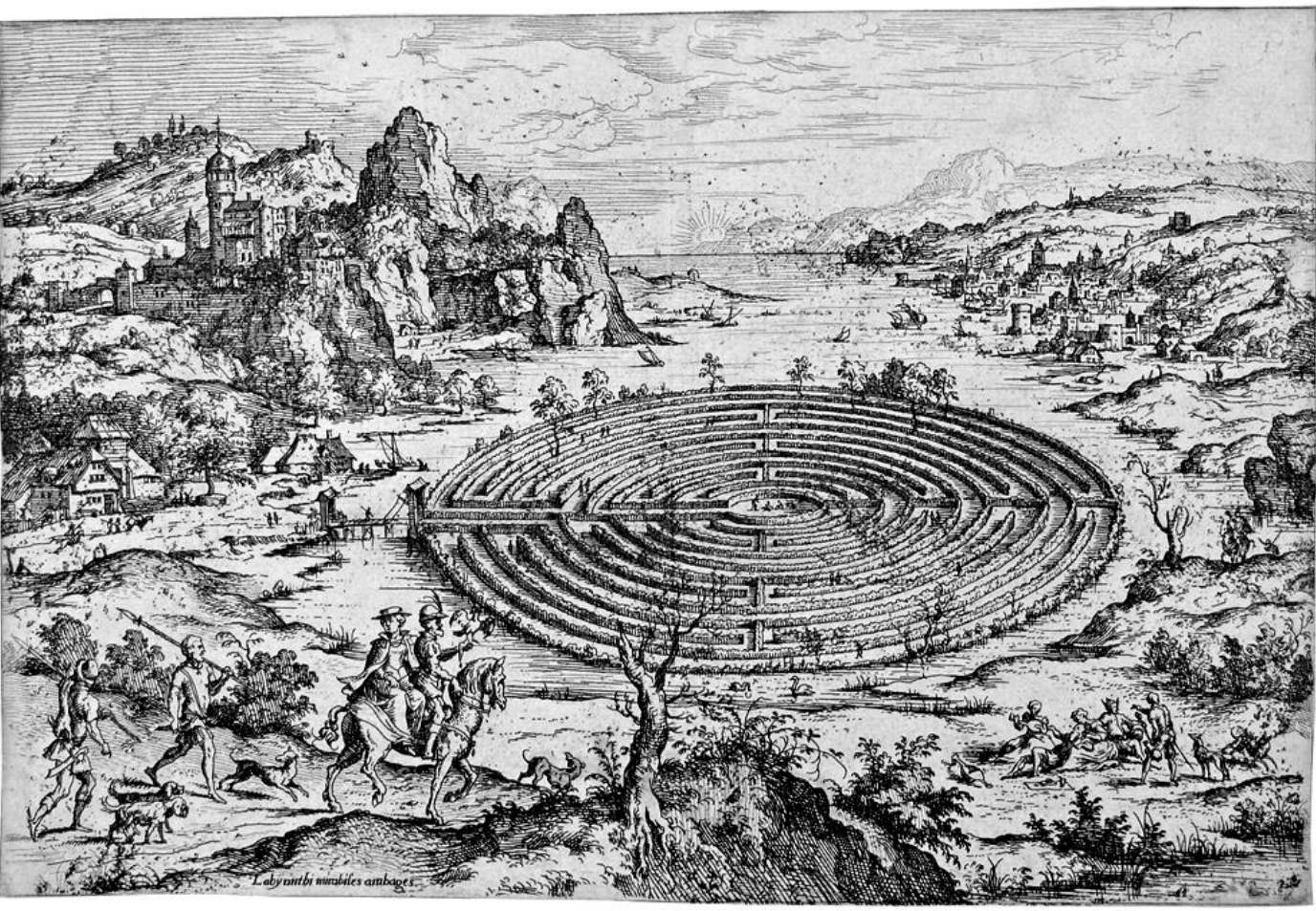


UVOD *u* PROGRAMIRANJE

MILAN SEGEDINAC

UVOD U PROGRAMIRANJE

Milan Segedinac

Naslov:	Uvod u programiranje
Jezik:	Srpski
Verzija:	1.0.0
Autor:	Milan Segedinac
Godina izdanja:	2025.
Autorska prava:	Copyright © 2025. Milan Segedinac. Sva prava zadržana.
Licenca:	Ovo delo je licencirano pod Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0).
Pravni tekst licence:	https://creativecommons.org/licenses/by-nc-nd/4.0/
Ukratko, dozvoljeno je:	Kopirati i deliti delo u originalnoj formi uz obavezno navođenje autora.
Zabranjeno je:	Menjati, prerađivati ili koristiti delo za komercijalnu dobit.
Slika na koricama:	Hijeronim Kok, Kritski lavirint, 1551. – 1558.

Ova knjiga je posvećena Zori Konjović.

Zahvaljujem se Zori Konjović, Jovani Vidaković, Milanu Vidakoviću i Goranu Saviću na dragocenim komentarima i pomoći tokom pisanja knjige.

Takođe, zahvaljujem se Milanu Vidakoviću i Branku Milosavljeviću, profesorima na čijim predmetima sam bio asistent kada sam počinjao da radim. Ova knjiga je, u velikoj meri, rezultat iskustva koje sam tada stekao.

Posebno se zahvaljujem Lazaru Velickom, bez koga ova knjiga ne bi bila završena.

Sadržaj

Uvod – šta je programiranje i kako ga učiti?

- Program, programiranje i računar
- Principi kompozabilnosti, apstrakcije i parametrizacije u programiranju
- Imperativno i deklarativno programiranje

"Svi mi imamo dva života: *istinski* koji snivamo u detinjstvu, i sanjamo ga i dalje, kao odrasli, kroz neku izmaglicu; i *lažni* koji živimo u zajedništvu s drugima i koristimo u praktične svrhe, onaj gde nas na kraju strpaju u neki sanduk."

Fernando Pesoa, Knjiga nespokoja

"Od svega korisnog što su me učili pobegao sam kroz dvorišni prozor."

Fernando Pesoa, Trafika

Učenje programiranja

Programiranje je, nema sumnje, veština koju je poželjno savladati. Ali oko samog savladavanja programiranja postoje izvesne nesuglasice. Jedni nam obećavaju da ćemo naučiti da programiramo u programskom jeziku po izboru za 24 sata, a drugi nam govore da nam trebaju decenije posvećenog rada da bismo postali programeri. I jedni i drugi su, u stvari, u pravu: zaista vrlo brzo možemo naučiti neke stvari iz programiranja, ali je potrebno mnogo vremena da bismo postali *pravi* programeri. Ipak, nemojte da vas to brine: programiranje postaje zanimljivo čim počnemo da učimo da programiramo!

Odakle krenuti?

Ukoliko rešimo da naučimo da programiramo, postavlja nam se pitanje odakle da krenemo? I gotovo uvek se to pitanje javlja u sledećem obliku. Da li je najbolje da prvo počnemo da učimo programski jezik C ili Java, ili je Python pravi programski jezik za početnike? Ili možda treba da počnemo da učimo HTML i CSS, pa onda pređemo na JavaScript? Ova pitanja su po svojoj prirodi pogrešna: učenje programiranja nije isto što i učenje programskog jezika.

Programski jezik – prozor u koncepte programiranja

Ako već umemo da programiramo, učenje novog programskog jezika je jednostavno; ako ne umemo, učenje bilo kog programskog jezika je bespredmetno – čemu će nam služiti naučeno? Novi programski jezik ima smisla da učimo jedino ako će promeniti način na koji *gledamo* na programiranje. Naravno, i kada počinjemo da učimo da programiramo, samo učenje programiranja mora pratiti i učenje programskog jezika u kom će koncepti

programiranja biti ilustrativni: programski jezik je, u tom slučaju, prozor u koncepte programiranja. Shodno tome, iako ovu knjigu prate ilustracije u programskom jeziku Haskell, ona nikako nije kurs Haskella!

Šta je programiranje? – pisanje uputstava za rešavanje problema

Pređimo sada na predmet ove knjige i hajde da vidimo *šta je programiranje*. Program je *uputstvo za rešavanje problema*, a programiranje je veština pisanja programa, odnosno *pisanje uputstava za rešavanje problem*. Termin program se koristi u tom smislu i u drugim domenima. Na primer, *studijski program* na fakultetu kaže koje predmete student treba da položi da bi postao softverski inženjer (ovde je problem koji se rešava: Kako postati softverski inženjer?). Ili *partijski program* određuje na koji način neka politička partija predlaže da se reši problem vođenja države.

Uputstvo za rešavanja problema sastoji se od pravila koja treba primeniti da bi se neki problem rešio. Na primer, problem pripreme ručka može se rešiti primenom pravila koja definiše recept za jelo.

Pravila toliko jednostavna da ih čak i mašina može izvršiti

Pravila rešavanja problema mogu se razbiti na manja potpravila. Potpravila se mogu dalje razložiti na još manja potpravila i tako sve dok ne stignemo do najsitnijih pravila, takozvanih *primitivnih* odnosno *prostih* pravila, koja *ne zahtevaju nikakvo razmišljanje*, odnosno koja su toliko jednostavna da ih *čak i mašina* (odnosno računar) *može izvršiti*.

Računar – univerzalna mašina za rešavanje problema

Računar je, onda, *mašina za rešavanje problema*. Kojih problema? Pa onih problema za koja ima uputstva kako se rešavaju, odnosno svih onih problema za koje smo napisali programe. A za koje probleme možemo napisati programe? Pretpostavka na kojoj se zasniva programiranje je da *svaki problem koji možemo rešiti može da se razloži na takva elementarna pravila*. Odnosno, za svaki problem koji može da se reši možemo napisati program koji ga rešava.

Svaki poziv uključuje rešavanje problema

Ali programiranje nema ekskluzivna prava na rešavanje problema, svaki poziv uključuje rešavanje problema. Obućar rešava probleme popravke cipela, pekar rešava probleme pripreme hleba, a lekar rešava probleme lečenja ljudi. Problemi koje oni rešavaju toliko su raznorodni, da se postavlja pitanje *da li možemo da nađemo principe koji su dovoljno generalni da se koriste za rešavanje svih problema*, bez obzira na to kom domenu ljudske delatnosti problemi pripadaju.

Hajde da pokušamo tako što ćemo posmatrati probleme koje svakodnevno rešavamo.

Dekompozicija

Složene probleme razbijamo na jednostavne

Mali broj problema sa kojima sa susrećemo možemo neposredno da rešimo: najčešće problem na koji smo naišli moramo da razbijemo na potprobleme, pa onda te potprobleme na njihove potprobleme dok ne dođemo do jednostavnih problema, koje možemo direktno da rešimo (odnosno, kao što smo videli, koji su toliko jednostavni da ih čak i mašina može rešiti). Zatim rešenja tih jednostavnih problema spajamo i dobijamo rešenje originalnog složenog problem. Na primer, rešili smo da otputujemo na more. Prvo moramo da rezervišemo smeštaj, pa onda da kupimo avionske karte. (Zaboravili smo da napomenemo da pre svega ovoga moramo zaraditi novac za putovanje.) Pa, veče pre putovanja, moramo da se spakujemo. Vidimo da je akcija „uzeti pasoš iz fioke i staviti ga u torbu sa dokumentima“ tek mali deo rešenja velikog problema „odlaska na more“, i da konkretne akcije koje preduzimamo dobijaju smisao samo u svetlu širih problema koje rešavamo. Kažemo da složene probleme *dekomponujemo*. Ovako dobijene komponente rešenja mogu se kombinovati na različite načine da bismo dobili rešenja različitih problema. Na primer, kupovina avio karata može da bude deo rešenja problema odlaska na more, ali i deo rešenja problema službenog putovanja.

Kompozabilnost

Iste komponente se mogu kombinovati na različite načine da bi se dobila rešenja različitih problema

Pored toga, mali broj problema je takav da sami možemo da ih rešimo cele. Na primer, ako smo odlučili da otputujemo na more, oslonićemo se na avio-prevoznika i jedine tačke dodira koju ćemo imati sa avio-prevoznikom će biti rezervacija karata i samo putovanje. Neće nas interesovati nabavka goriva za avion, redovni servisi aviona, zapošljavanje osoblja i nabavka hrane koje se servira u avionu. Kažemo da smo te detalje *apstrahovali*. Ako ne bismo primenjivali princip apstrakcije, teško da bismo mogli da rešimo bilo koji problem. Zamisli samo koliko bi, bez principa apstrakcije, bio težak problem pripreme doručka: morali bismo sami da uzgojimo pšenicu; pa da sameljemo brašno; napravimo testo sa starterom jer nemamo kvasac; odvojimo deo testa kao starter za naredni hleb; sakupimo drva; ispečemo hleb u peći koju smo sami sagradili od cigala koje smo sami napravili... (Koliko bi komplikovano tek bilo da na taj način dođemo do putera za doručak?) Pored principa *dekompozicije* (problem razbijamo na manje potprobleme) prilikom rešavanja problema koristimo i princip *apstrakcije*: neke detalje

Apstrakcija

Od latinske reči *abstractus*, što znači odvučen, odvojen, udaljen

Proces zanemarivanja detalja

rešavanja problema prosto prepuštamo drugima da reše, odnosno *zanemarujemo detalje koji su za nas nevažni*. Kada programiramo, nužno koristimo apstrakciju – ako ništa drugo, apstrahovaćemo način na koji računar izvršava primitivna pravila. Na primer, kada programiramo u programskom jeziku Haskell, nećemo se baviti pitanjem kako radi sabiranje, nego ćemo prihvatiti da je to primitivna operacija koju računar prosto 'ume' da uradi.

Parametrizacija i uopštavanje

Kako bi izgledalo uputstvo za kuvanje čaja? Zagrejemo vodu, sipamo je u šolju i ubacimo kesicu čaja. A kako bi izgledalo uputstvo za kuvanje kafe? Zagrejemo vodu, sipamo je u šolju i dodamo instant kafu. Ova dva uputstva su jako slična, razlikuju se samo u tome što u slučaju čaja u vrelu vodu dodamo kesicu čaja, a u slučaju kafe dodamo instant kafu. Mogli bismo napraviti uopšteno uputstvo: „zagrej vodu, sipaj je u šolju i dodaj X", pa da iz njega izvedemo konkretna uputstva tako što u prvom slučaju slovo X zamenimo rečima „kesicu čaja", a u drugom slučaju rečima „instant kafu". U tom slučaju primenili smo princip parametrizacije, a X se naziva parametar. Parametrizacija nam omogućuje da, umesto da pravimo uputstva samo za pojedinačne slučajeve, pravimo uputstva koja se odnose na široke skupove problema.

Principi programiranja

Kompozabilnost, apstrakcija i parametrizacija su tri najvažnija alata u rešavanju problema i, shodno tome, tri osnovna principa programiranja. Ova knjiga će stoga posmatrati programiranje u svetlu ova tri principa.

Šta obuhvata učenje programiranja?

Skup principa koji se koristi prilikom rešavanja problema je mali, ali je skup načina na koji se pišu uputstva za rešavanje problema velik. Neka uputstva su izuzetno detaljna (kažemo *niskog nivoa apstrakcije*), neka su vrlo apstraktna (*visokog nivoa apstrakcije*). Neka nam govore koje korake da preduzmemo da bi problem bio rešen, neka samo kako rešenje treba da izgleda. Na primer, recept nam govori koje sastojke, na koji način i u kom redosledu treba da upotrebimo da bismo napravili kolače. Ili uputstvo iz IKEAe nam govori koje korak i u kom redosledu treba da preduzmemo da bismo sklopili policu. Isto tako, neki programi su *skupovi instrukcija koje treba da se sprovedu korak po korak*. Često ćete u literaturi naići na ovu vrstu programa. Takve programe zovemo *imperativnim*. Programski jezici Java, C, C++, Python, C# i JavaScript su programski jezici namenjeni za pisanje imperativnih programa. Međutim, ta

Imperativno i deklarativno programiranje

definicija programa je preuska: nisu sva uputstva zadata kao serije instrukcija. Na primer, plan za izgradnju kuće ne kaže kako i kojim redosledom treba da se slažu cigle, nego opisuje (obično vizuelno) konfiguraciju kuće (raspored i dimenzije prostorija), koliko treba da budu debeli zidovi i gde da budu otvori za vrata i prozore. Ili lista za kupovinu ne kaže u koju prodavnicu treba ići i kojim redosledom uzimati robu sa rafova, nego samo nabraja šta treba uzeti. Ovakva uputstva su *deklarativni programi*. Postoje programski jezici poput Prologa koji su namenjeni za pisanje deklarativnih programa. Ova knjiga se bavi deklarativnim programiranjem u programskom jeziku vrlo visokog nivoa apstrakcije – programskom jeziku Haskell.

Ova knjiga je na prvom mestu namenjena svima koji se prvi put susreću sa programiranjem, bez obzira na uzrast. Potrebno je samo da poseduju znanja koja se stiču osnovnim obrazovanjem .Voleo bih da bude jednako korisna učeniku viših razreda osnovne škole kao i osobi srednjih godina koja je rešila da promeni profesiju i postane programer. U stvari, ambicija mi je da napišem knjigu koja će nekima biti *prva knjiga iz programiranja*. Na drugom mestu, ova knjiga je namenjena programerima koji žele da „ojačaju temelje“. Hteo bih i da naglasim šta da ne očekujete od ove knjige: ova knjiga, sama za sebe, najverovatnije neće pomoći učenicima da dobiju višu ocenu iz programiranja. Neće biti ni adekvatna literatura za pripremu za takmičenja nadarenim učenicima. Studentima neće pomoći da lakše polože predmete koji uključuju programiranje. Neće vam pomoći ni da se pripremite za intervju za pozicije programera. Broj tema koje će ova knjiga pokriti je mali i neće vam dati ono što je potrebno samo za potpuno savladavanje veštine programiranja. Ali je skup tema istovremeno obuhvatan, njen sadržaj i način prezentacije treba čitaocu da obezbede fundamentalna znanja o programiranju koja su neophodna za dublje razumevanje programiranja i dalje razmišljanje o programiranju.

Šta da (ne) očekujete od ove knjige?

Stoga se nadam da će svoje čitaoce pokrenuti na jedno dugo putovanje kroz programiranje. Ukoliko se to tebi ne desi, nemoj odustajati: nije do tebe, do knjige je. Pređi na drugu, treću, dok ne naiđeš na onu pravu. A i kada „naiđeš na onu pravu“, budi svestan da je ono što su ti dale prethodno pročitane knjige uticalo na tvoju spoznaju „one prave“ i da ćeš morati da čitaš sve nove i nove knjige da bi „ostao u formi za programiranje“ .

Putovanje kroz programiranje

Struktura knjige

Knjiga je osmišljena tako da priču o programiranju prate pitanja i zadaci. Da bi razumeo knjigu, potrebno je da ove vežbe radiš uporedo sa čitanjem knjige, odnosno da ne nastavljaš sa čitanjem dok ne (pokušaš da) rešiš zadatke. Neki zadaci su jednostavni, neki su teški. Ima i trik pitanja, za koja sam se potrudio da ih maskiram da izgledaju naivno, koja su osmišljena da te navedu da uvidiš nelogičnosti koje se ne mogu objasniti onim što smo do tada naučili, pa nam treba da uvedemo nove koncepte.

Kurs

Idealno bi bilo da ovu knjigu prati kurs u kome grupa polaznika približno istog nivoa znanja, početnika, prolazi kroz knjigu, a da kurs vodi iskusan *programer* (ne nužno nastavnik) koji ume da prepozna na šta vežbe „ciljaju“. Tada bi svaki problem trebalo diskutovati u grupi, a programer koji vodi kurs bi umeo da usmeri diskusiju u dobrom smeru. Naravno, mislim da knjigu vredi čitati i ukoliko ne postoji mogućnost organizovanja ovakvog kursa.

Kritike

Odmah ću te upozoriti: ova knjiga je ostrašćena i iznosi lične stavove autora. Očekujem kritike, pre svega od strane već formiranih programera: da se *knjiga ne bavi pojmovima koji se svakodnevno koriste u programerskoj praksi* (kao što su, na primer, varijable ili kontrola toka programa ili pojmovi objektno orjentisanog programiranja); da je previše *apstraktna*; da je *anahrona*; da je *beskorisna*; i naravno, da *to nije programiranje*[1]. To je u redu: ova knjiga ne govori šta programiranje *jeste*, nego šta bi programiranje *moglo da bude*. Ona rasvetljava jednu ideju, jedan pogled na programiranje, a kada je jedna strana predmeta osvetljena, druge moraju ostati u senci.

[1] Pogotovo očekujem pitanje *zašto danas, kada veštačka inteligencija piše programski kod, učiti programiranje*? Upravo zbog toga što se programski kod može generisati ova knjiga nije kurs programskog jezika (na primer programskog jezika Haskell), nego rasvetljava osnovne principe programiranja. Programiranje nikada nije ni bilo (samo) pisanje programskog koda, nego je oduvek bilo razumevanje i opisivanje sveta u kome živimo, a to će još dugo biti zadatak čoveka. Veštačka inteligencija nam je, naizgled paradoksalno, pomogla da tu činjenicu uvidimo. Zato ohrabrujem kritičko korišćenje veštačke inteligencije kao pomoćnog alata u učenju programiranja.

Pitanja i zadaci

- Nabroj primere uputstava za rešavanje problem sa kojima se susrećeš u svakodnevnom životu. Za svaki od primera koji si naveo prepoznaj da li je imperativan (govori koje korake treba da preuzmemo da bi rešili problem) ili je deklarativan (govori šta treba da se uradi).

Programi

- Koje sve korake je potrebno obaviti da bi se:
  - Zamenile knjige u biblioteci;
  - Pripremio ručak;
  - Otišlo sa prijateljima u bioskop.

  Jedan zajednički korak u ovim problemima je prevoz do lokacije (prevoz do biblioteke, prevoz do prodavnice, prevoz do bioskopa). Koje još zajedničke korake imaju ovi problemi?

Dekompozicija problema

- Koje od koraka iz navedenih problema obavljaš ti, a u kojima učestvuju i drugi i kako?

Apstrakcija

- Diskutuj u grupi do kojih zaključaka si došao? Šta znači da se isti problem može rešiti na puno načina? Da li svi ljudi probleme dekomponuju na iste potprobleme? Da li se svi apstrahuju od istih problema?

Diskusija

- Identifikuj probleme koje svakodnevno obavljaš, a koji se mogu parametrizovati.

Parametrizacija

- Diskutuj u grupi do kojih zaključaka si došao.
- Napravi plan za rođendansku proslavu za druga ili drugaricu. Plan treba da ima sledeće elemente:
  1. *Dekompozicija problema* – koje sve korake treba preduzeti da bi se organizovao rođendan. Na primer, napraviti spisak gostiju, pripremiti pozivnice, ...
  2. *Apstrakcija* – koje od koraka trebaš da osmisliš sam, a koje ćeš dati nekom drugom da uradi. Na primer, za pravljenje torte ne moraš sam da daš recept, nego taj korak možeš prepustiti poslastičaru.
  3. *Parametrizacija* – koji koraci zavise od konkretnih vrednosti koje se mogu zameniti parametrima. Na primer, nabavka pića je korak koji zavisi od broja gostiju na rođendanu.

Planiranje rođendanske proslave

- Pretpostavka na kojoj se zasniva programiranje je da se problemi mogu dekomponovati do pravila koja su toliko jednostavna da ne zahtevaju (ljudsko) razmišljanje, već ih može rešiti i mašina. Šta misliš, da li je to zaista tako? Argumentuj svoj stav u grupi.

Mašina za rešavanje problema

Izrazi

- Prost izrazi
- Složeni izrazi
- Prioritet operatora
- Konstante
- Program kao složeni izraz

U ovom poglavlju ćemo odgovoriti na pitanje kako se izvršava računarski program. Da bismo odgovorili na ovo pitanje, počećemo sa pisanjem jednostavnog programa. Prvo ćemo pokrenuti GHCI[2]. Hajde da probamo da unesemo broj – to je najjednostavniji program koji možemo da napišemo. Taj broj koji smo uneli biće ispisan u redu ispod.

```
> 1
1
> 5
5
> 100
100
> -10
-10
```

Prost izrazi

Zapis vrednosti

Uneli smo 1 i dobili smo broj 1. Uneli smo 5 i dobili smo broj 5. Uneli smo 100 i dobili smo broj 100. Uneli smo -10 i dobili smo broj -10. Programsko okruženje primi vrednost (1, 5, 100, -10) i vrati tu vrednosti. Ovi zapisi vrednosti koje zadajemo nazivaju se *prosti* odnosno *primitivni izrazi*. Oni su nešto najjednostavnije što programski jezik podržava – kada primi prost izraz GHCI „sam ume“ da vrati njegovu vrednost. I da bude potpuno jasno: pri izvršavanju programa na kraju se sve završava prostim izrazima.

[2] U ovoj knjizi neću objašnjavati koji je softver potrebno instalirati za primere koje navodim niti kako se taj softver instalira, ne zbog toga što mislim da je to nevažno, nego zbog toga što ne želim da se skreće fokus sa osnovne ideje, pa da knjiga postane uputstvo za instaliranje softvera. Sve što ti treba vezano za GHC koji ćemo koristiti naći ćeš na internetu.

Ali, pored toga što može da ponovi broj koji smo uneli, programsko okruženje može i da sabere brojeve.

```
> 1 + 1
2
> 2 + 3
5
> 3 + 7
10
> 5 + 10
15
```

Složeni izrazi

Prosti izrazi povezani operatorima

Uneli smo 1 + 1 i dobili smo 2. Uneli smo 2 + 3 i dobili 5. Uneli smo 3 + 7 i dobili 10. Uneli smo 5 + 10 i dobili 15. Od čega se sastoji „2 + 3“? Sastoji se od dva prosta izraza, 2 i 3, i operatora + između njih. Takve izraze, koji su nastali *spajanjem prostih izraza pomoću operatora* kao što je +, nazivamo *složeni izrazi*. Kada programsko okruženje primi složeni izraz, *evaluira ga* (sračuna) i vrati nam dobijenu vrednosti. Za izraz „2 + 3“ vrednost evaluacije će biti 5. Kao što GHCI „zna“ šta znači 5, „zna“ i šta treba da radi operator +, odnosno kao što je 5 primitivni izraz, + je *primitivni operator* koji možemo koristiti za pravljenje složenih izraza.

Računar bolje evaluira izraze nego čovek

Sada već vidimo u čemu je računar superioran u odnosu na čoveka: ako ima dobro definisano uputstvo za rešavanje problema (a to je uputstvo koje je razrađeno do nivoa pravila toliko jednostavnih da ne zahtevaju razmišljanje nego čak i mašina može da ih izvrši), računar nikada neće praviti lapsuse. Ako bismo probali u glavi da saberemo 256878561241546 i 56548658413548 vrlo je verovatno da bismo pogrešili. Računar će svaki put tačno evaluirati izraz.

```
> 256878561241546 + 56548658413548
313427219655094
```

Naravno, možemo zadati računaru zadatak da sabere i više brojeva.

```
> 1 + 2 + 3 + 4 + 5
15
```

Šta se prilikom evaluacije ovog izraza dogodilo? Evaluacija je imala četiri koraka:

1. U prvom koraku su sabrana dva krajnja leva broja, odnosno evaluiran je podizraz 1 + 2, pa je izraz postao 3 + 3 + 4 + 5
2. U drugom koraku su sabrana dva krajnja leva broja, odnosno izraz je postao 6 + 4 + 5
3. U trećem koraku su sabrana dva krajnja leva broja, pa je izraz postao 10 + 5
4. U četvrtom koraku su sabrana dva krajnja leva broja, pa je izraz postao 15. Pošto više nema šta da se sabira, pošto je računar došao do kraja evaluacije, vraća se dobijena vrednost.

Evaluacija složenih izraza

Evaluacija tog izraza prikazana je listingom ispod. Vidimo da se svaki korak evaluacije sastoji od spajanja dva prosta izraza operatorom koji ih povezuje.

```
1 + 2 + 3 + 4 + 5
=> (1 + 2) + 3 + 4 + 5
=> 3 + 3 + 4 + 5
=> (3 + 3) + 4 + 5
=> 6 + 4 + 5
=> (6 + 4) + 5
=> 10 + 5
=> 15
```

Kao da smo imali izraz `(((1 + 2) + 3) + 4) + 5`; saberi `1` i `2`, pa dodaj na rezultat 3, pa dodaj na rezultat 4, pa dodaj na rezultat `5`.

Prilikom evaluacije izraza u svakom koraku izraz se pojednostavio sve dok nismo došli do *tačke u kojoj više nije bilo šta da se pojednostavljuje*, odnosno do tačke u kojoj je izraz sveden na najjednostavniji mogući oblik[3]. Evaluacija izraza je upravo to – *pojednostavljivanje izraza do najjednostavnijeg mogućeg oblika*. Pri tome se u svakom koraku evaluacije eliminiše se po jedan podizraz koji čine dva prosta izraza povezana operatorom.

