetaka koje ćemo kasnije pojasniti.

```
  površina (Pravougaonik (_, _, a, b)) = a * b


instance Figura Trougao where
  obim (Trougao (x1, y1, x2, y2, x3, y3)) = a + b + c
    where
      a = sqrt $ (x1 - x2)^2 + (y1 - y2)^2
      b = sqrt $ (x2 - x3)^2 + (y2 - y3)^2
      c = sqrt $ (x3 - x1)^2 + (y3 - y1)^2

  površina (Trougao (x1, y1, x2, y2, x3, y3)) = sqrt . product $ [s, s-a, s-b, s-c]
      where
        a = sqrt $ (x1 - x2)^2 + (y1 - y2)^2
        b = sqrt $ (x2 - x3)^2 + (y2 - y3)^2
        c = sqrt $ (x3 - x1)^2 + (y3 - y1)^2
        s = (a + b + c) / 2


instance Figura Krug where
  obim (Krug (_, _, r)) = 2 * pi * r

  površina (Krug (_, _, r)) = pi * r^2
```

Više o klasama tipova naučićemo na primerima poznatih klasa koje su već definisane u Haskelu.

13.2. *Equality*

U klasi `Eq` nalaze se svi tipovi čije vrednosti je moguće porediti pomoću operatora `==` i `/=`. Definicija klase `Eq` je sasvim jednostavna:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Primer 1. Definišimo tip koji predstavlja racionalan broj. Kako je svaki racionalan broj oblika a/b za neka sva cela broja a i $b \neq 0$, to tip `Racionalan` možemo definisati kao uređen par celih brojeva:

```
newtype Racionalan = R (Int, Int)
```

Pokušaj korišćenja vrednosti tipa `Racionalan` sa operatorima klase `Eq` dovešće do greške, jer tip `Racionalan` nije instanca ove klase:

```
ghci> a = R (1, 2)
ghci> b = R (5, 10)
ghci> a == b

<interactive>:1:3: error:
    • No instance for (Eq Racionalan) arising from a use of ‘==’
    • In the expression: a == b
      In an equation for ‘it’: it = a == b
```

Da bismo mogli da poredimo vrednosti tipa `Racionalan`, instanciraćemo `Eq Racionalan`:

```
instance Eq Racionalan where
    (R (a, b)) == (R (c, d)) = a * d == c * b
    (R (a, b)) /= (R (c, d)) = a * d /= c * b
```

Ponovnim učitavanjem koda u *GHCi*, dobijamo mogućnost poređenja vrednosti tipa `Racionalan`:

```
ghci> a = R (1, 2)
ghci> b = R (5, 10)
ghci> a == b
True
ghci> a /= b
False
```

Primetimo da deluje nepotrebno definisati obe funkcije `==` i `/=`. Zaista, ako bi smo poznavali jednu od ove dve funkcije, bilo bi prirodno da izvedemo onu drugu kao njenu negaciju (ako su dve vrednosti jednake onda nisu različite, i obrnuto). Haskel jezik nam omogućuje i to. Osim što u klasi možemo deklarisati tipove funkcija, možemo definisati i neke od tih funkcija preko ostalih a zatim navesti minimalni skup funkcija koje moraju biti definisane u instanci[125].

Konkretno, puna definicija klase `Eq` je[126]:

```
class Eq a where
  (==), (/=)            :: a -> a -> Bool

  x /= y                = not (x == y)
  x == y                = not (x /= y)
  {-# MINIMAL (==) | (/=) #-}
```

Linija `{-# MINIMAL (==) | (/=) #-}`, koja samo podseća na komentar, naziva se **pragma**, i služi da kompajleru ukaže da je barem jednu od funkcija `==` i `/=` dovoljno i potrebno implementirati. Kroz naredne primere upoznaćemo se detaljnije sa sintaksom pragme *MINIMAL*.

Svaka od te dve funkcije se može izvesti preko one druge, i u samoj definiciji klase `Eq` su navedena ta izvođenja. Međutim, da se ne bismo vrteli u krug sa definicijama[127], svaka instanca `Eq` klase mora sadržati barem jednu od definicija operatora `==` ili `/=`.

Primer 2. *Nastavak prethodnog primera.* Za instancu `Racionalan`, dovoljno je bilo napisati sledeće

```
instance Eq Racionalan where
    (R (a, b)) == (R (c, d)) = a * d == c * b
```

Umesto definicije za `==`, mogli smo da definišemo `/=`.

13.3. *Ordering*

Klasu `Ord` čine svi tipovi koje je moguće porediti pomoću operatora `<`, `<=`, `>` i `>=`. Definicija klase `Ord` je sledeća:

[125] Pritom, taj skup neophodnih definicija ne mora biti jedinstven...
[126] Gore smo naveli izmenjenu definiciju klase `Eq`, radi jednostavnosti
[127] Ako definišemo `==` kao negaciju `/=`, a `/=` kao negaciju `==`, tada bi svako poređenje dovelo do beskonačne rekurzije.

```
class (Eq a) => Ord a where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <= y = (compare x y) /= GT

    x >= y = y <= x
    x > y = not (x <= y)
    x < y = not (y <= x)

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
    {-# MINIMAL compare | (<=) #-}
```

Ovde vidimo nešto drugačiju definiciju klase. Umesto sa `class Ord a where` definicija klase započinje sa `class (Eq a) => Ord a where`. Izraz `(Eq a) =>` predstavlja klasno ograničenje. Kao i kod tipova vrednosti, klasnim ograničenjima ograničavamo skup tipova koji mogu biti postavljeni na mesto tipske promenljive `a`. U ovom slučaju, sa klasnim ograničenjem garantujemo da će neki tip pripadati nekoj drugoj klasi pre nego što ga pridružimo ovoj klasi.

U slučaju klase `Ord`, ima smisla zahtevati da tip već pripada klasi `Eq` jer pojam jednakosti vrednosti neophodan za razlikovanje funkcija `>` i `>=`[128]. Prema tome, klasa `Ord` je podklasa klase `Eq`.

Klasa `Ord` propisuje funkciju `compare :: a -> a -> Ordering` koja upoređuje dve vrednosti, uobičajene relacije (operatore koje vraćaju logičke vrednosti) za poređenje elementa kao i dve binarne funkcije `min` i `max` koje respektivno vraćaju manji odnosno veći od argumenata.

Tip `Ordering` je tip koji sadrži tri vrednosti `LT`, `EQ`, `GT` koje označavaju rezultat poređenja dve vrednosti. Vrednost `compare a b` je `LT`, `EQ`, `GT` ako je vrednost `a` strogo manja, jednaka, strogo veća od vrednosti `b`.

Definicije operatora `<`, `>`, `>=` koriste operator `<=` i prilično su jasne. Jasne su i definicije binarnih funkcija `min` i `max` koje takođe koriste operator `<=`. Operator `<=` je definisan preko funkcije `compare`. Funkcija `compare` preko funkcije `<=`. Kao i kod klase `Eq`, i ovde imamo slučaj beskonačne rekurzije kroz definicije. Stoga je potrebno, i dovoljno, u instanci definisati samo jednu od funkcija `<=` ili `comapre`.

I ovde imamo `MINIMAL` pragmu koja sugeriše kompajleru da je neophodno da instanca sadrži barem definiciju funkcije `comapre` ili `<=`. U instanci je moguće definisati i ostale funkcije (u tom slučaju te definicije instance će potisnuti definicije u definiciji klase), ali za tako nečim gotovo nikad nema potrebe.

> **Primer 3.** Načinimo tip `Racionalan` instancom `Ord` klase. Pošto smo već u prethodnim primerima implementirali `Eq Racionalan` instancu, dovoljno je da implementiramo `compare` funkciju.

[128] Drugim rečima, ako možemo da poredimo vrednosti sa `>` i `>=`, tada `a == b` možemo definisati kao

```
(a >= b) && not (a > b).
```

Sa druge strane ako možemo da poredimo vrednosti sa `==` i `/=`, to nam ništa ne govori o poređenju vrednosti u smislu operatora `<`, `<=`, `>`, `>=`. Dakle, funkcije koje definiše klasa `Eq` su elementarnije od onih funkcija klase `Ord`. Stoga ima smisla zahtevati da `Ord` bude podklasa klase `Eq`.

```
instance Ord Racionalan where
  compare (R (a, b)) (R (c, d)) = compare (a * d) (c * b)
```

U implementaciji smo iskoristili činjenicu da je odnos između brojeva a/b i c/d isti kao odnos između brojeva $a \cdot d$ i $c \cdot b$.

13.4. *Show* i *Read*

Klasu `Show` čine oni tipovi čije se vrednosti mogu prezentovati u vidu niske. Za nas je važno da definicija klase `Show` podrazumeva funkciju `show`:

```
class Show a where
  show :: a -> String
```

Ovde nismo naveli kompletnu definiciju klase, ali je dovoljno znati da minimalno instanciranje `Show` klase podrazumeva samo implementaciju funkcije `show`.

Klasa `Show` je neophodna za ispisivanje vrednosti u *GHCi* okruženju. Do sada smo koristili `deriving (Show)` konstrukciju koja je omogućavala da se tip automatski pridruži `Show` klasi tj . da se *izvede* instanca. Sada znamo kako možemo implementirati sopstvene instance.

Primer 4. Ako želimo da izvedemo `Show Racionalan`, dovoljno je definiciju tipa promenimo u sledeću:

```
newtype Racionalan = R (Int, Int) deriving (Show)
```

Pri izvođenju, `show` funkcija će kreirati nisku koja liči veoma na sam kod s kojim je konstruisana vrednost.

```
ghci> a = R (2, 3)
ghci> show a
"R (2,3)"
ghci> a
R (2,3)
```

Kada u *GHCi* unesemo neki izraz, taj izraz će se evaluirati, i dobijena vrednost će se prikazati kao niska dobijena primenom `show` funkcije.

Za tip `Racionalan` je bolje implementirati posebnu `show` funkciju, i ne koristiti izvođenje. U tom slučaju treba ukloniti `deriving (Show)`, a dodati instancu `Show Racionalan`:

```
instance Show Racionalan where
  show (R (x, y)) = show x ++ "/" ++ show y
```

Primetimo da u definiciji koristimo takođe `show` funkciju, ali primenjenu na celobrojne vrednosti. Možemo da kažemo da `show` sa leve i desne strane znaka `=`, ne predstavljaju iste funkcije.

Prikazivanje vrednosti je sada smislenije:

```
ghci> a = R (2, 3)
ghci> show a
"2/3"
ghci> a
2/3
```

Klasa `Read` je suprotnost klasi `Show`. Instance klase `Read` su tipovi čije se vrednosti mogu "pročitati" iz niske. Kako je "čitanje" vrednosti iz niske značajno složenije nego zapisivanje u nisku i definicija klase `Read` je složenija. Stoga ovde nećemo navoditi definiciju klase.

Zgodno je ipak znati za funkciju `read :: Read a => String -> a` koja se može koristiti sa instancama klase `Read`.

Primer 5. Ugrađeni tipovi poput numeričkih su instance klase `Read`. Stoga, sa `read` funkcijom možemo pročitati vrednost iz niske.

```
ghci> read "233"
*** Exception: Prelude.read: no parse
```

Razlog zašto smo dobili izuzetak je taj zato što interpreter "ne zna" kakva vrednost je zapisana u nisci. Tip funkcije `read` je `Read a => String -> a`. Vidimo da je kodomen parametrizovan, što znači da tip izraza `read "233"` *a priori* može da bude bilo koji tip klase `Read`. Stoga je potrebno deklarisati tip izraza `read "233"`:

```
ghci> read "233" :: Int
233
ghci> read "233" :: Float
233.0
ghci> read "-233" :: Double
-233
```

Mnogi drugi tipovi su takođe instance klase `Read`:

```
ghci> read "True" :: Bool
True
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(42,'a')" :: (Int, Char)
(42,'a')
```

13.5. *Number* i *Fractional*

Klasu `Num` čine svi tipovi koji predstavljaju nekakav skup brojeva (celih, racionalnih, realnih, kompleksnih...). Definicija klase `Num` je:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Ova klasa propisuje tri operatora `+`, `-` i `*` za koje je jasno šta predstavljaju. Funkcija `negate` negira vrednost (daje inverz u odnosu na sabiranje). Funkcija `abs` daje apsolutnu vrednost, dok `signum` daje znak. Funkcija `fromInteger` prevodi vrednost tipa `Integer`[129] u vrednost instance.

[129] Setimo se, `Integer` je tip koji predstavlja celobrojne vrednosti u proizvoljnoj preciznosti

Klasa `Fractional` je podklasa klase `Num` koja dozvoljava i deljenje vrednosti:

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
```

Funkcija `fromRational` transformiše vrednost tipa `Rational` u vrednost instance. Do sada tip `Rational` nismo spominjali, ali u pitanju je tip koji se koristi za prezentovanje racionalnih brojeva u proizvoljnoj tačnosti.

Funkcija `recip` daje recipročnu vrednost broja[130]. Dovoljno je implementirati samo jednu od funkcija `(/)` i `recip`, jer su one u definiciji klase definisane jedna preko druge:

```
recip x = 1 / x
x / y   = x * recip y
```

Definicije klasa `Num` i `Fractional` približno odgovaraju definicijama algebarskih struktura *prstena* i *polja*.

Primer 6. Uvedimo tip `Racionalan` u navedene klase. Imajući na umu zakone poput

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm cb}{cd} \qquad \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd} \qquad |\frac{a}{b}| = \frac{|a|}{|b|}$$

kodiranje je veoma jednostavno:

```
instance Num Racionalan where
  (R (a, b)) + (R (c, d)) = R (a*d + c*b, b*d)
  (R (a, b)) - (R (c, d)) = R (a*d - c*b, b*d)
  (R (a, b)) * (R (c, d)) = R (a*c, b*d)
  negate (R (a, b)) = R (negate a, b)
  abs (R (a, b)) = R (abs a, abs b)
  signum (R (a, b)) = R (signum a * signum b, 1)
  fromInteger n = R (fromIntegral n, 1)
```

instanciranje `Fractional Racionalan` je još jednostavnije, jer je dovoljno implementirati samo dve funkcije

```
instance Fractional Racionalan where
  recip (R (a, b)) = R (b, a)
  fromRational r = R (numerator r, denominator r)
```

Funkcije `numerator` i `denominator` su funkcije kojima se može pristupiti brojiocu i imeniocu vrednosti tipa `Rational`.

Jasno je da tipovi poput `Int`, `Integer`, `Float`, `Double` i `Rational` pripadaju klasi `Num`. Od navedenih tipova, samo prva dva ne pripadaju klasi `Fractional`. Iako deluje da su samo ovakvi tipovi instance ovih klasa, `Num` i `Fractional` mogu sadržati još mnoge zanimljive primere što pokazuju naredni zadaci.

[130] Recipročna vrednost vrednosti x je vrednost y takva da je $xy = 1$. U kontekstu `Fractional` klase, smatra se da važi

```
x * y = fromRational 1,
```

iako to nije propisano standardom.

Zadatak 1. Definisati tip kompleksnih brojeva sa

```
newtype Kompleksan = K (Double, Double).
```

Instancirati `Num Kompleksan` i `Fractional Kompleksan`. Apsolutnu vrednost implementirati kao normu kompleksnog broja, a znak kompleksnog broja implementirati tako da važi

```
abs z * signum z = z.
```

U narednim zadacima, funkcije `abs` i `signum` implementirati proizvoljno.

Zadatak 2. Definisati tip `Z4` koji predstavlja skup ostataka pri deljenju sa 4 tj

$$\mathbb{Z}_4 = \{0, 1, 2, 3\}.$$

Ovaj tip definisati kao

```
newtype Z4 = Z4 Int.
```

Instancirati `Num Z4` pri čemu voditi računa da se sve operacije vrše po modulu 4. Na primer $3 + 2 \underset{4}{=} 1$ i $2 \cdot 2 \underset{4}{=} 0$.

Zadatak 3. Analogno prethodnom zadatku definisati tip `Z5` koji prezentuje

$$\mathbb{Z}_5 = \{0, 1, 2, 3, 4\},$$

i instancirati `Num Z5`. Kreirati (sa ili bez računara) tablicu množenja za u skupu $\mathbb{Z}_5$ (naravno, u odnosu na množenje modulo 5). Utvrditi da je množenje u $\mathbb{Z}_5$ *regularno* tj. da je proizvod dva nenula elementa uvek različit od nule (iz prethodnog primera vidimo da to ne važi za $\mathbb{Z}_4$). Koristeći ovu činjenicu, definisati recipročnu vrednost svakog elementa iz $\mathbb{Z}_5$ i instancirati `Fractional Z5`

Zadatak 4. Definisati tip za predstavljanje polinoma sa celobrojnim koeficijentima kao

```
newtype Polinom = Polinom [Int].
```

Uvesti ovaj tip u klase `Eq`, `Show` i `Num`.

14. Algebarski tipovi podataka

Do sada smo videli kako možemo kreirati nove tipove koristeći `newtype` konstrukciju. U ovoj lekciji predstavićemo jedan opštiji način kreiranja novih tipova. Prva razlika u ovom načinu kreiranja tipova je ta što se umesto ključne reči `newtype` koristi `data`. Definicija tipa `T` sada će počinjati sa `data T = ...`, a umesto jednog konstruktora od sada ćemo potencijalno imati više konstruktora.

Tipove ćemo kreirati od postojećih tipova koristeći dve operacije: proizvod i sumu. Tipovi koji se dobijaju uz ove dve operacije nazivaju se **algebarski tipovi podataka**.

Navedene operacije odgovaraju pojmovima *Dekartovog proizvoda* i *unije* u teoriji skupova. Kao što ćemo uvideti, sličnost se ne završava na tome: proizvod i suma tipova poštuju zakone poznate iz algebre skupova i algebre prirodnih brojeva[131].

14.1. Proizvod

Proizvod dva tipa `A` i `B` je tip čije vrednosti odgovaraju kombinacijama vrednosti tipova `A` i `B`. Proizvod tipova u Haskelu se jednostavno konstruiše na sledeći način:

```
data P = MkP A B
```

Ovde, `P` je naziv novog tipa, a `MkP` je konstruktor.

Svaka vrednost gore definisanog tipa `P`, je oblika `MkP a b` gde su `a :: A` i `b :: B` neke vrednosti. Podudaranje oblika dozvoljava da vrednosti tipa `P` "dekonstruišemo" na ovaj način.

> **Primer 1.** Vektor dvodimenzionalne ravni možemo definisati kao proizvod tipova `Float` i `Float` na sledeći način:
>
> ```
> data Vektor = Vektor Float Float deriving (Show)
> ```
>
> Vrednosti se konstruišu sa konstruktorom:
>
> ```
> ghci> v = Vektor 2 1
> ```
>
> Konstruktor se koristi i za dekonstrukciju vrednosti sa tehnikom podudaranja oblika:
>
> ```
> dužinaVektora :: Vektor -> Float
> dužinaVektora (Vektor x y) = sqrt (x**2 + y**2)
> ```
>
> ```
> ghci> dužinaVektora v
> 2.23606797749979
> ```

Nije nužno postaviti iste tipove u proizvod, niti ih mora biti dva.

> **Primer 2.** Naredni tip predstavlja jednu osobu (njeno ime, godine, i to da li je državljanin Srbije):
>
> ```
> data Osoba = Osoba String Int Bool deriving (Show)
> ```
>
> Ovde, kao i u nekim prethodnim primerima sa `newtype` konstrukcijom, tip i konstruktor imaju isto ime
>
> Pri radu sa tipom `Osoba` takođe koristimo podudaranje oblika. Ako želimo da "izvučemo" ime iz osobe, možemo napisati narednu funkciju:

[131] Ipak, ovo nije razlog zašto se ovi tipovi nazivaju *algebarski*. Postoje dublji razlozi za ovaj naziv.

```
imeOsobe :: Osoba -> String
imeOsobe (Osoba ime _ _) = ime
```

Prethodna funkcija odgovara onome što u objektno orijentisanim jezicima nazivamo *getter* (metoda kojom se pristupa nekom atributu objekta). Lako je implementirati i *setter*, tj. funkciju koja menja jednu vrednost u proizvodu:

```
promeniIme :: String -> Osoba -> Osoba
promeniIme novoIme (Osoba _ godine drzavljanin) = Osoba novoIme godine drzavljanin
```

Zadatak 1. Napisati getere i setere za sva polja vrednosti tipa `Vektor` i `Osoba`.

Proizvod tipova može da sadrži i samo jedan tip. U tom slučaju proizvod je analogan `newtype` konstrukciji koju smo ranije upoznali.

14.1.1. Uređene n-torke kao proizvodi tipova

Pažljiv čitalac će uvideti da smo koncept proizvoda tipova već imali kod uređenih n-torki. I zaista tip `Osoba` iz prethodnog primera je u nekom smislu ekvivalentan tipu `([Char], Int, Bool)`. Svaku vrednost tipa `Osoba` možemo konvertovati u vrednost tipa `([Char], Int, Bool)` i obrnuto.

Sličnost nije slučajna, jer uređene n-torke jesu proizvodi tipova. Na primer, uređen par je *apstraktni* tip definisan sa

```
data (,) a b = (,) a b.
```

Slično su definisane i druge uređene n-torke

```
data (,,) a b c = (,,) a b c,
data (,,,) a b c d = (,,,) a b c d
```

i tako dalje. Primetimo da tipovi i konstruktori imaju isto ime!

Haskel kompajler dozvoljava da se ovaj konstruktor upotrebljava na malo drugačiji način, pa se može pisati `(1, 'a')` umesto `(,) 1 'a'`.

Koliko god ovo delovalo čudno, možemo se zaista uveriti da `(,)` jeste jedno ime koje dele konstruktor tipa i konstruktor vrednosti:

```
ghci> :t (,)
(,) :: a -> b -> (a, b)
ghci> :k (,)
(,) :: * -> * -> *
```

Zadatak 2. Koristeći proizvod tipova (bez korišćenja uređenih parova), definisati tip kompleksnih brojeva `Kompleksan` oslanjajući se na `Double` vrednosti. Instancirati `Num Kompleksan` i `Fractional Kompleksan`.

14.1.2. Jedinični tip

Konstrukcija proizvoda tipova ima oblik `K T₁ T₂ … Tₙ`, gde su `T₁`, … `Tₙ` neki tipovi, a `K` konstruktor. U specijalnom slučaju možemo napraviti proizvod čiji konstruktor ne uzima ni jedan dodatni tip:

```
data T = K
```

Konstruktor `K` je funkcija arnosti 0, odnosno konstanta tipa `T`. Drugim rečima, tip `T` sadrži samo jednu vrednost a to je `K`. Zbog toga za `T` takođe kažemo da je **jedinični tip** baš kao za `()`. Zapravo, tip `()` je takođe definisan kao prazan proizvod:

```
data () = ()
```

Ovo je još jedan primer gde konstruktor i tip imaju isto ime.

14.2. Suma

Suma dva tipa `A` i `B` je tip koji sadrži vrednosti koje odgovaraju svim vrednostima tipova `A` i `B`. Suma tipova može se shvati kao unija dva tipa. Suma tipova se vrši navođenjem više konstruktora razdvojenih vertikalnom crtom. Svaki od konstruktora predstavlja konstruktor proizvoda, i može imati proizvoljno mnogo argumenata.

Primer 3. Tip `SlovoIliBroj` koji sadrži i slova i brojeve može se konstruisati kao

```
data SlovoIliBroj = Slovo Char | Broj Int deriving (Show)
```

Sada postoje dva konstruktora vrednosti:

```
ghci> a1 = Slovo 'a'
ghci> a2 = Broj 2
ghci> :t a1
a1 :: SlovoIliBroj
ghci> :t a2
a2 :: SlovoIliBroj
```

Vrednosti `a1` i `a2` imaju isti tip, iako predstavljaju različite oblike podataka.

Konstruktore koristimo sa podudaranjem oblika. Svaka vrednost tipa `SlovoIliBroj` može biti ili oblika `Slovo x` ili oblika `Broj x`, te podudaramo takve oblike:

```
daLiJeSlovo :: SlovoIliBroj -> Bool
daLiJeSlovo (Slovo x) = True
daLiJeSlovo (Broj x) = False
```

Moguće je "sabrati" više tipova istovremeno, odnosno navesti više od dva konstruktora.

Primer 4. Sledeći tip označava dužinu u različitim mernim jedinicama

```
data Dužina = Metar Float | Milja Float | SvetlosnaSekunda Float
              deriving (Show)
```

Za tip `Dužina` možemo da definišemo ovakvu funkciju konverzije:

```
uMetre :: Dužina -> Dužina
uMetre (Milja x) = Metar (1609.344 * x)
uMetre (SvetlosnaSekunda x) = Metar (299792458 * x)
uMetre x = x
```

> Tip `Dužina` predstavlja uniju tri `Float` tipa. Ali iako su tipovi u sumi isti, ova unija je *disjunktna*. Drugim rečima vrednosti `Metar 1`, `Milja 1` i `SvetlosnaSekunda 1` su međusobno potpuno različite vrednosti, iako se svaka od ovih vrednosti dobila primenom konstruktora na `1 :: Float`

Napomenimo da poredak konstruktora u sumi nije od značaja. Tako na primer, definicija

```
data SlovoIliBroj = Slovo Char | Broj Int
```

definiše isti[132] tip kao i definicija `data SlovoIliBroj = Broj Int | Slovo Char`. Za razliku od toga, u proizvodima poredak tipova je bitan. Tako na primer, definicija

```
data SlovoIBroj = SIB Int Char
```

je sasvim različita od definicije[133]

```
data SlovoIBroj = SIB Char Int
```

14.2.1. Suma jediničnih tipova

Jedinični tipovi nisu mnogo korisni sami po sebi, ali su veoma korisni kada se koriste unutar sume. Na primer, sada lako (i logično) možemo da predstavimo tipove sa konačno mnogo članova (takozvane *enumeracije*):

```
data Pol = Muško | Žensko
```

```
data ZnakKarte = Herc | Karo | Tref | Pik
```

```
data Boje = Crna | Bela | Crvena | Plava | Zelena | Žuta
```

Do sada smo se susreli sa više tipova koji su suma jediničnih tipova (a da to nismo ni znali):

1. Logički tip `Bool` je definisan kao suma jediničnih tipova `True` i `False`.
2. Tip `Ordering` je definisan kao suma jediničnih tipova `LT`, `EQ` i `GT`. Podsećamo da se vrednosti ovog tipa koriste kao rezultati poređenja.

> **Zadatak 3.** Kreirati algebarski tip `Ruka`, koji može predstavljati *papir*, *kamen* ili *makaze*. Kreirati funkciju `uporedi :: Ruka -> Ruka -> Ordering` koja određuje poredak među različitim rukama.
>
> *Rešenje.*
>
> ```
> data Ruka = Papir | Kamen | Makaze
>
> uporedi :: Ruka -> Ruka -> Ordering
> uporedi Kamen  Kamen  = EQ
> uporedi Papir  Papir  = EQ
> uporedi Makaze Makaze = EQ
> uporedi Makaze Papir  = GT
> uporedi Kamen  Makaze = GT
> uporedi Papir  Kamen  = GT
> uporedi _      _      = LT
> ```

[132] Tačnije, svaka od ove dve definicije se može zameniti sa onom drugom u kodu, i ostatak koda neće morati da se promeni.

[133] Ove dve definicije su naravno ekvivalentne u smislu da mogu predstaviti vrednosti koje su u jednoznačnoj korenspodenciji (više o tome u nastavku lekcije). Međutim, ako se jedna od ove dve definicije zameni sa drugom, ta zamena će zahtevati i promenu ostatka koda (na svim mestima u kodu gde su iskorišćeni konstruktori).

Zadatak 4. Kreirati algebarski tip `Dan` čije vrednosti prezentuju dane u nedelji.

Rešenje.

