

PROGRAMSKE PARADIGME

Stefan Mišković

Matematički fakultet, Beograd

Sadržaj

1	Uvod u programske paradigme	3
1.1	Odnos programskih jezika i paradigmi	3
1.2	Proceduralna paradigma	3
1.3	Objektno-orijentisana paradigma	4
1.4	Funkcionalna paradigma	4
1.5	Logička paradigma	5
1.6	Ostale programske paradigme	5
2	Uvod u Python	7
2.1	Osnovne osobine Pythona	7
2.2	Rad u interaktivnom okruženju	7
2.3	Rad sa datotekama	9
2.4	Promenljive i osnovni tipovi	10
2.5	Operatori i izrazi	12
2.6	Ulaz i izlaz	14
2.7	Komentari i osnovna pravila pisanja	16
3	Kontrola toka i funkcije u Pythonu	19
3.1	Uslovno izvršavanje	19
3.2	Petlja while	20
3.3	Petlja for i funkcija range	22
3.4	Definisanje i pozivanje funkcija	24
3.5	Parametri i povratne vrednosti	26
3.6	Rekurzivne funkcije	29
4	Kolekcije u Pythonu	32
4.1	Liste	32
4.2	Pristup elementima i izmena listi	33
4.3	Prolazak kroz liste i osnovne operacije	36
4.4	Torke i skupovi	40
4.5	Rečnici	42
5	Objektno-orijentisano programiranje u Pythonu	47
5.1	Klase i objekti	47
5.2	Atributi, metode i konstruktor	48
5.3	Nasledivanje	51
5.4	Polimorfizam	53
6	Uvod u Haskell	56
6.1	Osnovne osobine Haskell	56
6.2	Funkcionalni način izračunavanja	57
6.3	Rad u interaktivnom okruženju	58
6.4	Rad sa datotekama	61

7	Tipovi i funkcije u Haskellu	64
7.1	Osnovni tipovi i zaključivanje tipova	64
7.2	Tipovi i definisanje funkcija	66
7.3	Funkcije sa više argumenata	68
7.4	Uslovni izrazi, ograðene i lokalne definicije	71
7.5	Podudaranje oblika	74
7.6	Polimorfni tipovi i tipske klase	76
8	Liste i rekurzija u Haskellu	80
8.1	Liste i osnovne operacije	80
8.2	Rasponi i izlistavanje	83
8.3	Podudaranje oblika nad listama	85
8.4	Rekurzivne funkcije	88
8.5	Rekurzivne funkcije nad listama	90
9	Funkcije višeg reda u Haskellu	93
9.1	Funkcije kao argumenti i lambda izrazi	93
9.2	Funkcije map, filter i zipWith	95
9.3	Funkcije foldr i foldl	98
9.4	Kompozicija funkcija	100
10	Definisanje novih tipova u Haskellu	103
10.1	Tipski sinonimi	103
10.2	Novi tipovi i konstruktori	104
10.3	Parametrizovani tipovi i tip Maybe	106
10.4	Tipovi sa imenovanim poljima	108
11	Poređenje programskih paradigmi	110
11.1	Razliĉiti pristupi rešavanju problema	110
11.2	Proceduralni, objektno-orijentisani i funkcionalni pristup	111
11.3	Prednosti i ograniĉenja razliĉitih pristupa	112
11.4	Izbor paradigme u savremenim programskim jezicima	113

1 Uvod u programske paradigme

1.1 Odnos programskih jezika i paradigmi

Programski jezik predstavlja skup pravila i zapisa pomoću kojih se opisuju podaci i postupci koje računar treba da izvrši. Postoji veliki broj programskih jezika, među kojima su C, Pascal, Python, Java, Haskell i Prolog.

Programi se mogu razlikovati ne samo po jeziku u kome su napisani, već i po načinu na koji je organizovano rešavanje problema. Zadatak se, na primer, može rešavati nizanjem koraka, organizovanjem podataka i postupaka u objekte, primenom funkcija ili zadavanjem činjenica i pravila. Ovakvi opšti načini organizovanja programa nazivaju se programske paradigme.

Razmatraćemo četiri osnovne programske paradigme: proceduralnu, objektno-orijentisanu, funkcionalnu i logičku.

Jedna paradigma može biti podržana u više programskih jezika. Proceduralni pristup se, na primer, koristi u jezicima C, Pascal i Python. Sa druge strane, isti programski jezik može podržavati više paradigmi. Python omogućava i proceduralno i objektno-orijentisano programiranje. Haskell je jedan od najpoznatijih predstavnika funkcionalnog programiranja, dok je Prolog karakterističan predstavnik logičkog programiranja.

Različite paradigme pokazuju da isti problem ne mora uvek da se rešava na isti način. U narednim podsekcijama upoznaćemo osnovnu ideju svake od navedenih paradigmi i prikazati je na jednostavnom primeru.

1.2 Proceduralna paradigma

Proceduralna paradigma zasniva se na tome da se problem rešava zadavanjem niza koraka koje računar treba da izvrši. Program opisuje postupak kojim se, polazeći od početnih podataka, dolazi do željenog rezultata.

U proceduralnim programima često se koriste promenljive čije se vrednosti mogu menjati tokom izvršavanja. Pomoću uslova bira se koje naredbe treba izvršiti, dok se pomoću petlji određene naredbe mogu ponavljati više puta. Delovi programa koji obavljaju određeni zadatak mogu se izdvojiti u funkcije.

Među poznatim programskim jezicima koji podržavaju proceduralno programiranje nalaze se C, Pascal i Python. Jednostavan proceduralni program u Pythonu može izgledati ovako:

```
suma = 0

for broj in range(1, 6):
    suma = suma + broj

print(suma)
```

Ovaj program računa zbir brojeva od 1 do 5. Na početku se promenljivoj `suma` dodeljuje vrednost 0. Ona će tokom izvršavanja sadržati zbir brojeva koji su do tog trenutka obrađeni.