Evaluacija

Pojednostavljivanje izraza

[3] U ovom slučaju najjednostavniji oblik na koji je izraz mogao da se svede je prost izraz, ali ne mora to uvek da bude. Međutim, važno je napomenuti da Haskell koristi lenju (odloženu) evaluaciju, što znači da se izrazi ne evaluiraju do kraja automatski, već samo onoliko koliko je potrebno da bi se dobio rezultat koji je trenutno potreban.

Prioritet operatora

Pored toga što može da sabira, programsko okruženje može i da oduzima, množi i deli brojeve. Shodno tome, možemo i da koristimo operatore -, * i / kada pravimo složene izraze od numeričkih prostih izraza. Ovo su takođe primitivni operatori koje GHCI „sam ume“ da izvrši. Međutim, različiti operatori imaju različit prioritet u evaluaciji: prvo će se izvršavati množenja i deljenja, a tek onda sabiranja i oduzimanja. Pogledajmo evaluaciju sledećeg izraza

```
> 1 + 2 * 3 - 4 / 5
6.2
```

Evaluacija je imala sledeće korak:

1. Izvršeno je množenje, 2 * 3, pa je izraz postao 1 + 6 – 4 / 5
2. Zatim je izvršeno deljenje, 4 / 5 pa je naš izraz postao 1 + 6 – 0.8
3. Zatim je izvršeno sabiranje, 1 + 6, pa je izraz postao 7 – 0.8
4. Na kraju je izvršeno oduzimanje pa je izraz postao 6.2

Koraci evaluacije prikazani su listingom ispod.

```
1 + 2 * 3 - 4 / 5
=> 1 + (2 * 3) - (4 / 5)
=> 1 + 6 - (4 / 5)
=> 1 + 6 - 0.8
=> 7 - 0.8
=> 6.2
```

Pitanja i zadaci

- Obrati pažnju da je rezultat evaluacije isti bez obzira da li je prvo eliminisan podizraz `2 * 3` ili `4 / 5`. Objasni zašto.

Kompozabilnost izraza

Kao što vidimo, prilikom evaluacije izraza, izraz se „razbija“ na podizraze do nivoa dva prosta izraza koji se neposredno evaluiraju, odnosno da se složen izraz dobija *kompozicijom* prostih izraza. Da li možemo nad izrazima da primenimo druge principe koje smo spomenuli u uvodu, odnosno principe apstrakcije i parametrizacije? Odgovor na ovo pitanje je, naravno, da.

Konstanta

Pored toga što izraz možemo da gradimo kompozicijom od jednostavnijih izraza koje povezujemo operatorima, izraz možemo i da *apstrahujemo*. Izrazu bismo mogli da dodelimo *ime* i svaki sledeći put kada nam zatreba da iskoristimo to ime.

```
> e = 2.71828
> e
2.71828
```

Parametrizacija

Na svakom mestu na kom bismo naveli ime izraza *e*, ono bi bilo zamenjeno brojem 2.71828. Ovakve imenovane izraze nazivamo *konstante*. Više ne bismo morali da pamtimo koja je vrednost konstante *e*, bilo bi dovoljno da znamo kako se zove i čemu služi. Naravno, sa desne strane operatora = mogao je da se nađe i složeni izraz, kao što je prikazano listingom ispod.

```
> x = 1 + 2 * 3 - 4 / 5
```

U tom slučaju konstanti `x` je dodeljena vrednost koja se dobija evaluacijom izraza sa desne strane operatora =, odnosno `6.2`. Naravno, konstanta je kasnije mogla da se javi kao gradivni blok u narednim izrazima, kao što je prikazano listingom ispod.

```
> 3 * 2 + x
```

Prilikom evaluacije, ime konstante x biva zamenjeno vrednošću konstante, odnosno izraz postaje `3 * 2 + 6.2` i dalje se evaluira kao i svaki drugi izraz koji smo do sada videli.

```
3 * 2 + x
=> 3 * 2 + 6.2
=> (3 * 2) + 6.2
=> 6 + 6.2
=> 12.2
```

Izvršavanje programa

Evaluacija izraza

Ispostavlja se da je odgovor na pitanje iz uvoda, *kako da napišemo uputstvo za rešavanje problema*, veoma jednostavan: *napisaćemo ga kao složeni izraz*. Taj složeni izraz sastojaće se od elementarnih gradivnih blokova, *prostih izraza* i *primitivnih operatora*, koje GHCI „sam ume“ da evaluira odnosno izvrši. Izvršavanje programa je onda evaluacija izraza. Rezultat izvršavanja programa dobićemo kada izraz više ne bude mogao da se pojednostavi.

Pitanja i zadaci

- Navedi korake u evaluaciji sledećih izraza:
  - 3 – 2 + 1 – 4 + 5
  - 3 – (2 + (1 – 4) + 5)
  - 3 – 2 + 1 – 4 * 5
  - 3 – 2 + (1 – 4) * 5

Da li su dobijeni rezultati za sve primere jednaki? Objasni zašto je to tako.

- Diskutuj o dobijenim koracima evaluacije u grupu.
- Pokreni primere u programskom jeziku Haskell. Da li si negde pogrešio?
- Diskutuj o greškama u grupi. Da li ima istih grešaka koje je dobilo više učenika u grupi? Ako ima, zašto?

Funkcije

- Parametrizacija i parametrizovani izrazi
- Izvršavanje funkcije
- Parcijalno izvršavanje funkcije

U prethodnom poglavlju smo se susreli sa izrazima. Videli smo kako, primenom principa kompozicije, od prostih izraza gradimo složene izraze. Takođe, videli smo da izraze možemo da apstrahujemo tako što im dodelimo naziv, odnosno što ih dodelimo konstantama. I videli smo da se izvršavanje programa svodi na evaluaciju izraza. U ovom poglavlju ćemo dati odgovor na pitanje da li nam treba još neki gradivni element osim izraza za pisanje programa.

Parametrizacija

Videli smo da program (uputstvo za rešavanje problema) možemo da napišemo kao *složeni izraz*, koji je sačinjen od *prostih izraza* povezanih *primitivnim operatorima*. Tako napisan program dajemo programskom okruženju (u našem slučaju to je GHC) koje ga evaluira i vraća vrednost evaluacije izraza. Složeni izraz se evaluira tako što se razbija na podizraze dok se ne dođe do prostih izraza. Prosti izrazi se direktno evaluiraju i na njih se primenjuju operatori u skladu sa prioritetom operatora. Odnosno, vidimo da se evaluacija završava prostim izrazima – programsko okruženje prosto „ume“ da vrati vrednost `5` kada unesemo „5“. Međutim, u izrazu, na mestu prostog izraza može se naći i *parametar*. Na primer, umesto `1 + 2 * 3` mogli bismo imati `1 + 2 * x` i dobiti izraz koji će se različito evaluirati u zavisnosti od vrednosti koju ima parametar `x`. Ako `x` ima vrednost `3`, izraz će se evaluirati u `7`. Ako x ima vrednost `2`, izraz će se evaluirati u `5`.

Parametrizovani izrazi

Ovde se susrećemo sa važnim pojmom *parametrizacije*. Zamisli da je radio aparat napravljen tako da hvata samo jednu radio stanicu – takav radio aparat ne bi bio naročito upotrebljiv. S druge strane, radio aparat kome možemo da promenimo frekvenciju na kojoj sluša okretanjem dugmeta, odnosno čija je frekvencija parametrizovana, može da se koristi za hvatanje mnogo različitih stanica. Slično tome, u programiranju možemo da napravimo parametrizovane izraze koje nazivamo *funkcije*.

Pitanja i zadaci

- Gde se sve susrećeš sa pojmom parametrizacije? Koji sve uređaji u svakodnevnom domaćinstvu su parametrizovani? Diskutuj u grupi.

Anatomija funkcije

Kada pišemo funkciju (parametrizovani izraz), treba da navedemo kako će njeni parametri *zvati*. Takođe, da bismo završili evaluaciju izraza, treba i da navedemo koje vrednosti će parametri imati. Listingom ispod prikazano je kako jedna funkcija izgleda.

```
> (\x -> 1 + 2 * x) 3
7
```

Parametar i izraz

Iako na prvi pogled deluje zastrašujuće kriptično, u stvari je jako jednostavno. Imamo izraz sa jednim parametrom, koji se zove x. To smo rekli fragmentom \x -> . Odnosno, rekli smo: „u izrazu koji sledi x će biti parametar“. Ovu sekciju, u kojoj navodimo parametre koje ćemo koristiti nazivamo *zaglavlje funkcije*. Zatim imamo sam izraz: 1 + 2 * x, koji se naziva *telo funkcije*. I na kraju imamo vrednost koju ćemo dodeliti parametru x prilikom izvršavanja funkcije, a to je vrednost 3, koju nazivamo *argument*. Argument je konkretna vrednost koju parametar prima.

Pitanja i zadaci

- Koja je razlika između parametra i prostog izraza?
- Koja je razlika između parametra i argumenta? Kako zadajemo argumente?
- Kako se u primerima uređaja iz domaćinstva zadaju argumenti? Koje vrednosti ovi argumenti mogu da imaju?

Izvršavanje funkcije

Izvršavanje funkcije je takođe vrlo jednostavno. Programsko okruženje će prosto uzeti telo funkcije i svaku pojavu parametra x zameniti vrednošću argumenta. Tako će naš izraz, od `1 + 2 * x` postati `1 + 2 * 3`. Dalje će programsko okruženje izraz evaluirati na jednak način kao što je evaluiralo i sve druge izraze do sada, odnosno, prvo će pomnožiti `2` i `3`, pa dobiti izraz 1 + 6, pa će sabrati `1` i `6` i dobiti `7`.

Ništa nas ne sprečava da se parametar x pojavi na više mesta u telu funkcije. Mogli smo da imamo sledeći izraz.

Više pojava parametra u telu funkcije

```
> (\x -> x * x) 3
9
```

Kako bi se izvršila ova funkcija? Svaka pojava parametra `x` u telu funkcije bila bi zamenjena vrednošću argumenta, pa smo dobili izraz `3 * 3`. Zatim je ovaj izraz evaluiran i dobijena je vrednost `9`.

Naravno, i u ovom slučaju se radi o izrazu koji se evaluira. Listingom ispod je prikazana evaluacija navedenog izraza.

```
 (\x -> x * x) 3
=> 3 * 3
=> 9
```

Postavlja se pitanje *zbog čega* bismo parametrizovali izraz ako mu odmah pošaljemo vrednost parametra? Iako je ovo čest scenario u mnogim programskim jezicima (na primer u programskom jeziku JavaScript neposredni poziv neimenovane funkcije se jako često koristi), mnogo je češća situacija da se funkciji dodeli još i ime.

Imenovane funkcije

```
> sqr x = x * x
> sqr 3
9
> sqr 4
16
```

Ovde smo kreirali jednu funkciju kojoj smo dali ime `sqr` za koju smo rekli da će imati jedan parametar `x`. To smo postigli fragmentom `sqr x` . Telo funkcije je parametrizovani izraz `x * x`. Nakon toga, svaki put kada nam treba da izračunamo kvadrat nekog broja, dovoljno je da pozovemo ovu funkciju navodeći njeno ime i vrednost argumenta `x`, na primer `sqr 3` ili `sqr 4`. Na mestu gde smo pozvali funkciju ime funkcije biće zamenjen telom funkcije u kome će svaka pojava parametra x biti zamenjena vrednošću argumenta. Tako će `sqr 3` postati `(\x -> x * x) 3`, pa će svako `x` biti zamenjeno sa `3` i dobićemo izraz `3 * 3`, koji će se evaluirati u `9`. Evaluacija je prikazana listingom ispod.

Poziv imenovane funkcije

```
sqr 3
=> (\x -> x * x) 3
=> 3 * 3
=> 9
```

Pitanja i zadaci

Objasni zašto se funkcije imenuju?

Apstrakcija

Kada smo funkciju jednom napisali (ili je dobili napisanu u biblioteci funkcija), telo te funkcije nas više ne interesuje. Programski jezici često nude biblioteke napisanih funkcija čije implementacije ne moramo da znamo da bismo koristili programski jezike. Da bismo funkciju koristili, dovoljno je samo da znamo ime funkcije i parametre koje vraća, a izvršavanjem funkcije (zamenom pojava parametara u telu argumentima iz poziva funkcije i evaluacijom tako dobijenog izraza) baviće se programsko okruženje. Odnosno, potrebno nam je samo da znamo *šta* funkcija radi, a ne interesuje nas više *kako* ona to radi. Ovo ponašanje susrećemo stalno u svakodnevnom životu. Na primer, da bismo slušali muziku na radio aparatu, *ne moramo da znamo kako da napravimo radio aparat*. Dovoljno je da znamo kako da podesimo željenu talasnu dužinu i time namestimo radio stanicu. Ili da bismo vozili automobil, *ne moramo da znamo da napravimo automobil*. Zamisli koliko bi svet bio komplikovan kada bismo za svaki uređaj koji koristimo morali da znamo i kako radi! Sa ovim pojmom zanemarivanja nebitnih detalja smo se već susretali i susretaćemo se kroz celu knjigu: to je *apstrakcija*. *Funkcija nam omogućuje da apstrahujemo detalje implementacije i da se fokusiramo samo na upotrebu.* Time smo dobili odgovor na pitanje koje smo malopre postavili „*zbog čega* bismo parametrizovali izraz?“ – *da ne bismo morali da ga pamtimo,* odnosno da *bismo mogli da ga apstrahujemo*.

Pitanja i zadaci

- Koje od uređaja koje si naveo bi umeo i da napraviš, a koje umeš samo da koristiš?

Svaka funkcija vraća vrednost

Poziv svake funkcije završiće se nekim izrazom. Kada primi izraz, programsko okruženje ga evaluira i na kraju evaluacije vraća dobijenu vrednost. Shodno tome, i poziv svake funkcije vraća vrednost – vrednost evaluacije izraza dobijenog kada se pojave parametara zamene vrednostima argumenata.

Funkcija više parametara

Videli smo da možemo da napišemo funkciju sa jednim parametrom. Da li bi funkcija mogla da ima više parametara? Ako bi, kako bi se ti parametri zamenili vrednostima argumenata? Hajde da pogledamo primer:

```
> (\x y -> x + y) 2 3
5
```

Napisali smo neimenovanu funkciju koja primi dva parametra, `x` i `y`. Telo funkcije je izraz u kome se saberu ova dva parametra, odnosno `x + y`. Funkcija je odmah pozvana sa argumentima `2` i `3` i dobijena je vrednost `5`. Vidimo da funkcija može da ima više parametara. A kako je izgledalo njeno izvršavanje?

Pitanja i zadaci

- Koji od uređaja koje si naveo imaju više parametara? Koji su to parametri? Zbog čega imaju više parametara?

Izvršavanje funkcije više parametara

U prvom koraku izvršavanja funkcije parametar x je zamenjen vrednošću `2`. Time je izraz koji smo zadali postao `(\y -> 2 + y) 3` . Nakon toga, u sledećem koraku je parametar y zamenjen vrednošću `3`. Time je izraz postao `2 + 3`. Ovaj izraz je evaluiran i dobijena je vrednost 5. Kao što vidimo, izvršavanje funkcije više parametara svodi se na seriju izvršavanja funkcija od po jednog parametra s leva u desno. Znači prvo će biti zamenjen parametar `x`, pa u sledećem koraku parametar `y`. U svakom koraku se eliminiše po jedan parametar, slično kao što smo u evaluaciji izraza imali da se u svakom koraku eliminiše po jedan podizraz sačinjen od dva prosta izraza i jednog operatora. Ovakav način izvršavanja funkcija, u kome svaka funkcija primi po jedan parametar i vrati jednu vrednost, naziva se *kariing*, po logičaru Haskelu Kariju.

```
 (\x y -> x + y) 2 3
=> ((\x -> (\y -> x + y)) 2) 3
=> (\y -> 2 + y) 3
=> 2 + 3
=> 5
```

Prilikom izvršavanja funkcije više parametara ne moramo poslati sve parametre.

```
> increment = (\x y -> x + y) 1
> increment 2
3
```

Delimično izvršavanje funkcije

Ime `increment` je dodeljeno povratnoj vrednosti poziva funkcije `(\x y -> x + y) 1`. Prilikom poziva funkcije, argument `1` je zamenio svaku pojavu konstante x u telu funkcije, koja je postala `(\y -> 1 + y)`. Više nije bilo argumenata za zamenu parametara, pa je time i završeno izvršavanje funkcije, a imenu `increment` je dodeljena funkcija koja prosleđeni parametar uvećava za `1`. Pozivanje funkcije sa manje argumenata nego što funkcija ima parametara nazivamo *delimično izvršavanje funkcije*. Evaluacija tog izraza prikazana je listingom ispod.

```
(\x y -> x + y) 1
=> (\x -> (\y -> x + y)) 1
=> \y -> 1 + y
```

Šta je rezultat evaluacije izraza?

U prethodnom poglavlju videli smo da se evaluacija izraza odvija dok se složen izraz ne svede na prost izraz. Ali da li je to uvek tako? U primeru delimičnog izvršavanja funkcije videli smo da evaluacija izraza (u ovom slučaju parametrizovanog izraza) ne mora da rezultuje prostim izrazom. Evaluacija izraza se u stvari završava kada *izraz više ne može da se pojednostavljuje*, što ne mora uvek da bude isto što i prost izraz.

Retko ćemo biti u situaciji da imamo delimično izvršavanje neimenovane funkcije. Mnogo češće ćemo biti u situaciji da delimično izvršavanje koristimo u kombinaciji sa imenovanim funkcijama.

```
> add x y = x + y
> increment = add 1
> increment 5
6
```

Konkretizacija

Napravili smo jednu opštu funkciju `add`, koja sabira dva broja. Zatim smo iz nje, delimičnim izvršavanjem, izveli konkretniju funkciju `increment`, koja zadati broj uvećava za `1`.

Pitanja i zadaci

- Kako se konkretizacija funkcije odnosi prema pojmu apstrakcije? Da li je apstraktnija funkcija `add` ili funkcija `increment`? Diskutuj u grupi.

Prefiksna i infiksna notacija

Funkcije smo do sada pozivali tako što navedemo njeno ime nakon kog sledi lista argumenata. Ovaj način pozivanja funkcija naziva se *prefiksna notacija*, jer naziv funkcije prethodi listi argumenata. Međutim, nekada bi nam bilo jednostavnije da funkciju dva parametra pozovemo tako što bismo njeno ime naveli *između* argumenata. Bilo koja funkcija dva parametra može se koristiti u ovoj, *infiksnoj* notaciji prosto tako što se njeno ime navede između dva *bektik* karaktera (`).

```
> add x y = x + y
> 3 `add` 5
8
```

Takođe, bilo koji operator može da se koristi u prefiksnoj notaciji tako što se navede u zagradama.

```
> (+) 3 5
8
```

Sada vidimo da smo funkciju `increment` mogli implementirati još jednostavnije.

```
> increment = (+) 1
> increment 3
4
```

Prosti izrazi i funkcije

Iako na prvi pogled izgleda kao nevažan tehnički detalj programskog jezika Haskell, ova opaska o prefiksnoj i infiksnoj notaciji baca novo svetlo na uvodnu priču o izrazima. Videli smo da se složeni izrazi sastoje od prostih izraza (`1`, `5`, `100`, `-10`) i operatora kojima se prosti izrazi povezuju (`+`, `-`, `*`, `/`). Sada vidimo da su i operatori samo posebno zapisani pozivi funkcija (u infiksnoj notaciji), odnosno da nam trebaju jedino *primitivni izrazi* i *funkcije* (koje su i same parametrizovani *izrazi*) da bismo pisali složene izraze.

Primitivne funkcije

Videli smo da funkciju `sqr` (kvadriranje) možemo napisati koristeći množenje (`sqr x = x * x`) i da funkciju `increment` možemo napisati koristeći sabiranje (`increment = (+) 1`). Videli smo i da su sabiranje i množenje takođe funkcije i rekli smo da su funkcije parametrizovani izrazi. Sada se postavlja pitanje *kako bi izgledao parametrizovani izraz za sabiranje*? Kako je funkcija `(+)` *implementirana*? Ako pogledamo dublje u implementaciju programskog jezika Haskell, doći ćemo do toga da je sabiranje *primitivna funkcija*, čija implementacija dolazi sa programskim okruženjem, kao i oduzimanje, množenje i deljenje. Kao i u slučaju evaluacije prostih izraza (kako programsko okruženje „ume“ da, kada unesemo primitivni izraz „`5`“ to evaluira baš u `5`?) i izvršavanje primitivnih funkcija je prepušteno programskom okruženju[4]. Ovaj zaključak u potpunosti je u skladu sa opaskom iz uvoda da je programiranje pisanje uputstava za rešavanje problema tako što zadajemo pravila koja se dekomponuju do primitivnih potpravila. I da su ta potpravila toliko jednostavna da čak i mašina „ume“ da ih izvršava, odnosno da za njihovo izvršavanje nije potrebna ljudska pamet. Kao što smo složene izraze gradili od primitivnih izraza, i funkcije gradimo od primitivnih funkcija koristeći principe dekompoziciji, parametrizacije i apstrakcije.

Samo izrazi

U prethodnom poglavlju zaključili smo da se programi pišu kao složeni izrazi koji se sastoje od prostih izraza koji su povezani operatorima. Sada smo videli da su operatori funkcije, a funkcije su takođe (parametrizovani) izrazi. Onda složene izraze pravimo tako što primitivne izraze povezujemo parametrizovanim izrazima, odnosno jedini *gradivni materijal koji koristimo kada pišemo programe su izrazi*.

[4] Naravno, način na koji se na računaru zaista obavlja sabiranje dva broja nije misterija, ali prevazilazi opseg ove knjige. Zato, za sveobuhvatnije razumevanje programiranja, uz ovu knjigu, preporučujemo i izučavanje arhitekture računara.

Tipovi

- Tip određuje upotrebu
- Logički tip
- Tipovi i funkcije
- Uvođenje novog tipa

Tip je upotreba

Ako bismo hteli da zidamo kuću susreli bismo se sa različitim *vrstama* građevinskog materijala. Imali bismo daske, cigle, crep,... Šta sve pomoću nekog građevinskog materijala možemo da uradimo zavisi od vrste, odnosno *tipa* građevinskog materijala. Na primer, dasku bismo mogli da presečemo testerom za drvo, a ciglu ne bismo. Dve daske bismo mogli da povežemo ekserom, dve cigle ne bismo. Cigle bismo mogli da povežemo malterom, a daske ne bismo. Kao i u građevinarstvu, i u svakoj drugoj oblasti ljudske delatnosti *tip* predmeta *određuje kako predmet može da se upotrebi*. I u programiranju *tip podatka* određuje *šta sve sa tim podatkom možemo da uradimo*, odnosno *koje funkcije nad tim podatkom možemo da izvršimo*.

Pitanja i zadaci

- Koje sve tipove susrećemo u svakodnevnom životu? Koji tipovi namirnica postoje kada kuvamo jela? Kako ovi tipovi određuju upotrebu namirnica?
- Kako se tipovi namirnica zadaju u receptu za jelo?

Kako naučiti novi tip?

Videli smo da programiranje podrazumeva korišćenje osnovnih gradivnih blokova za sastavljanje složenih uputstava za rešavanje problema. To znači da, kada hoćemo da naučimo novi tip, treba da naučimo koje vrednosti taj tip ima i kako te vrednosti možemo da koristimo. Odnosno treba da naučimo kako možemo da pravimo složene izraze od tih vrednosti.

Logički tip

U prethodnim primerima smo se susreli samo sa brojčanim vrednostima i funkcijama koje manipulišu brojčanim vrednostima, (sabiranjem, oduzimanjem, množenjem, deljenjem; zatim smo definisali novu funkciju, `sqr`, koja kvadrira brojeve). Međutim, pored brojeva, u programiranju se javljaju i drugi tipovi. Sada ćemo pogledati još jedan tip, namenjen za predstavljanje logičkih vrednosti.

Logičke vrednost

Rečenica koju izgovaramo može da bude ili tačna ili netačna. Tačnost i netačnost predstavljaju se logičkim vrednostima koje su tipa *Bool*. Postoje samo dve vrednosti, odnosno samo dva prosta izraza ovog tipa, `True` i `False`.

```
> True
True
> False
False
```

Pitanja i zadaci

- Navedi 5 rečenica koje su uvek tačne i 5 rečenica koje su uvek netačne. Šta je potrebno da bi rečenica bila uvek tačna ili uvek netačna?

Upotreba logičkog tipa

Logički tip može se koristiti i za predstavljanje drugih pojava, a ne samo logičke tačnosti iskaza. Na primer, prekidač za svetlo može biti uključen (`True`) ili isključen (`False`). Ili učenik može biti prisutan na času (`True`) ili odsutan (`False`).

Pitanja i zadaci

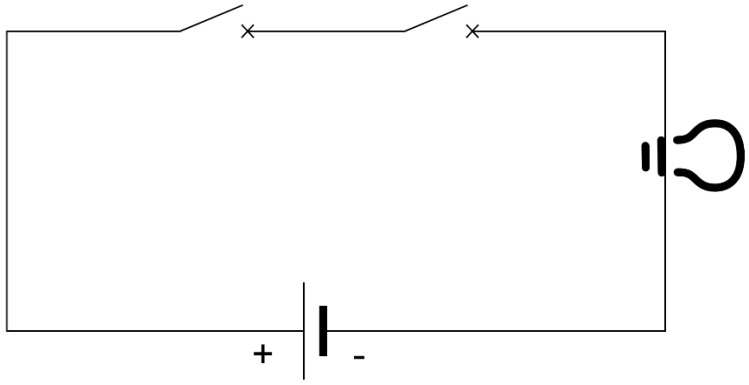
- Koje sve pojave mogu da se predstave logičkim tipom podataka? Diskutuj u grupi.

Logički operatori

Videli smo da od brojčanih vrednosti možemo da pravimo složene izraze upotrebom primitivnih operatora sabiranja, oduzimanja, množenja i deljenja. Pošto se ovi operator primenjuju na numeričke (brojčane) vrednosti, kažemo da su to numerički operatori. Slično je i sa logičkim vrednostima: postoji skup operatora koji se primenjuju nad logičkim vrednostima i koje zovemo logički operatori. Od logičkih vrednosti možemo graditi složene logičke izraze upotrebom operatora *konjunkcije* (`&&`, logičko i), *disjunkcije* (`||`, logičko ili) i *negacije* (`not`, logičko ne). Pogledajmo ove operatore.

Logičko i

Zamislimo sijalicu koja je povezana na dva prekidača kao na slici 1.



Slika 1 – Logičko i

Ova sijalica će biti upaljena jedino ako su *oba* prekidača uključena. Ako je uključen samo jedan od njih, ili ako nijedan od prekidača nije uključen i sijalica će biti ugašena. Ovakvu vezu nazivamo *logičko i*.

Vidimo da će vrednost evaluacije izraza koji sadrži logičko i biti `True` jedino ako oba operanda imaju vrednost `True`. U svakom drugom slučaju, vrednost evaluacije izraza će biti `False`.

Na listingu ispod prikazana je upotreba logičkog i u programu.

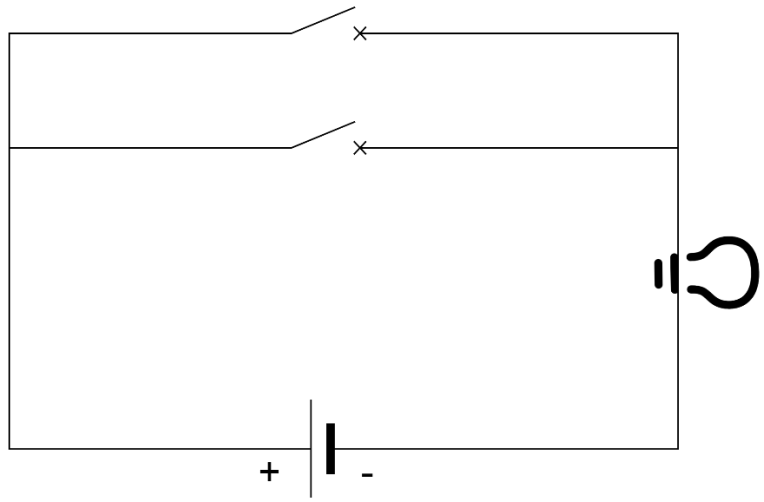
```
> True && True
True
> True && False
False
> False && True
False
> False && False
False
```

Pitanja i zadaci

- *Logičko i* se jako često koristi u svakodnevnom životu. Da bismo mogli da vozimo bicikl potrebno je da je napumpana i prednja i zadnja guma. Smisli još primera upotreba *logičkog i* u svakodnevnom životu i diskutuj primene sa kolegama i grupi.

Logičko ili

Pored logičkog i, podjednako važan logički operator je *logičko ili*. Ono će dve logičke vrednosti povezati tako da će rezultat biti `True` ako je makar jedna od vrednosti tačna. Na slici 2 prikazana je sijalica povezana na dva prekidača, ali tako da će ona biti upaljena ako je makar jedan od prekidača uključen.