```
data Dan = Ponedeljak
    | Utorak
    | Sreda
    | Četvrtak
    | Petak
    | Subota
    | Nedelja
```

Radi preglednosti zgodno je definiciju prelomiti u više linija. Pri prelomu, nije neophodno da svaki konstruktor bude u posebnoj liniji niti je neophodno da uspravne crte budu jedna ispod druge. Neophodno je samo da definicija nije podeljena na više delova sa praznim linijama.

Zadatak 5. U lekciji o rekurziji pokazan je zadatak sa Hanojskim kulama. U tom zadatku, iskoristili smo tip `Char` da bismo predstavili različite štapove (A, B ili C). Definisati tip `Štap` koji sadrži tri vrednosti, i prepraviti rešenje pomenutog zadataka tako da se umesto tipa `Char` koristi novi tip.

14.3. Možda-tip

Sada ćemo kreirati tip `MoždaBroj` kojim možemo da predstavimo ili jedan `Float` broj ili izostanak bilo kakve smislene vrednosti[134]. Ovaj tip ćemo konstruisati kao sumu tipa `Float` i jediničnog tipa `Ništa`:

```
data MoždaBroj = SamoBroj Float | Ništa
```

Tip `MoždaBroj` je koristan kad god hoćemo da radimo u programu sa nekim vrednostima koje možda nisu ni zadate.

Primer 5. Ako očitavamo temperaturu s nekog senzora, onda je dobro to očitavanje predstaviti `Float` vrednošću. U nekim situacijama naš senzor ne mora vraćati očitanu temperaturu (usled nekih hardverskih problema, itd...), i tada treba koristiti specijalnu vrednost koja označava da do očitavanja temperature nije ni došlo. Nezgodno bi bilo koristiti vrednost 0 (ili neku drugu vrednost) jer se tada ne mogu razlikovati ispravna očitavanja temperature 0°C od neispravnih.

Stoga je u ovom slučaju zgodno koristiti `MoždaBroj` tip za prezentovanje rezultata očitavanja senzora. Vrednost poput `SamoBroj 2` označava da je temperatura zaista 2°C, dok vrednost `Ništa` označava da do očitavanja nije došlo.

Sa `MoždaBroj` vrednostima nije lako raditi kao sa `Float` vrednostima, ali nije ni preteško napisati funkcije koje su nam potrebne:

```
apsolutnaRazlika :: MoždaBroj -> MoždaBroj -> MoždaBroj
apsolutnaRazlika (SamoBroj x) (SamoBroj y) = SamoBroj $ abs $ x - y
apsolutnaRazlika _ _ = Ništa
```

Funkcija će vratiti vrednost oblika `SamoBroj _` i kad god prosledimo dve vrednosti istog oblika. U svim drugim slučajevima, barem jedna od prosleđenih vrednosti će biti `Ništa`, i stoga ima smisla vratiti samo vrednost `Ništa`

[134] Nešto poput vrednosti `null` ili `undefined` u nekim jezicima

```
podeli :: Možda Broj -> Možda Broj -> Možda Broj
podeli (SamoBroj x) (SamoBroj y) =
    if y == 0
        then Ništa
        else SamoBroj $ x / y
podeli _ _ = Ništa
```

Neke funkcije su takve, da i dve "dobre" vrednosti, mogu da daju "lošu" vrednost. Deljenje nulom nije definisano, pa u slučaju kada je `y == 0` vraćamo `Ništa`. Kao i u prethodnoj funkciji, ako je jedan od argumenata `Ništa`, vrednost `Ništa` i vraćamo.

Slično tipu `Možda Broj` možemo konstruisati[135] tip `Možda Niska`:

```
data Možda Niska = SamoNiska String | Ništa
```

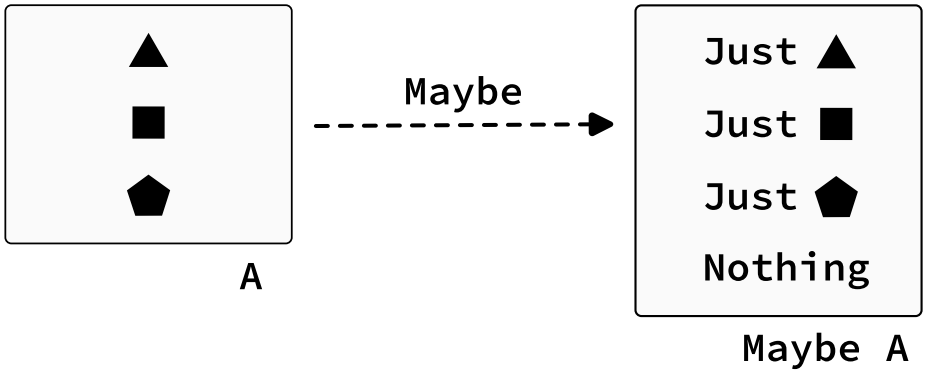
Konstrukcija tipa je `Možda Niska` je u potpunosti analogna konstrukciji tipa `Možda Broj`. Jasno je da je ovakva konstrukcija korisna za bilo koji tip, stoga možemo definisati naredni apstraktni tip

```
data Možda t = Samo t | Ništa
```

Upravo ovako je definisan apstraktni tip `Maybe` koji je dostupan u Prelidu:

```
data Maybe t = Just t | Nothing.
```

Od sada, svaki tip oblika `Maybe a` nazivaćemo **možda-tip**.



Šematski prikaz `Maybe` apstraktnog tipa. Tip `A` sadrži tri vrednosti prezentovane sa tri poligona, dok tip `Maybe A` sadrži četiri vrednosti. Možemo da smatramo da apstraktni tip `Maybe` transformiše tip `A` u `Maybe A`, poput neke funkcije na nivou tipova. Naravno `Maybe` nije funkcija, i zbog toga je na ilustraciji strelica označena sa isprekidanom linijom umesto sa punom linijom. Sa druge strane, konstruktor `Just :: A -> Maybe A` jeste funkcija.

> **Zadatak 6.** Koje vrednosti sadrži tip `Maybe Bool`?
>
> *Rešenje.* `Just True`, `Just False` i `Nothing`.

> **Zadatak 7.** Koje vrednosti sadrži tip `Maybe (Maybe Bool)`?
>
> *Rešenje.* `Just (Just True)`, `Just (Just False)`, `Just Nothing` i `Nothing`

[135] Obratite pažnju da u istom modulu nije moguće definisati tip `Možda Broj` i `Možda Niska` jer imaju istoimeni konstruktor `Ništa`. Naravno, rešenje je da promenimo ime konstruktora u npr `NištaNiska` ili `BezNiske`. U nastavku ćemo radi jednostavnosti zadržati naziv `Ništa`.

14.3.1. Totalne funkcije sa možda tipovima

Možda-tipovi se često koriste da bi se parcijalnim funkcijama pridružile odgovarajuće totalne funkcije. Ideja je da se za one vrednosti domena za koje je funkcija nedefinisana, prosto vrati `Nothing` vrednost.

Primer 6. Funkcija `head :: [a] -> a` koja vraća glavu niza nije totalna: pozivanje `head []` dovodi do izuzetka, jer prazan niz nema glavu. Zbog toga, u jednom od standardnih modula, definisana je funkcija `listToMaybe`:

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

Dakle, ako niz nije prazan `listToMaybe` će vratiti glavu niza upakovanu u `Just`. Ako je niz prazan, `listToMaybe` će vratiti `Nothing`.

Zadatak 8. Definisati totalne varijante funkcija `tail`, `init`, `last` koristeći možda tipove.

14.4. Zapisi

Koliko god proizvod tipova delovao korisno, u nekim situacijama pristupanje elementima proizvoda može dovesti do glomaznog i nečitljivog koda.

Primer 7. Na svakoj elektronskoj ličnoj karti su sačuvane i sledeće niske: ime, prezime, mesto stanovanja, adresa stanovanja, kao i broj koji predstavlja godinu rođenja. Ako formiramo novi tip koji će u sebi sadržati navedene informacije, dolazimo do glomazne definicije:

```
data Osoba = MkOsoba String String String String Int
```

Primetimo da iz ove definicije ne znamo koja koordinata predstavlja koju informaciju. Jedino znamo da koordinata tipa `Int` predstavlja godinu rođenja. Ovu dvosmislenost možemo rešiti pisanjem komentara pored definicije

Možemo se odlučiti da prva niska predstavlja ime, druga prezime, treća mesto a četvrta adresu. Tada funkcije za rad sa tipom `Osoba` deluju poput sledećih:

```
punoIme :: Osoba -> String
punoIme (MkOsoba ime prezime _ _ _)
    = ime ++ " " ++ prezime

punaAdresa :: Osoba -> String
punaAdresa (MkOsoba _ _ mesto adresa _)
    = adresa ++ ", " ++ mesto

punoletna :: Int -> Osoba -> Bool
punoletna trenutnaGodina (MkOsoba _ _ _ _ godinaRođenja)
    = trenutnaGodina - godinaRođenja >= 18
```

Osim što dekonstrukcije podatka daju dugačak kod, programer je u obavezi da pamti šta svaka od koordinata u proizvodu predstavlja. Takođe, veliki problem se javlja kada se definicija tipa promeni nakon što su funkcije napisane. Tada je potrebno izmeniti svako podudaranje oblika u kodu, što može biti zahtevan posao.

Da bi se sprečili problemi opisani u primeru, i olakšao programerima posao, Haskel dozvoljava nešto drugačiju definiciju proizvoda koju nazivamo **zapis**[136]. U definiciji zapisa svakoj od koordinata se dodeljuje ime - **labela**, koje se može kasnije iskoristiti za pristupanje toj koordinati. Niz imena koordinata se navodi u vitičastim zagradama.

Nastavljajući priču iz prošlog primera, možemo definisati tip `Osoba` kao zapis:

```
data Osoba = MkOsoba {
    ime :: String,
    prezime :: String,
    godinaRođenja :: Int,
    mestoStanovanja :: String,
    adresaStanovanja :: String,
}
```

Vrednosti tipa `Osoba` je i dalje moguće konstruisati navođenjem vrednosti nakon konstruktora, ali je moguće i iskoristiti sintaksu za zapise:

```
osoba1 = MkOsoba "Petar" "Petrović" 2000 "Beograd" "Njegoševa"

osoba2 = MkOsoba {
    ime = "Jovana",
    prezime = "Jovanović",
    godinaRođenja = 1990,
    mestoStanovanja = "Novi Sad",
    adresaStanovanja = "Zmaj Jovina",
}
```

Korišćenjem sintakse za zapise, programer nije u obavezi da ispoštuje redosled koordinata, ali mora navesti vrednosti za svaku koordinatu.

Definisanjem tipa `Osoba` sintaksom za zapise, zapravo je kreirano pet funkcija `ime :: Osoba -> String`, `prezime :: Osoba -> String`, `godinaRođenja :: Osoba -> Int`, `mestoStanovanja :: Osoba -> String`, `adresaStanovanja :: Osoba -> String`. Ove funkcije se nazivaju **projekcije**, i vraćaju odgovarajuće koordinate:

```
ghci> :t ime
ime :: Osoba -> String
ghci> ime osoba1
"Petar"
ghci> ime osoba2
"Jovana"
```

Prema tome, labele zapisa se koriste poput getera u objektno orijentisanim jezicima[137]. Što se tiče promene vrednosti koordinate (odnosno setera), to se može izvršiti navođenjem labele sa novom vrednošću koordinate:

[136] eng. *record*

[137] Na primer, kôd `ime osoba2` odgovara nečemu što bi se Javi/C++-u napisalo kao `osoba2.ime`.

```
ghci> ime osoba1
"Petar"
ghci> osoba3 = osoba1{ime = "Miloš"}
ghci> ime osoba3
"Miloš"
```

Vrednosti koordinata vrednosti `osoba3` su iste kao odgovarajuće vrednosti koordinata `osoba1`, osim vrednosti polja `ime` koja je `"Miloš"`.

Pri podudaranju oblika, moguće je "uhvatiti" samo labele koje su nam potrebne. Tako bi se funkcije od malopre mogla ovako napisati:

```
punoIme :: Osoba -> String
punoIme (MkOsoba {ime = ime, prezime = prezime})
    = ime ++ " " ++ prezime

punaAdresa :: Osoba -> String
punaAdresa (MkOsoba {mestoStanovanja = mesto, adresaStanovanja = adresa})
    = adresa ++ ", " ++ mesto

punoletna :: Int -> Osoba -> Bool
punoletna trenutnaGodina (MkOsoba {godinaRođenja = godinaRođenja})
    = trenutnaGodina - godinaRođenja >= 18
```

14.5. Algebra tipova

U matematici, **algebra**[138] označava disciplinu koja proučava zakonitosti koje važe u formalnim sistemima, pri čemu se te zakonitosti proučavaju i iskazuju kroz *algebarske izraze*. Na primer, činjenica da je

$$(5 + 2)(5 - 2) = 25 - 4$$

spada u domen *aritmetike*, dok iskaz

$$(m + n)(m - n) = m^2 - n^2$$

spada u domen algebre prirodnih brojeva. Ovaj iskaz je tačan za sve prirodne brojeve m i n, i prethodno navedena činjenica je samo jedna instanca ovog algebarskog zakona. Međutim, algebra je mnogo šira od algebre brojeva. Algebra proučava i apstraktne zakonitosti koje važe sa operacijama nad vektorima, skupovima, funkcijama...

Na primer, algebarski izraz

$$(a \circledast b) \circledast c = (c \circledast b) \circledast a$$

važi za sve prirodne brojeve a, b i c ako operator $\circledast$ predstavlja sabiranje. Takođe važi i za sve dvodimenzionalne vektore a, b i c ako operator $\circledast$ predstavlja sabiranje vektora. Međutim, važi i za sve rotacije a, b i c oko neke ose p u prostoru, ako $\circledast$ predstavlja *nadovezivanje* (odnosno kompoziciju) rotacija. Slično i za skupove i za operaciju unije skupova... U svim navedenim slučajevima, ovaj algebarski zakon ima gotovo isti dokaz i zasniva se na osobinama komutativnosti i asocijativnosti odgovarajuće operacije.

14.5.1. Jednakost tipova

Algebarski zakoni se gotovo uvek navode u obliku jednakosti. Pošto želimo da proučimo algebru tipova, neophodno je prvo razumeti jednakost između tipova.

[138] Reč algebra potiče od arapskog izraza الجبر (*al-džebra*) koji se koristio za tehniku nameštanja kostiju. Arapski matematičar Al Hurezmi iz IX veka je iskoristio ovaj termin za jednu tehniku rešavanja jednačina, i termin je sa prevodom njegovog priručnika dospeo u Evropu. Interesantno, termin *algoritam* je dobijen od samog Al Hurezmijevog imena.

Sistem tipova u nekom programskim jeziku često spada u kategoriju *strukturalnih* ili *nominativnih* tipskih sistema. U **strukturalnom tipskom sistemu**, dva tipa su jednaka ako je njihova struktura ista (šta god to značilo za taj tipski sistem). Nasuprot tome, u **nominativnom (nominalnom) tipskom sistemu** dva tipa su jednaka ako su definisana istom definicijom, tj. ako su istog imena. Da bismo shvatili šta ove definicije znače, pogledaćemo zašto sistem tipova u Haskelu zapravo nije ni strukturalan niti nominativan.

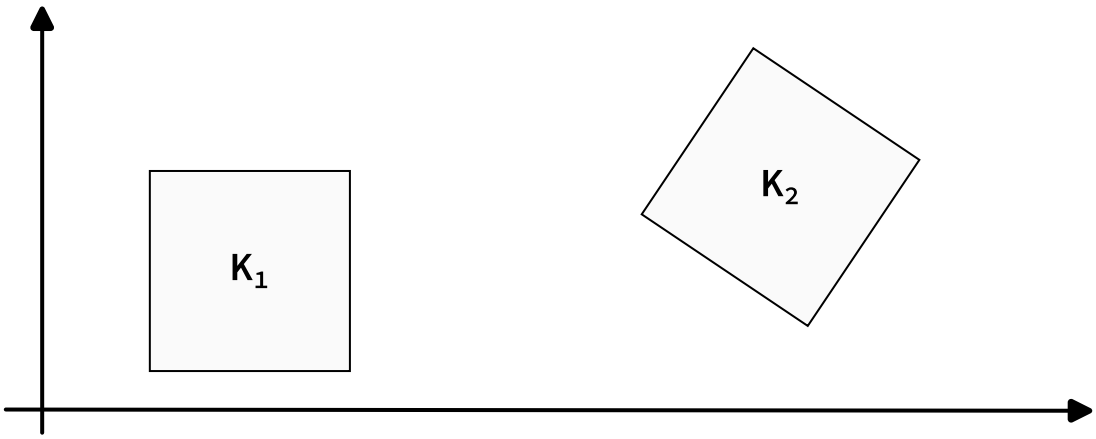
Neka su definisani tipovi `data A = A Int` i `data B = B Int`. Za ove tipove bismo rekli da imaju potpuno istu strukturu, vrednosti oba tipa su "sačinjena" od jedne celobrojne vrednosti. Ako bi sistem tipova Haskela bio strukturalan, tada ova dva tipa mogla slobodno da se "mešaju", pa bi tako na primer funkcija tipa `A -> Bool` mogla biti primenjena na vrednost tipa `B`. Međutim, lako se možemo uveriti da će takva primena dovesti do tipske greške. Prema tome, sistem tipova Haskela nije strukturalan.

Sistem tipova Haskela nije ni nominativan, jer tipski sinonimi definisani sa `type A = B`, dozvoljavaju da se tipovi `A` i `B` mešaju. Na primer, na svakom mestu gde se može koristiti vrednost tipa `[Char]`, može se iskoristiti vrednost tipa `String`. Prema tome, tipovi različitog imena mogu biti jednaki.

Tačnu definiciju jednakosti tipova ovde ne možemo dati, ali smo kroz prethode redove oslikali neke osobine jednakosti. Sam sistem bismo mogli da okarakterišemo kao nominalan modulo tipski sinonimi. Mi kao korisnici kompajlera i interpretera, možemo smatrati dva tipa u Haskelu jednakim ako ih možemo proizvoljno mešati (koristiti jedan umesto drugog ne uvodeći tipsku greške u kompilaciju). Kako sam kompajler proverava jednakost, ostavljamo za neka kasnija razmatranja.

14.5.2. Izomorfizam skupova

Jednakost tipova koju smo analizirali gore je previše stroga: videli smo da (ako ignorišemo tipske sinonime), tipovi su jednaki samo samim sebi[139]. Zbog toga želimo da uvedemo slabiji pojam jednakosti, takozvani *izomorfizam*, koji će nam omogućiti da lakše rezonujemo o nekim odnosima između tipova. Definiciju nove relacije uzećemo iz teorije skupova.



Dva kvadrata K_1 i K_2 nisu jednaka, ove dve figure se nalaze se na različitim mestima, i nemaju zajedničkih tačaka. Međutim, ovi kvadrati imaju iste dimenzije, i pomeranjem jednog možemo "savršeno preklopiti" drugi. Zbog toga, za ove kvadrate kažemo da su *kongruentni* i to obeležavamo $K_1 \cong K_2$. Mnogobrojne osobine koje proučavamo u geometriji (površina, obim, dužina dijagonale....) su potpuno iste za ova dva kvadrata, i za proučavanje ovih osobina nas ne interesuje sam položaj figura u prostoru. Uvođenjem pojma kongruentnosti, rasuđujemo o figurama na apstraktnijem nivou, zanemarujući osobine koje nam nisu interesantne.

Ideja je da skupove sa istim brojem elemenata poistovetimo. Ova ideja bi bila sasvim dobra, ako bismo radili samo sa konačnim skupovima. U matematici mnogi skupovi nisu konačni (kao u ostalom i mnogi tipovi u Haskelu), te je stoga potrebno osmisliti definiciju koja se ne oslanja na pojam broja.

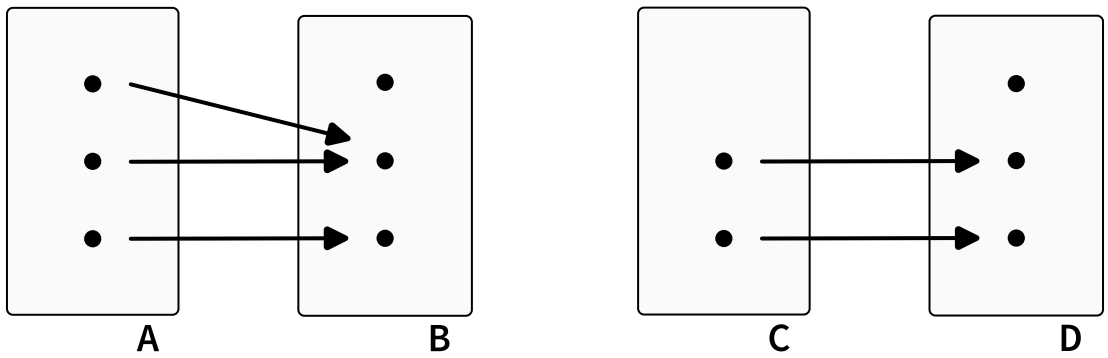
[139] To bi bilo kao da u algebri prirodnih brojeva kažemo da $m + n$ i $n + m$ nisu jednaki jer nisu isti kao nizovi simbola.

Primetimo da dva konačna skupova imaju isti broj elemenata ako se ti elementi mogu "povezati" tako da je svaki element iz jednog skupa povezan sa tačno jednim elementom iz drugog skupa. Ako ovakvo povezivanje formalizujemo kroz pojam funkcije, tada ćemo taj formalizam moći da koristimo i sa beskonačnim skupovima. Navedena razmišljanja su formalizovana kroz narednu definiciju.

Za skupove A i B kažemo da su **izomorfni**, i pišemo $A \cong B$ ako postoji **bijekcija**[140] između ova dva tipa, odnosno ako postoji funkcija $f : A \to B$ sa sledećim osobinama:

1. za svaki element $b \in B$ postoji element $a \in A$ takav da je $f(a) = b$ (*osobina surjektivnosti*)
2. ako je $f(a_1) = f(a_2)$ za $a_1, a_2 \in A$, tada mora važiti $a_1 = a_2$ (*osobina injektivnosti*)

Ako bi ove uslove preneli na grafičko prezentovanje funkcija koje smo do sada videli, gde vrednosti tipa predstavljamo kao tačke, a funkciju kao skup strelica između tih tački, tada nam uslov surjektivnosti funkcije $f : A \to B$ govori da je svaka tačka iz B povezana sa nekom tačkom iz A. Uslov injektivnosti nam govori da ne postoje dve tačke iz A koje su povezane sa istom tačkom iz B.



Na ilustraciji vidimo dva primera. Levo je prikazana funkcija $f : A \to B$, koja jeste surjektivna ali nije injektivna. Primetimo da skupovi A i B jesu izomorfni, ali f to ne dokazuje. Desno je prikazana funkcija $g : C \to D$ koja jeste injektivna, ali nije surjektivna.

Primer 8. Identička funkcija na skupu A, tj. funkcija $\mathrm{id}_A : A \to A$ definisana sa $\mathrm{id}_A(x) = x$, je bijekcija. Zaista, za svaki element $a \in A$ važi da je $a = \mathrm{id}_A(a)$ pa je stoga id_A surjektivna funkcija. Dalje, ako je $\mathrm{id}_A(a_1) = \mathrm{id}_A(a_2)$ tada je

$$a_1 = \mathrm{id}_A(a_1) = \mathrm{id}_A(a_2) = a_2,$$

pa je stoga id_A i injektivna funkcija.

Prema tome, za svaki tip A važi $A \cong A$.

Primer 9. Neka je U jednočlan skup. Tada za svaki skup postoji jedinstvena funkcija $u_A : A \to U$.

Ako je skup A prazan, tada u_A nije surjekcija, a ako skup A ima barem dva elementa tada u_A nije injekcija. Funkcija u_A je bijekcija ako i samo ako je A isto jednočlan skup.

Ako je $f : A \to B$ bijekcija, tada za svaki element $b \in B$ postoji element $a \in A$ takav da je $f(a) = b$ (zbog surjektivnosti). Štaviše takav, element mora biti jedinstven zbog injektivnosti. Stoga, možemo definisati funkciju tipa $B \to A$ koja svakom elementu $b \in B$ dodeljuje element $a \in A$ takav da važi $b = f(a)$. Ovakvu funkciju nazivamo **inverzna funkcija** funkcije f i obeležavamo sa f^{-1}. Dakle, ako je $f(a) = b$ tada je $f^{-1}(b) = a$. Kombinujući ove dve jednakosti dobijamo da je $f(f^{-1}(b)) = b$ i $f^{-1}(f(a)) = a$ za sve elemente $b \in B$ i sve $a \in A$. Ove jednakosti se mogu zapisati i kao

[140] Bijekcije se nazivaju i *uzajamno jednoznačne korespodencije.*

$$f^{-1} \circ f = \mathrm{id}_A \quad \text{i} \quad f \circ f^{-1} = \mathrm{id}_B,$$

gde su id_A i id_B identičke funkcije na skupovima A i B respektivno.

Važi i obrnuto: ako postoje funkcije $f : A \to B$ i $g : B \to A$ takve da je $g \circ f = \mathrm{id}_A$ i $f \circ g = \mathrm{id}_B$, onda su ove funkcije bijekcije i jedna drugoj inverzne. Funkcija f je surjektivna funkcija, jer za proizvoljno $b \in B$, važi da $f(g(b)) = \mathrm{id}_B(b) = b$, što znači da f slika $g(b)$ u b. Funkcija f je injektivna, jer ako je $f(a_1) = f(a_2)$ tada je i $g(f(a_1)) = g(f(a_2))$ pa je i $a_1 = a_2$ zbog $g \circ f = \mathrm{id}_A$. Analogno se dokazuje i da je g bijekcija.

Navedenim razmatranjem smo dokazali da postoji bijekcija između A i B ako i samo ako postoje funkcije $f : A \to B$ i $g : B \to A$ takve da je $f \circ g = \mathrm{id}_B$ i $g \circ f = \mathrm{id}_A$. Uslov izomorfizma skupova smo mogli da definišemo koristeći i ovu karakterizaciju, ali smo ovako (dužim putem) videli smisao jednakosti.

Koristeći prethodnu karakterizaciju izomorfnih skupova, možemo lako da dokažemo još neke osobine relacije izomorfnosti. Ako je $A \cong B$ to znači da postoje funkcije $f : A \to B$ i $g : B \to A$ takve da je $f \circ g = \mathrm{id}_B$ i $g \circ f = \mathrm{id}_A$, a samim tim to znači da je $B \cong A$. Takođe, ako je $A \cong B$ i $B \cong C$, sledi da postoje funkcije $f_1 : A \to B$ i $g_1 : B \to A$ takve da je $f_1 \circ g_1 = \mathrm{id}_B$ i $g_1 \circ f_1 = \mathrm{id}_A$ i funkcije $f_2 : B \to C$ i $g_2 : C \to B$ takve da je $f_2 \circ g_2 = \mathrm{id}_C$ i $g_2 \circ f_2 = \mathrm{id}_B$. Dalje, važi da je[141]

$$(g_1 \circ g_2) \circ (f_2 \circ f_1) = g_1 \circ \mathrm{id}_B \circ f_1 = \mathrm{id}_A$$

i

$$(f_2 \circ f_1) \circ (g_1 \circ g_2) = f_2 \circ \mathrm{id}_A \circ g_2 = \mathrm{id}_C,$$

pa je $A \cong C$, jer funkcije $f_2 \circ f_1 : A \to C$ i $g_1 \circ g_2 : C \to A$ imaju neophodne osobine.

Dakle relacija $\cong$ poseduje svojstva kakva očekujemo od jednakosti. Svaki skup je izomorfan samom sebi. Ako je A izomorfan sa B, tada je i B izomorfan sa A. I, ako je A izomorfan sa B i B izomorfan sa C, tada je i A izomorfan sa C. Naravno, ovo je sasvim očekivano, jer smo relaciju $\cong$ definisali tako da za konačne skupove važi $A \cong B$ ako A i B imaju isti broj elemenata.

Motivisani skupovima, uvodimo pojam izomorfizma Haskel tipova. Za tipove `A` i `B` kažemo da su izomorfni, ako postoje funkcije `f :: A -> B` i `g :: B -> A` takve da važi `g . f = id` i `f . g = id`. Ako su tipovi `A` i `B` izomorfni, to označavamo sa `A ≅ B`.

Kao i slučaju konačnih skupova, dva tipa sa konačno mnogo vrednosti su izomorfna ako imaju isti broj elemenata.

> **Primer 10.** Tipovi `Bool` i `Maybe ()` su izomorfni. Da bismo to dokazali dovoljno je da konstruišemo par funkcija, čije obe kompozicije su identičke funkcije. Uzmimo `f :: Bool -> Maybe ()` sa
>
> ```
> f x = if x then Just () else Nothing
> ```
>
> i `g :: Maybe () -> Bool` sa
>
> ```
> g x = x /= Nothing.
> ```
>
> Lako se proverava da jednakosti `g (f x) = x` i `f (g y) = y` važe.

14.5.3. Algebarski zakoni

U narednim redovima ćemo izneti neke zakone algebre prirodnih brojeva, videti koji je pandan tim zakonima u algebri tipova, a zatim i dokazati iznete zakone. Da bi skratili naredne redove, i napravili

[141] Ovde koristimo dve činjenice. Prvo, kompozicija funkcija je asocijativna operacija te se zagrade mogu postavljati proizvoljno u izrazu sačinjenom od kompozicija. Drugo, identičke funkcije su neutrali za operaciju kompozicije te se mogu proizvoljno uklanjati iz izraza.

razliku između algebre brojeva i algebre tipova, uvešćemo skraćeni zapis za sumu i proizvod tipova. Sa `A ⊕ B` ćemo označiti sumu tipova `A` i `B`, a sa `A ⊗ B` proizvod tipova.

Za sabiranje prirodnih brojeva važi zakon *komutativnosti*: $m + n = n + m$ za sve prirodne brojeve m i n. Ako ovaj zakon "mehanički" prevedemo na algebru tipova dobijamo

`A ⊕ B ≅ B ⊕ A`.

Ovaj metaizraz[142] nam govori da je tip definisan sa kodom `S1 A | S2 B` izomorfan sa tipom definisanim sa `S1 B | S2 A`[143]. Da bi dokazali ovaj izomorfizam, definišimo narednu funkciju (jedna će biti dovoljna):