Naredba `for` određuje ponavljanje dela programa. U ovom primeru promenljiva `broj` redom dobija vrednosti 1, 2, 3, 4 i 5. Pri svakom prolasku kroz petlju trenutna vrednost promenljive `broj` dodaje se promenljivoj `suma`.

Vrednost promenljive `suma` zato se tokom izvršavanja redom menja od 0 preko 1, 3, 6 i 10 do konačne vrednosti 15.

U ovom primeru jasno se vidi osnovna ideja proceduralnog programiranja. Programer je

zadao korake koji se izvršavaju određenim redosledom, a vrednost promenljive `suma` postepeno se menja dok se ne dobije rezultat. Detalje korišćene Python sintakse upoznaćemo kasnije.

1.3 Objektno-orijentisana paradigma

Objektno-orijentisana paradigma zasniva se na organizovanju programa pomoću objekata. Objekat objedinjuje podatke koji ga opisuju i postupke koji se nad tim podacima mogu izvršavati.

Objekti iste vrste najčešće se prave na osnovu klase. Klasa predstavlja opis zajedničkih osobina i ponašanja takvih objekata. Podaci koji pripadaju objektu nazivaju se atributi, dok se funkcije povezane sa objektom nazivaju metode.

Objektno-orijentisano programiranje podržavaju brojni programski jezici, među kojima su Java i Python. Jednostavan primer u Pythonu može izgledati ovako:

```
class Pravougaonik:
    def __init__(self, sirina, visina):
        self.sirina = sirina
        self.visina = visina

    def površina(self):
        return self.sirina * self.visina

p = Pravougaonik(4, 3)
print(p.površina())
```

U ovom primeru klasa `Pravougaonik` opisuje pravougaonike. Svaki pravougaonik ima širinu i visinu, koje se čuvaju u atributima `sirina` i `visina`.

Funkcija `__init__` koristi se pri pravljenju novog objekta. U ovom primeru ona određuje početne vrednosti njegove širine i visine. Izrazom `Pravougaonik(4, 3)` pravi se objekat čija je širina 4, a visina 3. Taj objekat se zatim čuva pod imenom `p`.

Metoda `površina` računa površinu pravougaonika koristeći njegove attribute. Ova metoda se zatim poziva za objekat `p`, pa se dobija vrednost 12. Naredba `print` prikazuje dobijeni rezultat.

U ovom primeru podaci o pravougaoniku i postupak za računanje njegove površine objedinjeni su u istoj programskoj celini. Upravo takvo organizovanje programa pomoću objekata predstavlja osnovnu ideju objektno-orijentisanog programiranja.

1.4 Funkcionalna paradigma

Funkcionalna paradigma zasniva se na tome da se izračunavanja opisuju pomoću funkcija i izraza. Umesto zadavanja niza koraka kojima se postepeno menja stanje programa, rezultat se dobija izračunavanjem odgovarajućeg izraza.

Najpoznatiji predstavnik funkcionalnog programiranja jeste Haskell. Kao jednostavan primer, ponovo možemo razmotriti zbir brojeva od 1 do 5. U Haskellu se on može zapisati ovako:

```
sum [1..5]
```

Zapis `[1..5]` predstavlja listu brojeva 1, 2, 3, 4 i 5. Funkcija `sum` računa zbir elemenata te liste, pa je vrednost celog izraza 15.

U proceduralnom primeru isti rezultat dobili smo pomoću promenljive čija se vrednost postepeno menjala unutar petlje. U ovom primeru nema takve promenljive ni petlje. Umesto

toga, neposredno je zadat izraz čijim se izračunavanjem dobija rezultat.

Ovakvo oslanjanje na funkcije i izraze predstavlja osnovnu ideju funkcionalnog programiranja. Programski jezik Haskell i način definisanja funkcija u njemu detaljnije ćemo upoznati kasnije.

1.5 Logička paradigma

Logička paradigma zasniva se na opisivanju činjenica i pravila koja važe u posmatranom problemu. Na osnovu njih mogu se postavljati upiti i određivati odgovori koji proizilaze iz zadatih činjenica i pravila.

Najpoznatiji predstavnik logičkog programiranja jeste Prolog. Jednostavan program u Prologu može izgledati ovako:

```
roditelj(ana, marko).
roditelj(marko, petar).

baba_ili_deda(X, Z) :-
    roditelj(X, Y),
    roditelj(Y, Z).
```

Prve dve linije predstavljaju činjenice. One govore da je Ana roditelj Marka i da je Marko roditelj Petra.

Nakon činjenica zadato je pravilo. Ono kaže da je osoba *X* baba ili deda osobe *Z* ako postoji osoba *Y* takva da je *X* njen roditelj, a *Y* roditelj osobe *Z*. Velika slova *X*, *Y* i *Z* predstavljaju promenljive koje Prolog tokom zaključivanja povezuje sa odgovarajućim vrednostima.

Programu se zatim može postaviti upit:

```
?- baba_ili_deda(ana, petar).
```

Na osnovu zadatih činjenica i pravila Prolog dobija odgovor **true**. Zaista, Ana je roditelj Marka, a Marko je roditelj Petra, pa zadato pravilo važi za Anu i Petra.

U ovom primeru nije zadat niz koraka kojima treba proveriti odnos između Ane i Petra. Umesto toga, opisane su činjenice i pravilo koje ih povezuje, a Prolog na osnovu njih određuje odgovor. To predstavlja osnovnu ideju logičkog programiranja.

1.6 Ostale programske paradigme

Proceduralna, objektno-orijentisana, funkcionalna i logička paradigma predstavljaju osnovne pristupe koje smo do sada razmatrali. Pored njih postoje i drugi načini organizovanja programa. Neki od njih mogu se posmatrati kao posebne paradigme, dok su drugi blisko povezani sa osnovnim paradigmama ili predstavljaju njihove kombinacije.

Jedan takav pristup je skript programiranje. Skript program se najčešće sastoji od niza naredbi koje se izvršavaju u određenom okruženju. Skript jezici se često koriste za automatizaciju jednostavnih poslova i pri razvoju veb aplikacija. Među poznatim skript jezicima nalaze se JavaScript i TypeScript.