Slika 2 – logičko ili

Izrazi koji sadrže *logičko ili* evaluiraće se u `True` u svim slučajevima osim kada su oba operanda `False`. Ponašanje *logičkog ili* u programu prikazano je listingom ispod.

```
> True || True
True
> True || False
True
> False || True
True
> False || False
False
```

Pitanja i zadaci

- *Logičko ili* koristimo kada je dovoljno da je ispunjen bar jedan od dva uslova. Na primer, da bi neko mogao da uđe u biblioteku, mora da pokaže člansku kartu ili đačku knjižicu. Nije važno koji od ta dva dokumenta ima — dovoljno je da ima bar jedan, odnosno nije potrebno da ima oba. Smisli još primera upotreba *logičkog ili* u svakodnevnom životu i diskutuj primene sa kolegama u grupi.

Logičko ne

Pored ova dva logiča operatora (*logičko i* i *logičko ili*) postoji još jedan: *logičko ne*. Ova operator se razlikuje po tome što su *logičko i* i *logičko ili* povezivali dve logičke vrednosti, odnosno oni su *binarni operatori*, dok se *logičko ne* odnosi samo na jednu vrednost, odnosno ono je *unarni operator*. Odnosno, *logičko i* i *logičko ili* su funkcije dva parametra (kao, na primer funkcija +), dok je *logičko ne* funkcija jednog parametra (kao na primer funkcija `inc`).

Logičko ne negira vrednost, odnosno „obrne“ je: kada se primeni na `Flase` vrati `True`; kada se primeni na `True` vrati `False`.

```
> not True
False
> not False
True
```

Pitanja i zadaci

Logičko ne koristimo kada hoćemo da kažemo da nešto nije tačno. Na primer, ako nije lepo vreme, nećemo ići na izlet. Dakle, "ne" lepo vreme znači da pada kiša, duva jak vetar ili je previše hladno.

- Razmisli o još primerima kada koristimo *logičko ne* u svakodnevnom životu i diskutuj ih sa drugima u grupi.

Logički operatori su funkcije

Kao i u slučaju aritmetičkih operatora (sabiranja, oduzimanja, množenja i deljenja) i logički operatori su funkcije i mogu se koristiti u prefiksnoj notaciji. Tako smo mogli da napišemo funkciju za logičko i kao što je prikazano ispod.

```
> (&&) True True
True
```

Logičke funkcije

Kao što smo mogli da pravimo funkcije nad brojčanim vrednostima kao složene, parametrizovane izraze, možemo da pravimo i funkcije nad logičkim vrednostima. Napravićemo funkciju koja predstavlja značenje reči „ako“.

Pitanja i zadaci

- Šta znači reč „ako“? Diskutuj u grupi.

Šta *znači* kada kažemo „ako uradim domaći, dobiću dobru ocenu“? To znači da situacija u kojoj sam uradio domaći sigurno dobijam i dobru ocenu. Ali moguće je i da dobijem dobru ocenu i ako nisam uradio domaći – na primer tako što sam bio aktivan na času. Kako bi trebala funkcija „ako“ da se ponaša? Vrednost ove funkcije bila bi False jedino u slučaju kada sam uradio domaći i dobio lošu ocenu.

Domaći	Dobra ocena	
True	True	True
True	False	False
False	True	True
False	False	True

Ovu logičku funkciju zovemo *implikacija*, i možemo je napraviti koristeći tri osnovna logička operatora (i, ili, ne). Mogli bismo da je definišemo na sledeći način.

```
> implies p q = not p || q
> implies True True
True
> implies True False
False
> implies False True
True
> implies False False
True
```

Pitanja i zadaci

- Implementiraj funkciju *logičko i* koristeći *logičko ne* i *logičko ili*
- Implementiraj funkciju *logičko ili* koristeći *logičko ne* i *logičko i*.
- Koje još primene *implikacije* možeš da smisliš? Diskutuj u grupi.
- Kako bi još mogla da se implementira *implikacija*?
- Videli smo da u programskom jeziku implikacija može da se implementira koristeći *logičko ne* i *logičko ili*. Ako bi u programskom jeziku ima kao primitivni operator zadatu *implikaciju*, da li bi samo koristeći nju mogao da implementiraš druge logičke operatore? Koje logičke operatore bi mogao da implementiraš samo koristeći implikaciju?
- Još jedna interesantna logička funkcija je *ekskluzivno ili*. Ova funkcija je tačna jedino ako je samo jedna od dve logičke vrednosti tačna. Ako su obe netačne ili obe tačne, ona je netačna. Gde bi se u svakodnevnom životu koristila ova logička operacija?

- Kako bi implementirao *ekskluzivno ili*? Koje logičke operatore bi koristio?
- Da li bi, u programskom jeziku koji kao primitivu ima samo *ekskluzivno ili*, mogao da implementiraš ostale logičke operatora?
- Kako izgleda evaluacija sledećih izraza?
  1. True && False
  2. not True || False
  3. True && (not False)
  4. not (True && False)
  5. (not False || False) && (True && not True)

Kako se uči tip

Sada smo videli dve važne stvari: 1) uveli smo logički tip podataka; i 2) još važnije, videli smo kako se tip podataka uči. Kada hoćemo da naučimo novi tip podataka, treba da vidimo koje vrednosti taj tip može da ima, koje funkcije nad tim vrednostima možemo da pozivamo i za koje konkretne situacije u svetu se taj tip koristi. Svaki sledeći tip podataka koji budemo učili, učićemo na taj način.

Programiranje je celoživotno učenje (tipova)

Posao programera u najvećoj meri podrazumeva da pojave u svetu opisujemo programima koje pravimo. Sasvim je uobičajeno da jedan programer radi prvo na projektu za banku, potom razvija softver za biblioteku, a zatim učestvuje u razvoju video igre. Zbog toga ovaj posao podrazumeva da ćemo stalno učiti (i praviti) nove tipove. I zato je jasno zašto je naivno verovati da ćemo "jednog dana prosto naučiti da programiramo" – jer je programiranje po svojoj prirodi celoživotno učenje (tipova).

Pogrešni tipovi

Videli smo da tip podataka u stvari definiše upotrebu tog podatka, odnosno da nam kaže koje funkcije možemo pozvati nad tim podatkom. To znači da nam tip govori i u kojim funkcijama taj tip *ne može* da učestvuje. Šta bi desilo kada bismo vrednost probali da upotrebimo izvan onoga što je predviđeno njenim tipom? Hajde da probamo da saberemo dve logičke vrednosti.

Programsko okruženje treba da nam dozvoljava da vrednosti koristimo jedino u skladu sa njihovim tipom

```
> True + True

<interactive>:36:1: error:...
```

Ne bismo uspeli, odnosno dobili bismo grešku! Programsko okruženje nam ne dozvoljava (ili bi bar tako trebalo da bude) da

upotrebljavamo vrednosti osim u skladu sa njihovim tipom[5]. Vrlo slično kao što ne bismo uspeli kada bismo probali da testerom za drva presečemo ciglu – alat nije napravljen za to i neće raditi kako treba.

Signatura tipa

To znači da kada definišemo funkciju, pored naziva, parametara i tela, treba da kažemo i na šta je ta funkcija primenjiva. Odnosno definicija funkcije treba da sadrži još i specifikaciju kog tipa su parametri koje prima i kog tipa je vrednost evaluacije izraza koju vraća. Ova specifikacija tipova zove se *signatura tipa*[6]. Pogledajmo kako signatura tipa izgleda za operator *logičko ne*.

```
not :: Bool -> Bool
```

Logičko ne primi vrednost tipa `Bool` i vrati vrednost tipa `Bool`.

Kako stoji stvar sa binarnim operatorima? Kako bi izgledala signatura tipa operatora *logičko ili*?

```
(||) :: Bool -> Bool -> Bool
```

Tip funkcije i delimično izvršavanje funkcije

Iz prethodnog poglavlja smo saznali da sve funkcije primaju jednu vrednost i vraćaju jednu vrednost (kariing). Binarni operator `||` prima jednu vrednost tipa `Bool` i vraća jednu vrednost tipa `Bool -> Bool`. Sada smo slučajno otkrili novi tip: ovaj tip `Bool -> Bool` nam je već poznat iz prethodnog primera, jer je to funkcija koja prima `Bool` i vraća `Bool`! *I funkcija je, dakle, takođe tip*.

```
> f = (||) False
> f True
True
> f False
False
```

[5] Postoje programski jezici koja neće sprečiti neprikladnu upotrebu vrednosti. Za njih kažemo da su *slabo tipizirani*.

[6] Kako programsko okruženje zna signaturu tipa funkcije ako je mi nismo nigde napisali? Za to se koristi *zaključivanje tipova*. Implementacija zaključivanja tipova prevazilazi opseg ove knjige, ali je korisno izučiti je.

Kreirali smo funkciju `f` koja je delimično izvršena funkcija (`||`) i čija će signatura tipa biti `f :: Bool -> Bool`, odnosno koja prima logičku vrednosti i vraća logičku vrednosti.

Pitanja i zadaci

- U prethodnom primeru smo videli da je konstanta `f = (||) False` tipa `Bool -> Bool`. Kog tipa su sledeće konstante?
  1. `g = True`
  2. `h = not`
  3. `i = (&&) True`
  4. `j = (.) not not`
  5. `k = (\x -> x || (x && x))`

Relacioni operatori

Za sada vidimo da su tipovi svetovi za sebe: saberemo dva broja i dobijemo broj; kvadriramo broj i opet dobijemo broj; negiramo logičku vrednost i dobijemo logičku vrednost. Međutim, da li je moguće primeniti funkciju na jedan tip, a dobiti drugi tip? Odgovor je *da*. Na primer, primenom operatora poređenja, odnosno *relacionog operatora* (na primer operator ==) na dve brojčane vrednosti dobijamo logičku vrednost.

```
> 1 == 2
False
> 1 /= 2
True
> 1 > 2
False
> 1 < 2
True
> 1 >= 2
False
> 1 <= 2
True
```

Relacioni operatori su, redom, jednako, različito, veće, manje, veće ili jednako, i manje ili jednako.

Uslovni izraz

Videli smo da nam brojčani tip omogućuje da računamo. Ali za šta se u programima koristi i šta nam omogućuje logički tip? Omogućuje nam da pravimo *uslovne izraze*.

```
> if 3>2 then 3+2 else 3-2
5
```

Ako je `3` veće od `2`, onda evaluiraj izraz `3+2`. U suprotnom evaluiraj izraz `3-2`. Uslovni izraz sastoji se od *uslova* (izraza koji se evaluira u logičku vrednost), i dva izraza (true-izraza i false-izraza). Ukoliko se uslov evaluira u `True`, izvršiće se prvi izraz (`3+2`). Ukoliko se evaluira u `False`, izvršiće se drugi izraz (`3-2`).

Uslovni izraz i odlučivanje

Primere slične uslovnom izrazu često srećemo u svakodnevnom životu, svaki put kada treba da donesemo neku odluku. Tako, na primer, odlučivanje šta da obučemo na osnovu vremenskih uslova podseća na primenu uslovnog izraza. Ako je temperatura veća od 20°C, obući ću majicu kratkih rukava, u suprotnom ću obući majicu dugih rukava. U ovom primeru, uslov je temperatura > 20°C, dok su dva moguća izbora: majica kratkih rukava (ako je uslov tačan) i majica dugih rukava (ako uslov nije tačan).

Pitanja i zadaci

- Smisli primere uslovnog izraza iz svakodnevnog života. Diskutuj o ovim primerima u grupi.
- Da li je odlučivanje uvek između 2 opcije?

Uslovni izraz u funkciji

Uslovne izraze najčešće koristimo da definišemo delove funkcija. Time dobijamo da se funkcije različito ponašaju u zavisnosti od vrednosti parametara, odnosno da *kontekst u kom se izvršava izraz utiče na tok njegove evaluacije*. Pri tome, kontekst je određen vrednošću parametara koji su prosleđeni funkciji prilikom njenog poziva. Na primer, funkciju `absolute` bismo mogli implementirati kao.

```
> absolute x = if x >= 0 then x else (-x)
> absolute 3
3
> absolute (-3)
3
```

Ukoliko pošaljemo vrednost koja je veća od nule, funkcija će vratiti tu vrednost. Ukoliko pošaljemo vrednost manju od nule, vratiće vrednost sa poništenim negativnim predznakom.

Pogledajmo kako izgleda evaluacija ovog izraza.

```
absolute x = if x >= 0 then x else (-x)
absolute (-3)
=> (/x -> if x >= 0 then x else (-x)) (-3)
=> if (-3) >= 0 then (-3) else (-(-3))
=> if False then (-3) else (-(-3))
=> (-(-3))
=> 3
```

Prvo je u pozivu funkcije `absolute (-3)` naziv funkcije zamenjen njenim telom. Zatim je parametar `x` zamenjen prosleđenom vrednošću `(-3)`. Zatim je izraz u uslovu (`(-3) >= 0`) evaluiran i dobijena je vrednost `False`. Zatim je eliminisan deo uslovnog izraza koji se neće izvršiti i dobijen je izraz `(-(-3))`. Evaluacija ovog izraza rezultuje vrednošću `3` što je i najprostiji oblik na koji se početni izraz može svesti, odnosno kraj evaluacije.

Pitanja i zadaci

- Koja je signatura tipa ove funkcije? Da li upotreba uslovnog izraza ima posledice na signaturu tipa?

Pogledajmo ponovo funkciju `sqr` koja računa kvadrat broja. Ako joj prosledimo broj `1`, ona će vratiti broj `1`. Ako joj prosledimo broj `2`, vratiće nam broj `4`. Ako joj prosledimo broj `3`, vratiće nam broj `9`. U ovom slučaju funkcija može biti zapisana kao tabela (bila bi to zaista velika tabela!) čija jedna kolona su sve moguće vrednosti `x`, a druga kolona odgovarajuće vrednosti `sqr x`

x	sqr x
1	1
2	4
3	9
4	16
...	...

Funkcije su preslikavanja između tipova

Tabelarni zapis funkcije daje nam još jedan važan uvid u prirodu funkcija. Sada vidimo da funkcije, koje smo definisali kao parametrizovane izraze, možemo posmatrati na još jedan način: kao preslikavanja među tipovima. Funkcija `sqr` broj `1` je preslikala u broj `1`; broj `2` u broj `4`; broj `3` u broj `9`; ..., odnosno preslikala je brojčani tip na brojčani tip. Funkcija `not` preslika logički tip na logički tip (`True` je preslikala na `False`, a `False` na `True`). Kakvo je preslikavanje funkcija `(&&)`?

X		
True	Y	
	True	True
	False	False
False	y	
	True	False
	False	False

Funkcija `(&&)` je logički tip preslikala na *funkciju* koja logički tip preslika na logički tip. Analogno tome, funkcija `(==)` je brojčani tip preslikala na funkciju koja brojčani tip preslikava na logički tip.

Jednoznačno preslikavanje

To preslikavanje je takvo da funkcija za zadatu vrednost izvorišnog tipa uvek vrati istu vrednost odredišnog tipa. Svaki put kada pozovemo `sqr 3` dobićemo `9`, a kada pozovemo `sqr 4` dobićemo `16`. Ne može (ne sme) se desiti da pozovemo `sqr 3` pa da jednom dobijemo `9`, a drugi put da dobijemo `10`.

Pitanja i zadaci

- Za funkcije koje smo do sada spominjali (numerički operatori, logički operatori, relacioni operatori) objasni kakva su preslikavanja.
- Da li signatura tipa ima nekakve sličnosti sa dijagramom preslikavanja te funkcije?

Dva aspekta funkcije

1) Parametrizovani izraz
2) Preslikavanje tipova

U stvari na funkciju možemo gledati iz ptičije i žablje perspektive. Iz žablje perspektive, funkcija je izraz koji definiše kako se ulaz transformiše u izlaz. Iz ptičije perspektive, funkcija je preslikavanje između tipova. Ove dve perspektive upravo i opisujemo kada pišemo funkciju: 1) definišemo parametrizovani izraz koji odgovara toj funkciji; i 2) definišemo preslikavanje te funkcije signaturom tipa.

Komponovanje funkcija

Kada smo pričali o funkcijama kao parametrizovanim izrazima videli smo kako se princip apstrakcije primenjuje na funkcije. Za funkciju treba samo da znamo kako se zove i šta radi da bismo je koristili. Ne moramo da znamo kako je funkcija implementirana, odnosno ne moramo da znamo kako radi. Ali postavlja se pitanje da li se na funkcije odnosi i princip kompozabilnosti? Odnosno kako možemo da pravimo složene funkcije kao kompozicije jednostavnijih funkcija?

Komponovanje funkcija pomoću eksplicitnog poziva

Možemo na nekoliko načina. Pre svega, telo funkcije može sadržati poziv druge funkcije. Ako bismo imali dve funkcije, `sqr` koja nalazi kvadrat broj i `double` koja duplira broj, mogli bismo napisati funkciju `sqrDouble` koja nalazi kvadrat dupliranog broja.

```
> sqr x = x * x
> double x = 2 * x
> sqrDouble x = sqr (double x)
> sqrDouble 3
36
```

Kao u primeru `sqrDouble`, napiši kompozicije funkcija za sledeće slučajeve. Pretpostavi da već imaš definisane sledeće funkcije:

Pitanja i zadaci

```
double x = 2 * x
sqr x = x * x
negate x = -x
increment x = x + 1
isEven x = x `mod` 2 == 0
```

1. Napiši funkciju koja uvećava broj za `1`, pa ga zatim udvostručuje. (Nazovimo je npr., `doubleAfterIncrement`)
2. Napiši funkciju koja kvadrira broj, pa zatim stavlja minus ispred rezultata.
   (`negSqr`)
3. Napiši funkciju koja udvostručuje broj, zatim ga kvadrira, pa onda uvećava za `1`.
   (`sqrDoubleInc`)
4. Napiši funkciju koja proverava da li je kvadrat broja paran.
   (`isSqrEven`)
5. Napiši funkciju koja udvostručuje broj, zatim kvadrira rezultat, zatim stavlja minus ispred, i proverava da li je rezultat paran.
   (`isNegSqrDoubleEven`)

Komponovanje primenom . operatora

Kada smo napravili kompoziciju funkcije `sqrDouble x = sqr (double x)`, funkcije smo „povezali“ po parametru x. Ali ovaj parametar nam nije eksplicitno trebao. Dovoljno nam je bilo da znamo da se, nad prosleđenom vrednošću, prvo izvrši funkcija `double`, a zatim izvrši funkcija `sqr`. Za takvo povezivanje funkcija služi operator **.** (tačka) koji spaja funkcije. Odnosno, isto ponašanje smo mogli dobiti i da smo `sqrDouble` funkciju definisali kao na listingu ispod.

```
> sqrDouble = sqr . double
> sqrDouble 3
36
```

Pitanja i zadaci

- Primere iz prethodnog zadatka napiši upotrebom **.** operatora
- Obratite pažnju da je **.** operator običan binarni operator kao i `+` ili `&&`. Kako izgleda signatura tipa **.** operatora? Kog tipa su vrednosti koje prima, a kog tipa su vrednosti koje vraća?

Programiranje: učenje i pravljenje tipova

U ovom poglavlju smo videli da tipovi ograničavaju upotrebu podataka, a da se konkretna upotreba (način na koji obrađujemo podatke) definiše funkcijama. I rekli smo da se programiranje u velikoj meri svodi na učenje i pravljenje novih tipova. Učenje tipa je obuhvatalo učenje vrednosti koje tom tipu pripadaju i učenje funkcija koje se nad tim tipom mogu izvršiti. Naravno, upotrebu podatak uvek možemo da proširimo tako što uvedemo nove funkcije koje se nad tim tipom izvršavaju i tako povećamo skup stvari koje nad podacima tog tipa možemo da izvršimo.

Pravljenje novog tipa

Kompozabilnost

Ali rekli smo da možemo i *da napravimo nove tipove*. Kako bismo to uradili? Već smo videli da je jedan od osnovnih principa programiranja princip kompozabilnosti – složenije stvari pravimo od jednostavnijih. Složene izraze smo pravili povezivanjem jednostavnijih, složenije funkcije smo pravili od jednostavnijih funkcija. Isti princip se primenjuje i na tipove: nove tipove pravimo kompozicijom postojećih tipova.

Uvođenje novog tipa

Postoji nekoliko načina na koji se može napraviti novi tip. Najjednostavniji način je tako što ćemo prosto nabrojati vrednosti koje tom tipu propadaju.

```
> data Season = Spring | Summer | Autumn | Winter
deriving(Show, Eq)
> :{
| next season
|   | season == Spring = Summer
|   | season == Summer = Autumn
|   | season == Autumn = Winter
|   | season == Winter = Spring
| :}
> next Spring
Summer
```

Konstruktor tipa i konstruktor vrednosti

Kada smo kreirali tip uradili smo dve stvari: dali smo ime tipu i nabrojali vrednosti koje tip može da ima. Davanje imena tipu realizuje se pomoću *konstruktora tipa*. U ovom primeru konstruktor tipa je `data Season`. Navođenje vrednosti koje spadaju u taj tip realizuje se pomoću konstruktora vrednosti. U ovom primeru je to `Spring | Summer | Autumn | Winter`. Konstruktor vrednosti može da sadrži i informacije o funkcijama koje će biti automatski implementirane (`deriving`), ali o tome ćemo pričati više u poglavlju 6. Upotrebu vrednosti ovog tipa možemo dalje proširiti uvođenjem novih funkcija nad tim tipom, isto kao što smo

radili i sa primitivnim tipovima. U ovom primeru smo kreirali jednu funkciju, `next`, koja prima `Season` i vraća `Season`. Ova funkcija, za zadato godišnje doba vraća godišnje doba koje mu sledi. Ako funkciju `next` pozovemo za `Spring`, dobićemo vrednost `Summer`.

Pitanja i zadaci

- Napravi novi tip koji predstavlja dane u nedelji. Tip treba da ima sedam vrednosti — po jednu za svaki dan.
- Napravi novi tip koji predstavlja stanje semafora. Postoje tri vrednosti: crveno, žuto i zeleno.
- Napravi novi tip koji predstavlja školsku ocenu od 1 do 5, ali koristi konstruktore kao što su Nedovoljan, Dovoljan, Dobar, Vrlo dobar, i Odličan.
- Napravi novi tip koji opisuje deo dana: jutro, podne, popodne, veče i noć.
- Koje funkcije se mogu definisati nad tim tipovima? Diskutuj u grupi i napiši funkcije.

Kompozabilnost tipova

U ovom poglavlju smo videli da tipovi određuju upotrebu vrednosti i da skup tipova nije ograničen, nego je moguće uvesti nove tipove. Videli smo da se novi tipovi mogu uvesti tako što navedemo koje sve vrednosti taj tip može imati. Ali ostaje otvoreno pitanje da li nove tipove možemo kreirati koristeći princip kompozicije, pa, kao što smo nove funkcije pravili komponujući funkcije koje već imamo, kreirati nove tipove od tipova koji su nam već na raspolaganju? Na ovo pitanje daćemo odgovor u sledećem poglavlju.

Složeni tipovi

- Kompozicija tipova
- Lista
- String
- Par
- Sekvenca

Složena upotreba

Videli smo da nam tipovi omogućuju da predstavimo upotrebu vrednosti u programskim jezicima. Sada se postavlja pitanje *kako možemo da komponujemo tipove* da bismo predstavili složenu upotrebu?

U stvarnom svetu, retko radimo sa izolovanim vrednostima. Učenik ima više ocena, biblioteka ima više knjiga, vremenska prognoza ima temperaturu za svaki dan u nedelji. Dakle, umesto jedne vrednosti — često nam treba više vrednosti istog tipa. Tu dolazimo do liste, našeg prvog kompozitnog tipa.

Lista

Lista nam omogućuje da više vrednosti istog tipa okupimo u jednu novu vrednosti. Na primer, učenik je iz matematike dobio ocene 5, 5, 3, 4 i 5. Ove ocene bismo mogli predstaviti listom prikazanom u listingu ispod.

```
> grades = [5,5,3,4,5]
```

- Koje sve pojave koje srećeš u svakodnevnom životu mogu da se opišu listom? Navedi 5 primera i diskutuj u grupi.

Lista brojeva ima drugačiju upotrebu nego broj

Iako se u `grades` nalaze brojevi, `grades` nije brojčanog tipa. Konstanta `grades` je tipa *liste* brojeva, odnosno `[Int]`. Za listu ocena možemo postavljati pitanja koja ne bismo mogli ne bismo mogli da dobijemo odgovor za pojedinačne ocene. Na primer, možemo da pitamo koliko ocena iz matematike učenik ima.

```
> length grades
5
```

Pitanja i zadaci

- Koja je signatura tipa funkcije `length`?
- Da li bi funkcija `length` mogla da se upotrebi i nad listom logičkih vrednosti?

Indeksiranje

Naravno, često želimo da pristupimo pojedinačnim vrednostima u listi. Na primer, mogli bismo pitati: Koju je prvu ocenu iz matematike učenik dobio? Pojedinačnim vrednostima u listi, koje nazivamo *elementima*, pristupamo tako što navedemo na kom se rednom broju u listi nalaze, baš kao što učenik ima redni broj u dnevniku ili kao što sedište u vagonu voza ima svoj redni broj. Takav pristup elementima nazivamo *indeksiranje*, a redni broj elementa se zove *indeks*.

Mogli bismo pitati koja je bila prva ocena iz matematike koju je učenik dobio. Pristup prvoj oceni je prikazan u listingu ispod.

```
> grades !! 0
```

Operator indeksiranja

Operator indeksiranja u programskom jeziku Haskell je `!!` . Obrati pažnju da, kao i u većini drugih programskih jezika, indeksi u listama počinju od nule, a ne od jedan. Tako se prvi element u listi nalazi na poziciji `0`, drugi na poziciji `1`, i tako redom.

Ovakav način brojanja pozicija deo je tradicije programiranja. Iako u početku može delovati neobično, vrlo je lako navići se na njega.

Pitanja i zadaci

- Koja je signatura tipa operatora indeksiranja `(!!)` ?
- Da li je za indeksiranje bitno kog tipa su vrednosti koje se nalaze u listi?

Funkcije složenih tipova

Kao i u slučaju drugih tipova podataka, i u slučaju listi upotrebu proširujemo uvođenjem novih funkcija. To znači da možemo napisati funkcije koje se izvršavaju nad listama, odnosno koje kao parametre primaju liste. Na primer, možemo napisati funkciju `initialGrade` koja vraća prvu ocenu iz liste. Ova funkcija prima listu ocena kao parametar i vraća njen prvi element.

```
> initialGrade l = l !! 0
> initialGrade grades
5
```

- Koja je signatura tipa funkcije `initialGrade`?
- Kako izgleda evaluacija te funkcije?
- Da li evaluacija ove funkcije zavisi od broja elemenata u listi? Da li bi funkcija radila i da lista ima 10 elemenata, umesto 5?

Pitanja i zadaci

Kako bi izgledala funkcija koja računa prosek ocena učenika? Trebali bismo da saberemo sve ocene koje učenik ima i da ih podelimo brojem elemenata u listi. Videli smo da se broj elemenata u listi može dobiti pozivom funkcije `length`. Onda bi funkcija koja računa prosečnu ocenu mogla da izgleda kao u listingu ispod.

Funkcije nad elementima liste

```
> :{
| averageGrade l =
|   fromIntegral (l !! 0 + l !! 1 + l !! 2 + l !! 3 +
l !! 4)
|   / fromIntegral (length l)
| :}
> averageGrade [5,5,3,4,5]
4.4
```

Vidimo da prosečnu ocenu dobijamo tako što saberemo prvu, drugu, treću, četvrtu i petu ocenu iz liste koju smo poslali i dobijenu vrednost podelimo dužinom liste. Treba da napomenemo da funkcija `fromIntegral` pretvori ceo broj u decimalnu vrednost da bi se moglo izvršiti deljenje.

- Kako izgleda evaluacija funkcije `everageGrade` iz listinga iznad?
- Kako bi izgledala evaluacija da je poslata lista koja ima 6 ocena?
- Kako bi izgledala evaluacija da je poslata lista sa 4 ocene?

Pitanja i zadaci

Ovako smo dobili funkciju koja računa prosečnu ocenu ako učenik ima 5 ocena, ali šta kada dobije novu ocenu? Treba nam nova funkcija koja računa prosečnu ocenu za 6 ocena. Pa nam treba nova funkcija kada učenik ima 7 ocena, pa nova za 8 i tako dalje. Ovo sigurno nije dobro rešenje. Treba nam funkcija koja će se, prilikom evaluacije prilagoditi dubini liste nad kojoj se izvršava.

Koliko lista, toliko funkcija?

Razmislimo kako bismo izračunali zbir ocena učenika za listu sa proizvoljno mnogo ocena. Ako je lista prazna, zbir bi bio nula i prekinulo bi se izračunavanje. Ako nije prazna, uzeli bismo prvi element i dodali ga na zbir. Prvi element nam više ne treba, pa bismo

Kako sabiramo elemente u listi proizvoljne dužine?

ga izbacili iz liste. Nakon što ga izbacimo, proverili bismo da li je lista prazna. Ako je prazna, vratili bismo dosadašnji zbir. Ako nije, uzeli bismo ponovo njen prvi element, dodali ga na zbir i prvi element izbacili iz liste. I ove korake bismo ponavljali sve dok ne bi ispraznili listu. Hajde da probamo to da zapišemo kao program.

```
> :{
| sumGrades [] = 0
| sumGrades (x:xs) = x + sumGrades xs
| :}
> sumGrades [5,5,3,4,5]
22
```

Komplikovana ideja – elegantan program

Komplikovanu ideju izrazili smo elegantnim programskim kodom. Hajde da pogledamo šta se u ovom kodu dešava.