```
data S t1 t2 = S1 t1 | S2 t2

f :: S A B -> S B A
f (S1 a) = S2 a
f (S2 b) = S1 b

g :: S B A -> S A B
g (S1 b) = S2 b
g (S2 a) = S1 a
```

Dokažimo da važi `g . f = id :: S A B` Svaka vrednost `x :: S A B` je oblika `S1 a` ili `S2 b`, gde `a :: A` i `b :: B`. Ako je `x = S1 a`, tada je `f x = S2 a`, pa je `g (f x) = g (S2 a) = S1 a = x`. Ako je `x = S2 b` tada je `g (f x) = g (S1 b) = S2 b = x`. Dakle, važi `g (f x) = x` za svako `x :: S A B`, tj.

`g . f = id :: S A B -> S A B`.

Na potpuno isti način[144] se dokazuje i jednakost `f . g = id :: S B A -> S B A` odakle zaključujemo da je `S A B ≅ S B A`.

Sličnim rezonovanjem se dokazuje i zakon komutativnosti proizvoda

`A ⊗ B ≅ B ⊗ A`.

Određeni brojevi imaju posebna svojstva pri aritmetičkim operacijama. Na primer, broj 0 ima osobinu da važi

$$0 + n = n \quad \text{i} \quad 0 \times n = 0$$

za svaki prirodan broj n. Intuitivno deluje da je pandan broju 0 tip koji ne sadrži vrednosti, odnosno `Void`. Direktnim prevođenjem ovih jednakosti, dobijamo izomorfizme `Void ⊕ A ≅ A` i `Void ⊗ A ≅ Void` koje je potrebno dokazati.

Sledeći par funkcija dokazuju da je `Void ⊕ A ≅ A`:

[142] U ovoj sekciji u Haskel kodu će pojavljivati simboli ⊕ i ⊗. Njih samo koristimo da bi imenovali nove tipove, tj. da bi samo ime, poput `A ⊕ B`, ukazivalo kojom konstrukcijom je dobijen taj tip. Ovi simboli ne mogu se koristiti u Haskel kodu koji se kompajlira.

[143] Imena konstruktora smo proizvoljno uzeli.

[144] Pažljiv čitalac će primetiti da su `f` i `g` samo tipski konkretne instance parametarski polimorfne funkcije `h :: S x y -> S y x`. Zapravo, čitav ovaj dokaza je mogao da se svede na pokazivanje da važi `h . h = id`.

```
data S t1 t2 = S1 t1 | S2 t2

f :: S Void A -> A
f (S2 a) = a

g :: A -> S Void A
g a = S2 a
```

Možda deluje kao da funkcija `f` nije totalna, ali ona je definisana za sve vrednosti iz `S Void A`. Kako ne postoji vrednost tipa `Void`, sledi da ne postoji ni vrednost oblika `S1 x :: S Void A`.

Provera izomorfizma je jednostavna: svako `x :: S Void A` je oblika `S2 a` za neko `a :: A`. Sledi da je

$$\texttt{g (f x) = g (f (S2 a)) = g a = S2 a = x}$$

i

$$\texttt{f (g a) = f (S2 a) = a}.$$

Za dokazivanje `Void ⊗ A ≅ Void` iskoristićemo naredne funkcije:

```
data P t1 t2 = P t1 t2

f :: P Void A -> Void
f (P v _) = v

g :: Void -> P Void A
g v = case v of {}
```

Umesto što smo definisali `P`, mogli smo da iskoristimo tip uređenih parova `(,)`.

U ovom slučaju dokaz je jednostavan[145]: ne postoji vrednost `x :: Void` te jednakost važi `f (g x) = x` za sve `x :: Void`. Isto, ne postoji vrednost tipa `P Void A` pa i jednakost `g (f x) = x` važi uvek za `x :: P Void A`.

Dakle tip `Void` se ponaša u algebri tipova onako kako se ponaša broj 0 u algebri prirodnih brojeva. Broj 1 ima osobinu da je $1 \times n = n$ za sve prirodne brojeve n. Pandan ovom zakonu je

$$\texttt{() ⊗ A ≅ A},$$

što se dokazuje korišćenjem funkcija:

```
data P t1 t2 = P t1 t2

f :: P () A -> A
f (P _ a) = a

g :: A -> P () A
g a = P () a
```

Dokaz za `f . g = id` i `g . f = id` nećemo navoditi, jer nije ništa komplikovaniji od prethodnih. Istaknimo da smo ovde umesto `()` mogli da iskoristimo bilo koji tip koji ima jednu vrednost. Štaviše, možemo dokazati nešto opštije tvrđenje: ako je $\texttt{A}_1 \cong \texttt{A}_2$ i $\texttt{B}_1 \cong \texttt{B}_2$, tada je i

$$\texttt{A}_1 \oplus \texttt{B}_1 \cong \texttt{A}_2 \oplus \texttt{B}_2$$

i

[145] Dokaz se zasniva na tome da je iskaz $p \Rightarrow q$ tačan kad p nije tačno (*vacuous truth*). Iz iskaza *x je tipa Void*, možemo zaključiti bilo šta jer ne postoji vrednost tipa `Void`.

$A_1 \otimes B_1 \cong A_2 \otimes B_2$.

14.6. Zadaci

Zadatak 9. Kreirati algebarski tip podataka `Vektor` koji predstavlja vektor u trodimenzionalnom prostoru (Koristiti `Float` tip za koordinate). Kreirati funkcije za sabiranje vektora, množenje vektora skalarem, skalarnog množenja vektora, vektorskog množenja vektora i računanja dužine vektora.

Zadatak 10. Kreirati algebarski tip `Valuta` koji može da predstavi neke valute (npr, RSD, EUR, USD) i funkcije koje vrše konverziju između ovih valuta. Kreirati algebarski tip `Svota` koji sadrži jednu vrednost tipa `Float` i vrednost tipa `Valuta`. Kreirati algebarski tip `Banka` koji predstavlja nekoliko domaćih banaka. Kreirati tip `Račun` koji sadrži vrednost tipa `Int` (broj računa) i vrednost tipa `Banka` (predstavlja banku u kojoj je otvoren račun). Tip `Račun` se koristi samo za identifikaciju, i nema u sebi podatke o promeni računa. Kreirati algebarski tip `Transakcija` koji sadrži vrednost tipa `Svota`, a zatim dve vrednosti tipa `Račun` koje predstavljaju račun sa kog se šalje odnosno na koji se šalje novac. Kreirati jednu proizvoljnu listu `Transakcija` od 5 članova ili više. Kreirati funkciju `promenaNaRačunu :: Račun -> [Transakcija] -> Svota` koja računa promenu stanja računa nakon svih izvršenih transakcija. Pretpostaviti da račun može da prime svote novca u bilo kojoj valuti. Krajnju promenu iskazati u evrima.

Zadatak 11. Kreirati tip `Tačka` koji predstavlja tačku u dvodimenzionalnoj ravni. Kreirati tip `Figura` koji može da predstavi pravougaonik, krug, trougao, i proizvoljni pravilni mnogougao u ravni. Kreirati funkcije `površina :: Figura -> Float` i `obim :: Figura -> Float` koje redom računaju površinu i obim figura. Kreirati funkciju `transliraj :: (Float, Float) -> Figura -> Figura` koja translira (pomera) figuru. Kreirati funkciju `skaliraj :: Float -> Figura -> Figura` koja skalira figuru za uneti faktor.

Zadatak 12. Kreirati tip `KompleksanBroj`. Implementirati funkcije koje sabiraju, oduzimaju, množe i dele kompleksne brojeve.

Zadatak 13. Kreirati tip `Matrica` koji predstavlja matricu dimenzija 3×3. Kreirati funkcije koje sabiraju matrice, skaliraju matricu, i računaju determinantu matrice. Kreirati tip `Vektor` koji predstavlja vektor trodimenzionalno prostora. Kreirati funkciju koja množi matricu i vektor.

15. Još o sintaksi

Postoji još par pojedinosti Haskel sintakse koje nismo do sada obradili, a koje ćemo u nastavku prezentovati.

15.1. Podudaranje oblika

Nakon što smo se upoznali sa algebarskim tipovima, možemo dati nešto precizniji opis mehanizma podudaranja oblika. Podudaranje oblika smo koristili pri definisanju funkcija. U opštem slučaju, ovakve definicije su sačinjene od više linija koje predstavljaju primenu funkcije na *oblike*[146]. Svaka linija ima formu *jednakosti*

```
f x₁ x₂ … xₘ = y
```

gde su `x₁`, `x₂` itd. neki oblici. Oblik može biti nešto od sledećeg:

1. Literal broja, odnosno izraz poput `2`, `3.14`, `0x3A`, itd...
2. Literal karaktera, odnosno izraz poput `'a'`, `'B'`, `'ш'`, `'貓'` itd...
3. Literal niske, odnosno izraz poput `"hello"`, `"world!!"`, itd...
4. Proizvoljno ime, odnosno izraz poput `x`, `ys`, `n`, itd...
5. Džoker `_`
6. Literal uređene n-torka oblika, odnosno izraz poput `(1, 'a')`, `(x, y, z)`, `(_,1,_,x)` itd...
7. Literal liste oblika, odnosno izraz poput `[]`, `[5]`, `[x,y,_]`, itd...
8. Konstruktor primenjen na oblike ili zapise, odnosno izraz poput `True`, `Just x`, `Vektor x y _` itd...

Svaki oblik koji možemo konstruisati samo kombinovanjem prethodno navedenih oblika. Primetimo da oblici n-torke, liste i algebarskog tipa podatka, dozvoljavaju da se konstruišu oblici proizvoljne složenosti[147].

Numerički literal, karakter ili niska u obliku će se podudariti sa argumentom funkcije, ako i samo ako je taj argument jednak[148] tom literalu. Oblik sačinjen od imena, podudariće se sa svakim argumentom, i to ime će imati vrednost argumenta sa desne strane jednakosti. Jedno ime može se koristiti samo na jednom mestu sa leve strane jednakosti[149]. Džoker `_` takođe se podudara sa svakom vrednošću argumenta, ali za razliku od imena džoker se može pojavljivati na više mesta sa leve strane jednakosti. Vrednost argumenata koji se podudario sa džokerom, ne može se iskoristiti sa desne strane jednakosti.

Oblici sa listama, n-torkama, i konstruktorima uglavnom u sebi sadrže neke druge *podoblike* (npr. literale). Ovi oblici će se podudariti sa argumentom funkcije ako i samo ako se svi podoblici podudare sa odgovarajućim vrednostima.

Primetimo da se liste mogu podudariti na dva načina: ili preko literala liste (npr `[1,2,3]`) ili korišćenjem konstruktora liste (npr: `x:xs`). Nekad se ova dva pristupa podudaranju lista koriste istovremeno (npr. u obliku `x:[_]`). Ovo takođe važi i za uređene `n`-torke koji se mogu podudariti korišćenjem konstruktora poput `(,)`, `(,,,)`. itd. ali se ovakav pristup retko koristi u praksi.

15.1.1. *As* sintaksa

Konstruktori se mogu koristiti na dva načina: kao funkcije pomoću kojih se konstruišu vrednosti nekog tipa (otuda i samo ime), ali i za *dekonstrukciju* vrednosti podudaranjem oblika. Neretko se dešava da u definiciji funkcije neki argument dekonstruišemo da bi u telu funkcije istu tu vrednost konstruisali.

> **Primer 1.** Neka je dat tip koji predstavlja petodimenzionalni vektor:

[146] eng. *pattern*

[147] Na primer, `Just x`, ali `Just (Just x)`, i `Just (Just (Just x))`, itd...

[148] U smislu operatora `==`

[149] Npr. ne možemo sa oblikom `[x, x]` podudariti dvočlane liste sa istim prvim i drugim argumentom.

```
data Vektor = Vektor Float Float Float Float Float
```

Neka su date i dve funkcije `g1, g2 :: Vektor -> Float`. Zamislimo da je potrebno napisati funkciju `f :: Vektor -> Float` koja za dati vektor `v` vraća vrednost `g1 v` ako je prva koordinata vektora `v` pozitivna, a u suprotnom `g2 v`. Jedna implementacija tražene funkcije bi mogla da glasi ovako:

```
f :: Vektor -> Float
f (Vektor x y z w q) =
    if x >= 0
        then g1 (Vektor x y z w q)
        else g2 (Vektor x y z w q)
```

Vidimo da u kodu nepotrebno uvodimo promenljive `y`, `z`, `w`, `q`. Vrednosti koje predstavljaju ove promenljive nisu bitne za samu funkciju `f`. Situacija bi nešto bolja bila da smo iskoristili `let in` ili `where` konstrukciju, ali i osnovni problem bi i dalje ostao: istu vrednost dekonstruišemo pa konstruišemo.

As sintaksa nam omogućuje da isti argument istovremeno vežemo za jedno ime ali i da iskoristimo podudaranje oblika. Sintaksa ima oblik

`ime@oblik`,

pri čemu se `ime` ne nalazi u obliku. Na ovaj način vrednost čitavog argumenta će biti dostupna kao `ime` sa desne strane jednakosti.

Primer 2. Koristeći *as* sintaksu za prethodni primer, funkciju `f` možemo definisati kao

```
f :: Vektor -> Float
f v@(Vektor x y z w q) =
    if x >= 0
        then g1 v
        else g2 v
```

As sintaksom smo istovremeno izvršili podudaranje oblika, u ovom slučaju to podudaranje oblika dekonstruiše vektor na koordinate, ali smo i imenu `v` dodelili vrednost čitavog vektora.

Zapravo, kako nas vrednosti `y`, `z`, `w`, `q` uopšte ne interesuju, možemo ih zameniti džokerima:

```
f :: Vektor -> Float
f v@(Vektor x _ _ _ _) =
    if x >= 0
        then g1 v
        else g2 v
```

15.2. *Case* notacija

Podudaranje oblika je do sada bilo isključivo vezano za definicije funkcija: da bismo iskoristili podudaranje oblika, morali smo da definišemo novu funkciju[150]. *Case* konstrukcija nam omogućava podudaranje oblika unutar *izraza*, bez potrebe za definicijom funkcije.

Case izraz započinje sa kodom `case x of` gde je `x` vrednost čiji oblik želimo da ispitamo. Nakon ovog početka, potrebno je navesti blok sačinjen od oblika i rezultata razdvojenih strelicom `->`.

[150] Osim u slučaju podudaranja oblika u parametru lambda funkcije, koji se može iskoristiti samo za dekonstrukciju jednog oblika.