Linija `sumGrades [] = 0` kaže „ako je lista prazna, vrati 0“.

Linija `sumGrades (x:xs) = x + sumGrades xs` kaže „ako lista nije prazna, uzmi njen prvi element (`x`) i pozovi funkciju `sumGrades` nad ostatkom liste `xs`“. Samo smo definiciju računanja zbira ocena, kako smo je opisali u prethodnom paragrafu zapisali kao program.

Rekurzivne funkcije

Ovde vidimo jednu jako važnu osobinu funkcija: funkcije mogu da pozovu same sebe. Takve funkcije se nazivaju *rekurzivne funkcije*. Rekurzivne funkcije nam omogućava da složenost evaluiranog izraza prilagodimo složenosti strukture podataka nad kojima se izraz izvršava. Odnosno, zahvaljujući rekurziji ne treba nam posebna funkcija za listu sa 5 i za listu sa 6 elemenata.

Pitanja i zadaci

- Kako izgleda evaluacija izraza `sumGrades [5,5,3,4,5]` ?
- Kako bi izgledala evaluacija ovog izraza `sumGrades [5,5,3,4]`?
- Kako rekurzivne funkcije omogućuju da se složenost izraza prilagodi složenosti liste nad kojom se izvršava? Diskutuj o ovome u grupi.
- Kako izgleda signatura tipa funkcije `sumGrades`?
- Iako na prvu loptu deluje opskurno, pojam rekurzije je u stvari jako intuitivan i koristimo ga stalno u svakodnevnim aktivnostima. Identifikuj 5 aktivnosti iz svakodnevice koje se zasnivaju na rekurziji. Diskutuj u grupi.

Sada, kada smo napisali rekurzivnu funkciju koja računa zbir ocena učenika, lako ćemo napisati funkciju koja računa prosečnu ocenu učenika.

Korišćenje rekurzivnih funkcija

```
> averageGrade l = fromIntegral (sumGrades l) / fromIntegral (length l)
> averageGrade [5,5,3,4,5]
4.4
```

Obrati pažnju da funkcija `averageGrade` nije rekurzivna iako poziva rekurzivnu funkciju `sumGrades`. Kada pišemo funkciju `averageGrade` uopšte ne moramo znati da se funkcija `sumGrades` računa rekurzivno. To je logično, pošto funkcije služe kao mehanizam apstrakcije – kada je koristimo treba samo da znamo kako se zove, koje parametre prima i kog tipa je vrednost koju vraća, a ne treba da znamo kako radi.

Apstrahovanje rekurzije

Pitanja i zadaci

- Napiši rekurzivnu funkciju `lengthGrades :: [Int] -> Int` koja vraća broj ocena u listi. Pri tome nemoj koristiti funkciju `length`.
- Napiši funkciju `elemGrade :: Int -> [Int] -> Bool` koja proverava da li se određena ocena pojavljuje u listi.
- Napiši funkciju `reverseGrades :: [Int] -> [Int]` koja obrće redosled ocena u listi.
- Napiši funkciju `maxGrade :: [Int] -> Int` koja nalazi najveću ocenu u listi. Pretpostavi da lista nije prazna.
- Napiši funkciju `countGrade :: Int -> [Int] -> Int` koja broji koliko puta se određena ocena pojavljuje u listi.
- Napiši funkciju `appendGrades :: [Int] -> [Int] -> [Int]` koja spaja dve liste ocena u jednu koristeći rekurziju.
- Napiši funkciju `filterGreaterThan :: Int -> [Int] -> [Int]` koja vraća listu svih ocena koje su veće od zadatog broja.
- Napiši funkciju `increaseGrades :: [Int] -> [Int]` koja vraća listu u kojoj su sve ocene osim petica uvećane za 1 koristeći rekurziju.

Funkcije nad listama

Kreiranje prazne liste

Kao i u slučaju drugih tipova koje smo do sada videli (brojčani tip, logički tip, tip funkcije), postoji određeni broj ugrađenih funkcija koje možemo pozvati nad listama. Hajde da pogledamo šta sve sa listama

možemo da uradimo. Prvo ćemo videti kako možemo napraviti listu i pristupiti njenim elementima.

Praznu listu kreiramo pomoću praznih uglastih zagrada. Tako bi se u konstanti `l`, nakon izvršavanja sledećeg koda, nalazila lista bez ijednog elementa.

```
> l = []
```

Kreiranje liste sa elementima

Najčešće korišćen način da napravimo listu koja nije prazna je tako što elemente nabrojim redosledom po kome idu, odvojene zarezom i obuhvaćene uglastim zagradama. Već smo se susreli sa takvim načinom kreiranja liste.

```
> l = [5,5,3,4,5]
```

Pitanja i zadaci

- Rekli smo da sve u programu možemo interpretirati kao izraz i da se taj izraz evaluira. Kako bi izgledala evaluacija izraza `[5,5,3,4,5]`?

Kons operator

Sintaktički šećer

Prethodni zadatak je bio trik pitanje. Konstrukcija liste pomoću uglastih zagrada je samo *sintaktički šećer*, „lepši“ način da se zapiše izraz pomoću kog se konstruiše liste. Izraz za kreiranje liste koristi operator dve tačke (`:` čita se kons) koji na početak zadate liste doda jedan element. Tako bi kod ispod kreirao listu koja ima samo jedan element, `5`.

```
> l = (5:[])
> l
[5]
```

Kako bismo mogli da kreiramo listu sa više elemenata? Prosto bismo sukcesivno gradili izraz koji koristi kons operator.

Kons i lista više elemenata

```
> l = (5 : (5 : (3 : (4 : (5 : [])))))
> l
[5,5,3,4,5]
```

Sada vidimo da je `[5,5,3,4,5]` u stvari samo skraćeni zapis za izraz `(5:(5:(3:(4:(5:[])))))`.

Pitanja i zadaci

- Kako izgleda evaluacija izraza `(5:(5:(3:(4:(5:[])))))`?

Kons operator nam otkriva jednu važnu osobinu liste. Pogledajmo listu `(1 : (2 : (3 : [])))`. Od čega se ona sastoji? Od elementa `1` i ostatka liste, `(2 : (3 : []))`. A od čega se ova manja lista sastoji? Od elementa `2` i ostatka liste `(3 : [])`. A ova lista? Od elementa `3` i ostatka, a to je prazna lista `[]`. U svakom koraku u listi imamo prvi element (*glavu* liste) i ostatak (*rep* liste). Postoje ugrađene funkcije za pristupanje glavi i repu liste, koje se zovu `head` i `tail`.

Glava i rep

```
> l = [1, 2, 3]
> head l
1
> tail l
[2,3]
```

Razdvajanje liste na glavu i rep već smo koristili kada smo pričali o rekurzivnim funkcijama nad listama. Pošto je u funkciji `sumGrades x` bio prvi element liste, a `xs` lista koju čine svi elementi posle njega, kažemo da je `x` je bio glava liste, a `xs` rep liste.

```
> :{
| sumGrades [] = 0
| sumGrades (x:xs) = x + sumGrades xs
| :}
> sumGrades [5,5,3,4,5]
22
```

Pitanja i zadaci

- Kako bi operator indeksiranja `!!` implementirao kao rekurzivnu funkciju?

Konkatenacija

Dve liste možemo spojiti (konkatenirati) koristeći operator `++`. Ovako dobijena lista sastoji se od elemenata prve liste nakon kojih slede elementi druge liste.

```
> grades = [5,5,3,4,5]
> newGrades = grades ++ [4,5]
> newGrades
[5,5,3,4,5,4,5]
```

Pitanja i zadaci

- Šta je rezultat izvršavanja izraza `[1,2] ++ [3,4]`?
- Da li postoji razlika između izraza `1 : [2,3]` i `[1] ++ [2,3]` ? Ako misliš da postoji, argumentuj u grupi.
- Napiši funkciju `doubleList` koja primi listu koja radi kao što je navedeno u listingu ispod

```
> doubleList [1,2,3]
[1,2,3,1,2,3]
```

Opseg i operator `..`

Pomoću operatora `..` možemo kreirati listu elemenata u zadatom opsegu. Tako bismo list u brojeva između `1` i `5` mogli dobiti kao `[1 .. 5]`, kao što je prikazano listingom ispod.

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

Isecanje liste

Još dve korisne funkcije za rad sa listama su `take` i `drop`. Funkcija `take n lista` vraća listu koja sadrži prvih `n` elemenata iz zadate liste, a funkcija `drop n lista` vraća listu bez prvih `n` elemenata.

```
> grades = [5,5,3,4,5]
> take 3 grades
[5,5,3]
> drop 3 grades
[4,5]
```

Provera da li element postoji u listi

Funkcija `elem` proverava da li se zadata vrednost nalazi u listi. Listing ispod prikazuje korišćenje ove funkcije.

```
> grades = [5,5,3,4,5]
> elem 3 grades
True
> elem 2 grades
False
```

Pitanja i zadaci

- Kako izgleda signatura tipa funkcije `elem`?

Funkcija koja se primenjuje na sve elemente liste

Ranije u ovom poglavlju bio je zadatak da se implementira funkcija `increaseGrades` koja je sve ocene osim petice uvećavala za jedan. Hajde da pogledamo kako izgleda implementacija te funkcije.

```
> increaseGrade grade = if grade /= 5 then (grade + 1)
else grade
> :{
| increaseGrades [] = []
|     increaseGrades     (x:xs)     =     (increaseGrade
x):(increaseGrades xs)
| :}
```

Prvo smo napravili funkciju `increaseGrade` koja se odnosi na jednu ocenu u listi. Ukoliko je ocena različita od 5, naša funkcija je uveća za `1`. Ukoliko je `5`, ostaje nepromenjena. Zatim smo napisali rekurzivnu funkciju `increaseGrades` koja prođe kroz listu i primeni ovu funkciju `increaseGrade` na svaki element liste. Tako smo dobili listu uvećanih ocena.

Ali ovaj obrazac „prođi kroz listu i primeni funkciju na svaki element liste“ nije karakterističan samo za uvećavanje ocena u listi. Pošto je funkcija u programskom jeziku Haskell takođe vrednost, odnosno pošto postoji tip funkcije, mogli smo napisati funkciju koja primi listu i funkciju i vrati listu u kojoj je funkcija primenjena na svaki element liste. Ovu funkciju ćemo nazvati `myMap`.

Funkcija koja se primenjuje na sve elemente liste

```
> increaseGrade grade = if grade /= 5 then (grade + 1)
else grade
> decreaseGrade grade = if grade /= 1 then (grade - 1)
else grade
> :{
| myMap f [] = []
| myMap f (x:xs) = (f x):(myMap f xs)
| :}
> myMap increaseGrade grades
[5,5,4,5,5]
> myMap decreaseGrade grades
[4,4,2,3,4]
```

Napisali smo dve funkcije, `increaseGrade` i `decreaseGrade`. Prva uvećava za 1 sve ocene osim petice. Druga imanjuje za 1 sve ocene osim jedinice. Zatim smo napisali rekurzivnu funkciju `myMap` koja primi listu i funkciju. Ova funkcija vrati listu vrednosti dobijenih primenom funkcije na svaki element liste. Sada vidimo da funkciju `myMap` možemo iskoristi za uvećavanje ali i za umanjivanje ocena.

Šablon izmene svih vrednosti u listi prema prosleđenoj funkciji je jako čest slučaj u programiranju, tako da nismo morali da implementiramo funkciju `myMap` – funkcija `map` koja radi baš ovako postoji implementirana u programskom jeziku Haskell.

`map` funkcija

```
> map increaseGrade grades
[5,5,4,5,5]
> map decreaseGrade grades
[4,4,2,3,4]
```

Pitanja i zadaci

- Koja je signatura funkcije tipa `map`?
- Koristeći `map` funkciju pretvori listu brojeva u listu njihovih negativnih parnjaka.
- Koristeći `map` funkciju pretvori listu brojeva u listu njihovih apsolutnih vrednosti.
- Koristeći `map` funkciju pretvori listu brojeva u listu logičkih vrednosti u kojoj je `True` ako je broj manji od `3`, a `false` ako je veće od `3`.

Funkcija `map` transformiše listu tako što na svaki element liste primeni prosleđenu funkciju. Da li bismo, koristeći `map` funkciju mogli da transformišemo listu ocena tako što bismo u njoj ostavili samo ocene veće od 3? Ako bismo imali listu `grades = [5,5,3,4,5]`, rezultat bi trebao da bude `[5,5,4,5]`. Hajde da probamo da napišemo taj programski kod.

```
> grades = [5,5,3,4,5]
> map (\x -> if x > 3 then x else 0) grades
[5,5,0,4,5]
```

`map` funkcija prepakuje listu, ali ne može da izbaci ili doda element

Ovako nismo izostavili vrednosti manje od `4`, nego smo ih zamenili vrednošću `0`. U stvari, `map` samo može da prepakuje vrednosti iz jedne liste u drugu, ali će uvek očuvati dužinu originalne liste. Funkcija `map` ne može da „skrati“ listu tako što iz nje izostavi vrednosti. Zato ćemo napisati funkciju koja radi to što nam treba.

```
> grades = [5,5,3,4,5]
> :{
| leaveOutGrades [] = []
|   leaveOutGrades   (x:xs)   =   if   x   >   3   then
(x:leaveOutGrades(xs)) else leaveOutGrades xs
| :}
> leaveOutGrades grades
[5,5,4,5]
```

Filtriranje liste

Da li bismo, kao u slučaju `map`, ovu funkciju mogli da uopštimo? Mogli bismo na napišemo funkciju koja primi listu i funkciju koja primi broj, a vrati `True` ili `False` u zavisnosti da li je uslov ispunjen. I koja izostavi sve vrednosti iz liste za koje je dobijena vrednost `False`. Nazovimo je `myFilter`.

```
> :{
| myFilter f [] = []
| myFilter f (x:xs) = if (f x) then (x:myFilter f xs)
else myFilter f xs
| :}
> myFilter (\x -> x > 3) grades
[5,5,4,5]
```

Pitanja i zadaci

- Koja je signatura funkcije `myFilter`?
- Objasni kako radi funkcija `myFilter`.

Naravno, kao i u slučaju `map` funkcije, i ova funkcija predstavlja jedan jako čest scenario korišćenja liste i postoji ugrađena u programskom jeziku Haskell. Zove se `filter`.

`filter` funkcija

```
> filter (\x -> x > 3) grades
[5,5,4,5]
```

- Napiši funkciju koja za zadatu listu brojeva vraća listu svih brojeva koji su strogo između 2 i 6 (ne uključujući 2 i 6).
- Napiši funkciju koja za zadatu listu brojeva vraća listu duplo većih vrednosti svih brojeva većih od 5.
- Napiši funkciju koja za zadatu listu brojeva vraća listu kvadrata samo onih brojeva koji su veći od 3.
- Koristeći `map` i `filter` funkcije napiši funkciju `expand` koja primi listu brojeva, a vrati listu sačinjenu tako što je za svaki element `x` napravljen opseg brojeva od `1` do `x`. Primer izvršavanja ove funkcije dat je listingom ispod.

Pitanja i zadaci

```
> expand [1,2,3]
[1,1,2,1,2,3]
```

Implementaciju ove funkcije diskutuj u grupi.

Koristili smo funkciju `sum` koja sabere sve elemente liste. Hajde da je napišemo. Suma prazne liste je 0. Suma liste koja nije prazna je vrednost glave liste sabrana sa sumom repa lista. Kada smo ovako definisali funkciju, sada je možemo zapisati u programskom jeziku Haskell.

Sumiranje liste

```
> :{
| mySum [] = 0
| mySum (x:xs) = x + mySum xs
| :}
> grades = [5,5,3,4,5]
> mySum grades
22
```

- Kako izgleda evaluacija izraza `mySum grades` iz primera iznad?

Pitanja i zadaci

Da li bismo ovu funkciju mogli da uopštimo, pa da može i da pronađe proizvod svih elemenata u listi? Šta bi bilo ako bismo operator kojim se glava liste dodaje na rezultat prosledili kao parametar funkcije? Hajde da probamo.

```
> :{
| myFold f [] = 0
| myFold f (x:xs) = f x (myFold f xs)
| :}
> grades = [5,5,3,4,5]
> myFold (*) grades
0
```

Rezultat je 0. Zbog čega? Hajde da pogledamo kako izgleda evaluacija ovog izraza.

```
myFold (*) [5,5,3,4,5]
= 5 * myFold (*) [5,3,4,5]
= 5 * (5 * myFold (*) [3,4,5])
= 5 * (5 * (3 * myFold (*) [4,5]))
= 5 * (5 * (3 * (4 * myFold (*) [5])))
= 5 * (5 * (3 * (4 * (5 * myFold (*) []))))
= 5 * (5 * (3 * (4 * (5 * 0))))
= 5 * (5 * (3 * (4 * 0)))
= 5 * (5 * (3 * 0))
= 5 * (5 * 0)
= 5 * 0
= 0
```

Kada je naša funkcija evaluirala izraz `myFold (*) []` u `0`, sva preostala množenja su rezultovala nulom. Znači za slučaj u kom tražimo proizvod svih vrednosti inicijalna vrednost koju šaljemo treba da bude `1`, a u slučaju u kom sabiramo treba da bude `0`. Hajde da ovu vrednost izdvojimo u parametar funkcije.

```
> :{
| myFold f acc [] = acc
| myFold f acc (x:xs) = f x (myFold f acc xs)
| :}
> myFold (*) 1 grades
1500
> myFold (+) 0 grades
22
```

Parametar funkcije `acc` ima naizgled čudan naziv. Tako se zove jer predstavlja akumulator, odnosno parametar u koji će se, tokom procesiranja liste, sakupljati (akumulirati) vrednosti. U prvom slučaju akumulator je proizvod, a u drugom je to zbir.

`fold` i akumulator

- Kako izgleda evaluacija izraza `myFold (*) 1 grades`?
- Kako izgleda evaluacija izraza `myFold (+) 0 grades`?

Pitanja i zadaci

Naravno, kao i u slučaju `map` i `filter`, i ova funkcija je toliko korisna da postoji implementirana u programskom jeziku Haskell. Pošto nije svejedno da li ovakvu transformaciju vršimo sa početak liste ili sa kraja liste, postoje dve verzije ove funkcije, `foldl` i `foldr`. Prema ovoj podeli, mi smo implementirali `foldr` funkciju.

`foldr` i `foldl`

```
> foldr (+) 0 grades
22
```

- Napiši izraz koji za zadatu listu brojeva vraća zbir kvadrata svih elemenata koji su veći od 2. Koristi `filter` da izdvojiš te brojeve, `map` da ih kvadriraš, i `foldr` da ih sabereš.
- Napiši izraz koji računa zbir apsolutnih vrednosti svih negativnih brojeva u listi. Prvo koristi `filter` da pronađeš negativne brojeve, `map` za računanje apsolutne vrednosti, a `foldr` za sabiranje.
- Za listu celih brojeva, pronađi pozitivne brojeve, udvostruči ih i izračunaj zbir. Koristi `filter` za izbor pozitivnih, `map` za dupliranje, i `foldr` za sabiranje.
- Napiši izraz koji za listu brojeva uzima samo one manje od 10, zatim za svaki računa razliku `10 - x`, i sve to sabira. Koristi `filter`, `map` i `foldr`.
- Implementiraj `myFoldl` funkciju.

Pitanja i zadaci

Još jedan primitivni tip, kao tip brojeva ili logički tip, koji se često koristi je karakter (`Char`). Ovaj tip služi za predstavljanje alfanumeričkih karaktera (slova i cifara) i specijalnih znakova (poput razmaka i interpunkcije). Karakter se zadaje tako što se navede između dva apostrofa.

Char – slova, cifre, interpunkcija

```
> c = 'a'
> c
'a'
```

Pojedinačne karaktere možemo porediti.

```
> 'a' == 'b'
False
> 'a' == 'a'
True
```

String – lista karaktera

Sam po sebi tip `Char` nije naročito interesantan, odnosno ne pruža nam puno mogućnosti upotrebe, pošto predstavlja samo pojedinačni znak. Vrlo retko smo u situaciji da nešto opisujemo jednim jedinim karakterom. Prava snaga karaktera se ispoljava kada se grupišu u listu. Lista karaktera služi za predstavljanje teksta i naziva se `String`.

S obzirom na činjenicu da je `String` lista karaktera, možemo ga zadati isto kao što smo zadavali i listu brojeva.

```
s = ['h','e','l','l','o',' ','w','o','r','l','d']
```

Sintaktički šećer

Međutim, ova notacija nije naročito praktična. Zamisli koliko bi bilo teško čitati knjigu koja bi bila štampana u ovom formatu! Zbog toga postoji *sintaktički šećer*, intuitivniji zapis stringova. Stringove možemo zadati tako što prosto tekst obuhvatimo navodnicima.

```
> s = "hello world"
```

Pitanja i zadaci

- Gde se sve u svakodnevnom životu koriste tekstualni podaci? Probaj da identifikuješ što raznorodnije primere, koji prevazilaze okvir knjiga i časopisa.
- Kako izgleda evaluacija izraza `"hello world"`

Lista karaktera

Pošto String nije novi tip podataka, nego je samo lista karaktera, nad njim možemo da radimo sve što smo mogli da radimo i sa listom brojeva. Odnosno možemo da indeksiramo string, pristupimo glavi i repu string i koristimo kons operator kao i u slučaju liste brojeva.

```
> s = "hello world"
> s !! 0
'h'
> head s
'h'
> tail s
"ello world"
> '!':s
"!hello world"
```

Naravno, možemo koristiti i funkcije za transformaciju, odnosno `map`, `filter` i `foldr/foldl`.

Pogledajmo kako bismo mogli, koristeći `map` funkciju da implementiramo program koji string prebaci u velika slova. Prvo nam treba funkcija koja jedno slovo (jedan karakter) pretvori u veliko. Pošto već postoji gotova funkcija za to, nećemo je implementirati nego iskoristiti postojeće rešenje.

```
> import Data.Char
> toUpper 'a'
'A'
> toUpper 'A'
'A'
> toUpper '5'
'5'
```

Uvoz biblioteke

Ključna reč `import` omogućava korišćenje funkcija iz gotovih Haskell biblioteka, a `import Data.Char` konkretno omogućava rad sa funkcijama za obradu karaktera. Vidimo da `toUpper` vrati veliko slovo ako je poslato malo slovo. Inače vrati vrednost koju smo joj poslali.

Pitanja i zadaci

- Kako izgleda signatura tipa funkcije `toUpper`?

Sada, kada smo uvezli funkciju `toUpper` jednostavno je poslati je `map` funkciji koja će transformisati ceo string u velika slova.

```
> s = "hello world"
> map toUpper s
"HELLO WORLD"
```

U istoj biblioteci `Data.Char` nalazi se i funkcija `isUpper` koja proverava da li je slovo veliko ili malo. Ako je veliko vrati `True`, ako je malo vrati `False`.

```
> isUpper 'H'
True
> isUpper 'h'
False
```

Ovu funkciju mogli bismo iskoristiti na primer da iz stringa izbacimo sva mala slova.

```
> s = "HeLLo WoRLD"
```

```
> filter isUpper s
"HLLWRLD"
```

Nad stringovima možemo koristiti i `foldr`/`foldl` funkcije. Na primer, mogli bismo napisati kod koji broji koliko puta se neko slovo pojavljuje u stringu.

```
> s = "hello world"
> foldr (\c acc -> if c == 'l' then acc + 1 else acc)
0 s
3
```

U ovom primeru smo `foldr` funkciji poslali jednu neimenovanu funkciju koja primi karakter (parametar `c`) i akumulator (parametar `acc`). U slučaju da je karakter baš `'l'` akumulator će se uvećati za `1`. Ukoliko nije, ostaće neizmenjen. Pošto `foldr` u ovom slučaju koristimo za brojanje, inicijalna vrednost akumulatora će biti `0`.

Pitanja i zadaci

- Koristeći `foldr` funkciju napiši programski kod koji obrne listu. Ako bismo ga primenili na `"hello world"` rezultat izvršavanja bi bio `"dlrow olleh"`. Obrati pažnju da akumulatorska kolekcija ne mora da bude numerička vrednost već može biti i lista.
- Funkcija `isAlpha` iz `Data.Char` vrati `True` ukoliko je karakter slovo. Ako nije, odnosno ako je cifra, interpunkcija ili specijalni karakter, vratiće `False`. Koristeći `isAlpha` i `foldr` napiši programski kod koji prebroji koliko ima slova u stringu.
- Istu funkcionalnost implementiraj koristeći `isAlpha`, `filter` i `lenght`.
- Koristeći `map` funkciju napiši programski kod koji svaki razmak u tekstu zameni crticom.

Do sada smo videli listu kao složeni tip koji se dobija ponavljanjem elemenata istog tipa. Tako smo imali listu brojeva ili listu karaktera, odnosno string. Ali da li složene tipove možemo graditi od različitih primitivnih tipova, tako da na primer imamo tip koji čine i brojevi i stringovi? Koja bi bila situacija u kojoj bismo to koristili?

Par

Ako bismo, na primer hteli da kažemo da je Petar Petrović dobio ocenu 5 na kontrolnom, treba nam složen tip u kom će prvi element biti ime tipa string (`"Petar Petrovic"`), a druga vrednost će biti ocena (broj `5`). Za to služi tip par.

```
> grade = ("Petar Petrovic", 5)
> grade
("Petar Petrovic",5)
```

Prvi i drugi element para

Par dobijemo tako što dve vrednosti odvojene zarezom obuhvatimo u zagrade. U listi smo elementima mogli da pristupamo po indeksu čime smo zadavali poziciju elementa u listi. Slično je i sa parovima – u njima elementima pristupamo pomoću funkcija `fst` (prvom elementu) i `snd` (drugom elementu).

```
> fst grade
"Petar Petrovic"
> snd grade
5
```

Pristup elementima para

Elementima para mogli smo pristupiti kroz dodelu vrednosti, kao što je prikazano listingom dole.

```
> student = ("Petar Petrovic", 5)
> (name, grade) = student
> name
"Petar Petrovic"
> grade
5
```

Vidimo da je ovako konstanta `name` dobila prvu vrednost, a konstanta `grade` dobila drugu vrednost.

Kompozicija složenih tipova

Postavlja se pitanje da li gradivni blok složenog tipa uvek mora da bude prost tip. Naravno, složene tipove možemo praviti i od složenih tipova, baš kao što smo složene izraze mogli da pravimo i povezivanje složenih izraza. Hajde da napravimo listu parova.

```
>:{
| grades = [
|    ("Petar Petrovic", 5),
|    ("Jovana Jovanovic", 5),
|    ("Stevan Stevanovic", 3),
|    ("Marko Markovic", 4),
|    ("Iva Ivkovic", 5)
|    ]
| :}
```

Napravili smo listu učenika sa njihovim ocenama. Nad ovom listom bismo mogli da primenimo sve operacije koje smo nad listama već radili. Tako bismo mogli da izdvojimo sve učenike koji su dobili ocenu 5.

```
> filter (\x -> (snd x) == 5) grades
[("Petar Petrovic",5),
("Jovana Jovanovic",5),
("Iva Ivkovic",5)]
```

Ovaj šablon korišćenja liste parova je toliko čest da postoji funkcija `zip` koja spoji dve liste tako što napravi novu listu čiji elementi su parovi po jednog elementa iz prve i druge liste.

```
> :{
| students = [
|   "Petar Petrovic",
|   "Jovana Jovanovic",
|   "Stevan Stevanovic",
|   "Marko Markovic",
|   "Iva Ivkovic"
|   ]
| :}
> grades = [5,5,3,4,5]
> zip students grades
[("Petar Petrovic",5),
("Jovana Jovanovic",5),
("Stevan Stevanovic",3),
("Marko Markovic",4),
("Iva Ivkovic",5)]
```

Sekvenca

Ali postoje situacije u kojima je potrebno prikazati više od dva podatka potencijalno različitih tipova u složenom tipu. Na primer, prirodnije bi bilo da podaci o ocenama imaju razdvojeno ime, prezime i ocenu. U stvari, par je specijalni slučaj složenog tipa koji se zove *sekvenca* (*tuple*) koji može da ima proizvoljno mnogo elemenata različitog tipa.

```
> student = ("Petar", "Petrovic", 5)
> student
("Petar","Petrovic",5)
```

Pristup elementima sekvence

Prvom i drugom elementu sekvence i dalje možemo pristupiti preko funkcija `fst` i `snd`, ali ostalim elementima moramo pristupati kroz dodelu. Mogli smo napisati funkcije koje pristupaju elementima sekvence i tako apstrahovati pristup kroz dodelu.

```
> firstName (fn, ln, g) = fn
> lastName (fn, ln, g) = ln
> grade (fn, ln, g) = g
> firstName student
"Pera"
> lastName student
"Peric"
> grade student
5
```

Kompozicija složenih tipova

Naravno, kao što lista može da sadrži sekvence i sekvenca može da sadrži listu. Tako bismo mogli napraviti listu sekvenci učenika, pri čemu svaka sekvenca koja predstavlja učenika ima ime, prezime i listu ocena. Faktički, napravili smo složeni tip podataka koji predstavlja školski dnevnik.

```
> :{
| students = [
|    ("Pera", "Peric", [5,4,5,5]),
|    ("Jovana", "Jovanovic", [5,5,4]),
|    ("Stevan", "Stevanovic", [3,3,2]),
|    ("Marko", "Markovic", [4,4,5]),
|    ("Iva", "Ivkovic", [5,5,5,5])
|    ]
| :}
```

Funkcije koje dobavljaju ime i prezime bile bi iste kao i u prvom slučaju, ali bi sada mogli dodati funkciju `averageGrades` koja računa prosečnu ocenu učenika.

```
> firstName (fn, ln, g) = fn
> lastName (fn, ln, g) = ln
> grades (fn, ln, g) = g
> average l = fromIntegral (sum l) / fromIntegral (len
l)
> averageGrades = average . grades
> averageGrades (students !! 0)
4.75
```

Da bi se izračunala prosečna ocena učenika, prvo je potrebno dobaviti listu ocena učenika, koja je lista celih brojeva, a zatim izračunati srednju vrednost te liste. Funkciju `averageGrades` zato možemo definisati kao kompoziciju funkcija `average` i `grades`.

Čitljiv ispis složenog tipa

Hajde sada da napišemo funkciju koja ispisuje listu u kojoj svaki red čine prezime učenika, ime učenika i njegova prosečna ocena. Prvo ćemo napisati funkciju koja prikazuje podatke o jednom studentu.

```
> :{
| studentToStr student =
|   firstName student ++ " " ++
|   lastName student ++ " " ++
|   show (averageGrade student) ++ "; "
| :}
> studentToStr (students !! 0)
"Pera Peric 4.75; "
```

Obrati pažnju da smo `firstName` i `lastName` konkatenirali pomoću operatora `++`, ali da smo za `averageGrade` morali da pozovemo funkciju `show`. To je zato što je povratna vrednost funkcije `average` broj, a da bismo broj konkatenirali sa stringom moramo prvo da ga konvertujemo u string. Funkcija `show` upravo pretvara broj u string i to nam omogućuje.

Da bismo dobili spisak svih studenta sa prosečnim ocenama, treba svaku pojedinačnu sekvencu koja predstavlja učenika da pretvorimo u string. To ćemo najlakše uraditi pomoću `map` funkcije.

```
> map studentToStr students
["Pera Peric 4.75; ",
"Jovana Jovanovic 4.666666666666667; ",
"Stevan Stevanovic 2.6666666666666665; ",
"Marko Markovic 4.333333333333333; ",
"Iva Ivkovic 5.0; "]
```

Sada, kada smo videli kako da listu sekvenci koje predstavljaju studente pretvorimo u listu stringova, ostaje nam još da sve stringove iz liste sakupimo u jedan veliki string. To ćemo uraditi pomoću funkcije `foldr`.

```
> foldr (++) "" (map studentToStr students)
"Pera Peric 4.75; Jovana Jovanovic 4.666666666666667;
Stevan Stevanovic 2.6666666666666665; Marko Markovic
4.333333333333333; Iva Ivkovic 5.0; "
```

Hajde da omogućimo unos nove ocene. Da bismo uneli novu ocenu, moramo znati ime, prezime učenika, kao i ocenu. Prvo ćemo napisati funkciju `updateStudent`. Ova funkcija proveri da li su se ime i prezime studenta kome treba uneti ocenu poklopili sa vrednostima u sekvenci koja predstavlja učenika. Ako nisu, ternarni izraz vrati sekvencu koja mu je i prosleđena. Ako jesu, vrati sekvencu sa dodatom ocenom.

Izmena vrednosti složenog tipa

```
> :{
|     updateStudent     gradeToAdd     targetFirstName
targetLastName (firstName, lastName, grades) =
|      if firstName == targetFirstName && lastName ==
targetLastName
|             then (firstName, lastName, gradeToAdd :
grades)
|          else (firstName, lastName, grades)
| :}
```

Unos nove ocene u listu učenika je sada jednostavan. Prosto treba primeniti `updateStudent` pomoću `map` na celu listu.

```
> :{
|      insertGrade      gradeToAdd      targetFirstName
targetLastName studentList =
|      map  (updateStudent  gradeToAdd  targetFirstName
targetLastName) studentList
| :}
> insertGrade 5 "Pera" "Peric" students

[("Pera","Peric",[5,5,4,5,5]),("Jovana","Jovanovic",[
5,5,4]),("Stevan","Stevanovic",[3,3,2]),("Marko","Mar
kovic",[4,4,5]),("Iva","Ivkovic",[5,5,5,5])]
```

Obrati pažnju da je ulaz funkcije `updateStudent` ograničen pomoću parcijalne aplikacije funkcije, odnosno da su joj fiksirani parametri `gradeToAdd`, `targetFirstName` i `targetLastName`, kada je poslata `map` funkciji.

Ovaj pristup bismo, naravno, mogli da iskoristimo i za brisanje ocene studenta. Hajde da napišemo funkciju `deleteFirstGrade` koja primi ime i prezime učenika i listu učenika sa ocenama, a vrati listu u kojoj je za tog učenika obrisana prva ocena.

Brisanje vrednosti u složenom tipu

```
> :{
| students = [
|    ("Pera", "Peric", [5,4,5,5]),
|    ("Jovana", "Jovanovic", [5,5,4]),
|    ("Stevan", "Stevanovic", [3,3,2]),
|    ("Marko", "Markovic", [4,4,5]),
|    ("Iva", "Ivkovic", [5,5,5,5])
|    ]
| :}
> :{
|    updateStudent    targetFirstName    targetLastName
(firstName, lastName, grades) =
|     if firstName == targetFirstName && lastName ==
targetLastName then (firstName, lastName, tail grades)
|    else (firstName, lastName, grades)
| :}
> :{
|   deleteFirstGrade   targetFirstName   targetLastName
students =
|    map (updateStudent targetFirstName targetLastName)
students
| :}
> deleteFirstGrade "Pera" "Peric" students
[("Pera","Peric",[4,5,5]),("Jovana","Jovanovic",[5,5,
4]),("Stevan","Stevanovic",[3,3,2]),("Marko","Markovi
c",[4,4,5]),("Iva","Ivkovic",[5,5,5,5])]
```

Napisali smo funkciju `updateStudent` koja proveri da li su se ime i prezime učenika poklopili sa prosleđenim vrednostima, i ako jesu, vrati podatke o tom učeniku sa izbačenom prvom ocenom. Onda smo tu funkciju pomoću `map` primenili na sve elemente liste učenika.

- Da li bi funkcija za brisanje prve ocene mogla da se unapredi? Šta bi se desilo da je primenimo na listu učenika u kojoj neki od učenika nema nijednu ocenu?
- Napiši program koji rukuje katalogom knjiga. Svaka knjiga ima naslov, autora i broj strana.

```
books = [
  ("Na Drini ćuprija", "Ivo Andrić", 314),
  ("Seobe", "Miloš Crnjanski", 472),
  ("Hazarski rečnik", "Milorad Pavić", 366),
  ("Prokleta avlija", "Ivo Andrić", 121),
  ("Stranac", "Albert Camus", 123)
]
```

Zadaci:

1. Napiši funkcije `title`, `author` i `pages` pomoću kojih se pristupa pojedinačnim poljima knjige.
2. Napiši funkciju `shortBooks` koja prima listu svih knjiga a vraća listu knjiga kraćih od 200 strana.
3. Napiši funkciju `byAuthor` koja za zadatog autora vraća listu knjiga koje je napisao.
4. Napiši funkciju `bookToStr` koja konvertuje knjigu (sekvencu) u string.
5. Napiši funkciju `booksToStr` koja primi listu knjiga, a vrati string prilagođen za ispis.
6. Napiši funkciju `deleteBook` koja primi autora, naslov knjige i listu knjiga, a vrati listu knjiga sa izbačenom knjigom tog autora i pod tim naslovom.

- Napiši program koji rukuje kartama u bioskopu. Bioskopska karta ima ime na koje je rezervisana, film, broj karte i cenu karte, odnosno karta je sekvenca (`name`, `movie`, `number_of_tickets`, `ticket_price`)

```
sales = [
  ("Ana", "Dina", 2, 400),
  ("Milan", "Matrix", 1, 450),
  ("Jelena", "Dina", 3, 400),
  ("Nikola", "Oppenheimer", 1, 500)
  ("Milan", "Dina", 1, 400),
]
```

Zadaci:

1. Napiši funkcije `name`, `movie`, `number`, `number_of_tickets`, `ticket_price` kojima se pristupa poljima karte.
2. Napiši funkciju `totalRevenue` koja računa ukupnu cenu prodatih karata.
3. Napiši funkciju `customerToStr` koja, primi listu karti i ime na koji su izvršene rezervacija, a vrati string oblika

```
"Ana bought 2 tickets for Dune - total 800 RSD"
```

Obrati pažnju da ista osoba može imati rezervacije za različite filmove. Razmisli kako treba da izgleda format povratnog stringa u tom slučaju.

4. Napiši funkciju `filterByMovie` koja primi listu karata i naziv filma, a vrati listu karata za taj film.
5. Napisati funkciju `mostTickets` koja primi listu karata, a vrati ime osobe koja je rezervisala najviše karata.
6. Napisati funkciju `bestCustomer` koja primi listu karata, a vrati ime osobe koja je potrošila najviše novca na karte.
7. Napisati funkciju `averagePrice` koja izračunava prosečnu cenu karte.
8. Napiši funkciju `deleteReservation` koja primi ime osobe, naslov filma i listu karata, a vrati listu karata sa izbačenom rezervacijom za taj film i tu osobu.
9. Napiši funkciju `updateReservation` koja primi ime osobe, naslov filma, novi broj karti, novu cenu i listu rezervacija, a vrati novu listu rezervacija u kojoj je za tu rezervaciju postavljen novi broj karti i nova cena.

- Napiši program koji rukuje porudžbinama u restoranu. Svaka porudžbina ima broj stola i listu stavki, a svaka stavka ima naziv jela i cenu, odnosno program rukuje listom porudžbina oblika `(table_number, [(dish_name, price)])`. Primer liste je dat listingom ispod.

```
tables = [
  (1, [("Čorba", 250), ("Pljeskavica", 500), ("Sok",
150)]),
  (2, [("Pasta", 600), ("Sok", 150)]),
  (3, [("Salata", 300), ("Voda", 100)])
]
```

Zadaci:

1. Napiši funkciju `totalForTable` koja primi redni broj stola i listu porudžbini, a vrati ukupan iznos računa za taj sto.
2. Napiši funkciju `tableToStr` koja primi redni broj stola i listu porudžbina, a vrati string oblika "`Sto 1: Čorba – 250 RSD, Pljeskavica – 500 RSD, Sok – 150 RSD; Ukupno: 900 RSD`".
3. Napiši funkciju `mostExpensiveItems` koja primi listu porudžbina, a vrati listu sa najskupljim stavkama za svaki sto.
4. Napiši funkciju totalRevenu koja primi listu stavki, a vrati sumu svih cena za sve stavke.
5. Napiši funkciju `filterByOrder` koja primi naziv jela, a vrati listu rednih brojeva stolova koji imaju porudžbinu tog jela.
6. Napiši funkciju `deleteOrder` koja primi redni broj stola i listu porudžbina, a vrati listu iz koje je izbačena ta porudžbina.
7. Napiši funkciju `deleteItem` koja primi redni broj stola, naziv stavke i listu porudžbina i vrati listu u kojoj je za taj sto izbačena ta stavka.
8. Napiši funkciju `addItem` koja primi redni broj stola, naziv stavke, cenu i listu porudžbina, a vrati listu porudžbina u kojoj je ta, nova stavka dodata za sto pod zadatim rednim brojem.

Polimorfizam

- Značenje i kontekst
- Ad hoc polimorfizam
- Klase tipova
- Rekordi
- Parametarski polimorfizam

Značenje i kontekst

Šta znači reč „otvoriti“? Možemo „otvoriti prozor“ i u tom slučaju reč otvoriti označava fizičku radnju. Možemo „otvoriti firmu“, što znači da smo osnovali firmu. Možemo pred nekim „otvoriti srce“ odnosno biti potpuno iskreni. Možemo „otvoriti knjigu“ što znači da smo počeli da je čitamo. Kada se kartamo, možemo „otvoriti karte“. U stvari, reč „otvoriti“ se može upotrebiti na puno načina, odnosno može značiti puno toga. Šta neka reč konkretno znači zavisi od konteksta u kom je ta reč upotrebljena.

Funkcija sa više značenja, zavisno od konteksta ili

Programiranje verno odražava ovu osobinu jezika, pa *funkcija može da se ponaša različito u zavisnosti od konteksta u kom je upotrebljena*. A kontekst u kom je funkcija upotrebljena predstavljamo *tipovima vrednosti nad kojima se ona izvršava*. Funkcije koje se različito izvršavaju u zavisnosti od tipova vrednosti nad kojim se pozivaju zovemo *polimorfne funkcije*.

Različiti tipovi u istoj funkciji

Na polimorfizam bismo mogli gledati i na obrnut način – *vrednosti različitih tipova mogu se upotrebiti u istoj funkciji*. Primere ovakvog polimorfizma takođe često srećemo u svakodnevnom životu. Na primer, možemo reći „zapakuj u kutiju“. Nije bitno da li pakujemo knjigu, poklon ili jabuku — sve dok možemo da ih stavimo u kutiju, funkcija „zapakuj u kutiju“ ostaje ista. U tom slučaju, svi navedeni tipovi (vrste stvari koje se pakuju u kutiju) se tretiraju na isti način.

Dve vrste polimorfizma

Shodno tome, postoje dve vrste polimorfizma.

1. *Ad hoc polimorfizam* u kom pišemo različite implementacije iste funkcije u zavisnosti od tipova nad kojima se funkcija izvršava.
2. *Parametarski polimorfizam* u kom pišemo funkcije koje se izvršavaju na isti način za različite tipove.

Pitanja i zadaci

- Gde sve u svakodnevnom životu susrećemo polimorfizam? Koje polimorfne operacije susrećemo u gradnji kuće, u pripremi jela, u školovanju?

Polimorfizam je važna tema u programiranju i posvetićemo mu ovo poglavlje. Počećemo sa ad hoc polimorfizmom, odnosno, prvo ćemo videti kako možemo da pravimo različite implementacije funkcija u zavisnosti od tipova parametara.

Numerički tipovi podataka

U prethodnim poglavljima smo se susreli sa numeričkim podacima. Videli smo da možemo da napišemo neki broj, na primer 3 i da ga koristimo u složenom izrazu. I pričali smo o „numeričkom tipu“ pomoću kog se predstavljaju brojevi. Ali da li su *svi* brojevi istog tipa? Nisu. U programskom jeziku Haskell ima nekoliko tipova podataka koji predstavljaju brojeve:

1. `Int` – celi brojevi u ograničenom opsegu[7].
2. `Integer` – celi brojevi bez ograničenog opsega.
3. `Float` – manje precizni realni brojevi.
4. `Double` – precizniji realni brojevi.
5. `Rational` – racionalni brojevi, omogućuju tačne operacije sa razlomcima.
6. `Complex` – kompleksni brojevi.

Brojevi i polimorfizam

Postoje operacije, kao što je sabiranje, koje možemo da radimo nad svim brojčanim tipovima. Na primer sve brojeve možemo da sabiramo.

```
> (3 :: Int) + (2 :: Int)
5
> (3 :: Float) + (2 :: Float)
5.0
```

Obrati pažnju da smo, kada smo napisali `(3 :: Float)` u stvari programsko okruženje „naterali“ da izraz `3` tretira kao da je tipa `Float`.

[7] Opseg tipa `Int` je od -9223372036854775808 do 9223372036854775807 što je uslovljeno *fizičkom reprezentacijom* ovog tipa na računaru. Čitaoce koji hoće da nauči kako se tipovi prestavljaju u računaru upućujem na oblast *arhitektura računara*.

Ali postoje i operacije koje možemo da radimo samo nad nekim brojčanim tipovima. Na primer, brojeve tipa `Int` ne možemo da delimo, a brojeve tipa `Float` možemo.

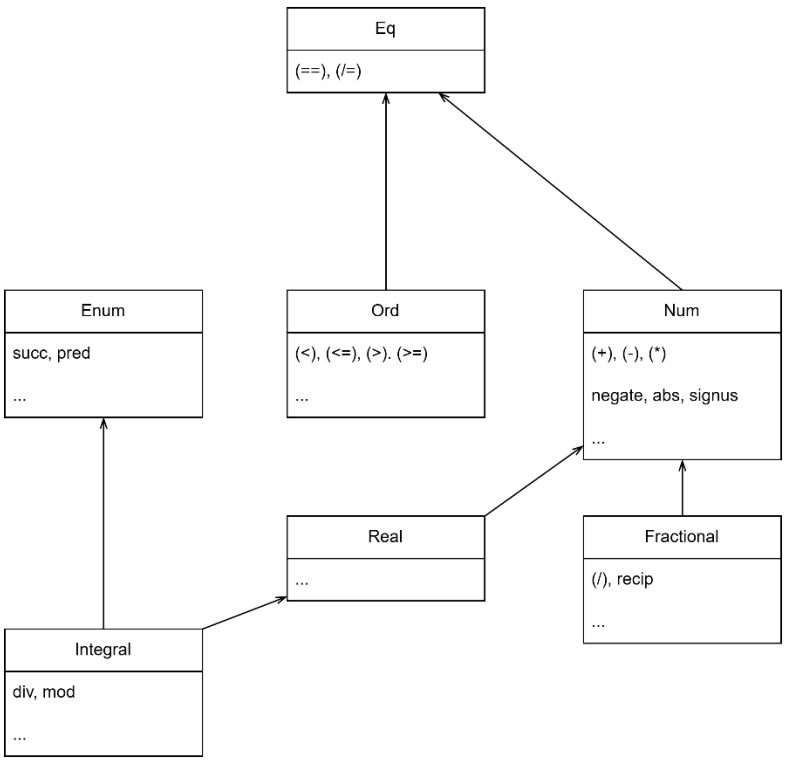
```
> (3 :: Int) / (2 :: Int)

<interactive>:14:12: error:
    * No instance for (Fractional Int) arising from a
use of `/'
    * In the expression: (3 :: Int) / (2 :: Int)
      In an equation for `it': it = (3 :: Int) / (2 ::
Int)

> (3 :: Float) / (2 :: Float)
1.5
```

Klase tipova

Kao što vidimo i `Int` i `Float`, možemo da sabiramo, ali ne možemo da delimo `Int`, nego samo `Float`. Odnosno, tipove možemo *grupisati* prema funkcijama koje nad njima možemo da pozovemo. Te grupe nazivamo *klase tipova* (eng., *type classes*). Hajde da pogledamo deo ugrađenih klasa tipova za rad sa brojevima i njihove funkcije u programskom jeziku Haskell.



Klasa `Eq`

Počećemo sa klasom tipova `Eq`. Naziv ove klase izveden je iz reči *equality*, odnosno ova klasa predstavlja tipove za čije vrednosti možemo da proverimo da li su jednake. Već smo videli da brojeve svih tipova možemo da uporedimo, odnosno možemo da nad njima pozovemo relacione operatore == i /=. Ali da li možemo uporediti i vrednosti koje nisu brojevi, koje su drugačijih tipova?

```
> True == False
False
> [1,2] == [1,2]
True
> "Hello world" == "Hello world"
True
> ("Pera", "Peric", [5,5,3,4,5]) == ("Pera", "Peric",
[5,5,3,4,5])
True
> ("Pera", "Peric", [5,5,3,4,5]) == ("Pera", "Peric",
[5,5,3,4])
False
```

Vidimo da možemo da poredimo i jednakost logičkih vrednosti, stringova, listi i ostalih složenih tipova. Klasa tipova nad kojima je moguće proveriti jednakost veoma je široka i obuhvata skoro sve tipove u programskom jeziku Haskell, uključujući i `Bool`, `String`, `List` i `Tuple`. Naravno, toj klasi tipova pripadaju i svi brojčani tipovi. Pored operatora provere jednakosti (==), ova klasa tipova zahteva da tipovi koji joj pripadaju podržavaju i operator provere nejednakosti (/=).

Pitanja i zadaci

- Kako radi poređenje jednakosti dve liste?
- Kako radi poređenje jednakosti dve sekvence?

Možemo da uporedimo da li su jednake dve vrednosti bilo kog tipa, sve dok taj tip pripada klasi tipova `Eq`. Ali da li možemo da uporedimo da li su jednake dve vrednosti različitih tipova?

```
> 3 == "3"

<interactive>:2:1: error:
    * No instance for (Num String) arising from the
literal `3'
    * In the first argument of `(==)', namely `3'
      In the expression: 3 == "3"
      In an equation for `it': it = 3 == "3"
```

Ne možemo. Pokušali smo da uporedimo vrednost tipa `Int` sa vrednošću tipa `String` i dobili smo grešku. Upravo tome nam tipovi služe: da odrede nad kojim vrednostima funkcija može, a nad kojim ne može da se pozove.

Kada smo, u poglavlju 4, pričali o signaturi tipa za funkciju, rekli smo da ona određuje kog tipa su parametri i povratna vrednost funkcije. Tako bi operator (==) mogao da primi `Int` i `Int`, a da vrati `Bool`.

```
(==) :: Int -> Int -> Bool
```

A mogao bi da primi i `String` i `String`, a da vrati `Bool`.

```
(==) :: String -> String -> Bool
```

A mogao bi da primi i `Tuple` i `Tuple`, a da vrati `Bool`. I `Bool` i `Bool`, a da vrati `Bool`. Ali ne bi mogao da primi `Int` i `String`, a da vrati `Bool`. Odnosno, treba nam mehanizam da kažemo da operator `(==)` može da primi vrednosti *bilo kog* tipa (koji pripada klasi `Eq`) sve dok su obe vrednosti *istog* tipa. I da će vratiti vrednost koja je tipa `Bool`. Za to nam služe *varijable tipa*. Hajde da pogledamo kako signatura operatora `(==)` zaista izgleda.

Varijabla tipa

```
...
class Eq a where
  (==) :: a -> a -> Bool
  ...
```

Prva linija koda kaže da imamo tip `a`, *bilo koji* tip koji pripada klasi `Eq`. Druga linija kaže da operator `(==)` primi dve vrednosti tipa `a`[8], a da vrati vrednost tipa `Bool`.

Pitanja i zadaci

- U poglavlju 5 je bilo nekoliko zadataka koji su tražili da se napišu signature tipova za funkcije. Ovi zadaci su osmišljeni tako da te navedu na napraviš "korisne greške" koje će ti u ovom poglavlju pomoći da razumeš varijable tipa. U GHCI (pomoću komande `:i`) proveri signature za te funkcije. Greške koje si napravio kada si inicijalno radio zadatke diskutuj u grupi.

[8] Iskaz da funkcija primi dve vrednosti treba uzeti sa zrncem soli. Naravno da zbog kariinga svaka funkcija prima jednu vrednost i vraća jednu vrednost. Ispravno bi bilo reći da operator `(==)` primi jednu vrednost tipa `a`, a vrati funkciju koja primi jednu vrednost tipa `a` i vrati vrednost tipa `Bool`.

Klasa `Ord`

Sledeća klasa tipova u dijagramu je `Ord`. Ova klasa tipova je *potklasa* klase `Eq`, što je predstavljeno strelicom koja ih povezuje. To znači da će svaki tip koji pripada klasi `Ord` automatski pripadati i klasi `Eq`. Ova klasa nameće obavezu da tipovi koji joj pripadaju moraju da podrže operatore manje (<), manje ili jednako (<=), veće (>) i veće ili jednako (>=). Klasa nameće još neke funkcije koje nećemo obrađivati u knjizi.

Logično je da možemo da uporedimo da li je jedan broj veći od drugog, odnosno da numerički tipovi pripadaju i ovoj klasi tipova.

```
> 1<=2
True
> 1>=2
False
```

Ali, kako stoji stvar sa drugim tipovima? Da li jedna lista može biti manja od druge? Ili da li je `False` manje od `True`? Odgovor je „da, u programskom jeziku Haskell“.

```
> True > False
True
> [2,2] > [1,2]
True
> [1,2] > [2,2]
False
> "Hello world" > "Hll wrld"
False
> ("Pera", "Peric", [5,5,3,4,5,5]) > ("Pera", "Peric",
[5,5,3,4,5])
True
> ("Pera", "Peric", [5,5,3,4,5]) > ("Pera", "Peric",
[5,5,3,4,5,5])
False
```

Klasa tipova `Ord` je takođe vrlo široka i obuhvata gotovo sve tipove u programskom jeziku Haskell.

Pitanja i zadaci

- Kako izgleda signatura tipa operatora `(>)`? Da li se i u njoj koriste varijable tipa?
- Samostalno istraži kako se operator `>` ponaša nad `Bool`, `String`, `List`, `Char` i `Tuple` tipovima. Šta znači da je jedan karakter veći od drugog? A šta znači da je jedan string veći od drugog? Da li postoje nešto zajedničko za poređenje stringova, listi i sekvenci? Zaključke diskutuj u grupi.

Sledeća klasa tipova u dijagramu je `Num`. Ona nalaže da nad vrednostima tipova koji joj pripadaju možemo da pozovemo operatore `(+)`, `(-)` i `(*)`, odnosno da možemo da ih sabiramo, oduzimamo i množimo. Ovoj klasi tipova pripadaju svi brojčani tipovi.

Klasa `Num`

```
> (3 :: Int) - (2 :: Int)
1
> (3 :: Float) - (2 :: Float)
1.0
```

Ali kako stoji stvar sa ostalim tipovima?

```
> "Hello world!" - "world!"

<interactive>:3:16: error:
    * No instance for (Num String) arising from a use
of `-'
    * In the expression: "Hello world!" - "world!"
      In an equation for `it': it = "Hello world!" -
"world!"
> True - False

<interactive>:4:6: error:
    * No instance for (Num Bool) arising from a use of
`-'
    * In the expression: True - False
      In an equation for `it': it = True - False
> [1,2,3] - [2,3]

<interactive>:5:1: error:
    * No instance for (Num [Integer]) arising from a
use of `it'
    * In the first argument of `print', namely `it'
      In a stmt of an interactive GHCi command: print
it
> ("Pera", "Peric") - ("Peric")

<interactive>:6:22: error:
    * Couldn't match type: [Char]
                     with: (String, String)
      Expected: (String, String)
        Actual: String
    *  In  the  second  argument  of  `(-)',  namely
`("Peric")'
      In the expression: ("Pera", "Peric") - ("Peric")
      In an equation for `it': it = ("Pera", "Peric")
- ("Peric")
```

Vidimo da string, logički tip, lista i sekvenca ne podržavaju operatore zahtevane ovom klasom tipova, što znači da joj ne pripadaju. U stvari, klasi tipova `Num` pripadaju numerički tipovi.

Pitanja i zadaci

- Primeti da klasa `Num` ne zahteva da vrednosti mogu i da se dele. Kakvi brojevi mogu da se sabiraju, oduzimaju i množe, ali ne mogu da se dele?

Klasa `Fractional`

Posebnu klasu tipova čine numerički tipovi za koje je definisano i deljenje. Ova klasa tipova zove se `Fractional` i ona je potklasa klase tipova `Num`. To znači da će svaki `Fractional` tip obavezno biti i `Num` tip. Ova klasa tipova uvodi operator `(/)`, odnosno operator deljenja. Uvođenje klase tipova `Fractional` je vrlo jasno – njeni tipovi služe za predstavljanje razlomljenih brojeva. Dakle tipovi koji predstavljaju cele brojeve, kao što je `Int`, neće pripadati ovoj klasi, a tipovi koji služe za predstavljanje razlomljenih brojeva, kao što je `Float`, hoće.

```
> (2 :: Int) / (4 :: Int)

<interactive>:7:12: error:
    * No instance for (Fractional Int) arising from a
use of `/'
    * In the expression: (2 :: Int) / (4 :: Int)
      In an equation for `it': it = (2 :: Int) / (4 ::
Int)
> (2 :: Float) / (4 :: Float)
0.5
```

Klasa `Enum`

Za razliku od razlomljenih brojeva, kod celih brojeva možemo da odredimo prethodnika i sledbenika. Zato celobrojni tipovi pripadaju klasi tipova `Enum`, koja uvodi funkcije `succ` i `pred`.

```
> succ (2 :: Int)
3
> pred (2 :: Int)
1
```

Klasa `Integral`

Na cele brojeve ne možemo da primenimo operator deljenja, ali možemo da primenimo celobrojno deljenje, i pri tome da dobijemo ostatak pri deljenju. Zato je uvedena klasa tipova `Integral`, koja zahteva da tipovi koji joj pripadaju imaju implementirane operatore `div` i `mod`. Ona je potklasa klase tipova `Enum`. Tip `Int` pripada ovoj klasi tipova.

```
> div 5 2
2
> mod 5 2
1
```

- Sam zaključi kojim klasama tipova pripada `Int`, a kojim pripada `Float`? Diskutuj u grupi. Zaključke proveri pomoću komande `:i` u GHCI.
- Koristivši funkciju `mod` napisati program koji za zadatu listu pronalazi sumu kvadrata svih parnih elemenata u listi.
- Napisati program koji nalazi sumu kvadrata svih elemenata liste koji se nalaze na parnim indeksima.