```
dužina :: [a] -> Int
dužina []     = 0
dužina (x:xs) = 1 + dužina xs

dužina' :: [a] -> Int
dužina' = (\ss -> case ss of
    []    -> 0
    x:xs  -> 1 + dužina' xs
)

dužina'' :: [a] -> Int
dužina'' ss = case ss of
    []    -> 0
    x:xs  -> 1 + dužina'' xs
```

Tri definicije funkcije koja računa dužinu liste. Prva definicija nam je dobro poznata. U drugoj definiciji je iskorišćen je `case` izraz unutar lambda izraza. Primetimo da je ovako napisan lambda izraz mogao da se navede bilo gde u kodu, i da nije bilo potrebe da se dodeljuje imenu `dužina`. Treća definicija je mešavina prethodne dve: parametar `ss` je vezan tehnikom podudaranja oblika (pri čemu je naveden samo jedan, opšti, oblik), a u telu funkcije je iskorišćena *case* notacija. Primetimo kako u *case* izrazu oblik `x:xs` nismo morali da obuhvatimo sa zagradama kao što je to slučaj u prvoj definiciji.

Za blok oblika koji sledi nakon koda `case x of` važe iste napomene kao i za blokove poput `where` ili `let in` blokova. Blok se može napisati i u jednoj liniji razdvajanjem slučajeva sa znakom `;`. Ako se blok prelomi u više linija, tada počeci svih linija moraju biti poravnati po vertikali.

> **Primer 3.** Iskoristimo primer iz prethodne glave za demonstraciju različitog preloma *case* izraza. Sve naredne definicije definišu istu funkciju
>
> ```
> data Dužina = Metar Float | Milja Float | SvetlosnaSekunda Float
>     deriving (Show)
>
> uMetre' :: Dužina -> Dužina
> uMetre' l = case l of
>     Milja m -> Metar (1609.344 * m)
>     SvetlosnaSekunda s -> Metar (299792458 * s)
>     x -> x
>
> uMetre'' :: Dužina -> Dužina
> uMetre'' l = case l of Milja m -> Metar (1609.344 * m)
>                        SvetlosnaSekunda s -> Metar (299792458 * s)
>                        x -> x
>
> uMetre''' :: Dužina -> Dužina
> uMetre''' l = case l of Milja m -> Metar (1609.344 * m); SvetlosnaSekunda s ->
> Metar (299792458 * s); x -> x
> ```

Možda isprva ne deluje kao mnogo korisna, ali *case* notacija može skratiti kôd.

> **Zadatak 1.** Napisati funkciju `moždaZbir :: Maybe [Maybe Int] -> Int` koja sabira sve vrednosti različite od `Nothing` u datom možda-nizu možda-brojeva. Iskoristiti *case* notaciju

15.3. Izlistavanje

U matematici, često se koristi zapis

$$\{x \in A \mid \Phi(x)\}$$

za definisanje skupa svih elemenata x iz skupa A takvih da važi neko svojstvo Φ. Na primer, $\left\{x \in \mathbb{Z} \mid x \underset{2}{\equiv} 0\right\}$ je skup svih celih brojeva deljivih sa 2 (odnosno skup svih parnih brojeva), a skup $\{x \in \mathbb{C} \mid x^3 = 1\}$ predstavlja skup svih rešenja jednačine $x^3 - 1 = 0$, itd... Ovakva notacija je podložna promenama, pa se umesto opisanog oblika koristi i

$$\{x \mid x \in A, \Phi(x)\}.$$

Takođe, često se koristi i zapis

$$\{f(x) \mid x \in A, \Phi(x)\}$$

koji označava skup slika pri funkciji f svih elemenata iz A koji poseduju svojstvo Φ. Na primer, $\left\{x^2 \mid x \in \mathbb{Z}, x \underset{2}{\equiv} 0\right\}$ je skup kvadrata svih parnih brojeva.

Haskel je preuzeo ovu korisnu matematičku sintaksu za *izlistavanje lista*[151]. Izlistavanje se navodi između uglastih zagrada (kako je to i slučaj sa listama u Haskelu), a koristi se i uspravna crta kao u matematičkom zapisu. Sa desne strane uspravne crte navodi se lista iz koje se neka promenljiva *izdvaja*, kao i niz predikata[152] koje vrednost promenljive mora da zadovolji. Na primer, ako je definisana lista

```
xs = [-3, -2, -1, 0, 1, 2, 3, 4] :: [Int],
```

tada se izraz[153]

```
[x | x <- xs, even x]
```

evaluira u listu `[-2, 0, 2, 4]`, a izraz

```
[x^2 | x <- xs, even x]
```

se evaluira u listu `[4, 0, 4, 16]`.

Kao što vidimo iz poslednjeg primera, pri izlistavanju Haskel lista je bitan poredak i ponavljanje elementa je dozvoljeno, dok je za skupove u matematici poredak nebitan, a elementi u njima ne mogu da se ponavljaju. Poredak pri izlistavanju Haskel lista je uslovljen poretkom izdvajanja elementa iz liste sa operatorom[154] `<-`. Elementi će biti "obrađeni" onim redom kojim su navedeni u listi iz koje se izdvajaju.

Predikata može biti više, pri čemu se predikati razdvajaju zarezom. Na primer, nastavljajući sa prethodnim primerima, zapis

```
[x | x <- xs, even x, (x `mod` 3) == 0]
```

se evaluira na listu svih brojeva iz liste `xs` koji su deljivi i sa `2` i sa `3`, a to je `[0]`. Takođe, moguće je u potpunosti izostaviti predikate, pa se

```
[x^2 | x <- xs]
```

evaluira u listu kvadrata svih elementa liste `xs`.

Osim izdvajanja i predikata, sa desne strane uspravne crte moguće je uvesti i lokalne deklaracije sa `let … = ….` Ovako deklarisana imena mogu se koristiti u izrazima koji slede nakon deklaracije. Na primer, lista svih brojeva iz `xs` deljivih sa 3 može se dobiti evaluacijom izraza

```
[x | x <- xs, let y = x `mod` 3, y == 0]
```

[151] eng. *list comprehension*
[152] izraza tipa `Bool`
[153] Setimo se, `even :: Integral a => a -> Bool` je funkcija koja određuje da li je broj paran.
[154] Mnogo više o ovom operatoru će biti rečeno u lekciji o monadama.

Zadatak 2. Zapisati listu kvadrata svih prirodnih brojeva manjih od 100 koji su deljivi sa svojom poslednjom cifrom.

Sintaksa izlistavanja jeste elegantna, ali ne donosi nam suštinski ništa novo. Svaki izraz oblika[155]

```
[f x | x <- xs, Φ₁(x), … Φₙ(x)]
```

gde su `Φ₁`, ..., `Φₙ` neki predikati, definiše istu listu kao i izraz

```
map f . filter Φₙ . … . filter Φ₁ $ xs.
```

Na programeru je da odluči koju od sintaksa će iskoristiti.

Zadatak 3. Rešiti prethodni zadatak korišćenjem `map` i `filter` funkcija.

15.3.1. Izlistavanje i podudaranje oblika

Prilikom izlistavanja, sa leve strane operatora `<-` moguće je postaviti i neki oblik. Tada će iz liste biti izdvojeni samo elementi koji se podudaraju sa tim oblikom.

Primer 4. Iz liste možda-brojeva

```
xs = [Just 2, Nothing, Just 6, Just 8, Nothing]
```

moguće je izdvojiti vrednosti iz `Just` "omotača" prostim izlistavanjem

```
[x | Just x <- xs].
```

Navedeno izlistavanje će se evaluirati u listu `[2, 6, 8]`.

Zadatak 4. Iz liste lista brojeva, tj. iz liste tipa `[[Int]]`, izdvojiti prve elemente svih lista dužine 2.

Podudaranje oblika kod operatora `<-` je jedna od prednosti izlistavanja u odnosu na pristup sa `map` i `filter` funkcijama. Iako je i ovom slučaju moguće izraziti iste programe i sa jednom i sa drugom tehnikom, tehnika izlistavanja dovodi do konciznijeg koda.

Zadatak 5. Iz liste možda-brojeva izdvojiti samo vrednosti iz `Just` konstruktora kao u prethodnom primeru, koristeći `map` i `filter` funkcije.

15.3.2. Izlistavanje iz više lista

Sa izlistavanjem moguće je iz više lista izdvojiti elemente, tako što će se sa desne strane uspravne crte navesti više izdvajanja sa operatorom `<-`. U slučaju izdvajanja iz dve liste, prvo će se "izdvojiti" svi elementi druge liste sa prvim elementom prve liste, pa svi elementi druge liste sa drugim elementom prve liste i tako dalje.... Sledeći primer ilustruje izdvajanja iz više lista.

Primer 5. Sa izlistavanjem iz liste brojeva i liste karaktera možemo lako generisati listu svih polja šahovske table. Svako polje predstavićemo kao uređen par prirodnog broja i karaktera na sledeći način

```
[(a, b) | a <- [1 .. 8], b <- ['A' .. 'H']].
```

Ovim dobijamo listu sa 64 elementa

```
[(1,'A'),(1,'B'),(1,'C') … (8,'H')]
```

[155] Ovo takođe važi i za izlistavanja u kojima se nalaze i deklaracije, što se može pokazati jednostavnim argumentom.

Ako poredak izdvajanja preokrenemo, dobićemo listu sa istim elementima ali u drugačijem poretku. Preciznije, izlistavanje

```
[(a, b) | b <- ['A' .. 'H'], a <- [1 .. 8]]
```

se evaluira u listu

```
[(1,'A'),(2,'A'),(3,'A') … (8,'H')].
```

Pri dodeli vrednosti sa operatorom `<-`, deklarisano ime će biti dostupno u svim narednim izrazima sa desne strane dodele (ali ne i u levim). Tako, možemo definisati i sledeće izlistavanje

```
[(a, b) | a <- [1 .. 3], b <- [1 .. a]]
```

koje se evaluira u

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

Zadatak 6. Naći sve pravougle trouglove obima manjeg od 100 čije su stranice celi brojevi. Trouglove predstaviti kao uređene trojke.

15.4. Pragme i ekstenzije

Pragma je instrukcija za Haskel kompajler smeštena u kodu. Pragme se zapisuju u obliku `{-# WORD … #-}`, gde `WORD` označava tip pragme, nakon čega mogu slediti dodatne informacije. Neki od dostupnih, i najčešće prisutnih, tipova pragmi su:

1. `LANGUAGE` omogućava ekstenzije Haskel jezika. O ekstenzijama ćemo govoriti u nastavku.
2. `OPTIONS_GHC` prosleđuje opcije *GHC* kompajleru, koje bi se inače prosledile kao opcije pri pokretanju kompajlera.
3. `WARNING` i `DEPRECATED` omogućava ispis poruka upozorenja prilikom kompilacije koda.
4. `MINIMAL` određuje minimalan skup metoda koje instanca neke klase mora da implementira.
5. `INLINE` i `NOINLINE` sugerišu kompajleru da (ne) izvrši određenu optimizaciju (*inlining*) prilikom generisanja izvršne datoteke.

Mi smo se u prošloj lekciji upoznali sa `MINIMAL` pragmom koju ćemo imati prilike ponekad da koristimo. Ostale pragme ćemo ređe koristiti, osim `LANGUAGE` pragme koja je verovatno najčešće korišćena pragma i koja je prisutna u gotovo svakom netrivijalnom Haskel projektu.

15.4.1. Ekstenzije

Pragma `LANGUAGE` omogućuje aktivaciju *Haskel ekstenzije* koja na neki način menja Haskel jezik. Ekstenzije mogu da dozvole određenu sintaksu koja nije validna bez te ekstenzije, da promene neku pojedinost sistema tipova, sistem modula, integraciju sa drugim jezicima, ili jednostavno da zabrane neke konstrukcije, itd.. Skup ekstenzija se menjao tokom decenija: nove ekstenzije su se dodavale i nastavljaju da se dodaju, neke ekstenzije su prevaziđene, a neke su postale deo standardnog jezika.

Pragma `LANGUAGE` mora se navesti na vrhu datoteke, pre deklaracije modula, i odnosi se samo na modul u kom je navedena:

```
{-# LANGUAGE LambdaCase #-}
module Main where

...
```

Primer aktivacije ekstenzije `LambdaCase`. Ako se aktivira više ekstenzija, tada se može navesti više pragmi, ili se u jednu pragmu postaviti više imena ekstenzija razdvojenih zarezima.

Većina ekstenzija je van domašaja ove lekcije, ali za sada možemo razumeti naredne:

1. `BinaryLiterals` dozvoljava pisanje celobrojnih literala u binarnom obliku. Ovakvi literali moraju se započeti sa `0b` ili `0B`. Na primer, literal `0b100` predstavlja 4.
2. `DuplicateRecordFields` dozvoljava da različiti zapisi imaju polja sa istim imenom.
3. `HexFloatLiterals` dozvoljava korišćenje heksadecimalnih brojeva sa zarezom: literal `0x0.1` predstavlja $\frac{1}{16}$, literal `0x0.0F` predstavlja $\frac{15}{256}$, itd...
4. `InstanceSigs` dozvoljava da se prilikom instanciranja klase navedu tipovi metoda.
5. `LambdaCase` dozvoljava da se lambda izraz oblika

   $$\texttt{\textbackslash x -> case x of \{}p_1\texttt{ -> }e_1\texttt{; … }p_n\texttt{ -> }e_n\texttt{\}}$$

   zapiše kao

   $$\texttt{\textbackslash case \{}p_1\texttt{ -> }e_1\texttt{; … }p_n\texttt{ -> }e_n\texttt{\}}.$$

   Ovo je jedna od najkorišćenijih ekstenzija.
6. `MagicHash` dozvoljava da se na kraju imena postavi znak `#`, i uvodi još neke oblike literala. Ova ekstenzija se najčešće koristi pri radu sa nekim *low-level* tipovima Haskel jezika.
7. `NoImplicitPrelude` stopira implicitni uvoz *Prelude* modula. Ovo oslobađa prostor imena.
8. `NumericUnderscores` dozvoljava zapis numeričkih literala sa znakom `_` radi bolje čitljivosti. To omogućava da npr. vrednost milion zapiše kao `1_000_000`. Ako je ova ekstenzija aktivirana, znak `_` u numeričkom literalu se u potpunosti ignoriše od strane kompajlera.
9. `OverloadedRecordDot` dozvoljava da se polju `b` nekog zapisa `a` pristupi sa `a.b` umesto sa `b a`. Ovakva sintaksa je analogna drugim jezicima gde se sa tačkom pristupa poljima nekog objekta (strukture).
10. `UnicodeSyntax` Dozvoljava da se umesto ASCII nizova poput `::`, `->`. `<-`, `=>`, itd, koriste UTF simboli poput ∷, →, ←, ⇒, itd... Ovo omogućava zapise poput `f :: Num a ⇒ a → Int`.

Ekstenzije su moćan deo Haskel jezika koje omogućuju programerima da olakšaju programiranje, ali i da istraže nove pravce u kojima bi Haskel jezik mogao da se razvija u budućnosti. Sa velikom moći dolazi i velika odgovornost, te je stoga potrebno razmisliti o posledicama[156] pre aktiviranja ekstenzija. U praksi, neke ekstenzije su često prisutne na projektima (npr. `LambdaCase`), dok se neke ekstenzije zaobilaze u širokom luku (npr. `UnicodeSyntax`).

> **Zadatak 7.** Uključiti ekstenziju `LambdaCase` i ponovo definisati funkciju `uMetre :: Dužina -> Dužina` koristeći novu sintaksu.

15.5. Programiranje bez tačaka

Programiranje bez tačaka[157] je stil pisanja programskog koda u kom se ne navode parametri funkcija. Ova tehnika je moguća zahvaljujući osobinama sintakse Haskela zbog čega je navodimo ovde (iako sama po sebi nije deo Haskel sintakse).

Suština programiranja bez tačaka leži u osobini da se lambda funkcija `\x -> f x` može zameniti sa funkcijom `f`. Zaista, funkcija koja primenjuje `f` na argument `x` i vraća dobijeni rezultat, ne razlikuje se od same funkcije `f`, i možemo ih smatrati istima[158]. Zamena izraza `\x -> f x` u `f` naziva se **eta konverzija**.

[156] Na primer, kada govorimo o ekstenzijama koje menjaju sintaksu, to su posledice koje se tiču čitljivosti koda.

[157] eng. *point-free programming*, poznato još kao i *tacit programming*

[158] U Haskelu funkcije `f`, `g` smatramo istim ako poseduju iste tipove, i za sve vrednosti `x` domena važi `f x = g x`. Napominjemo da se ovde radi "meta" rezonovanju, a u samom Haskelu je nemoguće u opštem slučaju porediti funkcije sa operatorom `==` (ili bilo kojim drugim). Razlozi ovog ograničenja nemaju veze sa Haskelom već sa teoretskim ograničenjima računarstva. Ipak možemo dati jednostavan argument u terminima Haskela: tip `[Bool]` sadrži beskonačno mnogo vrednosti, te se stoga dve funkcije `f, g :: [Bool] -> Bool` ne mogu uporediti jer bi to zahtevalo proveru `f x == g x` za beskonačno mnogo nizova `x`. Dakle, u opštem slučaju jednakost funkcija u Haskelu ne može se definisati.

Eta konverzija je jednostavan princip, ali u kodu često koristimo podudaranje oblika umesto lambda izraza, zbog čega prilika za eta redukcijom može biti skrivena "ispred nosa". Na primer, neka je data funkcija `h :: Int -> Int -> Bool` i neka je funkcija `g :: Int -> Bool` definisana sa

```
g x = h 10 x.
```

Ovakvo jednostavno podudaranje oblika nije ništa drugo nego skraćenica za definiciju sa lambda izrazom

```
g = (\x -> h 10 x).
```

Uzimajući u obzir da je `h 10 x` zapravo `(h 10) x`, vidimo da lambda izraz

```
(\x -> (h 10) x)
```

možemo eta konverzijom da svedemo na `h 10`. Samim tim originalna definicija funkcije `g` je postala sada

```
g = h 10.
```

Dakle, eta konverzija nam omogućuje "skraćivanje" najdesnijeg parametra u definicijama oblika

$$\texttt{g } x_1 \ \ldots \ x_n \texttt{ = IZRAZ } x_n,$$

pod pretpostavkom da se x_n ne pojavljuje u `IZRAZ`[159].

> **Primer 6.** Funkciju `zbir :: [Int] -> Int` koja vraća zbir svih elemenata liste, možemo definisati:
>
> ```
> zbir xs = foldr (+) 0 xs
> ```
>
> "Skraćivanjem" parametra `xs` dobijamo kod
>
> ```
> zbir = foldr (+) 0
> ```

Primetimo da u definiciji `sum = foldr (+) 0` iz primera, ne pojavljuje se parametar funkcije `sum`. Stoga za ovu definiciju kažemo da je napisana u stilu bez tačaka. Izraz *bez tačaka* potiče iz matematike, gde se često za argument funkcije kaže *tačka*[160] (iako možda ta funkcija nema veze sa geometrijskim tačkama). Kôd bez tačaka može biti elegantniji od "koda sa tačkama", ali ta elegantnost može vrlo lako da preraste u nečitljivost[161]. Ipak, dobro je razumeti ovu tehniku, jer je popularna među Haskel programerima i može se videti u svim projektima.

> **Primer 7.** Napišimo definiciju
>
> ```
> f x = 2 * x - 1
> ```
>
> u stilu bez tačaka. Prvo možemo izdvojiti operator oduzimanja u prefiksnu poziciju čime dobijamo
>
> ```
> f x = (-) (2 * x) 1.
> ```
>
> Pošto je za eta konverziju neophodno da parametar bude smešten na desnoj strani izraza, iskoristićemo `flip :: (a -> b -> c) -> b -> a -> c` kombinator koji obrće redosled parametra funkcija. Kombinator ćemo primeniti na funkciju `(-)`, čime stižemo do definicije
>
> ```
> f x = flip (-) 1 (x * 2).
> ```
>
> Ako postavimo i * u prefiksnu poziciju dobijamo

[159] Ovaj uslov nam zabranjuje da uradimo eta redukciju definicije `g x = max x x`, jer se `x` pojavljuje u izrazu `max x`

[160] često čujemo izraz *vrednost funkcije u tački*

[161] navedene osobine koda su subjektivne. Čak i za najjednostavniji kod bez tačaka, kakav smo videli u primeru, možemo reći da je nečitljiv, jer iz oblika definicije nije odmah jasno da je `zbir` funkcija.

```
f x = flip (-) 1 ((*) 2 x).
```

Vidimo da se ovde radi o kompoziciji funkcija `flip (-) 1` i `(*) 2` primenjenoj na `x`, pa je

```
f x = ((flip (-) 1) . ((*) 2)) x.
```

Prostom eta redukcijom stižemo do

```
f = (flip (-) 1) . ((*) 2)
```

U ovom primeru vidimo da ne treba po svaku cenu pisati kod u stilu bez tačaka[162].

Ironično, u stilu bez tačaka često se pojavljuje operator kompozicije koji se zapisuje sa tačkom. Ovo često buni početnike, te je dobro napomenuti da stil bez tačaka samo podrazumeva da se funkcije definišu bez navođenja parametra, a da ne podrazumeva ništa o upotrebi operatora kompozicije.

Pokušajmo sad da rešimo "problem" koji se često može sresti u Haskel kodu. Neka su date funkcije `f' :: A -> B -> C` i `g' :: C -> D`. U kodu nam je potrebna funkcija `h = \x y -> g' (f' x y)`, i voleli bismo ovu funkciju napišemo u stilu bez tačaka. Prva ideja koja nam pada na pamet je da iskoristimo operator kompozicije `.` i definišemo `h = g' . f'`. Međutim dobićemo tipsku grešku čim ovako definišemo `h`. Razlog tome je što je kodomen funkcije `f` tip `B -> C`[163]. To znači da domen funkcije `g'` mora biti isto `B -> C` ako želimo izvršimo kompoziciju, što naravno nije za naš primer. Dakle, funkcije više promenljivih ne možemo uvek lako komponovati sa `.`.

Pri traženju definicije bez tačaka, uvek je dobro krenuti od definicije za koju znamo da je dobra

```
h x y = g' (f' x y).
```

Videli smo da `h` nije kompozicija `g'` i `f'`, ali `h x` jeste kompozicija funkcija `g'` i `f' x`:

```
h x y = (g' . (f' x)) y.
```

Sada možemo izvršiti eta konverziju, čime dobijamo

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

Pomeranjem operatora kompozicije u prefiksnu poziciju dobijamo

```
h x = (.) g' (f' x),
```

što se može shvatiti kao primena funkcije `(.) g'` na `f' x`. Drugim rečima, ovde se radi o kompoziciji

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

koja se može eta konvertovati u

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

Ovim smo stigli do definicije "kompozicije" `f'` i `g'` koja je u stilu bez tačaka.

Ovaj postupak možda želimo da proizvoljne funkcije `f :: a -> b -> c` i `g :: c -> d`, a ne samo određene `f'` i `g'`. Stoga definišemo kombinator

```
comp_1_2 :: (c -> d) -> (a -> b -> c) -> (a -> b -> d)
```

sa

```
comp_1_2 g f = ((.) g) . f.
```

[162] Protivnici programiranja bez tačaka nazivaju ovaj still *pointless* umesto *pointfree*.

[163] Podsećamo da je u tipovima strelica `->` desnoasocijativna, što znači da je tip `f :: A -> B -> C` zapravo tip `f' :: A -> (B -> C)`

Jasno, za funkciju `h` iz prethodnog paragrafa važi `h = comp_1_2 g' f'`. Međutim, definicija `comp_1_2` više nije u stilu bez tačaka. Stoga transformišimo i ovu definiciju u stil bez tačaka. Prebacivanjem operatora kompozicije u prefiksnu poziciju od

```
comp_1_2 g f = ((.) g) . f
```

dobijamo

```
comp_1_2 g f = (.) ((.) g) f,
```

odakle eta konverzijom dobijamo

```
comp_1_2 g = (.) ((.) g).
```

Ovde imamo kompoziciju dve `(.)` funkcije, odnosno

```
comp_1_2 g = ((.) . (.)) g.
```

Poslednjom eta konverzijom dobijamo

```
comp_1_2 = (.) . (.).
```

> **Zadatak 8.** Napisati funkciju koja računa aritmetičku sredinu dva broja u stilu bez tačaka.

> **Zadatak 9.** U stilu bez tačaka dati definiciju kombinatora koji vrši kompoziciju jedne unarne funkcije i jedne ternarne funkcije
>
> ```
> comp_1_3 ::(d -> e) -> (a -> b -> c -> d) -> (a -> b -> c -> e)
> ```

> **Zadatak 10.** U stilu bez tačaka dati definiciju kombinatora koji vrši kompoziciju tri unarne funkcije
>
> ```
> comp_1_1_1 :: (c -> d) -> (b -> c) -> (a -> b) -> (a -> d).
> ```

16. Funktori

Koncept *funktora* je važan za Haskel. Iako isprva možda deluje kao nepotrebno apstraktan pojam, funktori zapravo opisuju veoma prirodnu transformaciju. Kroz mnogobrojne primere, videćemo da se funktori pojavljuju često.

16.1. Funktori

Pojam funktora je preuzet iz apstraktnih oblasti matematike, ali se u Haskelu taj pojam svodi na jednu klasu tipova. Pre nego što tačno definišemo nove pojmove, pokušaćemo da damo motivaciju.

16.1.1. Motivacija

U jednoj od prethodnih lekcija, upoznali smo se sa apstraktnim tipom `Maybe :: * -> *`. Kao što smo tada videli, `Maybe A` je tip koji osim vrednosti tipa `A` sadrži i vrednost `Nothing` koja predstavlja odsustvo vrednosti tipa `A`.

Zamislimo sada da od neke funkcije dobijamo vrednost tipa `Maybe Float`[164]. Ako želimo dalje da obrađujemo vrednost, moramo se osloboditi `Maybe` omotača:

```
dupliraj :: Maybe Float -> Float
dupliraj (Just a) = 2 * a
```

Nažalost funkcija `dupliraj` nije totalna. Ako joj prosledimo vrednost `Nothing` doći će do izuzetka i program će biti prekinut. Dakle, moramo vratiti vrednost tipa `Float` i u slučaju kada je prosleđena vrednost `Nothing`. Stoga funkcija sada izgleda ovako:

```
dupliraj :: Maybe Float -> Float
dupliraj (Just a) = 2 * a
dupliraj Nothing = 0
```

Vrednost `0` je proizvoljno odabrana, ilustracije radi.

Funkcija `dupliraj` je sada totalna, ali uočavamo drugi problem: izgubili smo informaciju koju `Maybe` tip nosi u sebi. Sada više ne možemo razlikovati vrednost `0` koja je "došla" iz `Just 0` od one koja je "došla" od `Nothing`. Ponekad je dozvoljeno napraviti takav gubitak informacije, ali često postoje i situacije kada nije. Zbog toga, definisaćemo funkciju `dupliraj'`:

```
dupliraj' :: Maybe Float -> Maybe Float
dupliraj' (Just a) = Just (2 * a)
dupliraj' Nothing = Nothing
```

Funkcija `dupliraj'` je totalna i ispravno obrađuje vrednost `Nothing`.

Primetimo da je kodomen funkcije `dupliraj'` ponovo možda-tip. Dakle, ako smo mislili da vrednost dobijenu primenom funkcije `dupliraj` prosledimo nekoj narednoj funkciji `f :: Float -> A`, tada moramo na sličan način definisati i funkciju `f' :: Maybe Float -> Maybe A`. Zaista, već smo ustanovili da verovatno ne želimo funkciju tipa `Maybe Float -> A` jer se takvom funkcijom gubimo informaciju koju možda-tip pruža. Stoga je potrebno svaku funkciju definisati u "možda-obliku", baš kao `dupliraj` i `dupliraj'`.

Na prvi pogled sve je u redu, jer je takve funkcije neznatno teže definisati. Ali kako budemo razvijali sve veći program, primetićemo da pišemo mnogo koda na potpuno isti način. Od funkcije `f :: A -> B` kreiraćemo funkciju `f' :: Maybe A -> Maybe B`, kodom poput sledećeg:

[164] Naveli smo ranije takav primer sa temperaturnim senzorom. Međutim primer može biti bilo koji podatak koji se dobija iz spoljnog sveta: očitavanje sa senzora, korisnički unos, selekcija iz baze, *http* zahtev ka serveru, itd... Svaka od ovih aktivnosti nam može vratiti neku konkretnu vrednost ili vrednost `Nothing` koja označava da je došlo do greške. I zaista, mnoge funkcije u Haskel bibliotekama vraćaju *možda-vrednosti*.