Pitanja i zadaci

Klasu tipova možemo da posmatramo kao skup tipova nad kojim je moguće izvršavati iste funkcije.

Novi tipovi i klase tipova

U poglavlju 4, kada smo pričali o tipovima, videli smo da skup tipova u programskom jeziku Haskell nije fiksan, odnosno da ga možemo proširiti i uvesti nove tipove. Za sada smo uveli jedan način kreiranja tipova: tipove smo kreirali pomoću konstruktora tipa i konstruktora vrednosti. Ovaj način kreiranja tipova podrazumevao je da se tipu dodeli ime i da se nabroje vrednosti koje mu pripadaju. Na ovaj način možemo reći da su godišnja doba proleće, leto, jesen i zima, da su svetla na semaforu crveno, žuto i zeleno, ili da su dani u nedelji ponedeljak, utorak, sreda, četvrtak, petak, subota i nedelja.

Ali kako se tipovi koje mi sami napravimo odnose prema klasama tipova? Kada kreiramo novi tip, kojim klasama tipova on da pripada?

```
> data Season = Spring | Summer | Autumn | Winter
> Spring == Summer

<interactive>:37:8: error:
    * No instance for (Eq Season) arising from a use
of `=='
    * In the expression: Spring == Summer
      In an equation for `it': it = Spring == Summer
```

Odgovor na naše pitanje je: ako ne kažemo ništa, tip koji kreiramo ne pripada nijednoj klasi. Greška koju smo dobili nam govori da naš tip ne pripada čak ni klasi tipova `Eq`, pa stoga nije moguće ni uporediti da li su dve vrednosti ovog tipa jednake. Ali možemo *reći* da novi tip pripada nekoj klasi tipova. To radimo tako što, nakon konstruktora tipa `Season`, kažemo da je `Season` instanca klase tipova `Eq` i navedemo implementaciju funkcija koje klasa `Eq` zahteva.

```
> :{
| instance Eq Season where
|     Spring == Spring = True
|     Summer == Summer = True
|     Autumn == Autumn = True
|     Winter == Winter = True
|     _ == _ = False
| :}
> Spring == Summer
False
> Spring == Spring
True
```

deriving

Za ovakve, osnovne klase tipova kao što je `Eq` ne moramo da eksplicitno implementiramo zahtevane funkcije, nego možemo da prepustimo samom okruženju da ih zaključi, navođenjem ključne reči `deriving`.

```
data Season = Spring | Summer | Autumn | Winter deriving
(Eq, Show)
```

Ovim smo rekli da tip `Season` pripada klasama tipova `Eq` (da je moguće porediti jednakost vrednosti ovog tipa) i `Show` (da je moguće vrednosti ovog tipa pretvoriti u stringove) i zahtevali smo da programsko okruženje samo zaključi implementaciju[9].

Na ovaj način možemo da opišemo puno fenomena, kao što su godišnja doba. Ali velik broj fenomena se ne može opisati na ovaj način.

Uvođenje novog tipa

Na primer, osoba ima ime i prezime – ako bismo pokušali da ovaj tip opišemo navođenjem svih mogućih vrednosti (kombinacijom svih mogućih imena i prezimena) dobili bismo ogroman spisak. Ovakve podatke smo opisivali sekvencama, pa bi osoba mogla da bude predstavljena sekvencom `(String, String)`, gde je prva vrednost ime, a druga prezime. Ovaj način je bio pogodan, ali jedan od problema je bio što su sve sekvence `(String, String)` istog tipa, bez obzira da li se radi o osobama (ime, prezime) ili knjigama (autor, naslov).

[9] Zaključivanje implementacije prevazilazi opseg ove knjige, ali znatiželjnog čitaoca ohrabrujem da samostalno istraži kako je implementirano.

```
> ("Pera", "Peric") == ("Na Drini cuprija", "Ivo
Andric")
False
```

Uporedili smo osobu i knjigu i dobili smo da nisu jednake. Ali one nisu samo nejednake; one su *neuporedive*, pošto suštinski pripadaju različitim tipovima. Zato nam programski jezik Haskell omogućuje da uvedemo novi tip koji je opisan na vrlo sličan način kao sekvenca.

```
> data Person = Person String String deriving (Eq,
Show)
> data Book = Book String String deriving (Eq, Show)
> p = Person "Pera" "Peric"
> b = Book "Ivo Andric" "Na Drini cuprija"
> p == b

<interactive>:68:6: error:
    * Couldn't match expected type `Person' with actual
type `Book'
    * In the second argument of `(==)', namely `b'
      In the expression: p == b
      In an equation for `it': it = p == b
```

Kada smo sada pokušali da uporedimo osobu i knjigu, dobili smo grešku koja kaže da su ove vrednosti neuporedive jer pripadaju različitim tipovima.

Naravno, treba nam mehanizam da pristupimo poljima ovako kreiranog tipa, odnosno da pristupimo imenu i prezimenu osobe.

```
> p = Person "Pera" "Peric"
> Person firstName lastName = p
> firstName
"Pera"
> lastName
"Peric"
```

Ovakav način pristupa poljima vrlo je sličan kao i pristup vrednostima u sekvenci.

```
> p = ("Pera","Peric")
> (firstName,lastName) = p
> firstName
"Pera"
> lastName
"Peric"
```

Kada smo podatke modelovali pomoću sekvenci, pokazalo se korisno da napišemo funkcije za pristup elementima, da ne bismo morali da pamtimo da li je ime prvo, a prezime drugo, ili je obrnuto.

```
> firstName (fn, ln) = fn
> lastName (fn, ln) = ln
> firstName p
"Pera"
> lastName p
"Peric"
```

Isti princip možemo primeniti i na tipove.

```
> data Person = Person String String deriving (Eq,
Show)
> firstName (Person fn ln) = fn
> lastName (Person fn ln) = ln
> p = Person "Pera" "Peric"
> firstName p
"Pera"
> lastName p
"Peric"
```

Rekord

Ovaj princip je toliko popularan da programski jezik Haskell uvodi skraćeni zapis za zadavanje funkcija za pristup poljima ovako definisanih tipova koji se zove *rekord*. Tip osobe smo mogli kreirati pomoću rekorda kao što je prikazano listingom dole.

```
> data Person = Person {firstName :: String, lastName
:: String} deriving (Eq, Show)
> p = Person "Pera" "Peric"
> firstName p
"Pera"
> lastName p
"Peric"
```

Kažemo da tip `Person` ima polje `firstName` koje je tipa `String` i `lastName` koje je tipa `String`.

Videli smo da, kada kreiramo tipove, možemo da ih uključimo u postojeće klase tipova. Ali, da li mi sami možemo da pravimo klase tipova u koje bismo uključili naše tipove? Odgovor na ovo pitanje je, naravno, da.

Uvođenje nove klase tipova

Hajde da napišemo program koji beleži informacije o osobama u sistemu. To mogu biti učenici, studenti i zaposleni. Učenik ima ime, prezime, školu i razred. Student ima ime, prezime, smer i godinu studija. Zaposleni ima ime, prezime i poziciju na poslu. Za predstavljanje učenika, studenata i zaposlenih koristićemo rekorde. Zatim ćemo ove tipove okupiti u jednu klasu tipova (`Display`) koja zahteva jednu funkciju (`display`) za prikaz osobe.I na kraju ćemo za svaki tip koji smo kreirali reći da pripada ovoj klasi tipova i implementiraćemo zahtevanu funkciju.

```
> :{
| data Pupil = Pupil
|   { pupilFirstName :: String
|   , pupilLastName  :: String
|   , pupilSchool    :: String
|   , pupilGrade     :: Int
|   }
| :}
> :{
| data Student = Student
|   { studentFirstName :: String
|   , studentLastName  :: String
|   , studentMajor     :: String
|   , studentYear      :: Int
|   }
| :}
> :{
| data Worker = Worker
|   { workerFirstName :: String
|   , workerLastName  :: String
|   , workerJobTitle  :: String
|   }
| :}
> :{
| class Display a where
|   display :: a -> String
| :}
> :{
| instance Display Pupil where
|  display p = pupilFirstName p ++ " " ++ pupilLastName
p ++
|               ", grade " ++ show (pupilGrade p) ++
|               ", school: " ++ pupilSchool p
| :}
> :{
| instance Display Student where
|     display s = studentFirstName s ++ " " ++
studentLastName s ++
|               ", year " ++ show (studentYear s) ++
|               ", major: " ++ studentMajor s
| :}
> :{
| instance Display Worker where
```

```
|      display w = workerFirstName w ++ " " ++
workerLastName w ++
|               ", job: " ++ workerJobTitle w
| :}
> p1 = Pupil "Pera" "Peric" "Gimnazija J.J Zmaj" 3
> display p1
"Pera Peric, grade 3, school: Gimnazija J.J Zmaj"
> s1 = Student "Marko" "Markovic" "Racunarstvo" 2
> display s1
"Marko Markovic, year 2, major: Racunarstvo"
> w1 = Worker "Jovan" "Jovanovic" "Senior software
developer"
> display w1
"Jovan Jovanovic, job: Senior software developer"
```

Sada možemo da iskoristimo ovo znanje da ponovo uradimo zadatak iz prethodnog poglavlja. Napravićemo školski dnevnik, ali uz korišćenje rekorda, a ne sekvenci za predstavljanje podataka. Dnevnik je lista učenika. Svaki učenik ima ime, prezime i listu predmeta. Svaki predmet ima naziv i listu ocena.

Lista rekorda

```
> :{
| data Subject = Subject
|   { subjectName   :: String
|   , subjectGrades :: [Int]
|   } deriving (Eq, Show)
| :}
> :{
| data Pupil = Pupil
|   { pupilFirstName :: String
|   , pupilLastName  :: String
|   , pupilSubjects  :: [Subject]
|   } deriving (Eq, Show)
| :}
> :{
| pupils =
|         [ Pupil "Jovana" "Petrovic"
|             [ Subject "Matematika" [5, 4, 5]
|             , Subject "Likovno" [5, 5]
|             ]
|         , Pupil "Marko" "Jankovic"
|             [ Subject "Matematika" [3, 4]
|             , Subject "Istorija" [4, 4, 5]
|             ]
|         ]
| :}
> pupils
[Pupil {pupilFirstName = "Jovana", pupilLastName =
"Petrovic", pupilSubjects = [Subject {subjectName =
"Matematika", subjectGrades = [5,4,5]},Subject
{subjectName = "Likovno", subjectGrades =
[5,5]}]},Pupil {pupilFirstName = "Marko",
pupilLastName = "Jankovic", pupilSubjects = [Subject
{subjectName = "Matematika", subjectGrades =
[3,4]},Subject {subjectName = "Istorija",
subjectGrades = [4,4,5]}]}]
```

Računanje vrednosti u listi rekorda

Omogućićemo da se za svaki predmet prikaže i prosečna ocena.

```
> :{
| average [] = 0
| average xs = fromIntegral (sum xs) / fromIntegral
(length xs)
| :}
> :{
| averageGrade subject = average (subjectGrades
subject)
| :}
> :{
| instance Show Subject where
|   show subject =
|     subjectName subject ++
|     " - Grades: " ++ show (subjectGrades subject) ++
|     ", Average: " ++ show (averageGrade subject)
|
| instance Show Pupil where
|   show pupil =
|       pupilFirstName pupil ++ " " ++ pupilLastName
pupil ++ ":\n" ++
|     unlines (map show (pupilSubjects pupil))
| :}
```

Implementacija `show` funkcije i priključivanje klasi

Implementirali smo funkciju `average` koja računa prosečnu vrednost za listu brojeva. Zatim smo implementirali funkciju `averageGrade` koja koristi funkciju `average` da izračuna prosečnu ocenu za predmet. Pošto hoćemo da prosečna ocena bude prikazana u ispisu, ne možemo koristi osnovnu `show` funkciju – ona samo prikaže polja rekorda. Zato moramo da implementiramo svoje `show` funkcije za tipove `Subject` i `Pupil`.

```
> :{
| addGradeToSubject name newGrade subject =
|   if subjectName subject == name
|    then Subject (subjectName subject) (subjectGrades
subject ++ [newGrade])
|     else subject
| :}
> :{
|  addGradeToPupil firstName lastName subjectName
newGrade pupil =
|       if pupilFirstName pupil == firstName &&
pupilLastName pupil == lastName
|     then Pupil
|          (pupilFirstName pupil)
|          (pupilLastName pupil)
|         (map (addGradeToSubject subjectName newGrade)
(pupilSubjects pupil))
|     else pupil
| :}
> :{
| insertGrade firstName lastName subjectName newGrade
pupils =
|   map (addGradeToPupil firstName lastName subjectName
newGrade) pupils
| :}
> insertGrade "Jovana" "Petrovic" "Matematika" 5 pupils
[Jovana Petrovic:
Matematika - Grades: [5,4,5,5], Average: 4.75
Likovno - Grades: [5,5], Average: 5.0
,Marko Jankovic:
Matematika - Grades: [3,4], Average: 3.5
Istorija - Grades: [4,4,5], Average: 4.333333333333333
]
```

Dodavanje vrednosti u složeni tip

Dodavanje nove ocene implementiramo na sličan način kao i u primeru iz prethodnog poglavlja. Napisali smo funkciju `addGradeToSubject` koja primi naziv predmeta, novu ocenu i rekord predmeta. Ako se prosleđeni naziv predmeta poklopio sa nazivom u predmetu, funkcija vrati novi predmet kome je u listu dodata prosleđena ocena. Ako se ne poklopi, vrati se originalni predmet.

Brisanje vrednosti iz složenog tipa

Zatim smo napisali funkciju `addGradeToPupil`. Ona primi ime i prezime učenika, naziv predmeta, novu ovcenu i rekord učenika. Ako se prosleđeni ime i preziem ne poklope sa imenom i prezimenom u rekordu učenika, vratiće se originalni rekord. Ako se poklope kreiraće se novi rekord za učenika u kom će se `addGradeToSubject` primeniti na predmete pomoću `map` funkcije.

Kako bismo napisali program koji briše prvu ocenu za zadati predmet i zadatog učenika?

```
> :{
| removeFirst [] = []
| removeFirst (_:xs) = xs
| :}
> removeFirst [1,2,3]
[2,3]
> :{
| removeFirstFromSubject name subject =
|   if subjectName subject == name
|     then Subject (subjectName subject) (removeFirst
(subjectGrades subject))
|     else subject
| :}
> :{
| removeFirstFromPupil firstName lastName subjectName
pupil =
|       if  pupilFirstName  pupil  ==  firstName  &&
pupilLastName pupil == lastName
|     then Pupil
|          (pupilFirstName pupil)
|          (pupilLastName pupil)
|            (map (removeFirstFromSubject subjectName)
(pupilSubjects pupil))
|     else pupil
| :}
> :{
|  removeFirstGrade  firstName  lastName  subjectName
pupils =
|      map  (removeFirstFromPupil  firstName  lastName
subjectName) pupils
| :}
> removeFirstGrade "Jovana" "Petrovic" "Matematika"
pupils
[Pupil {pupilFirstName = "Jovana", pupilLastName =
"Petrovic", pupilSubjects = [Subject {subjectName =
"Matematika",    subjectGrades    =    [4,5]},Subject
{subjectName    =    "Likovno",    subjectGrades    =
[5,5]}]},Pupil     {pupilFirstName     =     "Marko",
pupilLastName = "Jankovic", pupilSubjects = [Subject
{subjectName    =    "Matematika",    subjectGrades    =
[3,4]},Subject     {subjectName     =     "Istorija",
subjectGrades = [4,4,5]}]}]
```

Implementirali smo funkciju `removeFirst` koja primi listu i vrati tu listu sa uklonjenom prvom vrednošću. Zatim smo implementirali funkciju `removeFirstFromSubject` koja primi naziv premeta i rekord koji predstavlja predmet. Ako se prosleđeni naziv predmeta poklopi sa nazivom u rekordu, biće vraćen novi rekord koji će se od originalnog razlikovati po tome što je izbačena prva ocena iz liste. Ako se ne poklope, biće vraćen originalni predmet.

Zatim smo implementirali funkciju `removeFirstFromPupil` koja primi ime i prezime učenika, naziv predmeta i rekord koji predstavlja učenika. Ako se prosleđeni ime i prezime poklope sa imenom i prezimenom u rekordu, vratiće se novi rekord za učenika koji je identičan sa originalnim, osim što je pomoću `map` funkcije izbačena ocena za taj predmet. Ako se ne poklope, vratiće se originalni rekord.

Ovakav pristup možemo iskoristiti da uradimo i ostale zadatke iz prethodnog poglavlja.

Pitanja i zadaci

- Napiši program koji rukuje katalogom knjiga. Svaka knjiga ima naslov, autora i broj strana. Knjiga je predstavljena pomoću rekorda Book, a katalog knjiga je lista ovih rekorda.

```
> :{
| data Book = Book
|   { title  :: String
|   , author :: String
|   , pages  :: Int
|   } deriving (Show, Eq)
|
| books :: [Book]
| books =
|     [ Book "Na Drini cuprija"      "Ivo Andric"
314
|   , Book "Seobe"                "Milos Crnjanski"
472
|    , Book "Hazarski recnik"      "Milorad Pavic"
366
|     , Book "Prokleta avlija"       "Ivo Andric"
121
|    , Book "Stranac"               "Albert Camus"
123
|   ]
| :}
```

Zadaci:

1. Napiši funkciju `shortBooks` koja prima listu svih knjiga a vraća listu knjiga kraćih od 200 strana.
2. Napiši funkciju `byAuthor` koja za zadatog autora vraća listu knjiga koje je napisao.
3. Napiši funkciju `deleteBook` koja primi autora, naslov knjige i listu knjiga, a vrati listu knjiga sa izbačenom knjigom tog autora i pod tim naslovom.
4. Napiši funkciju `insertBook` koja prima ime autora, naslov knjige, broj strana i listu knjiga, a vraća novu listu knjiga u koju je dodata nova knjiga sa ovim podacima.

- Napiši program koji rukuje kartama u bioskopu. Bioskopska karta ima ime na koje je rezervisana, film, broj karte i cenu karte kao što je prikazano na listingu ispod.

```
> :{
| data Ticket = Ticket
|   { movieTitle      :: String
|   , numberOfTickets :: Int
|   , price           :: Int
|   } deriving (Show, Eq)
|
| data Customer = Customer
|   { customerName :: String
|   , customerTickets :: [Ticket]
|   } deriving (Show, Eq)
| :}
> :{
| reservations =
|   [ Customer "Ana"
|       [ Ticket "Dina" 2 400
|       ]
|   , Customer "Milan"
|       [ Ticket "Matrix" 1 450
|       , Ticket "Dina" 1 400
|       ]
|   , Customer "Jelena"
|       [ Ticket "Dina" 3 400
|       ]
|   , Customer "Nikola"
|       [ Ticket "Oppenheimer" 1 500
|       ]
|   ]
| :}
```

Zadaci:

1. Napiši funkciju `totalRevenue` koja računa ukupnu cenu prodatih karata.
2. Napiši funkciju `filterByMovie` koja primi listu rezervacija i naziv filma, a vrati listu rezervacija u kojoj su samo karte za taj film.
3. Napisati funkciju `mostTickets` koja primi listu rezervacija, a vrati ime osobe koja je rezervisala najviše karata.
4. Napisati funkciju `bestCustomer` koja primi listu rezervacija, a vrati ime osobe koja je potrošila najviše novca na karte.

5. Napisati funkciju `averagePrice` koja izračunava prosečnu cenu karte.
6. Napiši funkciju `deleteReservation` koja primi ime osobe, naslov filma i listu rezervacija, a vrati listu rezervacija iz koje je izbačena rezervacija karata za film sa primljenim naslovom i osobu sa primljenim imenom.
7. Napiši funkciju `updateReservation` koja primi listu rezervacija, ime osobe i naslov filma za koje postoji rezervacija, novi broj karti i novu cenu kartae, a vrati novu listu rezervacija u kojoj je za rezervaciju osobe sa zadatim imenom i zadatim naslovom filma postavljen novi broj karti i nova cena.

- Napiši program koji rukuje porudžbinama u restoranu. Svaka porudžbina ima broj stola i listu stavki, a svaka stavka ima naziv jela i cenu. Primer liste je dat listingom ispod.

```
> :{
| data OrderItem = OrderItem
|   { itemName  :: String
|   , itemPrice :: Int
|   } deriving (Show, Eq)
|
| data TableOrder = TableOrder
|   { tableNumber :: Int
|   , orderItems  :: [OrderItem]
|   } deriving (Show, Eq)
| :}
> :{
| orders =
|   [ TableOrder 1
|       [ OrderItem "Corba" 250
|       , OrderItem "Pljeskavica" 500
|       , OrderItem "Sok" 150
|       ]
|   , TableOrder 2
|       [ OrderItem "Pasta" 600
|       , OrderItem "Sok" 150
|       ]
|   , TableOrder 3
|       [ OrderItem "Salata" 300
|       , OrderItem "Voda" 100
|       ]
|   ]
| :}
```

Zadaci:

1. Napiši funkciju `totalForTable` koja primi redni broj stola i listu porudžbina, a vrati ukupan iznos računa za taj sto.
2. Napiši funkciju `mostExpensiveItems` koja primi listu porudžbina, a vrati listu sa po jednom najskupljom stavkom za svaki sto. Šta biva ako za neki sto postoji više stavki sa istom cenom koja je veća od drugih stavki?
3. Napiši funkciju `totalRevenu` koja primi listu stavki, a vrati sumu svih cena za sve stavke.
4. Napiši funkciju `filterByOrder` koja primi naziv jela, a vrati listu rednih brojeva stolova koji imaju porudžbinu tog jela.
5. Napiši funkciju `deleteOrder` koja primi redni broj stola i listu porudžbina, a vrati listu iz koje je izbačena ta porudžbina.
6. Napiši funkciju `deleteItem` koja primi redni broj stola, naziv stavke i listu porudžbina i vrati listu u kojoj je za taj sto izbačena ta stavka.
7. Napiši funkciju `addItem` koja primi redni broj stola, naziv stavke, cenu i listu porudžbina, a vrati listu porudžbina u kojoj je ta, nova stavka dodata za sto pod zadatim rednim brojem.

Ad hoc polimorfizam podrazumeva da se implementiraju različita tela iste funkcije u zavisnosti od tipova parametara. U ovom poglavlju smo videli da se, u programskom jeziku Haskell, ad hoc polimorfizam postiže pomoću klasa tipova koje nam omogućuju da navedemo koje funkcije tipovi koji im pripadaju moraju podržati. U tom slučaju, svaka implementacija funkcije bila je drugačija u zavisila je od tipova nad kojim se funkcija izvršava.

Parametarski polimorfizam

Ali videli smo da polimorfizam može da znači i da pravimo funkcije čija implementacija ne zavisi od tipova koji se prosleđuju, odnosno koje rade na isti način bez obzira na koje tipove se primene. Pogledajmo jedan primer ovakve *parametarski polimorfne* funkcije.

```
> identity x = x
> identity 3
3
> identity "Hello world!"
"Hello world!"
```

Napisali smo funkciju `identity` koja vrati vrednost koja joj je prosleđena. Ako prosledimo broj 3, ona vrati broj 3. Ako prosledimo string `"Hello world!"` ona vrati string `"Hello world!"`. Kako izgleda signatura ove funkcije?

```
> :i identity
identity :: p -> p
```

Vidimo da je funkcija definisana nad varijablom tipa (`p`). Znači funkcija preslikava bilo koji tip na samog sebe. Obratite pažnju da smo napisali polimorfnu funkciju, a da nigde nismo naveli kako implementacije za pojedinačne tipove izgledaju.

Parametarski polimorfne funkcije često se koriste za rad sa složenim tipovima. Na primer, mogli bismo napisati parametarski polimorfnu funkciju `swap` koja primi par i vrati par sa obrnutim vrednostima, u kom je prva vrednost postala druga, a druga postala prva.

```
> swap (x, y) = (y, x)
> swap (1, "Hello world!")
("Hello world!",1)
```

Pitanja i zadaci

- Kako izgleda signatura tipa funkcije `swap`?

Parametarski polimorfizam i strukture podataka

Pomoću parametarskog polimorfizma možemo kreirati korisne složene tipove. Jedan takav složeni tip je *stek*. Stek je uređenje podataka u kom se uvek pristupa podatku koji je poslednji dodat.

Stek

Zamisli jedan restoran u kome rade perač sudova i konobar. Perač pere tanjire, jedan po jedan, i svaki novi tanjir koji opere stavi na vrh gomile čistih tanjira. Gomila raste — prvi tanjir je dole, drugi je iznad njega, treći još iznad, i tako redom. Svaki sledeći tanjir ide na vrh.

Kada konobaru zatreba tanjir da usluži goste, on ne vadi onaj sa dna gomile. To bi bilo nezgodno jer bi morao da pomeri sve tanjire iznad njega. On uvek uzima tanjir koji je na vrhu, odnosno onaj koji je poslednji stavljen. Kad mu zatreba još jedan tanjir, opet uzima sledeći s vrha, i tako redom.

Po kom pravilu se koriste tanjiri u restoranu? Tanjir koji je poslednji stavljen, prvi se uzima. U programiranju, takva vrsta liste zove se stek.

Stek često srećemo i u drugim situacijama. Kada idemo na izlet, iz ranca ćemo prvo izvaditi ono što je na vrhu, odnosno ono što smo poslednje spakovali. Slično tome, kada se peku palačinke i slažu jedna na drugu na tanjir, ona koja je poslednja ispečena biće stavljena na vrh — i baš nju ćemo prvu uzeti.

Dodavanje u stek

`push`

Hajde da napišemo program koji predstavlja stek kao listu sa nekoliko funkcija. Mogli smo prepustiti da signature funkcija budu zaključene, ali napisaćemo ih sami da bismo bolje razumeli kako se u ovom kontekstu koristi parametarski polimorfizam. Prvo nam treba funkcija `push` koja dodaje element na vrh steka.

```
> :{
| push :: a -> [a] -> [a]
| push x xs = x:xs
| :}
> push 1 [2,3,4]
[1,2,3,4]
```

Ova funkcija primi element tipa `a` i listu elemenata tipa `a` (to je `[a]`) u koju se element dodaje, a vrati novu listu elemenata tipa `a` (ponovo `[a]`) u koju je prosleđeni element dodat.

Pitanja i zadaci

- Šta bi se desilo da funkciju `push` pozovemo sa jednim parametrom?

Preuzimanje iz steka

`pop`

Zatim nam treba funkcija `pop` koja preuzima element sa vrha steka. Pored toga što će ova funkcija da vrati element, ona treba i da ukloni preuzeti element sa vrha steka.

```
> :{
| pop :: [a] -> (a, [a])
| pop (x:xs) = (x, xs)
| :}
> pop [1,2,3,4]
(1,[2,3,4])
```

Obrati pažnju da smo preuzeti element i modifikovanu listu vratili kao par tipa `(a, [a])`.

Pitanja i zadaci

- Da li funkcija pop pokriva sve slučajeve? Šta bi se desilo da se pozove nad praznom listom?

Pristup vrhu steka

`peek`

Treba nam i funkcija koja vrati element sa vrha steka, bez da vraća modifikovanu listu. Tome služi funkcija `peek`.

```
> :{
| peek :: [a] -> a
| peek (x:xs) = x
| :}
> peek [1,2,3,4]
1
```

Provera da li je stek prazan

`isEmpty`

I treba nam funkcija koja proveri da li je stek prazan, `isEmpty`.

```
> :{
| isEmpty :: [a] -> Bool
| isEmpty [] = True
| isEmpty _ = False
| :}
> isEmpty []
True
> isEmpty [1,2,3,4,5]
False
```

Ova funkcija primi listu i vrati logičku vrednost koja kaže da li je prosleđena lista ima elemenata.

Parametarski polimorfizam steka

Ovako kreiran stek možemo da koristimo kao stek stringova, logičkih vrednosti, brojeva, vrednosti tipa koji sami kreiramo, odnosno vrednosti bilo kojih tipova. Zahvaljujući parametarskom polimorfizmu nemamo potrebe da za svaki konkretni tip implementiramo posebne funkcije.

Parametarski polimorfizam se često koristi za implementiranje struktura podataka[10] kao što je stek.

[10] Strukture podataka su veoma interesantna oblast, ali prevazilaze opseg ove knjige. Naravno, znatiželjni čitalac je upućen da samostalno istraži oblast strukture podataka i algoritama koji ih prate, kako bi stekao temeljnije znanje iz programiranja.

- Zamisli da si na autobuskoj stanici. Putnici dolaze jedan po jedan i staju u *red* da kupe kartu. Prvi putnik koji dođe staje na početak reda, zatim drugi staje iza njega, pa treći, i tako redom. Kada se šaltersko okno otvori, radnik ne poziva poslednjeg koji je došao (kao što bi bio slučaj da je u pitanju stek), već prvog — onog koji najduže čeka. Zatim dolazi sledeći, pa sledeći, i tako dok se red ne isprazni.