```
f' :: Maybe A -> Maybe B
f' (Just a) = Just (f a)
f' Nothing = Nothing
```

Ovde je `f :: A -> B` neka proizvoljna funkcija

Navedeni postupak je jednostavan, ali neprijatan za ponavljanje. Zbog toga želimo da konstruišemo jednu polimorfnu funkciju višeg reda koja će nas osloboditi zamornog kopiranja koda. Ta funkcija će kao parametar imati funkciju `f :: a -> b` a vratiće funkciju tipa `Maybe a -> Maybe b`. Na osnovu opisa nije teško napisati definiciju:

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

Ubrzo će postati jasan izbor imena ove funkcije...

Sada se `dupliraj''` može[165] elegantnije definisati:

```
dupliraj'' :: Maybe Float -> Maybe Float
dupliraj'' = maybeMap (2*)
```

Rečeno žargonom Haskel programera, funkcija `maybeMap` *podiže* funkciju tipa `a -> b` u funkciju tipa `Maybe a -> Maybe b`. Više nije neophodno svaku funkciju redefinisati tako da radi sa možda-tipovima, `maybeMap` na jedan uniforman način to radi za nas.

Kada radimo sa više funkcija, tada `maybeMap` možemo da iskoristimo sa operatorom kompozicije. Umesto da posebno od `f :: A -> B` i `g :: B -> C` definišemo odgovarajuće funkcije `f' :: Maybe A -> Maybe B` i `g' :: Maybe B -> Maybe C`, možemo jednostavno da `maybeMap` primenimo na `g . f`:

```
maybeMap (g . f) :: Maybe A -> Maybe C.
```

Ova kompozicija se mogla zapisati i kao

```
maybeMap g . maybeMap f.
```

Zaista, `maybeMap (g . f)` primenjena na vrednost `Just x` je po definiciji `Just (g (f x))`, a `maybeMap g . maybeMap f` primenjena na vrednost `Just x` je

```
maybeMap g (maybeMap f (Just x))
```

odnosno `maybeMap g (Just (f x))`, odnosno `Just (g (f x))`. I analogno se dokazuje jednakost i za `Nothing` vrednost[166].

Funkcija `maybeMap` ima još jednu zanimljivu osobinu: naime, važi jednakost

```
(maybeMap id) v = v
```

za svaku vrednost `v :: Maybe A`[167]. Odavde sledi da je `maybeMap id` upravo identička funkcija na `Maybe A`.

Opisane osobine funkcije `maybeMap` već smo videli ranije u drugom kontekstu: funkcija

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

[165] `(maybeMap (2*)) (Just x)` je po definiciji (funkcije `maybeMap`) `Just ((2*) x)`, što se svodi na `Just (2 * x)`. Sa druge strane `(maybeMap (2*)) Nothing` je po definiciji `Nothing`. To je upravo ono što smo želeli.

[166] Dokažite!

[167] `(maybeMap id) (Just x)` je po definiciji `(Just (id x))`, odnosno `Just x`. Sa druge strane, `(maybeMap id) Nothing` je `Nothing`.

primenjuje datu funkciju na svaki element date liste i vraća dobijeno listu. Ali `map` možemo takođe da shvatimo kao *podizanje* funkcije tipa `a -> b` u funkciju tipa `[a] -> [b]`, jer kada `map` primenimo na neku funkciju `f :: A -> B`, dobijamo funkciju tipa `[A] -> [B]`. Takođe, za funkciju `map` važe iste osobine: kompozicija `map g . map f` je ista kao kompozicija `map (g . f)`[168], i `map id` je identička funkcija.

Vidimo da je funkcija `map` analogna funkciji `maybeMap`, s tim što se `maybeMap` odnosi na apstraktni tip `Maybe :: * -> *` dok se `map` odnosi na apstraktni tip `[] :: * -> *`. Oba navedena tipska konstruktora možemo da shvatimo kao funkcije koje od nekog tipa `A` prave tip strukture koje sadrže vrednosti (ili vrednost) tipa `A`[169]. Odgovarajuće funkcije `map` i `maybeMap` omogućuju primenu neke funkcije na vrednosti koje su "upakovane" u ovim strukturama, pri čemu same strukture ostaju nepromenjene.

Kad god imamo kolekciju tipova (u ovom slučaju apstraktnih) i odgovarajućih funkcija koji imaju iste osobine, zgodno je da te tipove okupimo u jedinstvenu klasu, a odgovarajuće funkcije predstavimo zajedničkim imenom. Upravo zato uvodimo `Functor` klasu.

16.1.2. Definicija *Functor* klase

U Haskel jeziku, `Functor` je klasa tipova koja propisuje jednu funkciju i jedan operator:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a
    (<$) = fmap . const
    {-# MINIMAL fmap #-}
```

Prvo što bi trebalo da uočimo jeste da konkretni tipovi (tipovi vrste *) ne mogu pripadati ovoj klasi, jer je `fmap` tipa `(a -> b) -> f a -> f b` iz čega sledi da `f` mora biti apstraktni tip vrste `* -> *`[170].

Ono što nismo naveli u kodu su dva zakona koja `fmap` mora da zadovoljava:

1. *Funktor mora da čuva identičku funkciju*, tj. mora da važi
$$\texttt{fmap id} = \texttt{id}.$$
2. *Funktor mora da čuva kompoziciju funkcija*, tj. mora da važi
$$\texttt{fmap g . fmap h} = \texttt{fmap (g . h)}.$$

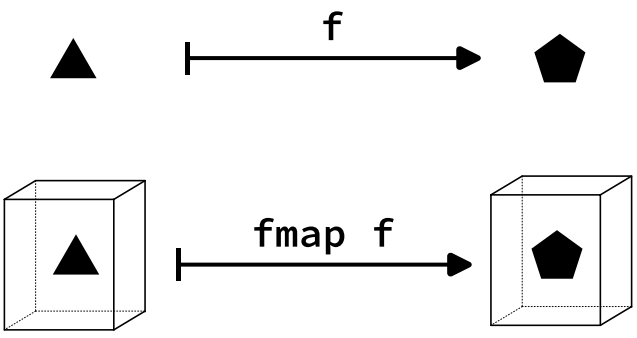
Kompajler ne može proveriti da li su navedeni zakoni zadovoljeni. Na programeru je da osigura da implementacija `fmap` za određenu instancu zadovoljava zakone. Svaki apstraktni tip koji je instanca `Functor` klase nazivamo **funktor**.

Operator `<$` koji je naveden u definiciji klase predstavlja podizanje `const` funkcije. Govoreći rečnikom struktura od malopre, sa operatorom `<$` zamenjuju se sve vrednosti u strukturi sa istom vrednošću. Ovaj operator je definisan preko `fmap` funkcije, i nije ga potrebno implementirati.

[168] Ovo je obrađeno u jednom od zadataka u sekciji o `map` funkciji.

[169] Vrednost tipa `Maybe A` se može shvatiti kao struktura koja sadrži jednu ili nijednu vrednost tipa `A`, a vrednost tipa `[A]` se može shvatiti kao struktura koja sadrži proizvoljno mnogo vrednosti tipa `A`.

[170] Tip funkcije (ili bilo koje druge vrednosti) može sadržati samo konkretne tipove, pa sledi da su `f a` i `a` vrste *. Dalje sledi da je `f` vrste `* -> *`.



Ilustracija ideje o funktoru. Neka funkcija `f` transformiše trouglove u petouglove, i neka postoji tip `Kutija A` koji predstavlja kutiju koja sadrži vrednost tipa `A`. Tada `fmap f` u kontekstu funktora `Kutija`, menja sadržaj kutije sa funkcijom `f`, dok samu kutiju (strukturu) čuva. Kao što ćemo videti, funktori nisu ograničeni samo na nekakve strukture podataka, već mogu biti i apstraktniji.

Primer 1. Kao što smo već nagovestili, apstraktni tip `[] :: * -> *` je funktor. Instanca `Functor []` je već definisana na sledeći način

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

Već smo videli da `map` zadovoljava oba zakona, ali ponovimo to još jednom. Prvo,

$$\begin{aligned} &\texttt{fmap id [}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{]} \\ =\ &\texttt{[id }x_1\texttt{, id }x_2\texttt{, … id }x_n\texttt{]} \\ =\ &\texttt{[}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{]} \end{aligned}$$

što znači da je

$$\texttt{fmap id xs = xs}$$

za svaku listu `xs`. Dakle `fmap id` je identička funkcija, pa je zadovoljen prvi zakon funktora. Drugo, kako je

$$\begin{aligned} &\texttt{(fmap g . fmap f) [}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{]} \\ =\ &\texttt{fmap g (fmap f [}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{])} \\ =\ &\texttt{[g (f }x_1\texttt{), g (f }x_2\texttt{), … g (f }x_n\texttt{)]} \\ =\ &\texttt{fmap (g . f) [}x_1\texttt{, }x_2\texttt{, … }x_n\texttt{],} \end{aligned}$$

sledi da je

$$\texttt{(fmap g . fmap f) xs = fmap (g . f) xs}$$

za svaku listu `xs`. Prema tome zadovoljen je i drugi zakon.

U slučaju ove instance operator `<$` zamenjuje sve vrednosti u listi sa jednom vrednošću:

```
ghci> True <$ [1,2,3,4,5]
[True,True,True,True,True]
```

Operator `<$` nije mnogo zanimljiv. I zaista, retko se javlja potreba za korišćenjem ovog operatora.

Primer 2. Sa apstraktnim tipom `Maybe :: * -> *` smo motivisali funktore u prethodnoj sekciji. Instanca `Functor Maybe` je takođe već definisana u Haskelu:

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

Definicija `fmap` u potpunosti odgovara definiciji `maybeMap` do koje smo sami došli ranije.

U kontekstu ove instance, operator `<$` zamenjuje vrednost u možda-tipu, ako ta vrednost postoji:

```
ghci> 2 <$ Just 7
Just 2
ghci> 2 <$ Nothing
Nothing
```

Primer 3. Apstraktni tip `(,) :: * -> * -> *` ne može biti funktor, jer ne poseduje odgovarajuću vrstu. Ali parcijalnom aplikacijom na neki tip `T` dobijamo apstraktni tip `(,) T :: * -> *` koji može biti funktor. Upravo je na taj način definisana instanca `Functor ((,) a)` za svaki tip `a`:

```
instance Functor ((,) a) where
    fmap f (x,y) = (x, f y)
```

Dakle u ovom kontekstu, `fmap` primenjuje funkciju na drugu koordinatu uređenog para, a operator `<$` zamenjuje drugu koordinatu sa nekom vrednošću:

```
ghci> fmap (+10) (1, 2)
(1, 12)
ghci> "Haskel" <$ (True, 2)
(True, "Haskel")
```

Slično su definisane i instance za `(,,) a` i `(,,,) a b`. Kao i kod uređenih parova, `fmap` i `<$` utiču samo na poslednju koordinatu.

U modulu `Data.Functor` definisane su mnoge instance poput navedenih. U ovom modulu je takođe definisan operator podizanja

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

kao sinonim za funkciju `fmap`. Ovaj operator smo već predstavili u lekciji o operatorima ali samo u kontekstu lista. Za razliku od `<$`, operator podizanja se zaista često koristi.

Zadatak 1. Neka je definisan apstraktni tip trodimenzionalnog vektora

```
data V3 a = V3 a a a.
```

Definisati instancu `Functor V3`.

16.2. Aplikativni funktori

16.2.1. Problem sa *Maybe* funktorom

Koliko god da je pojam funktora koristan, ipak nije mnogo zgodan za rad sa funkcijama više promenljiva. Na primer, ako želimo sabrati dve `Maybe Int` vrednosti, moramo implementirati funkciju poput

`maybeSum`:[171]:

```
maybeSum :: Maybe Int -> Maybe Int -> Maybe Int
maybeSum (Just a) (Just b) = Just $ a + b
maybeSum  _        _       = Nothing
```

Lako je definisati funkciju poput `maybeSum`, ali kao i ranije, ne želimo da ponavljamo kod. Umesto toga, želimo da već postojeće funkcije od dva (ili više) parametra podignemo na uniforman način do funkcija koja "rade" sa možda-tipovima. Kao što funkcija `fmap` omogućava da funkciju tipa `a -> b` primenimo na (vrednost tipa) `Maybe a` i dobijemo `Maybe b`, tako sad želimo način da funkciju tipa `a -> b -> c` primenimo na `Maybe a` i `Maybe b` i dobijemo `Maybe c`. Imitirajući implementaciju `maybeSum` stižemo do funkcije `maybeMap2`:

```
maybeMap2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
maybeMap2 f (Just a) (Just b) = Just $ f a b
maybeMap2 _  _        _       = Nothing
```

Funkcija `maybeMap2` podiže funkciju od dva parametra do funkcije koja "radi" sa možda-tipovima.

Funkcija `maybeMap2` značajno olakšava rad sa funkcijama dva parametra, ali ne rešava problem sa funkcijama tri i više parametra. Naravno, lako možemo implementirati odgovarajuće funkcije `maybeMap3`, `maybeMap4`, `maybeMap5`, ... ali nas to ponovo vodi ka ružnom kodu[172]. I dalje nećemo imati uniforman način podizanja funkcija više parametra na nivo možda-tipova.

Vratimo se korak u nazad, i pogledajmo zašto dobro poznata funkcija `fmap` nije zgodna za rad sa funkcijama dve promenljive. Neka je `zbir = (+) :: Int -> Int -> Int`. Primenom `fmap` na `zbir` i jednu `Maybe` vrednost dobijamo funkciju "upakovanu" u `Maybe` strukturu:

```
ghci> zbir = (+) :: Int -> Int -> Int
ghci> z = fmap zbir (Just 2)
ghci> :t z
z :: Maybe (Int -> Int)
```

Funkciju `zbir` definišemo samo iz razloga da bi se ograničili na tip `Int -> Int -> Int`. Operator `+` je polimorfan, i tip `Num a` bi samo nepotrebno otežao analizu koja sledi.

Zašto ovo ima smisla? Tip funkcije `zbir` je `Int -> Int -> Int`, odnosno eksplicitnije napisano

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

Ako funkcija

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

uzima za prvi argument funkciju tipa `a -> b`, tada će u slučaju funkcije `zbir`, tipska promenljiva `a` biti `Int` a promenljiva `b` biti `Int -> Int`. Stoga je i

```
fmap zbir :: Functor f => f Int -> f (Int -> Int)
```

odnosno

```
fmap zbir (Just 2) :: Maybe (Int -> Int).
```

[171] Diskutabilno je da li želimo baš ovakvu implementaciju `maybeSum`. U nekim slučajevima bi možda imalo smisla vraćati `Nothing` samo ako su oba argumenta `Nothing`, u suprotnom vratiti neku vrednost u `Just` konstruktoru. Ipak, u praksi najčešće želimo da funkcija vrati `Nothing` kada je barem jedan od argumenata `Nothing`

[172] Npr. ako u nekom trenutku promenimo broj parametra funkcije koju podižemo, tada na svakom mestu gde podižemo tu funkciju moramo i da menjamo odgovarajuću `maybeMap` funkciju.

Vrednost tipa `Maybe (Int -> Int)` predstavlja unarnu funkciju u `Maybe` strukturi. Setimo se da mi želimo da `f` primenimo na *dve* `Maybe Int` vrednosti, a vrednost `z` je nastala primenom na jednu vrednost. Stoga je potrebno funkciju u vrednosti `z` primeniti na još jednu `Maybe Int` vrednost. Zato ćemo definisati `maybeApply` funkciju[173]:

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

Definicija je jasna: ako su date vrednosti funkcije i argumenta, tada tu funkciju primenjujemo na argument i vraćamo rezultat u možda-tipu. U svakom drugom slučaju, barem jedna od vrednosti je `Nothing`, pa zato i vraćamo `Nothing`.

Sa funkcijom `maybeApply` možemo završiti prethodni primer iz terminala:

```
ghci> zbir = (+) :: Int -> Int -> Int
ghci> z = fmap zbir (Just 2)
ghci> maybeApply z (Just 3)
Just 5
```

Ako koristimo operator podizanja `<$>` (koji je sinonim za `fmap`) a `maybeApply` postavimo u infiksni oblik, kôd možemo elegantnije zapisati:

```
ghci> (zbir <$> Just 2) `maybeApply` (Just 3)
Just 5
ghci> (zbir <$> Nothing) `maybeApply` (Just 3)
Nothing
ghci> (zbir <$> Just 2) `maybeApply` Nothing
Nothing
```

Lepota ovog pristupa je što na sličan način možemo postupiti sa funkcijama više promenljiva. Na primer, neka je data neka funkcija

```
g :: A1 -> A2 -> A3 -> B.
```

Ako je `m1` neka vrednost tipa `Maybe A1`, tada je

```
g <$> m1 :: Maybe (A2 -> A3 -> B).
```

Na ovu vrednost možemo da delujemo sa `maybeApply` funkcijom. Ako je `m2` neka vrednost `Maybe A2` tipa, tada je

```
(g <$> m1) `maybeApply` m2 :: Maybe (A3 -> B).
```

Naravno, `maybeApply` nam opet omogućava da *možda-funkciju* primenimo na *možda-vrednost*. Tačnije

```
((g <$> m1) `maybeApply` m2) `maybeApply` m3
```

je vrednost tipa `Maybe B`. Funkciju `maybeMap2` od ranije možemo sada definisati i ovako:

```
maybeMap2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
maybeMap2 f a b = (f <$> a) `maybeApply` b
```

[173] Definicija veoma podseća na definiciju funkcije `fmap2`, ali postoje suštinske razlike. `fmap2` podiže binarnu funkciju na nivo možda-tipova, dok `maybeApply` primenjuje unarnu funkciju obmotanu u možda-tip na neku možda-vrednost.

16.2.2. Aplikativni funktori

Vratimo se na opšti primer funktora `f :: * -> *`. Ako želimo da funkciju proizvoljne arnosti podignemo na nivo funktora `f` neophodno je da posedujemo funkciju koja će vrednost tipa `f (A -> B)` transformisati u vrednost tipa `f A -> f B`, baš kao što `maybeApply` transformiše `Maybe (A -> B)` u `Maybe A -> Maybe B`. Videli smo da je takvu funkciju zgodno koristiti u infiksnom obliku, te ćemo je stoga definisati kao operator `<*>` kog nazivamo *operator aplikacije funktora*. Dakle, za neke funktore `f` možemo zahtevati postojanje operatora

```
(<*>) :: f (a -> b) -> f a -> f b.
```

Takve funktore nazivamo **aplikativni funktori**, jer dozvoljavaju neku vrstu aplikacije funkcije na vrednost.

Mi smo se već do sada upoznali sa "klasičnim" operatorom aplikacije

```
($) :: (a -> b) -> a -> b.
```

Operator podizanja

```
(<$>) :: (a -> b) -> f a -> f b
```

može se takođe[174] shvatiti kao neka vrsta operatora aplikacije. Operator `<$>` aplicira funkciju tipa `a -> b` na vrednost tipa `f a`. Operator

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

takođe u nekom smislu vrši apliciranje funkcije na vrednost. Primetimo koliko su tipovi ova tri operatora međusobno slična.

Naravno, ne možemo očekivati da će svaka implementacija funkcije `<$>` biti dobra. Na primer, da smo `maybeApply` definisali kao konstantnu funkciju tada ne bismo mogli smisleno da podižemo ostale funkcije:

```
badMaybeApply :: Maybe (a -> b) -> Maybe a -> Maybe b
badMaybeApply  _        _       = Nothing
```

```
ghci> (zbir <$> Just 2) `badMaybeApply` (Just 3)
Nothing
```

Rezultat svih primena je `Nothing`, što nije mnogo korisno.

Zbog toga, za operator `<*>` zahtevaćemo neke osobine (zakone), baš kao što smo zahtevali zakone za `fmap`.

16.2.3. Definicija *Applicative* klase

Kao i u slučaju funktora, pojam aplikativnog funktora formalizovaćemo kroz klasu. Klasa `Applicative` je definisana na sledeći način.

```
class Functor f => Applicative f where
    pure :: a -> f a

    (<*>) :: f (a -> b) -> f a -> f b
    (<*>) = liftA2 id
```

[174] Opet ističemo jednu suptilnu činjenicu. Tip

```
t = (a -> b) -> f a -> f b
```

možemo shvatiti kao tip funkcije koja uzima dve vrednosti, tipa `a -> b` i `a`, i daje vrednost tipa `b`. U tom smislu, `t` je tip funkcije koja aplicira funkciju na vrednost. Sa druge strane, funkcija tipa `t` može se shvatiti i kao funkcija koja uzima funkciju tipa `a -> b` i daje funkciju tipa `f a -> f b`. U tom smislu, tip `t` predstavlja tip funkcije koja podiže druge funkcije na nivo funktora `f`.

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)

(*>) :: f a -> f b -> f b
a1 *> a2 = (id <$ a1) <*> a2

(<*) :: f a -> f b -> f a
(<*) = liftA2 const

{-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Prvo što uočavamo u deklaraciji klase, je klasno ograničenje `Functor f => Applicative f` kojim se zahteva da sve instance klase `Applicative` budu instance i klase `Functor`. Nakon deklaracija funkcija, navedena je i pragma koja označava da svaka `Applicative` instanca sadrži implementaciju `pure` funkcije, kao i barem jednu od implementacija `<*>` i `liftA2` funkcija (otuda njihovo navođenje unutar `( | )`, nalik na algebarsku sumu).

Sa operatorom

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

smo se već upoznali u prethodnim redovima: ovaj operator primenjuje funkciju "upakovanu" u `f` funktor na vrednost koja je takođe "upakovana" u `f`. Videli smo da operator `<*>` dozvoljava podizanje funkcija više promenljiva na nivo funktora. Funkcija `liftA2` je specijalan slučaj ovakvog podizanja: to je funkcija koja podiže funkciju dva argumenta. Naravno, `liftA2` je moguće definisati preko `fmap` i `<*>` što smo i prikazali gore[175]. Zanimljivo je da je moguće uraditi i suprotno: definisati funkciju `<*>` preko funkcije `liftA2`. Ta definicija[176] je i data u definiciji klase:

```
(<*>) = liftA2 id.
```

Prema tome, `<*>` i `liftA2` definišu jedna drugu, i dovoljno je implementirati jednu od ovih funkcija.

Funkcija `pure :: a -> f a` predstavlja funkciju koja proizvoljnu vrednost "obmotava" u funktor, odnosno proizvoljnu vrednost podiže. Do sada smo se susretali sa pojmom podizanja, ali nismo zahtevali od funktora da podiže bilo koju vrednost, već samo funkcije. Na primer, u slučaju funktora `Maybe`, `pure` je baš konstruktor `Just`, dok je u slučaju funktora `[]` funkcija `pure` definisana kao `\x -> [x]`. U produžetku ćemo videti zašto je u ovim primerima funktora neophodno definisati `pure` baš ovako.

Funkcije `*>` i `<*` su definisane preko ostalih, i njih ćemo pojasniti kasnije.

Kao smo nagovestili, kao i u slučaju `fmap` funkcije, funkcije `Applicative` klase moraju zadovoljavati neke zakonitosti. Ove zakonitosti se odnose na `<*>` i `pure`, ali se mogu formulisati i za `liftA2` i `pure`. Zakoni su sledeći:

1. `fmap f x = pure f <*> x`.
2. `pure id <*> v = v`.
3. `u <*> pure y = pure ($ y) <*> u`.
4. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`.

[175] Mi smo naveli definiciju funkcije `maybeMap2` preko funkcije `maybeMap`. Ali `maybeMap2` jeste `liftA2` u kontekstu `Maybe` funktora, a `maybeMap` jeste `fmap`.

[176] Prvi argument `liftA2` funkcije je binarna funkcija, što `id` na prvi pogled nije. Međutim, zbog polimorfizma `id` možemo da instanciramo u funkciju tipa `(t1 -> t2) -> (t1 -> t2)`, što jeste tip binarne funkcije! Stoga je `liftA2 id` odgovarajućeg tipa

```
f (t1 -> t2) -> f t1 -> f t2.
```

Dalje, ako želimo da je `liftA2 f x` isto što i `(<*>) (fmap f x)`, tada je `liftA2 id x = (<*>) x`, odakle eta redukcijom sledi `liftA2 id` mora biti `<*>`.

5. `pure f <*> pure x = pure (f x)`.

Iako zakoni deluju komplikovano, zapravo su veoma prirodni. Kroz naredne primere videćemo konkretna značenja zakona.

16.2.4. Možda-tipovi kao aplikativni funktori

Kako smo same aplikativne funktore motivisali sa možda tipovima, nije iznenađujuće da je `Maybe` aplikativni funktor. Instanca `Applicative Maybe` je definisana na sledeći način

```
instance Applicative Maybe where
    pure = Just

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

Sa definicijom `<*>` u kontekstu možda-tipova smo se upoznali gore. Primenom `Just f` na `Just v`, želimo `Just (f v)`, dok primenom `Just f` na `Nothing` želimo `Nothing`. Međutim, to je upravo definicija za `fmap f m`. Sa druge strane, ako nam je umesto upakovane funkcije data `Nothing` vrednost, tada svakako želimo da vratimo `Nothing`.

Funkcija `pure` je implementirana tako da vrednost pakuje u `Just` funktor (tj. `pure x = Just x`). Možemo se zapitati zašto smo odabrali ovakvu definiciju, a ne `pure x = Nothing`. Razlog je sasvim jednostavan, ako bi bilo `pure x = Nothing`, tada bi `pure f <*> x` bilo `Nothing <*> x` odnosno `Nothing`, a to se razlikuje od `fmap f x` koje se svodi naravno na `Just (f x)`. Dakle, prvi zakon aplikativnih funktora bi bio prekršen. Sa druge strane, definicija `pure x = Just x` je saglasna sa prvim zakonom jer je tada `pure f <*> x` baš `Just f <*> x` odnosno `Just (f x)`.

Što se tiče ostalih zakona, oni se lako mogu proveriti. Drugi zakon je jednostavan i on nam govori da podizanje identičke funkcije mora biti identička funkcija u smislu aplikacije `<*>`. Konkretno `pure id` je po definiciji `Just id`, pa je

```
pure id <*> v = v
```

bez obzira da li je `v` oblika `Nothing` ili `Just x`.

Treći zakon je sličan drugom, i on se odnosi na podizanje sekcije `$ y`. Važi da je

```
Nothing <*> pure y = Nothing =
(Just ($ y)) <*> Nothing = pure ($ y) <*> Nothing
```

kao i

```
(Just f) <*> pure y = Just (f y) =
(Just ($ y)) <*> (Just f) = pure ($ y) <*> (Just f),
```

te je i treći zakon zadovoljen.

Četvrti zakon se odnosi na podizanje operatora kompozicije. Kratko rečeno, želimo da podizanje operatora kompozicije bude funkcija koja će vršiti kompoziciju. Napomenimo, da je `<*>` definisan kao levoasocijativan operator (kod definicije same klase `Applicative`). Za `Maybe` funktor možemo lako proveriti da zakon važi. Pretpostavimo da je `u = Just _u`, `v = Just _v` i `w = Just _w`., tada je

```
pure (.) <*> u <*> v <*> w =
((Just (.) <*> Just _u) <*> Just _v) <*> Just _w =
(Just (_u .) <*> Just _v) <*> Just _w =
Just (_u . _v) <*> Just _w =
Just ((_u . _v) _w) =
```

```
Just (_u (_v _w)) =
Just _u <*> Just (_v _w) =
Just _u <*> (Just _v <*> Just _w) =
u <*> (v <*> w)
```

Na sličan način se mogu dokazati i jednakosti kada je jedna od vrednosti `u`, `v` ili `w` jednaka `Nothing`.

Peti zakon se značajno jednostavnije proverava:

```
pure f <*> pure x = (Just f) <*> (Just x) =
Just (f x) = pure (f x).
```

> **Zadatak 2.** Neka je funkcija `inverz :: Float -> Maybe Float` definisana sa
>
> ```
> inverz x = if x == 0 then Nothing else Just (1 / x).
> ```
>
> Definisati funkciju `harm :: Float -> Float -> Maybe Float` koja vraća harmonijsku sredinu
>
> $$\frac{2}{\frac{1}{x}+\frac{1}{y}}$$
>
> dva broja x i y. Ako je jedan od brojeva nula, tada vratiti `Nothing`. Definisati funkciju korišćenjem metoda `Applicative` klase i funkcije `inverz`.

16.2.5. Liste kao aplikativni funktori

Da bi `[] :: * -> *` postao aplikativni funktor, potrebno je samo definisati funkciju `pure :: a -> [a]` i jednu od funkcija `(<*>) :: [a -> b] -> [a] -> [b]` ili `liftA2 :: (a -> b -> c) -> [a] -> [b] -> [c]`. Posmatrajući samo tipove navedenih funkcija, možemo dati mnogo različitih definicija funkcija. Naravno neće sve definicije biti dovoljno dobre, tako da zakoni aplikativnih funktora budu zadovoljeni. Ali, u slučaju lista postoji više mogućih definicija instance `Applicative [a]` koje zadovoljavaju pomenute zakone. Jedna od tih definicija je odabrana i implementirana u prelidu. Tu definiciju ćemo prikazati u nastavku:

Funkcija `pure` uzima vrednost nekog neodređenog tipa `a` i vraća vrednost tipa `[a]`. Kako je tip u potpunosti neodređen (nije ograničen nekom tipskom klasom), ne možemo mnogo što šta uraditi sa vrednošću. Za vrednost `x`, funkcija `pure` može vratiti `[]`, `[x]`, `[x, x]`, `[x, x, x]`, itd... Heuristički govoreći, konstantna funkcija `\x -> []` bi bila beskorisna, dok bi funkcije poput `\x -> [x, x]`, `\x -> [x, x, x]` vraćale "previše". Zbog toga, definisaćemo `pure` kao funkciju `\x -> [x]`.

Da bismo definisali `<*>` neophodno je da obratimo pažnju na zakone aplikativnih funktora. Prvo, iz

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

i iz `pure f = [f]` sledi da `[f] <*> [x1, … xn]` mora biti `[f x1, … f xn]`. Prema tome, `<*>` primenjuje funkcije iz leve liste na vrednosti iz desne liste. Postoji mnogo načina na koje se takva primena može definisati, i za sada nam nije jasno kako se više funkcija primenjuje na više vrednosti (za sada samo

[177] Na primer, možda bi operator `(<*>)` funkcije i vrednosti uparivao član po član poput `zip` funkcije koja uparuje vrednosti dve liste u listu parova. Ali kako mora važiti prvi zakon `fmap f x = pure f <*> x` sledi da ovakva definicija nije moguća. U suprotnom bi bilo

```
[f x1,f x2]
= fmap f [x1, x2]
= pure f <*> [x1, x2]
= [f] <*> [x1, x2] = [f x1],
```

što naravno nije tačno.

razumemo `[f] <*> xs`[177]. Zbog toga, `<*>` definišemo kao operaciju koja primenjuje svaku funkciju iz leve liste na svaku vrednost iz desne:

```
fs <*> xs = [ f x | f <- fs, x <- xs ].
```

Za ovako definisane `pure` i `<*>`, nije teško dokazati zakone aplikativnih funktora.

16.3. Strelice kao funktori

Još od prvih lekcija, koristili smo konstruktor `->`, takozvanu strelicu, za konstrukciju tipa funkcija. U lekciji o vrstama, uverili smo se da je strelica binarni tipski konstruktor.

Kako je strelica vrste `* -> * -> *`, strelica ne može biti funktor[178], ali ako fiksiramo jedan od argumenata, dobićemo tip odgovarajuće vrste. Za konkretan primer tokom narednog izlaganja odabraćemo `Int` kao tip koji ćemo fiksirati za domen ili kodomen. Da bi olakšali zapis, kreirajmo dva nova tipa:

```
type FromInt a = Int -> a

type ToInt a = a -> Int
```

Tipovi `FromInt` i `ToInt` su tipovi koji potencijalno mogu biti funktori jer poseduju odgovarajuću vrstu:

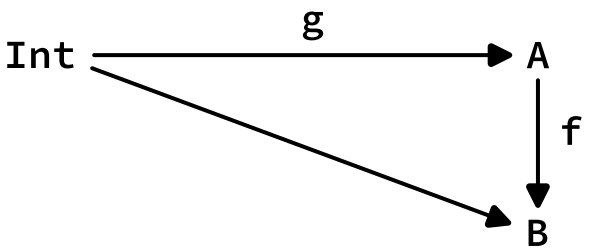
```
ghci> :kind FromInt
FromInt :: * -> *
ghci> :kind ToInt
ToInt :: * -> *
```

Tip `FromInt T` predstavlja tip svih funkcija *iz celih brojeva* u `T`, dok `ToInt T` predstavlja tip svih funkcija iz `T` *u tip celih brojeva*. Za početak, posvetimo se tipu `FromInt`.

Za definiciju instancu `Functor FromInt`, neophodno i dovoljno je definisati metodu `fmap :: (a -> b) -> FromInt a -> FromInt b`, tj, raspisujuci definiciju sinonima,

```
fmap :: (a -> b) -> (Int -> a) -> (Int -> b).
```

Definisanje metode `fmap` se svodi na definisanje funkcije tipa `Int -> B` ako su nam date funkcije `f :: A -> B` i `g :: Int -> A`. Kao što i naredna ilustracija pokazuje, nemamo mnogo izbora za definiciju: jedino sto možemo da uradimo je da napravimo kompoziciju `f . g :: Int -> B`:



Definicija instance je zapravo najkraća do sada:

```
instance Functor FromInt where
    fmap f g = f . g
```

Naravno, neophodno je i da proverimo dva zakona funktora.

Prvo, da li je `fmap id` identička funkcija. Naravno da jeste, jer je `id` neutralni element za kompoziciju tj.

[178] Funktori su tipovi vrste `* -> *`

$$\texttt{fmap id g = id . g = g}.$$

Drugo, da li je `fmap` distributivno u odnosu na kompoziciju? Kako je

$$\begin{aligned}
&\texttt{fmap (f2 . f1) g} \\
&\texttt{= (f2 . f1) . g} \\
&\texttt{= f2 . (f1 . g)} \\
&\texttt{= f2 . (fmap f1 g)} \\
&\texttt{= fmap f2 (fmap f1 g)} \\
&\texttt{= (fmap f2 . fmap f1) g}
\end{aligned}$$

za svako `g`, sledi da je

$$\texttt{fmap (f2 . f1) = fmap f2 . fmap f1},$$

pa je zadovoljen i drugi zakon funktora.

Primer 4. Definišimo konkretne vrednosti:

```
f :: Bool -> String
f b = show b

g :: FromInt Bool
g n = even n
```

Ako učitamo definicije funkcija (zajedno sa definicijom instance), možemo se uveriti u novi funktor:

```
ghci> m = fmap f g
ghci> :type m
FromInt String
ghci> m 2
"True"
ghci> m 3
"False"
```

Zadatak 3. Opisati operator `<$` u kontekstu funktora `FromInt`.

Naravno, sledeći korak je da ucinimo `FromInt` aplikativnim funktorom. Za definisanje metode

$$\texttt{pure :: a -> (Int -> a)}$$

ponovo nemamo mnogo izbora. Ako nam je data jedna vrednost `x :: A`, možemo samo na jedan način konstruisati funkciju tipa `A -> Int`, a ta jedinstvena funkcija je konstantna funkcija:

$$\texttt{\_ -> x}.$$

Definicija `<*>` je nešto komplikovanija, ali nas ponovo tipovi vode ka rešenju. U kontekstu tipa `FromInt`, operator `<$>` poseduje tip

$$\texttt{(Int -> (a -> b)) -> (Int -> a) -> (Int -> b)}.$$

Ako nam je data funkcija `f :: Int -> (A -> B)` i funkcija `g :: Int -> A`, tada možemo konstruisati funkciju tipa `Int -> B` tako što ćemo za svaku celobrojnu vrednost `Int`, aplicirati `f` i `g` na `n`, time dobiti vrednosti tipa `A -> B` i `B`, od kojih na očigledan način možemo dobiti vrednost tipa `B`. Kôd je ponovo sasvim jednostavan:

```
instance Applicative FromInt where
    pure x  = \_ -> x
    f <$> g = \n -> (f n) (g n)
```

Naravno, neophodno je dokazati da navedene definicije zadovoljavaju zakone aplikativnih funktora. Ovo ćemo prepustiti čitaocu, jer se ne razlikuje od provera koje smo do sada vršili.

> **Zadatak 4.** Dokazati da gorenavedena instanca `Applicative FromInt` zadovoljava pet zakona aplikativnih funktora.

Kao i do sada, uvek želimo da uopštimo kôd, tako da se odnosi na što više tipova. Prethodni argumenti za tip `FromInt` mogu se od reči do reči ponoviti za bilo koji drugi tip. Stoga, za svaki tip `T`, učinićemo tip `(->) T` instancom klasa `Functor` i `Applicative`. Srećom, Haskel dozvoljava da ovakva opšta definicija izrazi kroz deklaraciju klase korišćenjem polimorfnog tipa `(->) t`. Upravo tako su i definisane instance strelice u Prelidu:

```
instance Functor ((->) t) where
    fmap f g = f . g

instance Applicative ((->) t) where
    pure x  = \_ -> x
    f <*> g = \n -> (f n) (g n)
```

Vratimo se na tipski sinonim `ToInt`, koji konstruiše tip funkcija sa kodomenom `Int`. Da li je ovaj apstraktni tip funktor? Ako bi `ToInt` bio funktor, tada bismo od funkcija tipa `g :: A -> B` i `f :: A -> Int` mogli da konstruišemo funkciju tipa `B -> Int`. Međutim to je definitivno nemoguće, jer ako nam je data vrednost `B`, na tu vrednost ne možemo primeniti ni `f` ni `g`. Prema tome, `ToInt` ne može biti funktor, a samim tim ni aplikativni funktor. Isto važi i za svaki drugi apstraktni tip `FuncToT a = a -> T`.

16.4. IO funktor

Tip `IO A` predstavlja akciju (proceduru) koja kad se izvrši daje vrednost tipa `A`. Dakle, apstraktni tip `IO :: * -> *` prevodi konkretan tip `A` u tip akcije koja vraća vrednost tipa `A`. Ovo je veoma slično funktoru `FromInt` kog smo gore konstruisali. Razlika je u tome što vrednost tipa `IO A` ne zahteva argument da bi se dobila vrednost tipa `A` (akcije pokrećemo), dok vrednost tipa `FromInt` zahteva vrednost tipa `Int` da bi se dobila vrednost tipa `A` (funkcije primenjujemo).

Ako smo razumeli instance `Functor FromInt` i `Applicative FromInt`, onda naredne definicije bi trebalo da budu takođe razumljive. Ideja narednih implementacijama `fmap` i `<*>` je da se oslobodimo `IO` "omotača" pokretanjem akcija. Kao dodatnu pomoć za razumevanje, u komentarima su navedeni tipovi koji treba da poseduju metode u kontekstu tipa `IO`.

```haskell
instance Functor IO where
    -- fmap :: (a -> b) -> IO a -> IO b
    fmap f action = do
        r <- action
        return $ f r

instance Applicative IO where
    -- pure :: a -> IO a
    pure    = return

    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    actionF <*> actionX = do
        f <- actionF
        x <- actionX
        return $ f x
```

Obe instance su već definisane u standardnoj biblioteci.

Dokazivanje zakona funktora za date definicije je pravolinijski i nalik na dokazima koji su već izloženi u ovoj lekciji, te stoga neće biti navedeni.

Za kraj, primetimo da tip `IO A`, za razliku od svih drugih funktora koje smo analizirali do sada, ne predstavlja vrednosti koje "sadrže u sebi" vrednosti tipa `A`. Na primer, pri kompajliranju, vrednost tipa `IO Int` ne (mora da) sadrži vrednost tipa `Int`. Ovo navodimo ovde jer pomalo odstupa od intuicije koju smo ranije razvili o funktorima kao strukturama koje sadrže vrednosti (ilustracija sa kutijama).

17. Rekurzivni tipovi podataka

Rekurzivni tipovi podataka su tipovi čije vrednosti mogu "sadržati u sebi" vrednosti istog tipa. Ovakva apstraktna karakterizacija ne znači trenutno mnogo. Zbog toga ćemo u nastavku upoznati dva najznačajnija rekurzivna tipa, a to su *liste* i *stabla*. Kroz primere, uvidećemo da su rekurzivni tipovi podataka pogodni za predstavljanje vrednosti neograničene složenosti.

Važno je napomenuti da su rekurzivni tipovi specijalni slučaj algebarskih tipova podataka (koje smo već upoznali). Ono što izdvaja rekurzivne tipove podataka je to što se u definiciji tipa pojavljuje sam tip koji se definiše[179]

17.1. Liste

Konstruisaćemo tip `ListaBrojeva` čije vrednosti su liste s celobrojnim vrednostima[180]. Tip `ListaBrojeva` mora sadržati vrednost prazne liste, liste sa jednim elementom, liste sa dva elementa, liste sa tri elementa itd... Prazna lista se može predstaviti sa nularnim konstruktorom. Lista sa jednim elementom se može predstaviti sa unarnim konstruktorom, lista sa dva elementa sa binarnim konstruktorom, i tako dalje... Uzimajući ova razmatranja u obzir, dolazimo do naivne ideje o konstrukciji tipa `ListaBrojeva` kao sume konstruktora različite arnosti:

```
data ListaBrojeva =
    PraznaListaBrojeva
    | Lista1 Int
    | Lista2 Int Int
    | Lista3 Int Int Int
    | Lista4 Int Int Int Int
```

Obratite pažnju da je ovo samo algebarska suma podataka, prelomljena u više redova radi preglednosti

Odmah uočavamo problem: s definicijom poput navedene nikad ne možemo obuhvatiti sve liste. Koliko god mi konstruktora napravili, uvek će postojati potreba za još dužom listom tj. za konstruktorom još veće arnosti. Zbog toga moramo iskoristiti rekurzivnu konstrukciju. Lista može biti prazna lista, ili može biti lista koja je nastala spajanjem nekog elementa na početak već postojeće liste. Zbog toga, i tip lista konstruišemo kao sumu dva konstruktora: jedan konstruktor predstavlja praznu listu, dok drugi konstruktor nadovezuje element na listu manje dužine:

```
data ListaBrojeva = DodajBroj Int ListaBrojeva | PraznaListaBrojeva
```

> **Primer 1.** Listu od tri elementa `1`, `2`, `3` možemo konstruisati kao
>
> `DodajBroj 1 (DodajBroj 2 (DodajBroj 3 PraznaListaBrojeva))`.

Da bismo prikazivali vrednosti tipa `ListaBrojeva` u konzoli, pridružićemo tip `ListaBrojeva` klasi `Show`:

```
instance Show ListaBrojeva where
    show xs = "<" ++ prikaži xs ++ ">"
        where
            prikaži PraznaListaBrojeva = ""
            prikaži (DodajBroj y PraznaListaBrojeva) = show y
            prikaži (DodajBroj y ys) = show y ++ ", " ++ prikaži ys
```

[179] Baš kao što se u definiciji rekurzivne funkcije pojavljuje sama funkcija koja se definiše. Otuda i ime *rekurzivni tipovi podataka*.

[180] Već znamo da je to `[Int]`, ali ignorišimo to za trenutak.

Vrednosti tipa `ListaBrojeva` prikazujemo tako što elemente liste navodimo između znakova `<` i `>` koji podsećaju na izlomljene zagrade[181]. Za prikazivanje samih elemenata liste koristimo pomoćnu rekurzivnu funkciju `prikaži`. Sada možemo elegantno prikazivati vrednosti tipa `ListaBrojeva` u interaktivnom okruženju:

```
ghci> x = DodajBroj 1 (DodajBroj 2 (DodajBroj 3 PraznaListaBrojeva))
ghci> x
<1, 2, 3>
```

Zgodno je implementirati mnoge funkcije za rad sa tipom `ListaBrojeva`. Kako je i sama priroda tipa rekurzivna, i ove funkcije će biti (uglavnom) rekurzivne.

Dužinu liste je lako pronaći. Dužina prazne liste je `0`, a dužina liste nastale dodavanjem broja na listu `xs` je za jedan veća od dužine liste `xs`.

```
dužina :: ListaBrojeva -> Int
dužina PraznaListaBrojeva = 0
dužina (DodajBroj _ xs) = 1 + dužina xs
```

Funkcija `spoji` spaja dve liste vršeći rekurziju po prvoj listi.

```
spoji :: ListaBrojeva -> ListaBrojeva -> ListaBrojeva
spoji PraznaListaBrojeva xs = xs
spoji (DodajBroj x xs) ys = DodajBroj x (spoji xs ys)
```

Funkcija `obrni` obrće redosled elemenata liste

```
obrni :: ListaBrojeva -> ListaBrojeva
obrni PraznaListaBrojeva = PraznaListaBrojeva
obrni (DodajBroj x xs) = spoji (obrni xs) (DodajBroj x PraznaListaBrojeva)
```

Primetimo da rekurzivna funkcija `obrni` koristi u sebi drugu rekurzivnu funkciju `spoji`, zbog čega ova implementacija nije najefikasnija moguća.

> **Zadatak 1.** Kreirati tip `ListaBrojeva1` čije vrednosti su neprazne liste celobrojnih vrednosti.

> **Zadatak 2.** Kreirati tip `ListaBrojeva2` čije vrednosti su liste celobrojnih vrednosti koje imaju barem dva člana.

> **Zadatak 3.** Implementirati funkciju
>
> ```
> parni :: ListaBrojeva -> ListaBrojeva
> ```
>
> koja uklanja neparne elemente iz liste.

> **Zadatak 4.** Implementirati funkciju
>
> ```
> primeni :: (Int -> Int) -> ListaBrojeva -> ListaBrojeva
> ```
>
> koja primenjuje prosleđenu funkciju na svaki element liste.

[181] Odabrali smo izlomljene zagrade namerno da bi se ispis liste razlikovao od ispisa ugrađenih Haskel lista.

17.1.1. Apstraktni tip lista

Tip `ListaBrojeva` sadrži samo liste celobrojnih vrednosti. Jasno je da se sa sličnom konstrukcijom mogu definisati i drugi tipovi lista:

```
data ListaChar = DodajChar Char ListaChar | PraznaListaChar

data ListaBool = DodajBool Bool ListaBool | PraznaListaBool
```

Ovakvim pristupom za svaki tip moramo definisati odgovarajući tip lista, što nije praktično. Zbog toga, definisaćemo jedan apstraktan tip podataka `Lista :: * -> *` čiji parametar označava tip vrednosti sadržanih u listi. Definicija liste ostaje gotovo ista, ali se umesto konkretnog tipa pojavljuje tipski parametar `a`:

```
data Lista a = Dodaj a (Lista a) | PraznaLista
```

Nakon konstruktora `Dodaj` moraju biti navedeni konkretni tipovi (a ne apstraktni), te je stoga drugi parametar `Lista a` a ne `Lista`. Ovim je i utvrđena homogenost liste tj. obezbeđeno je da sve vrednosti moraju biti istog tipa. Na primer, ako posmatramo tip `Lista Int` tada konstruktor `Dodaj` ima tip `Int -> Lista Int -> Lista Int`, tj. `Dodaj` može dodati samo vrednost tipa `Int` na listu tipa `Lista Int`. Induktivno sledi da vrednost `Lista Int` sadrži samo vrednosti tipa `Int`.

Sa apstraktnim tipom se lako mogu predstaviti liste vrednosti bilo kog tipa. Na primer, sada `Lista Int` predstavlja tip koji odgovara tipu `ListaBrojeva` od malopre, a `Lista Bool` odgovara tipu `ListaBool` od malopre. Primetimo da sada konstruktori `Dodaj`, `PraznaLista` imaju isto ime za sve tipove, što dodatno olakšava zapis.

```
ghci> x = Dodaj 'a' (Dodaj 'b' PraznaLista)
ghci>:t x
x :: Lista Char
ghci> y = Dodaj True (Dodaj False PraznaLista)
ghci>:t y
y :: Lista Bool
```

Definicije funkcija `dužina`, `spoji`, `obrni` ostaju gotovo iste, ali se tipovi menjaju tako da se u njima pojavljuje tipska promenljiva. Na ovaj način ove funkcije postaju polimorfne i mogu se koristiti sa svim tipovima lista:

```
dužina :: Lista a -> Int
dužina PraznaLista = 0
dužina (Dodaj _ xs) = 1 + dužina xs

spoji :: Lista a -> Lista a -> Lista a
spoji PraznaLista xs = xs
spoji (Dodaj x xs) ys = Dodaj x (spoji xs ys)

obrni :: Lista a -> Lista a
obrni PraznaLista = PraznaLista
obrni (Dodaj x xs) = spoji (obrni xs) (Dodaj x PraznaLista)
```

> **Zadatak 5.** Za tip `Lista a` implementirati funkcije koje odgovaraju funkcijama `filter`, `map`, `zip` i `fold`.

Novi tip je takođe zgodno dodati u klasu `Show`. Implementacija funkcije `show` će biti potpuno ista kao implementacija te funkcije za tip `ListaBrojeva`, ali uz dodatnu pretpostavku. Za svaki tip `T`,

implementacija `show :: Lista T -> String` koristi u sebi implementaciju funkcije `show :: T -> String`, te je neophodno postaviti uslov o tipu `T` pri deklaraciji instance `Show (Lista T)`[182].