  Ovakvo ponašanje zovemo red — prvi koji uđe, prvi izlazi. U programiranju, *red* (eng., *queue*) je posebna vrsta liste u kojoj se novi elementi dodaju na kraj, a uklanjaju sa početka, baš kao u ovom primeru sa redom za autobusku kartu.

  Implementiraj red po uzoru na primer za stek. Samostalno smisli koje funkcije treba da budu podržane.

Monade

- Monada i dekompozicija na korake
- Lista kao monada
- Maybe monada
- IO monada
- Ispis u konzolu i čitanje sa tastature
- Rad sa fajlovima

Kako produžiti listu?

Pre nego što počnemo da se bavimo samim bočnim efektima, vratićemo se na jedan zadatak iz lista. U poglavlju 5, kada smo pričali o funkcijama `map` i `filter` nad listama bio je zadatak da se, koristeći ove funkcije, napiše funkcija `expand` koja primi listu brojeva, a vrati listu sačinjenu tako što je za svaki element `x` napravljen opseg brojeva od `1` do `x`. Za listu `[1,2,3]` trebali bismo da dobijemo listu `[1,1,2,1,2,3]`.

Naravno, zaključio si da je to nemoguće. To je i bio cilj tog zadatka: da te navede da zaključiš da nije moguće implementirati takvu funkciju samo koristeći samo `map` i `filter`.

Funkcija `map` je mogla samo da transformiše svaki element u listi, ali rezultat transformacije je bio lista iste dužine kao i početna. Funkcija `filter` je, doista, mogla da promeni dužinu liste, ali samo tako što bi je skratila, izbacila elemente koji ne odgovaraju navedenom kriterijum. Ali kako bi izgledala funkcija koja produži listu? Hajde da je implementiramo.

Funkcija koja menja i sadržaj i veličinu liste

Prvi parametar te funkcije bi svakako bio lista koja se obrađuje. Drugi bi mogao biti funkciju koja primi element liste, a vrati novu listu. Konačni rezultat bio bi lista sačinjena konkatenacijom listi koje se dobiju za pojedinačne elemente. U stvari, ona bi primila listu elemenata tipa `a`, i funkciju koja primi vrednost tipa `a` i vrati listu elemenata tipa `b`[11], a vrati listu elemenata tipa `b`. Kako bi izgledala signatura ove funkcije?

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

[11] Zašto ova funkcija ne vrati listu elemenata tipa `a`? Zato što, u opštem slučaju, može da se dogodi da elementi konačne liste ne budu istog tipa kao elementi početne liste. Na primer, listu brojeva mogli bismo transformisati u listu stringova.

Implementacija `myBind`

Ova funkcija bi bila rekurzivna i trebala bi da pokrije dva slučaja, jedan kada pošaljemo praznu listu i jedan kada pošaljemo listu sa elementima. Za praznu listu ova funkcija vraća praznu listu. Za listu koja ima elemente, ova funkcija uzme glavu liste, primeni funkciju na nju i na dobijeni rezultat konkatenira ono što se dobije kada se funkcija pozove za rep. Vrlo je slična kao implementacija `map` funkcije. Nazvaćemo je `myBind`.

```
> :{
| myBind :: [a] -> (a -> [b]) -> [b]
| myBind [] _ = []
| myBind (x:xs) f = f x ++ myBind xs f
|:}                                                    :}
> expand l = myBind l (\x -> [1..x])
> expand [1,2,3]
[1,1,2,1,2,3]
```

Evaluacija poziva `myBind`

Listingom ispod prikazana je eavluacija izraza `myBind [1,2,3] (\x -> [1..x])`.

```
myBind [1,2,3] (\x -> [1..x])
=> (f 1) ++ myBind [2,3] (\x -> [1..x])
=> [1] ++ myBind [2,3] (\x -> [1..x])
=> [1] ++ ((f 2) ++ myBind [3] (\x -> [1..x]))
=> [1] ++ ([1,2] ++ ((f 3) ++ myBind [] (\x -> [1..x])))
=> [1] ++ ([1,2] ++ ( [1,2,3] ++ []) )
=> [1] ++ ( [1,2] ++ [1,2,3] )
=> [1,1,2,1,2,3]
```

Bajnd operator >>=

Programski jezik Haskell ima svoju implementaciju ove funkcije. Ona se zove *bajnd* („veži“, eng. *bind*) i implementirana je operatorom >>= i radi isto kao i funkcija koju smo mi implementirali. Pogledajmo na listingu ispod kako bismo mogli implementirati funkciju `expand` koristivši bajnd operator.

```
> extend l = l >>= (\x -> [1..x])
> extend [1,2,3]
[1,1,2,1,2,3]
```

Vidimo da je jedina razlika u tome što se operator >>= koristi u infiksnoj notaciji, dok smo u primeru `myBind` koristili u prefiksnoj notaciji.

Bajnd i filtriranje

Bajnd operator može da produži listu. Ali da li može da filtrira listu? Da. Prosto će u situaciji u kojoj treba da izbaci element vratiti praznu listu. Pogledajmo kako bismo pomoću bajnd operatora mogli da napišemo program koji iz liste izbaci sve parne elemente.

```
> :{
| [1,2,3,4,5] >>= (\x ->
|          if x `mod` 2 == 0
|            then []
|            else [x]
|        )
| :}
[1,3,5]
```

Vidimo da, u slučaju da je broj paran, neimenovana funkcija vraća praznu listu. A u slučaju da element nije paran vraća listu koja ima samo taj element.

Bajnd i map

Da li bismo pomoću bajnd operatora mogli dobiti funkcionalnost koju omogućava `map`? Na primer, kako bi pomoću bajnd mogli da listu brojeva pretvorimo u listu njihovih kvadrata? Pogledajmo na listingu ispod kako bismo to uradili.

```
> :{
| [1,2,3,4,5] >>= (\x ->
|    [x * x]
|    )
| :}
[1,4,9,16,25]
```

Neimenovana funkcija koju prosledimo operatoru bajnd vrati listu koja ima samo jedan element, a to je kvadrat originalnog elementa.

Pitanja i zadaci

- Napiši funkciju `expandNeighbors` koja svaki broj `x` u listi proširi na `x-1, x, x+1`. Primer korišćenja ove funkcije je `expandNeighbors [3,5] == [2,3,4,4,5,6]`.
- Napiši funkciju `repeatSelf` koja svaki broj `x` iz liste ponovi `x` puta. Na primer, za listu `[1,3]` poziv ove funkcije bio bi `repeatSelf [1,3] == [1,3,3,3]`.
- Napiši funkciju `withPrevious` koja za svaki broj `x` iz liste napravi par `(x-1, x)`. Primer bi bio `withPrevious [3,5] == [(2,3),(4,5)]`.

- Napiši funkciju `stars` koja za listu brojeva vrati string u kom je svaki broj `x` predstavljen nizom od `x` zvezdica i odvojen od narednog niza karakterom `'|'`. Na primer `stars [2,1,3] == "**|*|***|"`.

Ulančavanje bajnd poziva

Sada bismo mogli da napišemo program koji listu brojeva prvo filtrira tako što iz nje izbaci sve parne brojeve, zatim dobijenu listu kvadrira, a nakon toga je proširi tako što za svaki kvadrat uzme sve brojeve od 1 do tog kvadrata. I sve ovo možemo implementirati koristeći samo bajnd — nije nam potrebno ništa drugo.

```
> :{
| [1,2,3,4,5] >>= (\x ->
|   if x `mod` 2 == 0
|     then []
|     else [x]
|   ) >>= (\y ->
|     [y*y]
|   ) >>= (\z ->
|     [1..z]
|   )
| :}
[1,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9,10,11,12,13,14
,15,16,17,18,19,20,21,22,23,24,25]
```

Koraci

Vidimo da funkcija bajnd uvodi novi način na koji zadajemo uputstvo za rešavanje problema. Kažemo *prvo izbaci* sve parne brojeve iz liste, *pa onda* kvadriraj sve preostale brojeve, *pa na kraju* za svaki kvadrat uzmi listu svih brojeva od 1 do tog kvadrata. Ovaj problem smo rešili u 3 *koraka*.

Bajnd i dekompozicija na korake

U stvari, bajnd operator nam omogućuje da program posmatramo kao *korake*, odnosno kao *seriju naredbi* koje se izvršavaju jedna za drugom nad podacima, pri čemu izlaz jedne naredbe može da utiče na sledeću. Time dobijamo novi alat za dekompoziciju problema: *dekompozicija na korake* koje je potrebno preduzeti pri rešavanju problema. Pri tome, nismo narušili početnu ideju ove knjige, a to je da je program izraz, a izvršavanje programa evaluacija izraza.

Sledeće zadatke uradi koristeći samo bajnd operator.

Pitanja i zadaci

- Napiši funkciju koja za zadatu listu brojeva vraća listu svih brojeva koji su između 2 i 6 (uključujući i 2 i 6).
- Napiši funkciju koja za zadatu listu brojeva vraća listu duplo većih vrednosti svih brojeva većih od 5.
- Napiši funkciju koja za zadatu listu brojeva vraća listu kvadrata samo onih brojeva koji su veći od 3.

Return

Pored operatora bajnd, kada radimo sa listama treba nam i funkcija koja element nekog drugog tipa „upakuje“ u listu. Za tip `Int` to znači da nam treba funkcija koja primi broj, a vrati listu koja sadrži taj broj kao jedini element. Ta funkcija se zove *return* („vrati“). Naravno, i nju bismo mogli samostalno implementirati, kao što je prikazano listingom ispod. U poglavlju 5 već smo se susreli sa `[]` što je skraćena notacija za *kons* operator koji kreira listu. Funkcija `return` je onda specijalni slučaj upotrebe operatora kons za kreiranje liste od jednog elementa.

```
> :{
| myReturn :: a -> [] a
| myReturn x = [x]
| :}
> myReturn 1
[1]
```

Kao i u slučaju map i bajnd funkcija i return funkcija dolazi sa programskim jezikom Haskell.

```
> :{
| [1, 2, 3] >>= (\x ->
|   if x `mod` 2 == 0
|     then return x
|     else []
|   )
| :}
[2]
```

Funktori i monade

Ove funkcije, map, bajnd i return, srešćemo i u mnogim drugim tipovima, a ne samo listama. Tipove koji podržavaju map funkciju nazivamo *funktorima*, a tipove koji podržavaju i bajnd i return funkcije nazivamo *monadama*[12]. Na primeru liste smo videli da se map funkcija može implementirati koristeći bajnd i return. Za ovaj zaključak lako je pokazati da važi i na bilo kom drugom tipu, a ne samo listi – prosto ćemo u implementaciji map funkcije koristiti samo bajnd i return. To znači da će sve monade istovremeno biti i funktori. Obrnuto nije slučaj.

Pitanja i zadaci

- Dokaži da su sve monade istovremeno i funktori

Then operator >>

Jedan čest slučaj korišćenja bajnd operatora je situacija u kojoj nas ne interesuje rezultat prethodnih operacija, nego samo želimo da se operacije izvršavaju jedna za drugom. Za to služi operator >> (then).

Pravila do notacije

Ovaj način programiranja, u kome se pomoću bajnd operatora program piše kao serija koraka koje treba preduzeti nad podacima je toliko učestao da za njega postoji skraćeni zapis koji se zove *do notacija*. Vidimo da je, prilikom korišćenja bajnd operatora glavni izvor nečitkosti programskog koda to što ima puno ulančanih anonimnih funkcija. U do notaciji operator <− zamenjuje bajnd operator kao što je prikazano tabelom ispod.

Bajnd operator	do notacija
`m >>= (\x -> ...)`	`x <- m`
`akcija >> ...`	`akcija`
`return vrednost`	`return vrednost`

Prevođenje bajnd u do notaciju

U primeru u kom smo koristili bajnd operator da iz liste izbacimo parne brojeve mogli smo koristiti do notaciju.

Do notacija i ulančani bajnd operatori

```
> :{
| do
|   x<-[1,2,3,4,5]
|   if x `mod` 2 == 0
|   then []
|   else return x
| :}
```

[12] Pojmove kao što su funktori i monada detaljno razrađuje oblast matematike koja se zove teorija kategorija. Poznavanje teorije značajno doprinosi znanju programiranja, ali nadilazi opseg ove knjige, pa znatiželjnog čitaoca upućujem da je dalje sam istraži.

Kako bi izgledao primer nekoliko koraka uzvezanih pomoću bajnd operatora, kada bismo ga preveli u do notaciju? Prevodili bismo jedan po jedan poziv bajnd operatora. Pogledajmo ponovo primer koda koji listu brojeva prvo filtrira tako što iz nje izbaci sve parne brojeve, zatim dobijenu listu kvadrira, a nakon toga je proširi tako što za svaki kvadrat uzme sve brojeve od 1 do tog kvadrata.

```
> :{
| [1,2,3,4,5] >>= (\x ->
|   if x `mod` 2 == 0
|     then []
|     else [x]
|   ) >>= (\y ->
|     [y*y]
|   ) >>= (\z ->
|     [1..z]
|   )
| :}
```

```
x <- [1,2,3,4,5]
if x`mod`2==0
  then []
  else [x]
```

```
y <- <prošli rezultat>
[y * y]
```

```
z <- <prošli rezultat>
[1 .. z]
```

Taj primer bi se mogao prevesti u čitljiviju do notaciju.

```
> :{
| do
|   x <- [1,2,3,4,5]
|   if x `mod` 2 == 0
|     then []
|     else do
|       y <- [x]
|       z <- [y * y]
|       [1..z]
| :}
```

Ugnježđeni do blok

Obrati pažnju na drugi do blok, koji će se izvršiti ako x nije deljivo sa 2. Uveli smo ga da bismo njime više iskaza (`y <- [x]`, `z <- [y * y]`, `[1..z]`) okupili u isti monadički blok.

Korišćenje do notacije vežbaćemo na istim primerima na kojima smo vežbali i bajnd.

Pitanja i zadaci

- Napiši funkciju koja za zadatu listu brojeva vraća listu svih brojeva koji su između 2 i 6 (uključujući 2 i 6).
- Napiši funkciju koja za zadatu listu brojeva vraća listu duplo većih vrednosti svih brojeva većih od 5.
- Napiši funkciju koja za zadatu listu brojeva vraća listu kvadrata samo onih brojeva koji su veći od 3.

Rekli smo da se map i bajnd mogu koristiti i na drugim tipovima podataka. Hajde da pogledamo još jedan primer tipa podataka koji je monada (i, naravno, funktor).

Izostale vrednosti

Često se susrećemo sa situacijom da vrednost u programu prosto ne postoji. Na primer, učenik bi mogao imati listu unetih ocena iz nekog predmeta, ali bi konačna ocena mogla da bude još uvek nezaključena. Kako bismo u programskom jeziku Haskell napisali da vrednost za zaključnu ocenu ne postoji?

Jedinični tip

U programskom jeziku Haskell postoji jedan specijalni, jedinični tip, odnosno tip junit („jedinica“,eng. *unit*) koji ima samo jednu vrednost, `()`, koji bismo mogli da iskoristimo da predstavimo ocenu koja ne postoji.

```
> finalGrade = ()
> finalGrade
()
```

Sve vrednosti u listi moraju biti istog tipa

Ali kako bismo mogli da predstavimo listu ocena, pri čemu neke još nisu zaključene, kombinujući brojeve i jedinični tip? Ne bismo mogli da kombinujemo brojeve i jedinični tip, pošto u programskom jeziku Haskell u listi sve vrednosti moraju biti istog tipa.

```
> finalGrades = [4,5,(),3]

<interactive>:11:16: error:
    * No instance for (Num ()) arising from the literal
`4'
    * In the expression: 4
      In the expression: [4, 5, (), 3]
      In an equation for `finalGrades': finalGrades =
[4, 5, (), ....]
```

Tip `Maybe`

Da bismo to mogli da uradimo, treba nam tip koji obuhvata i brojeve i nepostojeću vrednosti. Odnosno trebali bismo brojčani tip da proširimo junit vrednošću. U programskom jeziku Haskell postoji specijalni tip koji se zove `Maybe` i koji služi baš za tu namenu. Vrednost ovog tipa može biti ili `Nothing`, što znači da je prava vrednost ne postoji ili `Just x`, koja znači da vrednost postoji i da je baš `x`. Zato se i zove `Maybe`, jer vrednost možda postoji pa je `Just x`, a možda ne postoji pa je `Nothing`.

```
> finalGrade = Just 4
> finalGrade
Just 4
> finalGrade = Nothing
> finalGrade
Nothing
```

Koristeći ovaj tip mogli bismo napraviti listu zaključenih ocena, kao što smo planirali.

```
> finalGrades = [Just 4, Just 5, Nothing, Just 3]
> finalGrades
[Just 4,Just 5,Nothing,Just 3]
```

`Maybe` tip i map funkcija

Kao i tip liste, i na `Maybe` tip možemo da primenimo map funkciju. Kako bi map funkcija izgledala? Ukoliko je vrednosti `Nothing`, rezultat izvršavanja map funkcije će uvek biti `Nothing`. Ukoliko je `Just x` i ako smo prosledili funkciju `f`, rezultat će biti `Just f x`.

Pogledajmo kako bismo mogli da implementiramo map funkciju nad `Maybe` tipom.

```
myMap _ Nothing  = Nothing
myMap f (Just x) = Just (f x)
```

`fmap`

Vidimo da smo samo u Haskell kodu napisali ono što smo u prethodnom paragrafu rekli. Naravno, programski jezik Haskell dolazi sa ovom implementacijom, i ta funkcija se zove `fmap`.

```
> :{
| incrementGrade x =
|   if x == 5
|     then x
|     else x + 1
| :}
> incrementGrade 5
5
> incrementGrade 4
5
> fmap incrementGrade Nothing
Nothing
> fmap incrementGrade (Just 4)
Just 5
> fmap incrementGrade (Just 5)
Just 5
```

Napisali smo funkciju `incrementGrade` koja uveća za `1` svaku ocenu osim petice. Vidimo da, ako ovu funkciju pozovemo nad `5` dobijemo ponovo `5`, a ako je pozovemo nad `4` dobijemo `5`.

`Maybe` tip je funktor

Zatim smo ovu funkciju primenili na `Maybe` vrednosti pomoću `fmap`. Kada smo je primenili na `Nothing`, dobili smo ponovo `Nothing`. Ali kada smo je primenili na `Just 5` dobili smo `Just 5`, a kada smo je primenili na `Just 4`, dobili smo uvećanu ocenu `Just 5`.

Setimo se da činjenica da `Maybe` tip podržava map funkciju ovaj tip čini funktorom, baš kao što je i lista funktor.

`Maybe` tip i bajnd funkcija

Kao što možemo da primenimo map nad `Maybe` vrednostima, možemo da primenimo i bajnd, što ovaj tip čini monadom. Podsetimo se da bajnd operator primi monadu i funkciju koja vrednost transformiše u monadu, a vrati monadu. U slučaju `Maybe` tipa, bajnd operator će za `Nothing` vratiti `Nothing`, a na svaku drugu vrednost će primeniti funkciju koja je prosleđena.

```
myBind Nothing _ = Nothing
myBind (Just x) f = f x
```

I bajnd operator je podržan za `Maybe` tip u programskom jeziku Haskell.

`Maybe` tip je i monada

```
> :{
| incrementGrade x =
|   if x == 5
|     then Just x
|     else Just (x + 1)
| :}
> Just 4 >>= incrementGrade
Just 5
> Just 5 >>= incrementGrade
Just 5
> Nothing >>= incrementGrade
Nothing
```

Kao i u slučaju liste, bajnd operator smo mogli zameniti do notacijom.

`Maybe` tip i do notacija

```
> :{
| do
|   x <- Just 4
|   incrementGrade x
| :}
Just 5
```

Naša inicijalna ideja je bila da napišemo program koji primi listu zaključenih ocena, od kojih neke još nisu unete i sve ocene osim petice uveća za 1. Implementirali bismo ga tako što bismo primenili `incrementGrade` na listu `Maybe` vrednosti, kao što je prikazano na listingu ispod.

```
> :{
| incrementGrade x =
|   x >>= (\y ->
|     if y == 5
|       then Just y
|       else Just (y + 1))
| :}
> map incrementGrade finalGrades
[Just 5,Just 5,Nothing,Just 4]
```

U ovom pristupi, napisali smo funkciju koja uvećava ocenu (`incrementGrade`) i pomoću map funkcije smo je primenili na celu listu ocena. Na ovom primeru videli smo da monade mogu da se kombinuju, pa da možemo imati listu `Maybe` vrednosti. Takođe, mogli smo imati i `Mybe` liste ili bilo koju drugu kombinaciju monada.

Kombinovanje monada

Pitanja i zadaci

- Kakva je razlika između liste `Maybe` vrednosti i `Maybe` liste?
- Napiši funkciju `toText :: Maybe Int -> Maybe String` koja ocenu pretvara u string oblika `"Ocena: X"` ako ocena postoji. Ako ne postoji, pretvara se u `Nothing`.
- Napiši funkciju `safeDivide :: Int -> Maybe Int -> Maybe Int` koja broj deli sa datom vrednošću ako je deljenje moguće. Ako je vrednost kojom treba da se deli `0` ili ako je `Nothing`, rezultat je `Nothing`.
- Napiši funkciju `maybeSum :: Maybe Int -> Maybe Int -> Maybe Int` koja sabira vrednosti koji mogu i da ne postoje. Ako bilo koja od ove dve ne postoji, rezultat sabiranja je `Nothing`.

```
maybeSum (Just 2) (Just 3) == Just 5
maybeSum (Just 2) Nothing == Nothing
```

Program i svet

Razumevanje dekompozicije problema na korake, koju nam omogućuju monade, dovodi nas do centralne teme ovog poglavlja: kako program može da komunicira sa spoljašnjim svetom?

Programi menjaju svet

Ova knjiga je vođena idejom da je računarski program složeni izraz, a da je njegovo izvršavanje zapravo evaluacija tog izraza. Na primer, napisali smo `5 + 3`, i ovaj izraz je evaluiran u `8`. Ali da li je evaluacija izraza *sve* što programi treba da rade? Da li programi trebaju da rade *još nešto*?

Program koji upravlja robotom pokreće motor koji pomera veštačku ruku. Program koji upravlja pametnim stanom pali svetlo kada uzlazimo u zgradu. Slično tome, program može da štampa tekst, da pošalje mejl, izvrši uplatu u banci... Sve ove akcije rezultuju *promenom u svetu* u kom se program izvršava, a ne samo evaluacijom izraza.

U stvari, programi nam zato i trebaju, da bi pravili promene u svetu.

Svet utiče na programe

Isto tako, i svet u kom se programi izvršavaju utiče na program, pa će program klima uređaja aktivirati hlađenje kada je temperatura u prostoriji porasla iznad 26 stepeni Celzijusa, a program za plaćanje će ispisati poruku o grešci ukoliko ne postoje dovoljna sredstva na računu.

Monade i bočni efekti

`IO` monada

Prirodan način da predstavimo interakciju sa spoljašnjim svetom je da je organizujemo u korake. Na primer, ako bismo hteli da napišemo program koji preuzme korisnikovo ime i prezime sa tastature, a zatim pozdravlja korisnika, to bismo mogli da uradimo na sledeći način:

1) Preuzmi ime sa tastature
2) Preuzmi prezime sa tastature
3) Ispiši pozdrav u obliku `"Hello" ++ ime ++ prezime`

U ovom poglavlju smo videli da nam monade omogućuju upravo to, da program podelimo na korake. Zbog toga, bočni efekti u Haskell programima se ostvaruju primenom `IO` monade.

`IO` monada nam omogućava da program na kontrolisan način komunicira sa spoljnim svetom — tako što čita podataka sa tastature, ispisuje ih na ekran, radi sa fajlovima i slično. Obične funkcije u programskom jeziku Haskell nemaju bočne efekte. Zato pomoću `IO` monade ove efekte pakujemo u vrednosti koje se mogu obrađivati u koracima, a da pritom izvršavanje ostatka programa ostaje samo evaluacija izraza. Na ovaj način, Haskell jasno razdvaja deo programa koji je „čist“ i nema bočne efekte, od dela koji komunicira sa spoljašnjim svetom.

Kao i lista i `Maybe`, i `IO` je parametrizovani tip što znači da nosi u sebi neku vrednost konkretnog tipa. (Setimo se da je `Maybe Int` bio `Int` broj koji možda postoji, a možda ne postoji.)

Akcije

Slično tome, tip `IO String` predstavlja *akciju* koja, kada se izvrši, proizvede vrednost tipa `String`, ali istovremeno može imati i bočne efekte, poput čitanja sa tastature ili ispisivanja na ekran. Sam program zapravo ne sadrži direktno vrednost tipa `String`, već opisuje kako ta vrednost treba da se dobije kroz interakciju sa spoljnim svetom.

Razdvajanje čistog od nečistog

Ova struktura omogućava da u programskom jeziku Haskell „zarobimo“ bočne efekte unutar tipova i da ih jasno razlikujemo od čistih funkcija koje ne menjaju stanje sveta i na koje stanje sveta u kom se izvršavaju ne utiče. Tako, dok lista predstavlja kolekciju vrednosti, a `Maybe` opcioni (mogući) rezultat, `IO` akcije uključuju interakciju sa svetom i pri tome čuvaju bezbednost i čitljivost programa.

Čitanje sa tastature

Kako bi onda izgledala signatura funkcije koja čita string sa tastature?

```
getLine :: IO String
```

Ispis u konzolu

Ona ne primi ništa, a vrati `IO` monadu za `String`. To znači da će akcija `getLine`, kada se bude izvršila, proizvesti `String` vrednost. Obrati pažnju da `getLine` nije funkcija koja vrati `String` vrednost!

Kako bi izgledala signatura funkcije koja ispisuje vrednost u konzolu?

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

Ovo znači da `putStrLn` prima običan `String` kao ulaz i vraća akciju unutar `IO` monade (`IO ()`). Ta akcija, kada se izvrši, ispisuje dati tekst na ekran (standardni izlaz). Rezultat je jediničnog tipa, `()` znači da `putStrLn` ne vraća korisnu vrednost, zato što je njena svrha bočni efekat, odnosno ispisivanje, a ne vraćanje vrednosti.

Ove funkcije možemo iskoristiti za preuzimanje i ispis imena i prezimena, kako smo planirali.

```
> :{
| getLine >>= \firstName ->
| getLine >>= \lastName ->
| putStrLn ("Hello "++firstName++" "++lastName)
| :}
Pera
Peric
Hello Pera Peric
```

IO monada i do notacija

Kao što smo već videli kada smo obrađivali liste, učestala upotreba bajnd operatora rezultuje nečitkim programskim kodom, pa ćemo ovaj program napisati u do notaciji.

```
> :{
| do
|   firstName <- getLine
|   lastName <- getLine
|   putStrLn ("Hello "++firstName++" "++lastName)
| :}
Pera
Peric
Hello Pera Peric
```

Čitanje iz fajla

Čitanje sa tastature i ispis na ekran konceptualno se ne razlikuju mnogo od čitanja iz fajlova i upisa u fajl. Čitanje iz fajla je slično čitanju sa tastature, jedino je potrebno još i zadati putanju na kojoj se fajl nalazi. Za čitanje iz fajla koristi se funkcija `readFile` koja primi putanju do fajla, a vrati `IO` monadu za `String`. To znači da će ova funkcija, kada se izvrši akcija čitanja iz fajla, proizvesti string koji odgovara tekstualnom sadržaju fajla.

```
> :{
| do
|   fileContent <- readFile "c:/files/input.txt"
|   putStrLn fileContent
| :}
Hello World!
```

Upis u fajl

Upis u fajl sličan je ispisu na ekran, uz razliku da je potrebno zadati i putanju do fajla u koji se sadržaj ispisuje.

```
> :{
| do
|  writeFile "c:/files/output.txt" "Hello Haskell!"
| :}
```

Ova funkcija primi putanju do fajla i sadržaj koji u fajl treba da se snimi, a vrati `IO ()` monadu. Setimo se da jedinični tip `()` u monadi znači da sadržaj monade nije bitan, nego da nam samo treba akcija koju će monada izvršiti, što je u ovom slučaju ispis u fajl.

Pitanja i zadaci

- Za korisnički unos `x` upiši u fajl brojeve od `1` do `x`, svaki broj u poseban red.
- U fajlu su unete temperature, za svaki dan po jedna vrednost, i vrednosti su razdvojene uspravnom crtom. Napisati program koji računa prosečnu temperaturu i ispisuje je u konzolu.

Projekat

- Tekst projekta
- Moduli
- Organizacija projekta
- Objašnjenje implementacije projekta

Projekat: Školski dnevnik

U ovom poglavlju napravićemo sintezu svega o čemu smo pričali u knjizi i implementiraćemo jedan projekat - konzolnu aplikacija za vođenje školskog dnevnika. Aplikacija rukuje sledećim podacima:

Podaci

1. Učenik – redni broj, prezime, ime
2. Predmet – naziv
3. Ocene – ocene koje je učenik dobio u toku polugodišta i zaključna ocena koja može da ne postoji.