```
instance Show a => Show (Lista a) where
    show xs = "<" ++ prikaži xs ++ ">"
        where
            prikaži PraznaLista = ""
            prikaži (Dodaj y PraznaLista) = show y
            prikaži (Dodaj y ys) = show y ++ ", " ++ prikaži ys
```

> **Zadatak 6.** Uvesti tipove `Lista a` u klase `Eq` i `Ord`, `Functor` i `Applicative`.

17.1.2. Liste u Haskelu

U Haskelu liste su implementirane upravo onako kako smo i mi to gore uradili:

```
data [] a = [] | a : ([] a)
```

Sa navedenim kodom je definisan jedan apstraktni tip `[] :: * -> *`. Pod istim imenom je definisan i konstruktor prazne liste `[]`. Drugi konstruktor je konstruktor "dodavanja" `:`. Kako se ime `:` sastoji od specijalnog znaka, a ne slova, taj konstruktor je infiksni konstruktor[183], te se navodi između argumenata.

Haskel kompajler dozvoljava da se tip `[] a` zapiše kao `[a]`. Ovaj izuzetak u Haskel sintaksi značajno poboljšava čitljivost tipova. Takođe, izuzetak postoji i u konstrukciji samih vrednosti, te se vrednost

$$\texttt{a:b:c:[]},$$

može zapisati i kao `[a, b, c]` i slično.

17.2. Stabla

Za liste se kaže da su *linearne* strukture podataka, jer elementi na prirodan način čine niz. Svaki element liste, osim poslednjeg, poseduje jedinstvenog sledbenika. Isto tako, svaki element, osim prvog, poseduje jedinstvenog prethodnika. Koliko god liste bile korisne, liste nisu uvek najbolji izbor za reprezentaciju podataka.

Stablo je rekurzivna struktura podataka sačinjena od *čvorova*, u kojoj svaki čvor sadrži *vrednost* i nekoliko manjih stabala (od kojih neka mogu biti prazna). Specijalno, **binarno stablo** je stablo u kom svaki čvor sadrži vrednost i dva manja stabla[184].

Tip binarnog stabla koje sadrži u sebi vrednosti tipa `Int` možemo kreirati na sledeći način: prazno stablo ćemo kreirati sa nularnim konstruktorom `PraznoStablo`, dok ćemo čvorove kreirati sa konstruktorom `Čvor`. Konstruktor `Čvor` zavisi od tri parametra: vrednosti tipa `Int`, kao i od dve vrednosti tipa `StabloBrojeva`:

[182] Gore o tome nismo razmišljali o ovome, jer `Int` pripada klasi `Show`. Ali tip poput `Int -> Int` ne pripada ovoj klasi, te je besmisleno zahtevati da i `Lista (Int -> Int)` pripada ovoj klasi.

[183] U lekciji o operatorima smo naveli da operatori ne mogu da sadrže karakter `:`. Međutim, za infiksne konstruktore to pravilo je suprotno: infiksni konstruktori *moraju* početi sa `:`. Ostale stvari koje smo naveli o operatorima važe i za infiksne konstruktore. Između ostalog, moguće je deklarisati prioritet i asocijativnost infiksnog konstruktora sa odgovarajućim `infix` deklaracijama.

[184] Koja pritom mogu biti prazna

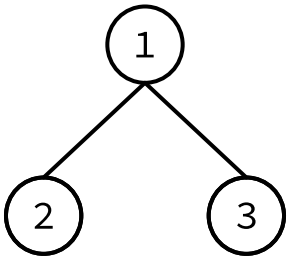
```
data StabloBrojeva =
    PraznoStablo
    | Čvor Int StabloBrojeva StabloBrojeva
    deriving (Show, Eq)
```

Izvedene instance `Show StabloBrojeva` i `Eq StabloBrojeva` su sasvim dovoljne za nas.

Primer 2. Vrednost

```
stablo1 = Čvor 1
    (Čvor 2 PraznoStablo PraznoStablo)
    (Čvor 3 PraznoStablo PraznoStablo)
```

predstavlja naredno stablo sa tri čvora:

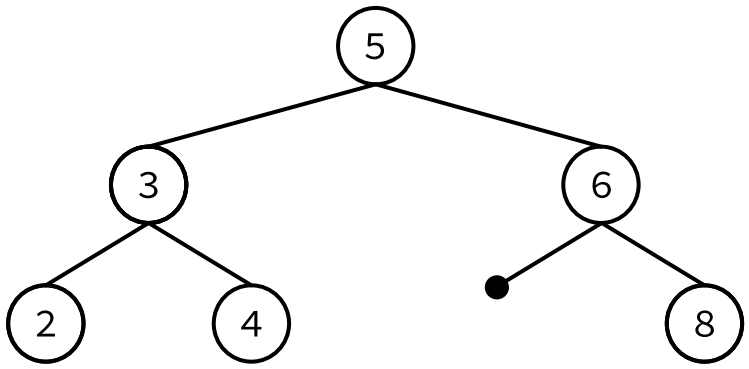


Ovo stablo je sačinjeno od čvora koji sadrži vrednost 1 kao i dva manja stabla. Ta dva manja stabla su sačinjena od po jednog čvora koji sadrži vrednost 2, odnosno 3. Ovi čvorovi u sebi sadrže prazna stabla (koje ne predstavljamo na slikama).

Za čvor 1 kažemo da je *roditelj* čvorova 2 i 3, dok za čvorove 2 i 3 često kažemo i da su *deca* čvora 1. Za jedinstveni čvor koji se nalazi na samom vrhu stabla, ovde je to 1, kažemo da je *koren* stabla. Za čvorove koji nemaju decu kažemo da su *listovi* stabla: u ovom stablu to su 2 i 3.

Primer 3. Naredna vrednost predstavlja stablo sa šest čvorova:

```
stablo2 = Čvor 5
    (Čvor 3
        (Čvor 2 PraznoStablo PraznoStablo)
        (Čvor 4 PraznoStablo PraznoStablo))
    (Čvor 6
        PraznoStablo
        (Čvor 8 PraznoStablo PraznoStablo))
```



U ovom stablu, 5 je koren, a čvorovi 2, 4 i 8 su listovi. Primetimo da u stablu čvor 6 ima samo jedno dete.

Stabla su pogodne strukture podataka za modelovanje mnogih odnosa koje srećemo u svakodnevnici[185].

Prvo bi bilo zgodno napisati funkciju `brojČvorova` koja određuje broj čvorova u stablu[186]. Naravno, funkciju ćemo implementirati rekurzivno, podudaranjem vrednosti po konstruktorima. Broj čvorova u praznom stablu je 0. Broj čvorova u stablu `Čvor a l r` je za jedan veći od zbira broja čvorova u podstablima `l` i `r`:

```
brojČvorova :: StabloBrojeva -> Int
brojČvorova PraznoStablo = 0
brojČvorova (Čvor _ l r) = 1 + brojČvorova l + brojČvorova r
```

```
ghci> brojČvorova stablo1
3
ghci> brojČvorova stablo2
6
```

Vrednosti `stablo1` i `stablo2` su definisana u prethodnim primerima.

Još jedna numerička karakteristika je *visina* stabla koja predstavlja najveći broj koraka od korena stabla do nekog lista. Drugim rečima, visina stabla predstavlja broj "generacija" prisutnih u stablu. Visinu stabla ćemo takođe izračunati rekurzivno: visina praznog stabla je 0, a visina stabla `Čvor a l r` je maksimum visina stabala `l` i `r` uvećan za jedan:

```
visinaStabla :: StabloBrojeva -> Int
visinaStabla PraznoStablo = 0
visinaStabla (Čvor _ l r) = 1 + max (visinaStabla l) (visinaStabla r)
```

```
ghci> visinaStabla stablo1
2
ghci> visinaStabla stablo2
3
```

[185] Na primer, pokušajte da sa stablom predstavite porodično stablo: sve vaše direktne pretke (roditelje, roditelje roditelja, itd...). Uvidećete da je porodično stablo zaista jedno binarno stablo. Isprva zbunjujuće, ali u takvom predstavljanju *deca* čvorova će zapravo biti *roditelji* osobe...

[186] Kao što funkcija `length` određuje broj elemenata u listi.

Korisna je i funkcija koja proverava da li se neka vrednost nalazi u stablu[187]. Strategija za pretragu je sledeća: ako čvor sadrži vrednost koju tražimo, tada se pretraga završava. U suprotnom, pretragu nastavljamo rekurzivno, pretražujući oba stabla:

```
elementStabla :: Int -> StabloBrojeva -> Bool
elementStabla _ PraznoStablo = False
elementStabla v (Čvor a l r)
    | v == a = True
    | v /= a = elementStabla v l || elementStabla v r
```

```
ghci> elementStabla 6 stablo1
False
ghci> elementStabla 6 stablo2
True
```

Zadatak 7. Implementirati funkciju

```
zbir :: StabloBrojeva -> Int
```

koja sabira vrednosti svih čvorova u stablu.

Zadatak 8. Implementirati funkciju

```
dupliraj :: StabloBrojeva -> StabloBrojeva
```

koja duplira vrednost svakog čvora, dok samu strukturu stabla ne menja.

Zadatak 9. Implementirati funkciju

```
najveći :: StabloBrojeva -> Int
```

koja pronalazi najveći element u stablu.

17.2.1. Uređena stabla

Stabla u kojima vrednost u svakom čvoru je veća od svake vrednosti u odgovarajućem levom podstablu, i manja od svake vrednosti odgovarajućem desnom podstablu, nazivamo **uređena stabla**.

Zadatak 10. Napisati funkciju

```
uređeno :: StabloBrojeva -> Bool
```

koje proverava da li je stablo uređeno.

Primer 4. Stablo `stablo1` nije uređeno jer je vrednost u korenskom čvoru manja od vrednosti u levom podstablu.

Stablo `stablo2` je uređeno jer je 3 veće od 2 a manje od 4, 6 je manje od 8, a 5 je veće 3, 2 i 4, i manje od 6 i 8.

Uređena stabla su značajna jer se neki algoritmi mogu efikasnije implementirati u slučaju uređenog stabla. Na primer, prilikom pretrage uređenog stabla često je potrebno izvršiti značajno manje poređenja: ako tražimo neku vrednost m u stablu čiji korenski čvor je n tada je dovoljno nastaviti pretragu samo u

[187] Kao što funkcija `elem` proverava da li se vrednost nalazi u listi.

levom (odnosno desnom) podstablu ako je $m < n$ (odnosno ako je $m > n$). Oslanjajući se na činjenicu da je korenski čvor veći (odnosno manji) od svih vrednosti u levom (odnosno desnom) podstablu, možemo ignorisati jedno od podstabala. Kôd za pretragu uređenog podstabla je sledeći:

```
elementUređenogStabla :: Int -> StabloBrojeva -> Bool
elementUređenogStabla _ PraznoStablo = False
elementUređenogStabla v (Čvor a l r)
    | v == a = True
    | v <  a = elementUređenogStabla v l
    | v >  a = elementUređenogStabla v r
```

> **Primer 5.** Funkcija `elementStabla` primenjena na vrednosti `8` i `stablo2` će imati sledeći tok: pošto je vrednost korenskog čvora (`5`) različita od `8`, funkcija će rekurzivno proveriti da li se vrednost `8` nalazi u levom podstablu. Pri toj pretrazi, funkcija ponovo rekurzivno proverava vrednosti čvorova 3, 2 i 5. Pošto levo podstablo ne sadrži traženu vrednost, prelazi se na desno podstablo. Prvo se proverava vrednost korena tog podstabla (čvor 8), a zatim se prelazi na desno dete, gde se pronalazi tražena vrednost. Primetimo da je algoritam obišao celokupno stablo pri pretrazi.
>
> Pogledajmo sada izvršavanje funkcije `elementUređenogStabla` primenjene na iste vrednosti `8` i `stablo2` (ovo stablo je uređeno, te ima smisla koristiti ovu funkciju). Pošto je tražena vrednost veća od vrednosti korenskog čvora 5, algoritam rekurzivno nastavlja pretragu na desnom podstablu. Vrednost `8` je takođe veća od vrednosti (sada korenskog) čvora 6, te se ponovo prelazi na desno podstablo. Međutim korenski čvor sada poseduje traženu vrednost, i pretraga se uspešno prekida. Primetimo da smo obišli samo dva čvora pre nego što smo prekinuli pretragu.

Kao što vidimo iz prethodnog primera, pretraga uređenog stablo može biti značajno efikasnija nego pretraga neuređenog stabla. Zaista, ako je dato stablo koje poseduje n generacija, tada to stablo može posedovati

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

čvorova. Pretraga ovakvog stabla bi u najgorem slučaju zahtevala $2^{n+1} - 1$ rekurzivnih poziva funkcije (računajući i prvi poziv), jer je neophodno da algoritam obiđe sve čvorove da bi dao definitivan odgovor. Sa druge strane, ako je to stablo uređeno, tada je u najgorem slučaju dovoljno izvršiti samo n rekurzivnih poziva.

> **Zadatak 11.** Implementirati funkciju
>
> ```
> najvećiUređeno :: StabloBrojeva -> Int
> ```
>
> koja pronalazi najveći element u uređenom stablu.

17.2.2. Apstraktni tip stabla

Naravno, u čvorovima stabla korisno je postavljati i vrednosti drugih tipova osim `Int`. Stoga je zgodno napraviti apstraktni tip `Stablo :: * -> *` koji će omogućiti uopštenu konstrukciju binarnog stabla. Jedini parametar ovog apstraktnog tipa, biće tip koji označava tip vrednosti sadržane u čvoru. Gore navedenu rekurzivnu definiciju je potrebno neznatno izmeniti:

```
data Stablo a = PraznoStablo | Čvor a (Stablo a) (Stablo a)
    deriving (Show, Eq)
```

Obratite pažnju da su imena konstruktora ostala ista, te stoga se ova definicije ne može nalaziti u istoj datoteci kao definicija tipa `StabloBrojeva`. Imena su namerno ostavljena ista, da bi bilo jasnije o čemu se radi.

Funkcije poput `brojČvorova` i `visinaStabla` imaju potpuno istu definiciju ali im se tip neznatno menja:

```
brojČvorova :: Stablo a -> Int
brojČvorova PraznoStablo = 0
brojČvorova (Čvor _ l r) = 1 + brojČvorova l + brojČvorova r

visinaStabla :: Stablo a -> Int
visinaStabla PraznoStablo = 0
visinaStabla (Čvor _ l r) = 1 + max (visinaStabla l) (visinaStabla r)
```

Imajte na umu da ove dve definicije ne mogu biti u istoj datoteci kao prethodne definicije, jer se tipovi razlikuju.

Funkcija `elementStabla` mora koristiti operator `(==) :: Eq a => a -> a -> Bool`. Stoga je potrebno da i tip funkcije `elementStabla` bude ograničen `Eq` klasom. Osim toga, definicija funkcije je potpuno ista:

```
elementStabla :: Eq a => a -> Stablo a -> Bool
elementStabla _ PraznoStablo = False
elementStabla v (Čvor a l r)
    | v == a = True
    | v /= a = elementStabla v l || elementStabla v r
```

Ako želimo da govorimo i o uređenim stablima, tada se moramo ograničiti na klasu `Ord`.

> **Zadatak 12.** Implementirati funkciju
>
> `uređeno :: Ord a => Stablo a -> Bool`
>
> koja proverava da li je stablo uređeno.

> **Zadatak 13.** Implementirati funkciju
>
> `elementUređenogStabla :: Ord a => a -> Stablo a -> Bool`
>
> koja proverava da li vrednost pripada uređenom stablu.

17.2.3. Stabla kao funktori

U jednoj od prethodnih lekcija obradili smo funktore. Funktor je apstraktni tip `T :: * -> *` koji dozvoljava implementaciju funkcije mapiranja

`fmap :: (a -> b) -> T a -> T b`,

koja zadovoljava određene osobine. Na primeru lista i možda-tipova, uvideli smo da o `fmap` možemo misliti kao o funkciji koja primenjuje prosleđenu funkciju na svaku vrednost tipa `a` sadržanoj u vrednosti tipa `T a`, pri čemu se sama struktura vrednosti tipa `T a` čuva.

I goredefinisani tip `Stablo :: * -> *` može biti instanca klase `Functor`. Primetimo prvo da `Stablo` je odgovarajuće vrste `* -> *`. Zatim, ako razmislimo o tome šta `fmap` radi na primeru drugih funktora, lako

se dolazi do ideje da `fmap` primenjuje datu funkciju na vrednost u svakom čvoru stabla[188].

```
instance Functor Stablo where
    fmap f (Čvor a l r) = Čvor (f a) (fmap f l) (fmap f r)
    fmap _ PraznoStablo   = PraznoStablo
```

Za instanciranje `Functor` klase, dovoljno je implementirati `fmap` funkciju.

Potrebno je uveriti se da ovako definisana `fmap` funkcija zadovoljava dve zakonitosti:

1. Podizanje identičke funkcije je identička funkcija, tj `fmap id = id`
2. Podizanje kompozicije je kompozicija podizanja, tj. `fmap (g. f) = fmap g . fmap f`.

Zakone možemo dokazati totalnom indukcijom po visini stabla. Na primer, dokažimo prvi zakon[189]. Baza indukcije su stabla visine 0, a to su prazna stabla. Funkcija `fmap id` primenjena na vrednost `PraznoStablo` je po definiciji `PraznoStablo`. Dalje, za induktivni korak pretpostavimo da tvrđenje važi za sva stabla čija visina je ne veća od n i dokažimo da onda važi i za stabla visine $n+1$. Neka je `Čvor a l r` stablo visine $n+1$. Funkcija `fmap id` primenjena na vrednost `Čvor a l r` je po definiciji

```
Čvor (id a) (fmap id l) (fmap id r) = Čvor a (fmap id l) (fmap id r).
```

Kako su `l` i `r` stabla visine ne veće od n, po induktivnoj pretpostavci važi da je `fmap id l = l` i `fmap id r = r`. Prema tome, primena funkcije `fmap id` na vrednost `Čvor a l r` daje istu tu vrednost. Indukcijom sledi da ovo važi za sva stabla, pa je `fmap id` zaista `id :: Stablo a -> Stablo a`.

> **Zadatak 14.** Klasa `Functor` definiše i operator `<$` kao `fmap . const`. Šta u kontekstu instance `Functor Stablo`, radi operator `<$`?

> **Zadatak 15.** Kreirati tip `TernarnoStablo :: * -> *` koji predstavlja stablo u kom svaki element ima tri direktna potomka. Implementirati funkcije za rad sa ovim tipom. Instancirati `Functor TernarnoStablo`

17.3. Aritmetički izrazi

Česta primena rekurzivnih tipova podataka je predstavljanje izraza (bilo matematičkih izraza, bilo izraza nekog programskog jezika). Izrazi su vrednosti koji koje mogu biti neograničeno složenosti, zbog čega su rekurzivni tipovi zaista neophodni za njihovo predstavljanje[190]. Na primer, ako posmatramo izraze koji su sačinjeni samo od prirodnih brojeva, sabiranja i množenja, tada lako možemo naći neograničeno mnogo izraza, pri čemu složenost svakog izraza je proizvoljno velika, npr:

$$1+1,$$

$$22*(1+6),$$

$$((10+32)*18)*(333+4)...$$

Da bismo našli pogodan način predstavljanja matematičkih izraza, za primer pogledajmo rekurzivnu strukturu izraza

$$(17+3)*(10*(1+1)).$$

[188] Ovo je samo heuristički argument, do kog smo došli iskustvom. U nastavku ćemo dokazati da ovako definisana `fmap` funkcija zaista zadovoljava odgovarajuće osobine.

[189] Konstruišite analogni dokaz za drugi zakon.

[190] Istaknimo da ovde ne govorimo o definisanju pojma funkcija uz pomoć aritmetičkih izraza u Haskelu, već o definisanju samog pojma *izraza* kao Haskel vrednosti.

Navedeni izraz predstavlja proizvod dva manja izraza $17 + 3$ i $10*(1 + 1)$. Prvi od tih izraza je zbir dva broja, dok je drugi proizvod broja i još manjeg izraza $1 + 1$. Na ovom primeru vidimo da su izrazi sačinjeni od manjih izraza ili brojeva koji se kombinuju uz pomoć dve operacije.

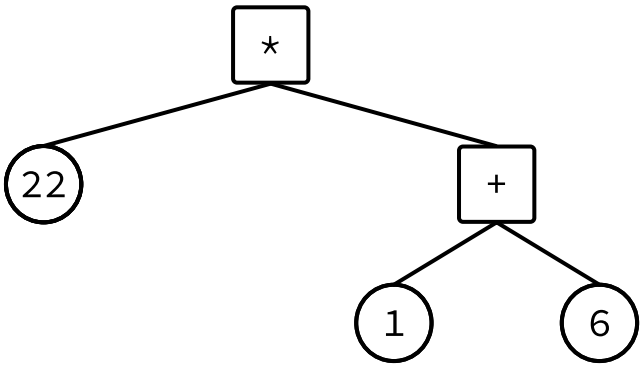
Navedeno razmatranje nas dovodi do rekurzivne definicije tipa `Izraz`. Vrednosti tipa `Izraz` konstruisaćemo na tri načina. Prvo, svaki prirodan broj za sebe čini izraz. Drugo, ako su data dva izraza, onda je i njihov zbir izraz. Treće, ako su data dva izraza onda je i njihov proizvod izraz:

```
data Izraz =
    Broj Int
    | Zbir Izraz Izraz
    | Proizvod Izraz Izraz
```

Primer 6. Sada izraz $22*(1 + 6)$ možemo da predstavimo sa vrednošću:

`Proizvod (Broj 22) (Zbir (Broj 1) (Broj 6))`.

Ova vrednost se može predstaviti i kao naredno stablo:



Kao što vidimo, vrednosti tipa `Izraz` se mogu shvatiti kao stabla, čiji listovi su brojevi, dok ostali čvorovi predstavljaju operacije sabiranja ili množenja. Stabla poput ovih nazivamo **sintaksna stabla**. Sa sintaksnim stablima se mogu prezentovati matematički izrazi ali i izrazi nekog programskog jezika, te je stoga ova struktura ključna za programe poput kompajlera i interpretera.

Naravno, novodefinisani tip želimo da uvedemo u neke klase. Najkorisnije je implementirati funkciju `show`, koja je će izraz predstaviti u obliku na koji smo do sada navikli.

```
instance Show Izraz where
    show (Broj a) = show a
    show (Zbir a b) = "(" ++ show a ++ "+" ++ show b ++ ")"
    show (Proizvod a b) = "(" ++ show a ++ "*" ++ show b ++ ")"
```

Druga funkcija koju je najlogičnije implementirati je funkcija `izračunaj :: Izraz -> Int`, koja sračunava izraz u jedinstvenu vrednost. Funkciju implementiramo rekurzivno: sračunavanjem broja se dobija taj isti broj, dok se sračunavanjem zbira (proizvoda) dva izraza dobija zbir (proizvod) vrednosti koje su dobijene sračunavanjem odgovarajućih podizraza:

```
izračunaj :: Izraz -> Int
izračunaj (Broj a) = a
izračunaj (Zbir a b) = izračunaj a + izračunaj b
izračunaj (Proizvod a b) = izračunaj a * izračunaj b
```

Zadatak 16. Kreirati tip `MatematičkiIzraz` čije vrednosti mogu da prezentuju matematičke izraze sačinjene od nepoznate x, realnih brojeva, operacija sabiranja, oduzimanja, množenja, deljenja i stepenovanja kao i funkcija poput sinusa i kosinusa. Napisati funkciju koja izračunava ove izraze u zavisnosti od vrednosti nepoznate. Implementirati funkciju koja računa izvod matematičkog izraza. Implementirati funkciju koja određuje da li je izraz polinom. Implementirati funkciju koja pronalazi neodređen integral polinoma.

Zadatak 17. *Za ovaj zadatak je neophodno uraditi prethodni zadatak* Ako ste upoznati sa *LaTeX* jezikom za zapisivanje matematičke notacije, konstruišite funkciju

```
uLatex :: MatematičkiIzraz -> String
```

koja daje *LaTeX* kôd za dati izraz. Koristite *LaTeX* naredbe poput `\times`, `^`, `\frac`, `\left(` i `\right)`, i tako dalje...

Zadatak 18. Definisati tip `LogičkiIzraz` koji predstavlja logičke izraze sačinjene od logičkih vrednosti i operacija konjunkcije i disjunkcije. Uvesti tip `LogičkiIzraz` u `Show` klasu tipova. Implementirati funkciju `izracunajLogički :: LogičkiIzraz -> Bool` koja izračunava zadati logički izraz.

Zadatak 19. *Za ovaj zadatak je neophodno uraditi prethodni zadatak* Neka je data funkcija `f :: Int -> Bool` sa `f x = x /= 0`. Konstruisati funkciju `konvertuj :: Izraz -> LogičkiIzraz` koja aritmetički izraz prevodi u logički izraz tako što sabiranje prevodi u disjunkciju, množenje u konjunkciju a brojeve u skladu sa datom funkcijom `f`. Na primer, izraz `2 * (7 + 0)` se konvertuje u $\top \wedge (\top \vee \bot)$. Da li važi

```
f . izračunaj = izracunajLogički . konvertuj
```

18. Parseri

Parser je funkcija koja od niza nekih simbola generiše vrednosti. Najčešće parseri služe da se niske (liste karaktera) transformišu u neke vrednosti poput brojeva ili struktura podataka.

> **Primer 1.** Funkcija `read :: Read a => String -> a` koja "čita" vrednost iz niske, nije ništa drugo nego jedan parser.
>
> ```
> ghci> read "39.5" :: Float
> 39.5
> ghci> read "[True, False]" :: [Bool]
> [True, False]
> ```

> **Primer 2.** I sami smo konstruisali jedan parser u lekciji o listama. U pitanju je funkcija `uBroj :: [Char] -> Int` koja iz niske parsira brojeve.

Parsiranje je generalno težak problem. Stoga ne čudi što je tokom istorije računarstva razvijeno mnogo teorije o parserima i konkretnim tehnikama za njihovo konstruisanje. Svakako, teorija o parserima prevazilazi obime ove knjige, ali možemo ovde prikazati jednu tehniku za konstrukciju parsera. U pitanju su takozvani **parser kombinatori** koji su zapravo funkcije višeg reda kojima se jednostavniji parseri kombinuju u složenije.

18.1. Tip parsera

Parsiranjem bi trebalo nisku (koja je lista karaktera) pretvoriti u neku vrednost tipa `T`. Na primer, parser celobrojnih brojeva bi trebalo da nisku transformiše u vrednost tipa `Int`, pa bi tip ovakvog parsera bio `String -> Int`. Parser logičkih lista bi trebalo da nisku pretvori u vrednost tipa `[Bool]`, pa bi tip ovakvog parsera bio `String -> [Bool]`. Stoga ćemo definisati novi (apstraktni) tip

```
Parser a = String -> a,
```

koji može predstaviti parsere za vrednosti bilo kog tipa. Međutim, kroz ovaj novi tip je neophodno obezbediti još par stvari.

1. Prvo, nećemo svaki parser konstruisati kao jedinstvenu funkciju. Složenije parsere ćemo konstruisati kombinovanjem manjih funkcija. Svaka od tih funkcija parsiraće jedan karakterističan deo niske (Na primer, parser `Float` vrednosti može biti sačinjen od tri parsera: prvog koji parsira ceo deo, drugog koji parsira tačku, i trećeg koji parsira razlomljeni deo. Slično, parser logičkih nizova može biti sačinjen od parsera koji parsira zagrade, parsera za literale logičke vrednosti, parsera za zarez, parsera za beline, itd...). Parsere ćemo kombinovati tako što će svaki naredni parser, parsirati ono "što ostane" nakon pokretanja prethodnih parsera. Stoga jedan parser vrednosti `T` neće vraćati samo vrednost tipa `T`, već uređen par tipa `(T, String)`, gde druga koordinata označava ostatak niske koja se parsira.
2. Drugo, ne možemo očekivati da će svako parsiranje biti uspešno. Iz niske `"Hi!"` ne možemo parsirati broj niti logičku listu. Zbog toga, parser će zapravo vratiti možda-vrednost `Maybe (T, String)`. Ako je parser uspeo, rezultat je oblika `Just (x, s)` za neku nisku `s` i vrednost `x :: T`. U suprotnom, parser vraća `Nothing`.

Uzimajući prethodno u obzir, stižemo do definicije tipa

```
newtype Parser a = P (String -> Maybe (a, String))
```

Od sada, parser celih brojeva je tipa `Parser Int`, a parser lista logičkih vrednosti je tipa `Parser [Bool]`, itd...

Sa navedenom definicijom tipa biće zgodna funkcija koja "pokreće" parser nad niskom:

```
parsiraj :: Parser a -> String -> Maybe (a, String)
parsiraj (P p) s = p s
```

Kreirajmo parser koji parsira jedno slovo iz niske. Taj parser ima tip `Parser Char`. Sa konstruktorom `P` obuhvatićemo funkciju koja uzima jedno slovo sa početka niske. Ako je niska prazna, parser će vratiti `Nothing`.

```
slovo :: Parser Char
slovo = P f
    where
        f [] = Nothing
        f (x:xs) = Just (x, xs)
```

Lokalno definisana funkcija `f` vrši glavnu ulogu. Međutim, da bi dobili vrednost tipa `Parser Char`, neophodno je da "upakujemo" ovu funkciju u konstruktor `P`. Naravno, postavlja se pitanje zašto je bilo neophodno da definišemo novi tip a ne tipski sinonim koji koji ne bi zahtevao konstruktor. Razlog tome je što za novi tip možemo definisati instance klasa `Functor` i `Applicative`, što nije moguće sa tipskim sinonimom koji se odnosi na tip funkcije.

Testiranje parsera je veoma jednostavno (i veoma čitljivo!):

```
ghci> parsiraj slovo "Hello world!"
Just ('H', "ello world!")
ghci> parsiraj slovo ""
Nothing
```

Nakon slova, najprirodnije je kreirati parser cifre. Parser cifre treba da nam vrati numeričku vrednost cifre, ako ulazna niska počinje sa cifrom.

```
cifra :: Parser Int
cifra = P f where
    f []     = Nothing
    f (x:xs) =
        if x `elem` ['0' .. '9']
        then Just (fromEnum x - fromEnum '0', xs)
        else Nothing
```

Tip karaktera pripada klasi `Enum`, te je moguće koristiti funkciju `fromEnum` koja daje *Unicode* vrednost karaktera. Oduzimanjem te vrednosti od *Unicode* vrednosti karaktera `'0'`, dobijamo numeričku vrednost koju karakter cifre predstavlja.

```
ghci> parsiraj cifra "9abc"
Just (9, "abc")
ghci> parsiraj cifra "1234"
Just (1, "234")
ghci> parsiraj cifra "Hello!"
Nothing
```

> **Zadatak 1.** Kreirati funkciju `maloSlovo :: Parser Char` koja parsira samo mala latinična slova (tj. karakter iz raspona `['a' .. 'z']`).

Često je zgodno da kreiramo funkciju koja kreira nekakav parser na osnovu prosleđenih argumenata. Parser `simbol :: Char -> Parser Char` je primer takve funkcije. Funkcija `simbol` uzima jedan karakter i vraća parser koji parsira isključivo taj karakter, a za sve ostale karaktere podbacuje.

```
simbol :: Char -> Parser Char
simbol c = P $ \s -> case s of
    x:xs -> if x == c then Just (c, xs) else Nothing
    _    -> Nothing
```

```
ghci> parsiraj (simbol 'H') "Hello!"
Just ('H', "ello!")
ghci> parsiraj (simbol 'A') "Hello!"
Nothing
```

> **Zadatak 2.** Kreirati funkciju `niska :: String -> Parser String` koja na osnovu niske `s` kreira parser koji parsira isključivo `s`.

18.2. Kombinatori

Parseri koje smo do sad konstruisali su bili "atomički". Za kreiranje složenijih parsera, poput onih koji parsiraju čitav kôd nekog programskog jezika, potrebno je jednostavnije parsere ukombinovati u složenije. Takvo kombinovanje vršimo pomoću funkcija koje nazivamo **parser kombinatori**. Parser kombinatori o postojećih parsera prave nove parsere.

Verovatno najjednostavniji, netrivijalni, parser kombinator je kombinator koji na uspešan rezultat nekog parsera primenjuje datu funkciju `f`:

```
primeni :: (a -> b) -> Parser a -> Parser b
primeni f p = P $
        \s -> case parsiraj p s of
            Just (x, s') -> Just (f x, s')
            Nothing -> Nothing
```

> **Primer 3.** Imaćemo prilike da se uverimo u narednom primerima zašto je `primeni` koristan, a sada dajemo jedan jednostavan primer da bi demonstrirali kako ovaj kombinator funkcioniše. Primenom funkcije $f(x) = x^2$ na parser cifre, dobijamo parser koji sa početka niske parsira kvadrat cifre.
>
> ```
> ghci> kvadrat = primeni (\x -> x*x) cifra
> ghci> parsiraj kvadrat "3"
> Just (9, "")
> ```

Veoma koristan je parser kombinator *ili*, koji uzima dva parsera i kreira novi parser koji pokušava da pokrene prvi parser a u slučaju neuspeha i drugi parser. Ovakav kombinator ima sasvim jednostavnu definiciju:

```
ili :: Parser a -> Parser a -> Parser a
ili p1 p2 = P $
        \s -> case parsiraj p1 s of
            Just (v, s') -> Just (v, s')
            Nothing -> parsiraj p2 s
```

Primer 4. Koristeći kombinator `ili` možemo lako napraviti parser koji parsira samo simbole `x` i `o`:

```
ghci> iksoks = ili (simbol 'x') (simbol 'o')
ghci> parsiraj iksoks "xoxox"
Just ('x', "oxox")
ghci> parsiraj iksoks "aaaa"
Nothing
```

Kombinator `ili` je veoma zgodno postaviti u infiksni oblik. tako smo mogli da definišemo

```
iksoks = (simbol 'x') `ili` (simbol 'o').
```

Iako možda deluje neobično, ovakva sintaksa doprinosi čitljivosti jer zapis podseća na prirodne rečenice: *parsirati simbol "x" ili simbol "o".*

Zadatak 3. Napisati parser koji koji simbole `x` i `o` parsira kao logičke vrednosti `True` i `False`.

Do sada smo parsirali samo jedinstvene vrednosti. Sada ćemo konstruisati parser kombinator tipa `Parser a -> Parser [a]` koji parsira listu vrednosti istog tipa, tako što pokreće prosleđeni parser dokle god može. Nazovimo ovaj kombinator `više` jer rezultat (može) sadržati više vrednosti istog tipa. Definicija kombinatora mora sadržati rekurzivne pozive, što ćemo uspostaviti sa pomoćnom rekurzivnom funkcijom `r`:

```
više :: Parser a -> Parser [a]
više (P f) = P $ \s -> (Just r s)
    where
        r [] = []
        r xs = case f xs of
            Just (v, xs') -> v : r xs'
            Nothing       -> []
```

Zadatak 4. Konstruisati kombinator `zatim :: Parser a -> Parser b -> Parser (a , b)`, koji pokreće prvi a zatim i drugi parser, i vraća uređen par rezultat parsiranja.

Zadatak 5. Parser `više p` može da vrati i praznu listu kao uspešnu vrednost što često nije poželjno. Definisati kombinator `baremJedan :: Parser a -> Parser [a]`, koji mora da parsira barem jednu vrednost (tj, nikad neće vratiti praznu listu).

Rešenje. Umesto da direktno napišemo definiciju kombinatora, iskoristićemo postojeće kombinatore. Ako je dat parser `p :: Parser A`, tada parser

```
p `zatim` (više p)
```

parsira vrednost tipa `(A, [A])`. Stoga je dovoljno sa kombinatorom `primeni` spojiti ove dve ove koordinate u jednu listu sa funkcijom `\(x,xs) -> x:xs` (ovo je naravno funkcija `uncurry (:)`). Prema tome, možemo definisati

```
baremJedan p = primeni (uncurry (:)) (p `zatim` (više p)).
```

Sada kada smo se upoznali sa najvažnijim parser kombinatorima, možemo lako konstruisati jedan veoma koristan parser. U pitanju je parser celobrojnih vrednosti. Uspešnim pokretanjem parsera

```
više cifra
```

dobijamo vrednost tipa `[Int]` koja predstavlja cifre sa početka ulazne niske. Stoga je dovoljno napisati funkciju `cifreUBroj :: [Int] -> Int` koja će niz cifara svesti na broj. Naravno, ovde nam je potreban rekurzivan prolazak kroz listu, pri čemu ćemo čuvati "trenutnu vrednost" koja se pri svakom koraku uvećava 10 puta (tj za jedno decimalno mesto). Direktna implementacija[191] izgleda ovako:

```
cifreUBroj :: [Int] -> Int
cifreUBroj xs =  r 0 xs where
    r v []     = v
    r v (x:xs) = r (10 * v + x) xs
```

Sada, parser `broj :: Parser Int` može da se definiše kao

```
broj = primeni cifreUBroj $ više cifra.
```

```
ghci> parsiraj broj "123abc456"
Just (123, "abc456")
ghci> parsiraj broj "abc"
Nothing
```

Umesto `više` možemo da iskoristimo kombinator `baremJedan` da bi dobili `Nothing` ako ulazna niska ne počinje sa brojem.

> **Zadatak 6.** Konstruisati parser celobrojnih vrednosti bez korišćenja kombinatora.

18.3. Aplikativni parseri

Tip funkcije

```
primeni :: (a -> b) -> Parser a -> Parser b
```

neodoljivo podseća na tip metode

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

sa kojom smo se upoznali i lekciji o funktorima. Kako je i tip `Parser` jedan apstraktni tip vrste `* -> *` ima smisla instancirati `Functor Parser` koristeći istu definiciju koju smo dali za `primeni`

```
instance Functor Parser where
    fmap f p = P $
        \s -> case parsiraj p s of
            Just (x, s') -> Just (f x, s')
            Nothing -> Nothing
```

Rezultat "mapiranja" funkcije `f` na neki parser `p`, je novi parser koji na uspešan rezultat parsiranja parsera `p` primenjuje funkciju `f`.

Sada je neophodno dokazati da ovako data definicija zadovoljava dva zakona.

Prvo, za svaki parser `p` mora važiti da je

```
fmap id p = p.
```

Kako je parser `p` oblika `P p'` za neku funkciju `p'`, važi sledeće

[191] Jednostavnije, funkciju `cifreUBroj` smo mogli da definišemo i kao

```
foldl (\a x -> 10*a + x) 0
```

odnosno još kraće

```
foldl ((+) . (*10)) 0
```

```
fmap id p = fmap id (P p')
= P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 (id x, s'); ℕ -> ℕ
= P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 (x, s'); ℕ -> ℕ
= P p' = p,
```

gde smo sa `𝕁` i `ℕ` skraćeno označili konstruktore `Just` i `Nothing`. Ovim je prvi zakon dokazan.

Za dokaz drugog zakona, neka su date funkcije `f :: A -> B` i `g :: B -> C`. Tada za svaki parser `p = P p' :: Parser A` važi

```
fmap (g . f) p = fmap (g . f) (P p')
= P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 ((g . f) x, s'); ℕ -> ℕ
= P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 (g (f x), s'); ℕ -> ℕ
= fmap g $ P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 (f x, s'); ℕ -> ℕ
= fmap g $ fmap f $ P $ \s -> case p' s of 𝕁 (x, s') -> 𝕁 (x, s'); ℕ -> ℕ
= fmap g $ fmap f $ P p'
= (fmap g . fmap f) p,
```

čime je dokazan i drugi zakon funktora.

Sledeći korak je implementirati instancu `Applicative Parser`. Za metodu `pure :: a -> Parser a`, nemamo mnogo izbora: to će biti "konstantan" parser koji ne parsira ništa. Za metodu

```
<*> :: Parser (a -> b) -> Parser a -> Parser b
```

takođe nemamo mnogo opcija. Ako nam je dat parser `p1 :: Parser (A -> B)` i parser `p2 :: Parser A`, sve što možemo da uradimo da bi dobili vrednost `B` je da "pokrenemo" parser `p1` i onda sa dobijenim vrednostima "pokrenemo" parser `p2`. Naravno, ako neki od parsera ne uspe, tada ni "kombinacija" ovih parsera ne bi trebalo da uspe.

```
instance Applicative Parser where
    pure x = P $ \s -> Just (x, s)

    p1 <*> p2 = P $ \s -> primeni (parsiraj p1 s) p2
        where
            primeni Nothing       _ = Nothing
            primeni (Just (f,s')) p = parsiraj (fmap f p) s'
```

Pomoćna funkcija `primeni`, proverava da li je parsiranje sa `p1` uspelo. Ako jeste uspelo, tada se rezultat `(s', f)` prosleđuje parseru `p2`, tako što se parser `fmap f p1` pokreće nad ostatkom `s'`.

> **Zadatak 7.** Dokazati da navedena definicija poštuje zakone aplikativnih funktora.

Parseri koji parsiraju funkciju (tj. vrednosti tipa oblika `Parser (A -> B)`) su na prvi pogled veoma čudan pojam. Šta uopšte znači parsirati funkciju iz niske karaktera? Ali, setimo se koja je bila motivacija za uvođenje aplikativnih funktora: aplikacija funkcija više promenljiva na vrednosti u funktorima. Od sada koristeći `pure` i `<*>` možemo kombinovati parsere koristeći proizvoljne funkcije.

> **Primer 5.** Kreirajmo parser koji parsira jedno slovo, zatim cifru i zatim te dve vrednosti kombinuje u uređeni par. Iz prethodnih primera i zadataka imamo parsere `slovo :: Parser Char` i `cifra :: Parser Int`, a želimo parser `slovoCifra :: Parser (Char, Int)`.
>
> Funkcija

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

kombinuje dve vrednosti u uređen par[192]. Stoga, vrednost `pure f :: Parser (a -> b -> (a, b))` je moguće sa operatorom[193] `<*>` primeniti na dve vrednosti tipa `Parser A` i `Parser B`, da bi dobili vrednost `Parser (A, B)`. I zaista

```
ghci> f = \x y -> (x, y)
ghci> slovoCifra = f <$> slovo <*> cifra
ghci> parsiraj slovoCifra "D7"
Just (('D',7), "")
```

Idealno za parsiranje oznaka šahovskih polja!

Kako operatori `<*` i `*>` funkcionišu u kontekstu parsera? Po definiciji, operator

```
(<*) :: Parser a -> Parser b -> Parser a
```

je definisan kao

```
p1 <* p2 = liftA2 const p1 p2
```

što se svodi na

```
p1 <* p2 = (fmap const p1) <*> p2.
```

Ako je `p1 :: Parser A`, tada je `fmap const p1 :: Parser (a -> b)` parser koji je moguće primeniti sa `<*>` na bilo koji drugi parser, ali će parsirana vrednost zavisiti samo od parsera `p1`, dok će parsirana vrednost parsera `p2` biti ignorisana, pod uslovom da parser `p2` uspe. Ako parser `p2` nije uspešan, tada će i parser `p1 <* p2` podbaciti. Slično je i sa operatorom

```
(*>) :: f a -> f b -> f b
```

koji ignoriše rezultat levog parsera (pod uslovom da je taj parser uspeo).

Navedeni operatori nam omogućavaju da konstruišemo parsere koji moraju parsirati neke delove niske, ali ti delovi ne utiču na parsiranu vrednost. Primer za ovu situaciju je parsiranje brojeva sa decimalnom tačkom: tačka se mora naći u zapisu decimalnog broja, ali ne utiče nikako na vrednost tog broja.

Primer 6. Imitirajući ideju o rekurzivnoj primeni parsera, možemo na jednostavan način implementirati funkciju

```
niska :: String -> Parser String
```

koju smo spomenuli u prethodnom zadatku.

Prvo, parsiranje prazne niske iz bilo koje druge niske uvek treba da uspe. Ovo predstavlja bazni slučaj. Drugo, parsiranje neke niske `x:xs` iz niske `s` može da se izvede tako što se prvo parsira simbol `x`, pa ako to parsiranje uspe, rekurzivno se parsira i ostatak `xs`:

```
niska :: String -> Parser String
niska ""     = P $ \s -> Just ("", s)
niska (x:xs) = P $ \s -> case parsiraj (simbol x) s of
    Just (_, s') -> parsiraj ((x:) <$> niska xs) s'
    Nothing      -> Nothing
```

[192] Ovu funkciju kraće možemo zapisati kao konstruktor uređenog para `(,)`.
[193] Podsećamo da je `<*>` levoasocijativan operator.

```
ghci> parsiraj (niska "True") "Truehelloworld"
Just ("True","helloworld")
ghci> parsiraj (niska "False") "Falsebyeworld"
Just ("False","byeworld")
ghci> parsiraj (niska "hello") "bye"
Nothing
```

Zadatak 8. Definisati parser `decimalanBroj :: Parser Float` koji parsira broj oblika

$$\overline{a_1 \cdots a_n . b_1 \cdots b_m}.$$

Zadatak 9. Definisati parser

```
logičkeListe :: Parser [Bool]
```

koji parsira liste logičkih vrednosti `True` i `False`. Na primer, parser bi trebalo da ispravno parsira nisku `"[True,False]"`. Pretpostaviti da u nisci neće biti belina.

Zadatak 10. Definisati parser

```
preskočiBeline :: Parser ()
```

koji samo preskače karaktere beline `' '`, `'\n'` i `'\t'`.

Zadatak 11. Ponovo definisati parser `logičkeListe :: Parser [Bool]` ali tako da ispravno parsira niske u kojima se nalaze beline između logičkih vrednosti i zareza ili zagrada. Na primer, parser bi trebalo da ispravno parsira nisku poput

```
"[ True, False ]".
```

18.4. Parsiranje matematičkih izraza

U lekciji o rekurzivnim strukturama podataka, definisali smo tip `Izraz` kao algebarski tip

```
data Izraz = Broj Int | Zbir Izraz Izraz | Proizvod Izraz Izraz.
```

Takođe smo i definisali instancu `Show Izraz` na sasvim jednostavan način.

```
instance Show Izraz where
    show (Broj a) = show a
    show (Zbir a b) = "(" ++ show a ++ "+" ++ show b ++ ")"
    show (Proizvod a b) = "(" ++ show a ++ "*" ++ show b ++ ")"
```

Kako to obično biva, obrnuti proces, proces čitanja strukture iz niske, je značajno komplikovaniji za implementaciju, ali nam tehnika aplikativnih parsera može nam pomoći u ovom slučaju. Sa parserom kog budemo konstruisali, moći ćemo da iz niske pročitamo vrednost `Izraz`.

Pre nego što počnemo sa definisanjem parsera, moramo tačno definisati oblik niska koje ćemo parsirati. Na primer, složeni parseri izraza mogu ispravno parsirati izraze poput `"2 + 5"` ili `"2*-7+6"` jer su napravljeni tako da ignorišu beline (razmake), da vode računa o prioritetu operacija, podržavaju unarni operator negacije, itd... Da bismo skratili naredne redove, mi nećemo kreirati ovako složeni parser. Parser koji budemo konstruisali moći će da parsira samo one izraze koji su nastali sa funkcijom `show :: Izraz`

`-> String` koju smo gore definisali tj, izraze sačinjene isključivo od cifara i znakova `+`, `*`, `(`, `)`, pri čemu se oko svakog podizraza nalaze zagrade (osim oko samih brojeva).

Kako je i sama struktura tipa `Izraz` rekurzivna, i parser tipa `Parser Izraz` biće rekurzivan. Posebno ćemo konstruisati parser `izrazBroj` koji parsira brojeve, parser `izrazZbir` koji parsira zbir dva manja izraza, i parser `izrazProizvod` koji parsira proizvod dva manja izraza.

Pretpostavimo da su navedeni parseri definisani. Jedan izraz može biti broj, proizvod ili zbir manjih izraza. Zbog toga, da bi parsirali izraz, prvo ćemo pokušati da parsiramo broj. Ako to ne uspemo, onda ćemo pokušati da parsiramo zbir, a ako ni to ne uspemo pokušaćemo da parsiramo proizvod[194]. Sa kombinatorom `ili` lako dobijamo željeni parser:

```
izraz :: Parser Izraz
izraz = izrazBroj `ili` izrazZbir `ili` izrazProizvod
```

Ostaje još definisati `izrazBroj`, `izrazZbir` i `izrazProizvod`. Parser `izrazBroj` ćemo lako definisati pomoću parsera `broj :: Parser Int` kog posedujemo od ranije. Sve što je potrebno je da parsiranu vrednost (koja je tipa `Int`), transformišemo u vrednost tipa `Izraz`, što se naravno postiže podizanjem konstruktora `Broj :: Int -> Izraz` na nivo funktora `Parser`:

```
izrazBroj :: Parser Izraz
izrazBroj = fmap Broj broj
```

Parser `izrazZbir` mora da funkcioniše na sledeći način: prvo je neophodno parsirati simbol za levu zagradu, zatim je potrebno parsirati levi sabirak, pa simbol plus, desni sabirak, i na kraju simbol desne zagrade. Rezultati parsiranja simbola nisu bitni (ako su uspešni), a rezultate parsiranja levog i desnog sabirka moramo ukombinovati u vrednost oblika `Zbir x y :: Izraz`. Naravno, u tome će nam pomoći podizanje konstruktora `Zbir`. Takođe, kako su oba sabirka izrazi sami za sebe, potrebno ih je parsirati sa `izraz` parserom. Ovo uspostavlja rekurziju.

```
izrazZbir :: Parser Izraz
izrazZbir =
    simbol '(' *>
    (Zbir <$> (izraz <* simbol '+') <*> izraz)
    <* simbol ')'
```

Parsiranje proizvoda je u potpunosti analogno parsiranju zbira:

```
izrazProizvod :: Parser Izraz
izrazProizvod =
    simbol '(' *>
    (Proizvod <$> (izraz <* simbol '*') <*> izraz)
    <* simbol ')'
```

Testiranjem se uveravamo da parser funkcioniše:

[194] Pritom, ovde uopšte nije bitno kojim redom pokušavamo pa pokrenemo parsere. U složenijim parserima, poredak je često bitan

```
ghci> parsiraj izraz "(1+2)"
Just ((1+2), "")
ghci> parsiraj izraz "((10+2)*32)"
Just (((10+2)*32), "")
```

Vrednost koju vidimo u prvoj koordinati je sintaksno stablo. Zbog načina na koje smo definisali `show`, ovo sintaksno stablo se prikazuje na sasvim jednostavan način.

Sadržaj knjige se redovno unapređuje i dopunjuje. Poslednja verzija je dostupna na haskel.ubavic.rs.

Generisano 07.09.2026 godine.