Aplikacija treba da omogući sledeće funkcionalnosti:

Funkcionalnosti

1. *Pregled spiska svih učenika* – sistem prikazuje spisak svih učenika, a za svakog se ispisuje redni broj, ime i prezime.
2. *Pregled spiska svih predmeta* – sistem prikazuje spisak naziva svih predmeta.
3. *Unos nove učenikove ocene na predmetu* – nastavnik unosi redni broj učenika, naziv predmeta i ocenu. Sistem dodaje unetu ocenu u spisak ocena tog učenika na navedenom predmetu.
4. *Pregled ocena jednog učenika na svim predmetima* – nastavnik unosi redni broj učenika. Sistem prikazuje spisak predmeta sa ocenama tog učenika.
5. *Pregled svih ocena jednog učenika na predmetu* – nastavnik unosi redni broj učenika i naziv predmeta. Sistem prikazuje spisak ocena tog učenika na navedenom predmetu.
6. *Pregled svih ocena svih učenika na predmetu* – nastavnik unosi naziv predmeta. Sistem prikazuje spisak učenika sa ocenama za taj predmet.
7. *Pregled ocena svih učenika na svim predmetima* – sistem prikazuje spisak svih učenika sa predmetima i ocenama.

Moduli

Zadaci koje smo do sada radili bili su relativno mali, najveći su imali nekoliko desetina linija koda, i pisali smo ih direktno u konzolu GHCI. Međutim, sa ovim projektom to nije slučaj. Kada ga završimo, imaće preko 300 linija koda. Bilo bi krajnje nepraktično da ga napišemo u konzoli. Ali bilo bi nepregledno i kada bi ceo ovaj program bio jedan jedini veliki fajl. Zato se programski kod u projektima raspoređuje u *module*.

Šta je modul

Modul je fajl koji sadrži povezane definicije — funkcije, tipove podataka i konstante — koje zajedno čine neku celinu. U ovom projektu, na primer, jedan modul će sadržati funkcije za čitanje podataka iz fajla i zapisivanje podataka u fajl. Modulima organizujemo kod tako da bude pregledniji i lakši za održavanje i ponovno korišćenje.

Deklaracija modula

Fajl se u programskom jeziku Haskell proglasi modulom tako što se na samom početku fajla napiše deklaracija modula. Ta deklaracija počinje ključnom rečju `module`, zatim ide ime modula (koje mora početi velikim slovom), pa lista onoga što modul izvozi (tj. stavlja na raspolaganje drugim modulima), i na kraju ključna reč `where`.

Deklaracija modula za rad sa fajlovima iz našeg projekta prikazana je listingom ispod.

```
module Gradebook.IO
  ( loadGrades
  , saveGrades
  ) where
```

Modul se zove `Gradebook.IO` i izvozi dve funkcije: `loadGrades` i `saveGrades`.

Uvoz modula

Kada nam treba funkcionalnost koju neki modul omogućava, prosto ćemo je uvesti pomoću ključne reči `import`. Ako želimo da uvezemo samo određene funkcije, možemo navesti njihova imena u zagradama. Listingom ispod prikazan je uvoz funkcija iz modula `Gradebook.IO`.

```
import Gradebook.IO (loadGrades, saveGrades)
```

Već vidimo da moduli imaju dve funkcije: prvo, oni omogućuju *dekompoziciju* programa na logički povezane celine; i drugo, oni obezbeđuju *apstrakciju* time što razdvajaju javno dostupne funkcije od onih koje nisu izvezene pa su ostale zatvorene u modulu.

Namena modula:

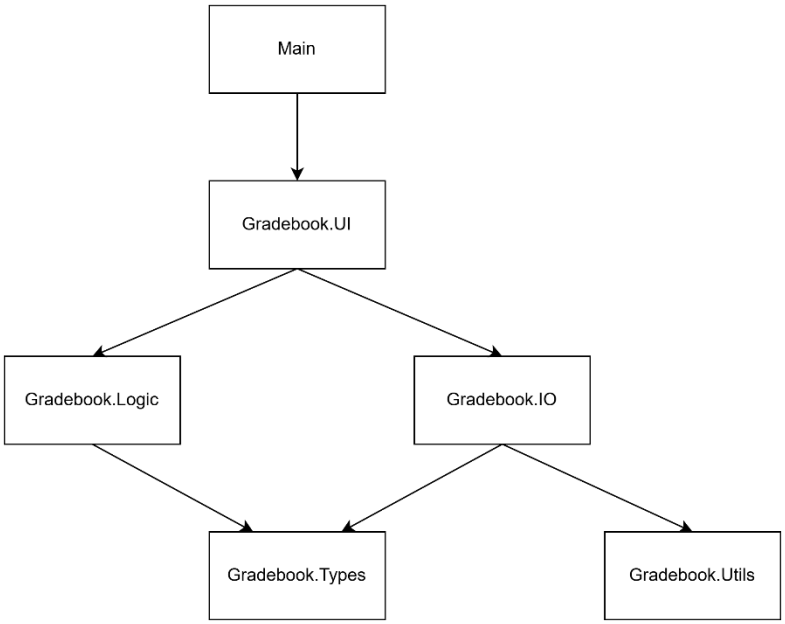
Dekompozicija i apstrakcija

Kako bismo mogli da organizujemo funkcionalnosti u ovom programu?

Organizacija projekta na modula

1. Sigurno nam trebaju korisnički tipovi koji predstavljaju učenike, predmete i ocene u zasebnom modulu. Taj modul ćemo nazvati `Gradebook.Types`.
2. Već smo rekli da ćemo u zaseban modul okupiti funkcionalnosti čitanja podataka iz fajlova i snimanja podatak u fajlove. Pošto se taj modul bavi ulazom i izlazom, nazvaćemo ga `Gradebook.IO`.
3. Trebaće nam funkcionalnosti koje implementiraju zahteve projekta. To bi, na primer bila funkcija koja primi redni broj učenika i listu svih ocena, a vrati ocene samo tog učenika. U programima ovakve funkcionalnosti čine *poslovnu logiku*, pa ćemo ovaj modul nazvati `Gradebook.Logic`.
4. Trebaće nam modul u kom su funkcionalnosti vezane za korisničke unose i prikaz. Pošto je taj modul zadužen za korisnički interfejs, nazvaćemo ga `Gradebook.UI`.
5. U zaseban modul ćemo staviti funkcionalnosti vezane za pokretanje programa. To će biti glavni modul u našem programu, od kog će početi izvršavanje program, pa ćemo ga nazvati `Main`.
6. U svakom programu postoje funkcionalnosti opšte namene, koje se koriste na različitim mestima u programu. Njih ćemo smestiti u modul `Gradebook.Utils`.

Slika ispod prikazuje kako su moduli u projektu povezani. Strelica označava uvoz.



Model podataka

Sada, kada smo organizovali naš projekat i osmisli kako ćemo rasporediti programski kod, možemo početi da ga razrađujemo. Prvo ćemo razraditi model podataka, odnosno tipove pomoću kojih ćemo opisati deo sveta kojim se program bavi. Ovi korisnički tipovi nalaze se u modulu `Gradebook.Types`.

Program treba da omogući čuvanje i prikaz učeničkih ocena na predmetima. To znači da ovaj program treba da opiše predmete, učenike i ocene. Već smo videli da se za takvu namenu u programskom jeziku Haskell mogu koristiti rekordi. Zato ćemo uvesti tri tipa rekorda: `Subject`, `Pupil` i `Grades`.

Rekord `Subject` je najjednostavniji – on predstavlja školski predmet i ima samo naziv.

```
data Subject = Subject
  { name :: String
  } deriving (Show, Eq)
```

Rekord `Pupil`, koji omogućava predstavljanje učenika, sadrži ime, prezime i redni broj učenika.

```
data Pupil = Pupil
  { pupilId    :: Int
  , lastName   :: String
  , firstName  :: String
  } deriving (Show, Eq)
```

Najsloženiji je rekord `Grades`, koji sadrži podatke o učeniku i predmetu, listu ocena i završnu ocenu.

```
data Grades = Grades
  { pupil          :: Pupil
  , subject        :: Subject
  , semesterGrades :: [Int]
  , finalGrade     :: Maybe Int
  } deriving (Show, Eq)
```

Čuvanje podataka

Pored toga što nam treba da podatke predstavimo u programu, treba nam i način da podatke sačuvamo u fajl i da ih učitamo iz fajla. Kako bismo mogli da podatke iz programa prevedemo u fajl? Jedan često korišćen format zapisa podatak u fajl svodi se na razdvajanje polja u podacima zarezom (eng. *comma-separadete values*, CSV). U tom formatu ocene jednog učenika na predmetu bile bi prevedene u string oblika.

CSV

```
15,Ristic,Andrej,Istorija,4;4;4,4
```

Klasa tipova `ToCSV`

Za čuvanje podataka u fajl nam treba funkcija koja primi rekord `Grades`, a vrati CSV string. Rekord `Grades` se sastoji od rekorda `Pupil`, rekorda `Subject`, liste ocena i zaključne ocene. Problem konverzije u CSV rešavamo pomoću ad hoc polimorfizma: uvodimo klasu tipova `ToCSV` koja ima jednu funkciju `toCSV` koja za svaki tip iz ove klase vrati CSV string reprezentaciju.

```
-- converters to CSV
class ToCSV a where
  toCSV :: a -> String

instance ToCSV Pupil where
  toCSV :: Pupil -> String
  toCSV p =
    show (pupilId p) ++ "," ++
    lastName p ++ "," ++
    firstName p

instance ToCSV Subject where
  toCSV :: Subject -> String
  toCSV = name

instance ToCSV Grades where
  toCSV :: Grades -> String
  toCSV g =
    toCSV (pupil g) ++ "," ++
    toCSV (subject g) ++ "," ++
     intercalate ";" (map show (semesterGrades g)) ++
"," ++
    maybe "-" show (finalGrade g)
```

Pošto problem konverzije u CSV rešavamo uvođenjem klase tipova, konverzija rekorda `Grades` se svodi na pozive funkcije `toCSV` za polje `pupil` i polje `subject` i konverziju dodatnih polja, `semesterGrades` i `finalGrades`.

Učitavanje podataka

Za učitavanje podataka iz fajla nam treba funkcija koja radi obrnuto od ovoga: ona primi CSV string, a vrati rekord `Grades`. Ona CSV string pretvori u listu segmenata tako što ga podeli po zarezu. Za to ćemo iskoristiti funkciju `splitOn` koja je implementirana u poglavlju o složenim tipovima. Od ovako dobijene liste kreiraće se rekord `Grades` pozivom funkcije `createGrades`.

```
makeGrades :: [String] -> Grades
makeGrades    [pupilIdStr,    lastName,    firstName,
subjectName, semesterGradesStr, finalGradeStr] =
  Grades
    { pupil = Pupil
        { pupilId = read pupilIdStr
        , lastName = lastName
        , firstName = firstName
        }
    , subject = Subject { name = subjectName }
      ,  semesterGrades  =  map  read  (splitOn  ';'
semesterGradesStr)
    , finalGrade = if finalGradeStr == "-" then Nothing
else Just (read finalGradeStr)
    }
```

Ova funkcija zadužena je za kreiranje rekorda `pupil` i `subject`, kao i za popunjavanje polja `semesterGrades` i `finalGrade`.

Kada su napisane funkcije za prevođenje rekorda u CSV string, i, obrnuto, CSV stringa u rekorde, čitanje i upis u fajl je jednostavno. Kada se čitaju podaci učitaćemo ceo fajl kao string, podeliti ga po novom redu i mapirati funkcijom `makeGrades`.

```
loadGrades :: IO [Grades]
loadGrades = do
    content <- readFile gradesFilePath
    _ <- evaluate (length content)
     return (map (makeGrades . splitOn ',') (lines
content))
```

Upis u fajl

Upis u fajl implementiramo tako što na listu ocena mapiramo funkciju `toCSV` i dobijene vrednosti spojimo u string koji se zatim snimi u fajl.

```
saveGrades :: [Grades] -> IO ()
saveGrades gradesList =
      writeFile gradesFilePath (unlines (map toCSV
gradesList))
```

Poslovna logika

Kada smo rešili učitavanje podataka iz fajla i snimanje podataka u fajl možemo pristupiti implementaciji poslovne logike u modulu `Gradebook.Logic`. Pregled spiska učenika i predmeta implementiramo tako što se lista rekorda `Grades` mapira na listu rekorda `Pupil`, odnosno `Subject`, a zatim se iz ovih lista uklone duplikati pomoću funkcije `nub`.

```
-- 1. list of all pupils
listPupils :: [Grades] -> [Pupil]
listPupils = nub . map pupil

-- 2. list of all subjects
listSubjects :: [Grades] -> [Subject]
listSubjects = nub . map subject
```

Za unos nove ocene koristimo isti pristup kao u poglavlju o složenim tipovima: pravimo funkciju koja primi redni broj učenika, naziv predmeta, novu ocenu i rekord `Grades`, koja, ukoliko se učenik i predmet poklope, vrati novi rekord `Grades` sa unetom ocenom. Ukoliko se ne poklope, vrati originalni rekord.

```
-- 3. input pupil's grade for a subject
updateGrade :: Int -> String -> Int -> Grades -> Grades
updateGrade pId subjectName newGrade grades =
  if pupilId (pupil grades) == pId && name (subject
grades) == subjectName
     then grades { semesterGrades = semesterGrades
grades ++ [newGrade] }
    else grades
```

Unos nove ocene za listu rekorda `Grades` se onda svodi na mapiranje funkcije `updateGrade`.

```
insertGrade :: Int->String->Int->[Grades]->[Grades]
insertGrade pupilId subjectName newGrade =
  map (updateGrade pupilId subjectName newGrade)
```

Preostale funkcije, koje prikazuju ocene jednog učenika na svim predmetima, ocene jednog učenika na svim predmetima, ocena svih učenika na jednom predmetu, kao i pregled svih ocena svode se na filtriranje liste rekorda `Grades`.

```
-- 4.   list of pupil's grades on all subjects
getStudentGrades :: Int -> [Grades] -> [Grades]
getStudentGrades pId = filter (\g -> pupilId (pupil g)
== pId)

-- 5. list of pupil's grades on one subject
getStudentSubjectGrades :: Int -> String -> [Grades] -
> [Grades]
getStudentSubjectGrades pId subjectName =
   filter (\g -> pupilId (pupil g) == pId && name
(subject g) == subjectName)

-- 6. list of all grades on one subject
getSubjectGrades :: String -> [Grades] -> [Grades]
getSubjectGrades subjectName =
  filter (\g -> name (subject g) == subjectName)
```

Konzolni korisnički interfejs

Kada imamo poslovnu logiku, treba da omogućimo korisniku da koristi aplikaciju. Korisnički interfejs je implementiran kao jednostavna konzolna aplikacija. Kada korisnik pokrene program, prikaže mu se meni kao na slici ispod.

```
Please choose an option:
1. All pupils
2. All subjects
3. Enter grade
4. Pupil's grades
5. Pupil's grades on a subject
6. All grades on a subject
7. All grades
8. Exit
```

Korisnik podatke unosi kroz konzolne dijaloge.

```
Enter subject name:
Matematika
```

Podaci iz aplikaciju prikazuju se tabelarno, u zavisnosti od akcije koju je korisnik odabrao.

```
No  | Last Name    | First Name | Subject         | Semester Grades | Final
-------------------------------------------------------------------------
1   | zivkovic     | Milan      | Matematika      | 3,4,3           | 3
2   | Djordjevic   | Anja       | Matematika      | 5,5,4           | 5
3   | Ilic         | Jelena     | Matematika      | 4,4,3           | 4
4   | Knezevic     | Filip      | Matematika      | 4,4,4           | 4
5   | Tanaskovic   | Stefan     | Matematika      | 3,4,4           | 4
6   | Lazarevic    | Teodora    | Matematika      | 4,4,3           | 4
7   | Stankovic    | Filip      | Matematika      | 3,4,3           | 3
8   | Petrovic     | Marko      | Matematika      | 3,4,5           | 4
9   | Milic        | Marija     | Matematika      | 5,5,5           | 5
10  | Obradovic    | Nikola     | Matematika      | 5,5,5           | 5
11  | Bogdanovic   | Nina       | Matematika      | 5,5,5           | 5
12  | Jovanovic    | Milica     | Matematika      | 5,4,5           | 5
13  | Nikolic      | Sara       | Matematika      | 4,4,4           | 4
14  | Radovic      | Luka       | Matematika      | 4,3,4           | 4
15  | Ristic       | Andrej     | Matematika      | 4,4,4           | 4
```

Korisnički meni je string konstanta.

```
-- user menu
showMenu :: String
showMenu =
  "\n\
  \Please choose an option:\n\
  \1. All pupils\n\
  \2. All subjects\n\
  \3. Enter grade\n\
  \4. Pupil's grades\n\
  \5. Pupil's grades on a subject\n\
  \6. All grades on a subject\n\
  \7. All grades\n\
  \8. Exit\n\
  \"
```

Korisnički meni

Formatiranje tabela

Da bismo omogućili tabelarni prikaz, treba nam formatiranje liste rekorda `Grades` u tabelu. Za svaki element ove liste treba da se kreira string koji odgovara redu u tabeli. Ovaj problem je vrlo sličan konverziji rekorda `Grades` u CSV string, pa ćemo ga rešiti na isti način: uvešćemo klasu tipova `ToTableData` koja će imati funkciju `toTableData`, i ovoj klasi tipova ćemo pridružiti rekorde `Grades`, `Pupil` i `Subject`.

```
class ToTableData a where
  toTableData :: a -> String

instance ToTableData Pupil where
  toTableData p =
    padRight 3 (show (pupilId p)) ++ " | " ++
    padRight 12 (lastName p) ++ " | " ++
    padRight 10 (firstName p)

instance ToTableData Subject where
  toTableData s = padRight 15 (name s)

instance ToTableData Grades where
  toTableData g =
    toTableData (pupil g) ++ " | " ++
    toTableData (subject g) ++ " | " ++
    padRight 15
      (if null (semesterGrades g)
         then "-"
          else init (concatMap (\x -> show x ++ ",")
(semesterGrades g))) ++
    " | " ++
    padRight 5 (maybe "-" show (finalGrade g))
```

Formatiranje tabela se onda svodi na ispis zaglavlja tabele i mapiranje `toTableData` na odgovarajuću listu.

```
formatPupils :: [Pupil] -> String
formatPupils ps =
  "No  | Last Name    | First Name\n" ++
  replicate 35 '-' ++ "\n" ++
  unlines (map toTableData ps)

formatSubjects :: [Subject] -> String
formatSubjects ss =
  "Subject\n" ++
  replicate 15 '-' ++ "\n" ++
  unlines (map toTableData ss)

formatGrades :: [Grades] -> String
formatGrades gs =
  "No  | Last Name    | First Name | Subject         |
Semester Grades | Final\n" ++
  replicate 70 '-' ++ "\n" ++
  unlines (map toTableData gs)
```

Korisničke dijaloge, gde postoji potreba za tim, odnosno kada oni preuzimaju više vrednosti, implementiramo kao zasebne funkcije koje korisnički unos vraćaju kao sekvence zapakovane u IO monadu.

Korisnički dijalog

```
-- menu options
enterGrade :: IO (Int, String, Int)
enterGrade = do
   putStrLn "Enter pupil ID:"
   pupilIdStr <- getLine
   putStrLn "Enter subject name:"
   subjectName <- getLine
   putStrLn "Enter grade:"
   gradeStr <- getLine
   return (read pupilIdStr, subjectName, read gradeStr)
```

Opcije menija sastoje se od korisničkih dijaloga i tabelarnih prikaza i implementiramo ih kao nezavisne funkcije koje okupimo u listu.

Opcije menija

```
option1 :: IO Bool
option1 = do
  grades <- loadGrades
  putStr (formatPupils (listPupils grades))
  return True

option2 :: IO Bool
option2 = do
  grades <- loadGrades
  putStr (formatSubjects (listSubjects grades))
  return True

option3 :: IO Bool
option3 = do
  grades <- loadGrades
  (pupilId, subjectName, newGrade) <- enterGrade
  saveGrades (insertGrade pupilId subjectName newGrade
grades)
  return True

option4 :: IO Bool
option4 = do
  grades <- loadGrades
  putStrLn "Enter pupil ID:"
  pupilIdStr <- getLine
    putStr   (formatGrades   (getStudentGrades   (read
pupilIdStr) grades))
  return True

option5 :: IO Bool
option5 = do
  grades <- loadGrades
  putStrLn "Enter pupil ID:"
  pupilIdStr <- getLine
  putStrLn "Enter subject name:"
  subjectName <- getLine
```

```
  putStr (formatGrades (getStudentSubjectGrades (read
pupilIdStr) subjectName grades))
  return True

option6 :: IO Bool
option6 = do
  grades <- loadGrades
  putStrLn "Enter subject name:"
  subjectName <- getLine
   putStr (formatGrades (getSubjectGrades subjectName
grades))
  return True

option7 :: IO Bool
option7 = do
  grades <- loadGrades
  putStr (formatGrades  grades)
  return True

exitProgram :: IO Bool
exitProgram = do
  putStrLn "Exiting..."
  return False

-- Menu list
options :: [IO Bool]
options = [option1, option2, option3, option4, option5,
option6, option7, exitProgram]
```

Glavni program

U glavnom programu preostaje nam samo da, u rekurzivnoj funkciji prikažemo meni, preuzmemo redni broj akcije sa tastature i tu akciju izvršimo. Ukoliko korisnik odabere akciju za izlaz iz programa, iz njene funkcije će biti vraćeno `False` i rekruzivna funkcija `mainLoop` će se završiti.

```
-- Main loop
mainLoop :: IO ()
mainLoop = do
  putStrLn showMenu
  input <- getLine
  options !! (read input - 1) >>= \continue ->
    if continue then mainLoop else return ()
```

Po uzoru na ovaj projekat implementiraj sledeće projekte.

1. Aplikacija će raditi kao jednostavan sistem za vođenje evidencije o knjigama. Aplikacija rukuje sledećim podacima:
   - Autor – ime i prezime autora.
   - Knjiga – naslov knjige, broj strana i autor koji je napisao knjigu.
   - Katalog – spisak svih knjiga u sistemu.

Aplikacija treba da omogući sledeće funkcionalnosti:

- Pregled svih knjiga kraćih od 200 strana – sistem prikazuje spisak svih knjiga koje imaju manje od 200 strana, zajedno sa naslovima i autorima.
- Pregled svih knjiga jednog autora – korisnik unosi ime i prezime autora. Sistem prikazuje sve knjige tog autora.
- Brisanje knjige iz kataloga – korisnik unosi ime i prezime autora i naslov knjige. Sistem iz kataloga uklanja knjigu koja odgovara tim podacima.
- Dodavanje nove knjige u katalog – korisnik unosi ime i prezime autora, naslov knjige i broj strana. Sistem dodaje novu knjigu u katalog.

2. Aplikacija za rezervaciju karata u bioskopu, koja treba da obezbedi rad sa sledećim podacima:

Bioskopska karta – svaka karta sadrži:

- naziv filma
- broj rezervisanih karata
- cenu pojedinačne karte

Korisnik – svaki korisnik ima:

- Ime
- prezime
- spisak karata koje je rezervisao

Aplikacija treba da omogući sledeće funkcionalnosti:

- Računanje ukupnog prihoda od prodatih karata (suma svih rezervacija).
- Prikaz rezervacija samo za jedan film (filtriranje podataka).
- Pronalazak korisnika koji je rezervisao najviše karata ukupno.
- Pronalazak korisnika koji je potrošio najviše novca na karte.
- Računanje prosečne cene pojedinačne karte u sistemu (ukupan prihod podeljen sa ukupnim brojem karata).
- Brisanje rezervacije za zadatu osobu i film.
- Ažuriranje rezervacije: promena broja karata i cene za određenog korisnika i film.

3. Aplikacija za evidenciju porudžbinama u restoranu, koja rukuje sledećim podacima:

Stavka porudžbine – za svako jelo koje je naručeno čuva se:

- naziv jela
- cena

Porudžbina za sto – za svaki sto čuva se:

- broj stola
- lista naručenih stavki

Aplikacija treba da omogući sledeće funkcionalnosti:

- Izračunavanje ukupnog iznosa računa za određeni sto.
- Pronalazak najskuplje stavke u porudžbini za svaki sto (jedna stavka po stolu).
- Računanje ukupnog prihoda restorana na osnovu svih porudžbina.
- Pronalazak stolova koji su naručili određeno jelo (npr. „Sok“).
- Brisanje cele porudžbine za određeni sto.
- Brisanje konkretne stavke iz porudžbine određenog stola.
- Dodavanje nove stavke u porudžbinu određenog stola.

Zaključak

- Šta dalje?
- Ka kurikulumu programiranja

Ova knjiga je nastajala dugo, preko 5 godina. Dragi čitaoče, nadam se da je tebi čitanje ove knjige pružilo makar deo zadovoljstva koje je meni pružilo pisanje. Sada, kada završavaš čitanje ove knjige, ukoliko si zainteresovan da dalje učiš da programiraš, nalaziš se pred pitanjem koji su sledeći koraci pred tobom?

Već sam napomenuo da ova knjiga nije iscrpan udžbenik programskog jezika Haskell. Ostalo je još puno toga da naučiš o ovom programskom jeziku i detaljnije izučavanje programskog jezika Haskell se svakako isplati ukoliko želiš da nastaviš da učiš da programiraš. S druge strane, pokrili smo dovoljno koncepata iz programiranja da možeš da počneš da učiš i druge programske jezike. Učenje objektno-orjentisanog programiranja i jezika kao što je Java će ti sigurno biti interesantno jer će ti dati drugačiji pogled na programiranje i moći ćeš da ga uporediš sa funkcionalnim programiranjem koje si sada naučio.

Objektno-orjentisano programiranje

Dalje, velik broj koncepata o kojima smo govorili u ovoj knjizi je teorijski zasnovan pa je stoga korisno da energiju posvetiš izučavanju matematičkih osnova programiranja. One pre svega obuhvataju lambda kalkulus i teoriju kategorija.

Matematičke osnove

Rasvetljavanje principa programiranja u ovoj knjizi otvorilo je i brojna pitanja vezana za samo računarstvo, od kojih svako upućuje na zasebnu oblast.

Oblasti računarstva

Videli smo da je računar kao mašina za rešavanja problema u stvari mašina za evaluaciju izraza. I videli smo da se problemi dekomponuju do nivoa koji je toliko jednostavan da ne zahteva nikakvo ljudsko razmišljanje, već čak i računar može da ih reši. Odnosno videli smo da postoji određeni broj primitivnih izraza i operacija koje računar sam ume da izvrši. Na pitanja koji su to primitivni izrazi i operatori i kako ih računar izvršava odgovor daje *arhitektura računara*, za koju mislim da je najbolji prvi naredni korak nakon savladavanja ove knjige.

Arhitektura računara

Kompajleri

Videli smo da su neka uputstva vrlo detaljna, a neka apstraktna. Upoznali smo se sa programskim jezikom Haskell, koji je vrlo visokog nivoa apstrakcije. Postavlja se pitanje kako računar ume da prevede program napisan u programskom jeziku visokog nivoa apstrakcije na skup instrukcija koje računar razume i ume da izvrši. Tim problemom se bavi teorija kompajlera.

Veštačka inteligencija

Jedno veliko pitanje na koje ova knjiga upućuje proizilazi iz definicije programiranja. Rekli smo da je programiranje pisanje uputstava za rešavanje problema. Ali i samo pisanje uputstava za rešavanje problema, samo programiranje, je takođe *problem*. Onda se postavlja pitanje da li možemo da napišemo programe koji *sami* nalaze uputstva za rešavanje problema? Ovim pitanjem se bavi oblast veštačke inteligencije.

Veb programiranje

Kada smo pričali o bočnim efektima, videli smo da programi nikada ne postoje sami za sebe, nego da uvek komuniciraju sa spoljašnjim svetom. Ova činjenica otvara velik broj pitanja. Na pitanje kako da naš program otvorimo i učinimo dostupnim što većem broju korisnika odgovor daje veb programiranje.

Arhitektura softvera

U poslednjem poglavlju, kada smo radili projekat, videli smo da su realni programi vrlo složeni. Tada se postavlja pitanje kako da kontrolišemo složenost programa, odnosno kako da pišemo programe sa čijom složenošću ćemo moći da se izborimo i kada budu rasli. Na to pitanje odgovor daje arhitektura softvera.

Baze podataka

Kada smo radili projekat videli smo da program treba da omogući rad sa realnim podacima. Najčešće su to veoma velike količine podataka. Na pitanja kako se rukuje velikom količinom realnih podataka odgovor daju baze podataka.

Bezbednost softvera

Ti podaci kojima se rukuje su često osetljivi i poverljivi. Na pitanje kako da rad programa učinimo sigurnim daje odgovor softverska bezbednost.

Ukoliko je ova knjiga uspela da te zainteresuje za dalja pitanja, ukoliko te je pokrenula da preduzmeš dalje korake na putu učenja programiranja, ostvarila je svoj cilj.