Jednostavan primer u JavaScriptu može izgledati ovako:

```
let stranica = 5
let površina = stranica * stranica
console.log(površina)
```

U ovom primeru najpre se zadaje dužina stranice kvadrata, zatim se računa njegova površina, a na kraju se prikazuje rezultat 25. Naredbe se izvršavaju redom u okruženju u kome je skript pokrenut.

Poseban pristup koristi se i kod jezika namenjenih radu sa bazama podataka. Kod njih se često ne opisuje postupak kojim treba pronaći podatke, već se zadaje koje podatke želimo da dobijemo. Najpoznatiji primer takvog jezika je SQL.

Na primer, sledeći upit izdvaja imena svih studenata čije je prezime Petrovic:

```
SELECT ime  
FROM studenti  
WHERE prezime = 'Petrovic';
```

U ovom primeru nije navedeno kako baza treba da pronađe odgovarajuće studente. Upit samo opisuje koji rezultat želimo, dok sistem baze podataka određuje način na koji će ga dobiti.

U nekim programima potrebno je da više poslova može da napreduje tokom istog vremen-skog perioda. Takav pristup naziva se konkurentno programiranje. Na primer, program može preuzimati veliku datoteku sa interneta, dok korisnik za to vreme nastavlja da radi u drugim njegovim delovima. Time se izbegava da jedan duži posao potpuno zaustavi ostatak programa.

Kada se govori o programskim jezicima, korisno je razlikovati ih od drugih jezika i formata koji se često koriste u računarstvu. HTML služi za opis strukture i sadržaja veb stranice, dok CSS određuje njen izgled. XML omogućava strukturisan zapis podataka, a JSON se često koristi za njihov zapis i razmenu između programa. Ovi jezici i formati ne predstavljaju programske jezike, pa zato ne pripadaju nijednoj programskoj paradigmi.

2 Uvod u Python

2.1 Osnovne osobine Pythona

Python je programski jezik opšte namene koji se odlikuje jednostavnim i čitljivim zapisom. Koristi se za različite vrste programa, od kratkih programa za automatizaciju svakodnevnih poslova do veb aplikacija i obrade podataka.

Python je nastao početkom devedesetih godina XX veka. Od početka je razvijan sa naglaskom na jednostavan i čitljiv zapis, tako da se osnovna ideja programa često može jasno uočiti već iz samog koda.

Jednostavan Python program može izgledati ovako:

```
cena = 120
kolicina = 3
ukupno = cena * kolicina

print(ukupno)
```

Program računa ukupnu cenu tri proizvoda čija je pojedinačna cena 120. U prvoj liniji promenljivoj `cena` dodeljuje se vrednost 120, dok se promenljivoj `kolicina` dodeljuje vrednost 3. U narednoj liniji njihove vrednosti se množe, a dobijeni rezultat dodeljuje se promenljivoj `ukupno`.

Poslednja linija koristi funkciju `print`, koja služi za prikazivanje vrednosti. Pošto je proizvod brojeva 120 i 3 jednak 360, izvršavanjem programa prikazuje se broj 360.

Već na ovom primeru mogu se uočiti neke osnovne osobine Python programa. Naredbe se zapisuju jednostavno i pregledno, promenljivama se mogu pridruživati vrednosti, a nad tim vrednostima mogu se vršiti različita izračunavanja. Promenljive, tipove podataka, operatore i funkcije detaljnije ćemo upoznati u narednim podsekcijama.

Python podržava različite načine organizovanja programa. U jednostavnim primerima najpre ćemo koristiti proceduralni pristup, u kome se naredbe izvršavaju određenim redosledom. Kasnije ćemo upoznati i objektno-orijentisano programiranje pomoću klasa i objekata.

Python kôd može se neposredno isprobavati u interaktivnom okruženju ili sačuvati kao program koji se kasnije pokreće. U narednim podsekcijama upoznaćemo oba načina rada.

2.2 Rad u interaktivnom okruženju

Za izvršavanje Python programa potrebno je instalirati Python. Nakon instalacije, može se pokrenuti iz terminala, odnosno komandne linije. U zavisnosti od operativnog sistema i načina instalacije, najčešće se koristi komanda `python` ili `python3`.

Pokretanjem Pythona na ovaj način dobija se interaktivno okruženje. U njemu se izrazi i naredbe mogu unositi neposredno, a rezultati se odmah mogu proveriti. Oznaka `>>>` pokazuje da Python čeka novi unos.

Na primer, mogu se neposredno izračunavati aritmetički izrazi:

```
>>> 2 + 3
5

>>> 10 - 4
6
```

```
>>> 3 * 5
15

>>> 10 / 4
2.5
```

Python izračunava vrednost unetog izraza i prikazuje rezultat. Kao i u matematici, redosled računanja zavisi od prioriteta operatora, a zagrade se mogu koristiti kada želimo da taj redosled promenimo:

```
>>> 2 + 3 * 4
14

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

U prvom izrazu množenje se izvršava pre sabiranja. U drugom primeru zagrade određuju da se najpre saberu brojevi 2 i 3, pa se dobijena vrednost pomnoži brojem 4.

U interaktivnom okruženju mogu se koristiti i tekstualne vrednosti. One se u Pythonu mogu zapisati između jednostrukih ili dvostrukih navodnika:

```
>>> "Zdravo"
'Zdravo'

>>> 'Python'
'Python'
```

Oba zapisa predstavljaju tekstualne vrednosti. Važno je samo da se na početku i na kraju koristi ista vrsta navodnika.

Kada želimo da neku vrednost izričito pokažemo, možemo koristiti funkciju `print`:

```
>>> print("Zdravo")
Zdravo

>>> print(2 + 3)
5
```

U interaktivnom okruženju vrednost samog izraza prikazuje se automatski. Funkcija `print`, sa druge strane, predstavlja naredbu da se određena vrednost prikaže. Ova razlika će biti naročito važna pri radu sa programima zapisanim u datotekama.

U interaktivnom okruženju mogu se redom unositi i naredbe kojima se određuju i koriste vrednosti promenljivih:

```
>>> cena = 120
>>> kolicina = 3
>>> ukupno = cena * kolicina
>>> ukupno
360
```

Prvim dvema promenljivama dodeljuju se vrednosti 120 i 3. Njihov proizvod zatim se dodeljuje promenljivoj `ukupno`. Kada se na kraju unese samo njeno ime, Python prikazuje vrednost

360.

Interaktivno okruženje je pogodno za brzo isprobavanje izraza i kratkih delova koda. Kada program postane duži ili želimo da ga sačuvamo i ponovo koristimo, pogodnije je zapisati ga u datoteci.

2.3 Rad sa datotekama

Kada želimo da sačuvamo program i kasnije ga ponovo izvršavamo, Python kôd zapisuje se u tekstualnoj datoteci. Python datoteke najčešće imaju ekstenziju `.py`. Na primer, datoteka `racun.py` može sadržati sledeći program:

```
cena = 120
kolicina = 3
ukupno = cena * kolicina

print("Ukupna cena je", ukupno)
```

Program najpre promenljivama `cena` i `kolicina` dodeljuje odgovarajuće vrednosti. Njihov proizvod zatim se izračunava i dodeljuje promenljivoj `ukupno`. Poslednja naredba prikazuje tekst zajedno sa izračunatom vrednošću, pa se dobija ispis `Ukupna cena je 360`.

Python datoteke mogu se pisati u različitim programima namenjenim uređivanju programskog koda. Jedan od često korišćenih alata je Visual Studio Code. Uz odgovarajuću Python ekstenziju on omogućava preglednije pisanje i jednostavno pokretanje Python programa.

Sačuvana datoteka može se pokrenuti i iz terminala. Ako se nalazi u trenutnom direktorijumu i zove se `racun.py`, u zavisnosti od sistema može se koristiti komanda

```
python racun.py
```

ili

```
python3 racun.py
```

Python tada čita sadržaj datoteke i izvršava naredbe redom kojim su zapisane. U prethodnom primeru najpre se određuju vrednosti promenljivih, zatim se računa ukupna cena, a na kraju se rezultat prikazuje.

Važna razlika u odnosu na interaktivno okruženje jeste to što se vrednost običnog izraza u datoteci ne prikazuje automatski. Na primer, program koji sadrži samo

```
2 + 3
```

neće prikazati nikakav rezultat. Ako želimo da se izračunata vrednost prikaže, potrebno je napisati

```
print(2 + 3)
```

nakon čega se pri izvršavanju programa prikazuje vrednost 5.

Pri radu sa programom njegov sadržaj se može menjati, sačuvati i ponovo pokrenuti. Na taj način moguće je postepeno dodavati nove naredbe i proveravati kako izmene utiču na rezultat.

Interaktivno okruženje korišćićemo prvenstveno za kratke izraze i jednostavne provere, dok ćemo većinu narednih primera zapisivati u datotekama.

2.4 Promenljive i osnovni tipovi

Podaci koji se koriste u programu mogu se čuvati pomoću promenljivih. Promenljiva ima ime kojim kasnije možemo pristupiti njenoj vrednosti. Vrednost se promenljivoj dodeljuje pomoću znaka `=`.

Na primer:

```
godine = 20
```

Ovom naredbom promenljivoj `godine` dodeljuje se vrednost 20. Nakon toga ime promenljive možemo koristiti u drugim izrazima:

```
godine = 20
sledece_godine = godine + 1

print(sledece_godine)
```

Promenljiva `sledece_godine` dobija vrednost 21, jer se na trenutnu vrednost promenljive `godine` dodaje 1. Na ovaj način jednom izračunata vrednost može se sačuvati i kasnije koristiti na više mesta u programu.

Vrednost promenljive može se tokom izvršavanja promeniti. Na primer:

```
broj = 5
print(broj)

broj = 8
print(broj)
```

Prvom dodelom promenljiva `broj` dobija vrednost 5, pa se najpre prikazuje taj broj. Na narednom dodelom njena vrednost postaje 8, pa drugi poziv funkcije `print` prikazuje 8. Ovakvo menjanje vrednosti promenljivih često se koristi u proceduralnim programima.

Svaka vrednost u Pythonu pripada određenom tipu. Tip određuje kakvu vrstu podatka vrednost predstavlja i koje se operacije nad njom mogu izvršavati. Među osnovnim tipovima koje ćemo najčešće koristiti nalaze se `int`, `float`, `str` i `bool`.

Tip `int` koristi se za cele brojeve. Na primer:

```
broj_studenata = 25
temperatura = -3
```

Obe vrednosti predstavljaju cele brojeve, pa su promenljive `broj_studenata` i `temperatura` povezane sa vrednostima tipa `int`.

Tip `float` koristi se za brojeve koji imaju decimalni deo. U Pythonu se pri njihovom zapisu koristi decimalna tačka:

```
visina = 1.82
prosek = 8.75
```

Vrednosti 1.82 i 8.75 pripadaju tipu `float`. Nad celim i decimalnim brojevima mogu se vršiti različita aritmetička izračunavanja, koja ćemo detaljnije razmotriti u narednoj podsekciji.

Tip `str` koristi se za tekstualne vrednosti. Tekst se, kao što smo već videli, može zapisati između jednostrukih ili dvostrukih navodnika:

```
ime = "Ana"
grad = 'Beograd'
```

Promenljive `ime` i `grad` u ovom primeru imaju tekstualne vrednosti. Više znakova koji zajedno čine tekst u Pythonu se naziva string, pa otuda i naziv tipa `str`.

Tip `bool` predstavlja logičke vrednosti. Postoje samo dve takve vrednosti, `True` i `False`:

```
polozio = True
pada_kisa = False
```

Logičke vrednosti koriste se kada neki uslov može biti ispunjen ili neispunjen. Njihova uloga će postati naročito važna kada budemo razmatrali poređenja i uslovno izvršavanje naredbi.

Tip neke vrednosti može se proveriti pomoću funkcije `type`. U interaktivnom okruženju, na primer, možemo dobiti:

```
>>> type(25)
<class 'int'>

>>> type(3.14)
<class 'float'>

>>> type("Python")
<class 'str'>

>>> type(True)
<class 'bool'>
```

Na ovaj način možemo proveriti kom tipu pripada bilo koja vrednost. Funkcija `type` može se primeniti i na promenljivu:

```
>>> godine = 20
>>> type(godine)
<class 'int'>
```

U Pythonu nije potrebno unapred navesti tip promenljive. Kada napišemo

```
godine = 20
```

Python na osnovu dodeljene vrednosti prepoznaje da je u pitanju ceo broj. Slično tome, iz naredbe

```
ime = "Ana"
```

vidi se da promenljiva ima tekstualnu vrednost.

Promenljivoj se kasnije može dodeliti i vrednost drugog tipa. Na primer:

```
podatak = 10
podatak = "deset"
```

Posle prve naredbe promenljiva `podatak` povezana je sa celim brojem, dok je posle druge povezana sa tekstom. Python ovakav zapis dozvoljava, mada je u preglednim programima po-

željno da se promenljive koriste dosledno i da njihova imena jasno ukazuju na podatke koje predstavljaju.

Odgovarajući izbor imena čini program lakšim za razumevanje. Imena poput `cena`, `ukupno` ili `broj_studenata` zato su obično pogodnija od neodređenih imena kao što su `x` ili `a`, osim kada su kratka imena prirodna za konkretan problem. Preciznija pravila za zapisivanje imena razmotrićemo na kraju ove sekcije.

2.5 Operatori i izrazi

Izraz predstavlja zapis čijim se izračunavanjem dobija neka vrednost. Najjednostavniji izrazi mogu biti pojedinačne vrednosti ili promenljive, dok se složeniji izrazi grade njihovim povezivanjem pomoću operatora.

Na primer:

```
2 + 3
cena * kolicina
```

Prvi izraz ima vrednost 5. Vrednost drugog izraza zavisi od trenutnih vrednosti promenljivih `cena` i `kolicina`.

Za osnovna aritmetička izračunavanja koriste se operatori `+`, `-`, `*` i `/`, koji predstavljaju sabiranje, oduzimanje, množenje i deljenje:

```
>>> 7 + 3
10

>>> 7 - 3
4

>>> 7 * 3
21

>>> 7 / 2
3.5
```

Operator `/` predstavlja obično deljenje i za brojeve tipa `int` i `float` daje rezultat tipa `float`. Pored njega, Python podržava operator `//`, koji daje rezultat deljenja zaokružen naniže, kao i operator `%`, koji daje ostatak pri deljenju:

```
>>> 7 // 2
3

>>> 7 % 2
1
```

Operator za stepenovanje zapisuje se kao `**`. Na primer:

```
>>> 2 ** 5
32

>>> 10 ** 2
100
```

Aritmetički operatori mogu se kombinovati u složenije izraze. Kao i u matematici, množenje i deljenje imaju veći prioritet od sabiranja i oduzimanja:

```
>>> 2 + 3 * 4
14
```

Ako želimo drugačiji redosled računanja, koristimo zagrade:

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

Pored aritmetičkih operatora, često je potrebno porediti dve vrednosti. Operator `==` proverava da li su vrednosti jednake, dok operator `!=` proverava da li su različite. Za poređenje veličine koriste se operatori `<`, `<=`, `>` i `>=`:

```
>>> 5 == 5
True

>>> 5 != 3
True

>>> 4 < 7
True

>>> 10 <= 6
False
```

Rezultat poređenja je logička vrednost `True` ili `False`. Takvi izrazi će biti posebno važni kada budemo određivali pod kojim uslovima određene naredbe treba da se izvrše.

Važno je razlikovati operator dodele `=` od operatora poređenja `==`. Naredba

```
broj = 5
```

dodeljuje promenljivoj `broj` vrednost 5, dok izraz

```
broj == 5
```

proverava da li promenljiva trenutno ima tu vrednost i kao rezultat daje `True` ili `False`.

Logički izrazi mogu se povezivati operatorima `and`, `or` i `not`. Operator `and` daje vrednost `True` kada su oba uslova ispunjena, dok je kod operatora `or` dovoljno da bude ispunjen bar jedan. Operator `not` menja logičku vrednost u suprotnu:

```
>>> True and False
False

>>> True or False
True

>>> not True
False
```

Ovi operatori se najčešće koriste zajedno sa poređenjima. Na primer, možemo proveriti da

li se broj nalazi između 1 i 10:

```
>>> broj = 7
>>> broj >= 1 and broj <= 10
True
```

Oba poređenja u ovom primeru imaju vrednost `True`, pa je i ceo izraz tačan. Slično tome, možemo proveriti da li je broj manji od 0 ili veći od 100:

```
>>> broj = 120
>>> broj < 0 or broj > 100
True
```

Kada se u istom izrazu koristi više različitih operatora, njihov prioritet određuje redosled izračunavanja. Nije potrebno pamtit sva pravila prioriteta. Kada izraz može biti nejasan, zagrade omogućavaju da se željeni redosled jasno naznači:

```
>>> (5 > 2) and (10 < 20)
True
```

Pri promeni vrednosti promenljive često se pojavljuju izrazi poput

```
broj = broj + 1
```

kojima se trenutna vrednost uvećava za 1. Python omogućava i kraći zapis iste naredbe:

```
broj += 1
```

Na sličan način mogu se koristiti i operatori `-=`, `*=` i drugi. Na primer:

```
cena = 100
cena += 20
cena *= 2

print(cena)
```

Nakon prve naredbe vrednost promenljive `cena` je 100. Drugom naredbom ona se uvećava za 20, a zatim se dobijena vrednost množi brojem 2. Na kraju promenljiva zato ima vrednost 240.

Operatori omogućavaju da od postojećih vrednosti gradimo nove izraze, vršimo izračunavanja i proveravamo različite uslove. Poređenja i logički izrazi biće naročito važni pri razmatranju kontrole toka programa.

2.6 Ulaz i izlaz

Programi često treba da prime određene podatke od korisnika, obrade ih i zatim prikažu dobijeni rezultat. U Pythonu se za jednostavno prikazivanje podataka koristi funkcija `print`, dok se za unos podataka koristi funkcija `input`.

Funkciju `print` već smo koristili u prethodnim primerima. Ona može prikazati broj, tekst, vrednost promenljive ili rezultat nekog izraza:

```
print(25)
```

```
print("Python")

broj = 7
print(broj)
print(broj * 2)
```

Izvršavanjem ovog programa dobija se:

```
25
Python
7
14
```

Funkciji `print` može se proslediti i više vrednosti. One se tada pri prikazivanju razdvajaju razmakom:

```
ime = "Ana"
godine = 20

print("Ime:", ime)
print("Godine:", godine)
```

Dobija se ispis:

```
Ime: Ana
Godine: 20
```

Na ovaj način tekst i vrednosti promenljivih mogu se jednostavno prikazati zajedno.

Kada želimo da korisnik sam unese podatak, koristimo funkciju `input`. Između zagrada može se navesti tekst koji objašnjava šta korisnik treba da unese:

```
ime = input("Unesi ime: ")
print("Zdravo,", ime)
```

Pri izvršavanju programa najpre se prikazuje poruka `Unesi ime:`, nakon čega program čeka da korisnik unese tekst. Uneta vrednost čuva se u promenljivoj `ime` i zatim se koristi u narednoj liniji.

Na primer, ako korisnik unese `Ana`, rezultat će izgledati ovako:

```
Unesi ime: Ana
Zdravo, Ana
```

Važno je znati da funkcija `input` unetu vrednost uvek vraća kao tekst. To važi i kada korisnik unese cifre. Na primer:

```
godine = input("Unesi broj godina: ")
```

Ako korisnik unese 20, promenljiva `godine` sadrži tekstualnu vrednost "20", a ne ceo broj 20. To se može proveriti funkcijom `type`:

```
>>> godine = input("Unesi broj godina: ")
Unesi broj godina: 20
>>> type(godine)
```

```
<class 'str'>
```

Zbog toga se sa takvom vrednošću ne mogu odmah vršiti uobičajena aritmetička izračunavanja. Ako želimo da uneseni podatak koristimo kao ceo broj, možemo ga pretvoriti pomoću funkcije `int`:

```
godine = int(input("Unesi broj godina: "))
sledece_godine = godine + 1

print("Sledece godine imaces", sledece_godine)
```

Ako korisnik unese 20, funkcija `input` najpre dobija tekst "20", a funkcija `int` ga pretvara u ceo broj 20. Zato se na tu vrednost može dodati 1, pa program prikazuje:

```
Unesi broj godina: 20
Sledece godine imaces 21
```

Kada očekujemo decimalan broj, umesto funkcije `int` koristi se funkcija `float`. Na primer:

```
visina = float(input("Unesi visinu: "))
print("Uneta visina je", visina)
```

Ako korisnik unese 1.82, promenljiva `visina` dobija decimalnu vrednost 1.82.

Unos i izračunavanje mogu se povezati u jednostavan program. Na primer, sledeći program učitava cenu jednog proizvoda i željenu količinu, a zatim računa ukupnu cenu:

```
cena = float(input("Unesi cenu proizvoda: "))
kolicina = int(input("Unesi kolicinu: "))

ukupno = cena * kolicina

print("Ukupna cena je", ukupno)
```

Ako korisnik unese cenu 120.5 i količinu 3, program računa njihov proizvod i prikazuje:

```
Unesi cenu proizvoda: 120.5
Unesi kolicinu: 3
Ukupna cena je 361.5
```

Ovaj primer pokazuje čest način organizovanja jednostavnog programa. Najpre se unesu potrebni podaci, zatim se nad njima izvrši odgovarajuće izračunavanje, a na kraju se dobijeni rezultat prikaže korisniku.

2.7 Komentari i osnovna pravila pisanja

Pri pisanju programa važno je da kôd bude ne samo ispravan, već i dovoljno jasan da se lako može pročitati i razumeti. Python ima jednostavna pravila zapisivanja, a pridržavanje nekoliko osnovnih pravila znatno doprinosi preglednosti programa.

Komentari služe za dodavanje objašnjenja koja nisu deo izvršavanja programa. U Pythonu komentar počinje znakom `#` i traje do kraja reda:

```
# Cena jednog proizvoda
```

```
cena = 120

# Racunanje ukupne cene
kolicina = 3
ukupno = cena * kolicina

print(ukupno)
```

Python zanemaruje sadržaj komentara pri izvršavanju programa. Oni služe programeru da kratko objasni značenje određenog dela koda. Komentare je korisno koristiti kada doprinose razumevanju programa, ali nije potrebno objašnjavati svaku pojedinačnu naredbu.

Pri izboru imena promenljivih postoje određena pravila. Ime može sadržati slova, cifre i donju crtu `_`, ali ne može početi cifrom. Na primer, sledeća imena su ispravna:

```
broj = 10
broj_studenata = 25
ocena1 = 9
```

Sa druge strane, zapis

```
2broj = 10
```

nije ispravan, jer ime počinje cifrom.

Python razlikuje velika i mala slova. Zbog toga `broj`, `Broj` i `BROJ` predstavljaju tri različita imena:

```
broj = 5
Broj = 10

print(broj)
print(Broj)
```

Izvršavanjem ovog programa prikazuju se vrednosti 5 i 10.

Poželjno je birati imena koja jasno opisuju podatke koje predstavljaju. Ako se ime sastoji od više reči, u Pythonu je uobičajeno da se one razdvoje donjom crtom:

```
ime_studenta = "Ana"
broj_poena = 85
prosecna_ocena = 8.75
```

Ovakva imena su obično preglednija od nejasnih skraćenica i olakšavaju razumevanje programa.

Neke reči u Pythonu imaju posebno značenje i ne mogu se koristiti kao imena promenljivih. Među njima su, na primer, `if`, `else`, `for`, `while`, `def`, `return` i `class`. Sa njihovim značenjem upoznavaćemo se postepeno kada budemo koristili odgovarajuće delove jezika.

Python naredbe se najčešće pišu jedna ispod druge i ne završavaju se tačkom-zarezom:

```
a = 5
b = 3
zbir = a + b
print(zbir)
```

Ovakav način zapisivanja doprinosi jednostavnom i preglednom izgledu Python programa.

Posebno važno pravilo u Pythonu odnosi se na uvlačenje pojedinih redova koda. Ono se koristi da označi koje naredbe pripadaju određenom delu programa. Ovu osobinu već smo videli u jednostavnom primeru sa petljom:

```
for broj in range(1, 4):  
    print(broj)
```

Naredba `print` uvučena je zato što pripada petlji i izvršava se za svaku od vrednosti 1, 2 i

3. Kada više naredbi pripada istom delu programa, one treba da budu jednako uvučene.

Za uvlačenje se uobičajeno koriste četiri razmaka:

```
for broj in range(1, 4):  
    print("Broj je", broj)  
    print("Njegov kvadrat je", broj ** 2)
```

Obe naredbe `print` u ovom primeru pripadaju petlji. Nepravilno uvlačenje može promeniti značenje programa ili dovesti do greške, pa je važno da se ono koristi dosledno. Njegovu ulogu detaljnije ćemo videti pri razmatranju uslova, petlji i funkcija.

3 Kontrola toka i funkcije u Pythonu

3.1 Uslovno izvršavanje

U dosadašnjim primerima naredbe programa uglavnom su se izvršavale redom kojim su zapisane. Međutim, često želimo da se neka naredba izvrši samo kada je određeni uslov ispunjen. Takvo ponašanje omogućava uslovno izvršavanje.

U Pythonu se za osnovno uslovno izvršavanje koristi naredba `if`. Na primer:

```
broj = 8

if broj > 0:
    print("Broj je pozitivan.")
```

Uslov `broj > 0` ima vrednost `True`, pa se naredba `print` izvršava. Ako bi promenljiva `broj` imala vrednost manju ili jednaku nuli, ova naredba bi bila preskočena.

Posle uslova navodi se dvotačka, a naredbe koje pripadaju naredbi `if` pišu se uvučeno. Na taj način Python određuje koji deo koda treba izvršiti kada je uslov ispunjen.

Često postoje dve mogućnosti i želimo da se uvek izvrši jedna od njih. Tada se uz `if` koristi i `else`. Na primer, možemo proveriti da li je uneti broj paran:

```
broj = int(input("Unesi ceo broj: "))

if broj % 2 == 0:
    print("Broj je paran.")
else:
    print("Broj je neparan.")
```

Operator `%` daje ostatak pri deljenju. Ako je ostatak pri deljenju broja sa 2 jednak nuli, uslov je ispunjen i izvršava se prva naredba `print`. U suprotnom se izvršava naredba koja pripada delu `else`.

Na primer, ako korisnik unese broj 12, program prikazuje:

```
Unesi ceo broj: 12
Broj je paran.
```

Ako postoji više od dve mogućnosti, može se koristiti `elif`. Na primer, broj možemo razvrstati kao pozitivan, negativan ili jednak nuli:

```
broj = int(input("Unesi ceo broj: "))

if broj > 0:
    print("Broj je pozitivan.")
elif broj < 0:
    print("Broj je negativan.")
else:
    print("Broj je nula.")
```

Python najpre proverava uslov uz `if`. Ako on nije ispunjen, proverava se uslov uz `elif`. Deo `else` izvršava se samo ako nijedan od prethodnih uslova nije ispunjen.

Kada postoji više delova `elif`, uslovi se proveravaju redom kojim su zapisani. Izvršava se samo prva grana čiji je uslov ispunjen, dok se ostale preskaču. Na primer:

```
temperatura = float(input("Unesi temperaturu: "))

if temperatura < 0:
    print("Temperatura je ispod nule.")
elif temperatura < 20:
    print("Temperatura je najmanje 0, a manja od 20 stepeni.")
else:
    print("Temperatura je najmanje 20 stepeni.")
```

Ako je temperatura 12, prvi uslov nije ispunjen, ali drugi jeste, pa se prikazuje odgovarajuća poruka. Deo `else` se tada više ne razmatra.

Uslov može biti i složeniji logički izraz. Za povezivanje više uslova koriste se operatori `and`, `or` i `not`, koje smo već upoznali. Na primer:

```
broj = int(input("Unesi broj: "))

if broj >= 1 and broj <= 10:
    print("Broj pripada intervalu od 1 do 10.")
else:
    print("Broj ne pripada intervalu od 1 do 10.")
```

Prva grana izvršava se samo ako su ispunjena oba uslova, odnosno ako je broj istovremeno najmanje 1 i najviše 10.

Operator `or` koristi se kada je dovoljno da bude ispunjen bar jedan od uslova. Na primer:

```
dan = input("Unesi dan: ")

if dan == "subota" or dan == "nedelja":
    print("Vikend je.")
else:
    print("Radni dan je.")
```

U ovom primeru prva poruka se prikazuje ako promenljiva `dan` ima vrednost `"subota"` ili `"nedelja"`. U svim ostalim slučajevima izvršava se deo `else`.

Uslovne naredbe mogu se nalaziti i unutar drugih uslovnih naredbi. Tada uvlačenje koda jasno pokazuje kojoj naredbi svaki deo pripada. Takvi slučajevi mogu biti korisni kada odluka zavisi od više uzastopnih provera, ali jednostavnije uslove je obično preglednije zapisati pomoću `elif` i logičkih operatora.

Uslovno izvršavanje omogućava da se tok programa prilagodi vrednostima podataka sa kojima program radi. Drugi važan način promene uobičajenog redosleda izvršavanja predstavlja ponavljanje naredbi, koje ćemo razmotriti pomoću petlji.

3.2 Petlja while

U programima je često potrebno da se iste naredbe izvrše više puta. Umesto da ih ponovno zapisujemo, možemo koristiti petlju. Petlja `while` ponavlja određene naredbe sve dok je zadati uslov ispunjen.

Jednostavan primer je ispis brojeva od 1 do 5:

```
broj = 1
```

```
while broj <= 5:
    print(broj)
    broj += 1
```

Pre svakog izvršavanja tela petlje proverava se uslov `broj <= 5`. Ako je njegova vrednost `True`, izvršavaju se uvučene naredbe koje pripadaju petlji. Nakon toga uslov se ponovo proverava.

Na početku promenljiva `broj` ima vrednost 1. Ta vrednost se prikazuje, a zatim se naredbom `broj += 1` uvećava za jedan. Isto se ponavlja za vrednosti 2, 3, 4 i 5. Kada promenljiva dobije vrednost 6, uslov više nije ispunjen i izvršavanje petlje se završava.

Rezultat programa je:

```
1
2
3
4
5
```

Promenljiva čija se vrednost menja pri svakom prolasku kroz petlju često služi za praćenje broja ponavljanja. U prethodnom primeru tu ulogu ima promenljiva `broj`.

Na sličan način možemo napraviti odbrojavanje:

```
broj = 5

while broj > 0:
    print(broj)
    broj -= 1

print("Kraj")
```

Petlja se izvršava dok je vrednost promenljive `broj` veća od nule. Njena vrednost se pri svakom prolasku smanjuje za jedan, pa se dobija:

```
5
4
3
2
1
Kraj
```

Naredba `print("Kraj")` nije uvučena, pa ne pripada petlji. Ona se izvršava samo jednom, nakon što se petlja završi.

Pri korišćenju petlje `while` važno je da se tokom njenog izvršavanja nešto promeni tako da uslov u nekom trenutku može postati netačan. Na primer:

```
broj = 1

while broj <= 5:
    print(broj)
```

U ovom programu vrednost promenljive `broj` se nikada ne menja. Uslov `broj <= 5` zato

ostaje ispunjen i petlja se ponavlja bez završetka. Takva petlja naziva se beskonačna petlja.

Petlja `while` je posebno korisna kada unapred ne znamo koliko puta će biti potrebno ponoviti određene naredbe. Na primer, možemo zahtevati od korisnika da unosi pozitivan broj sve dok unos ne bude odgovarajući:

```
broj = int(input("Unesi pozitivan broj: "))

while broj <= 0:
    print("Broj mora biti pozitivan.")
    broj = int(input("Unesi pozitivan broj: "))

print("Unet je broj", broj)
```

Prvi unos vrši se pre petlje. Ako uneseni broj nije pozitivan, uslov `broj <= 0` je ispunjen i program traži novi unos. To se ponavlja sve dok korisnik ne unese broj veći od nule.

Na primer, izvršavanje može izgledati ovako:

```
Unesi pozitivan broj: -4
Broj mora biti pozitivan.
Unesi pozitivan broj: 0
Broj mora biti pozitivan.
Unesi pozitivan broj: 7
Unet je broj 7
```

U ovom primeru unapred nije poznato koliko puta će korisnik uneti neodgovarajuću vrednost. Petlja se zato ne vezuje za unapred određen broj ponavljanja, već traje dok je određeni uslov ispunjen.

Petlja `while` pogodna je kada ponavljanje zavisi od nekog uslova i broj ponavljanja nije nužno poznat unapred. Kada želimo da prolazimo kroz unapred određen niz vrednosti, često je pogodnija petlja `for`.

3.3 Petlja for i funkcija range

Kada unapred znamo kroz koje vrednosti želimo da prolazimo, često je pogodnije koristiti petlju `for`. Ona omogućava da se iste naredbe izvrše za svaku vrednost iz određenog niza.

Za formiranje jednostavnih nizova celih brojeva često se koristi funkcija `range`. Na primer:

```
for broj in range(5):
    print(broj)
```

Izvršavanjem ovog programa dobija se:

```
0
1
2
3
4
```

Funkcija `range(5)` određuje vrednosti od 0 do 4. Pri svakom prolasku kroz petlju promenljiva `broj` dobija sledeću vrednost, a zatim se izvršavaju uvučene naredbe koje pripadaju petlji.

Važno je primetiti da vrednost 5 nije obuhvaćena. Kada se funkciji `range` prosledi samo jedan broj, vrednosti počinju od nule i završavaju se neposredno pre zadanog broja.

Početnu vrednost možemo navesti i posebno. Na primer:

```
for broj in range(1, 6):  
    print(broj)
```

U ovom slučaju dobijaju se vrednosti 1, 2, 3, 4 i 5. Prvi argument određuje početnu vrednost, dok drugi određuje granicu koja se ne uključuje.

Zato izraz

```
range(3, 8)
```

određuje vrednosti 3, 4, 5, 6 i 7.

Funkciji `range` može se zadati i treći argument, koji određuje korak između susednih vrednosti. Na primer, parne brojeve od 2 do 10 možemo prikazati ovako:

```
for broj in range(2, 11, 2):  
    print(broj)
```

Dobija se:

```
2  
4  
6  
8  
10
```

Početna vrednost je 2, granica je 11, a korak je 2. Vrednost promenljive `broj` zato se pri svakom prolasku povećava za dva.

Korak može biti i negativan, što omogućava prolazak kroz vrednosti u opadajućem redosledu:

```
for broj in range(5, 0, -1):  
    print(broj)
```

Rezultat je:

```
5  
4  
3  
2  
1
```

I ovde se krajnja vrednost ne uključuje. Pošto je granica 0, poslednja vrednost koju promenljiva `broj` dobija jeste 1.

Petlja `for` može se koristiti i za izračunavanja. Na primer, zbir svih celih brojeva od 1 do zadanog broja možemo izračunati na sledeći način:

```
n = int(input("Unesi pozitivan ceo broj: "))  
  
zbir = 0  
  
for broj in range(1, n + 1):
```

```
zbir += broj

print("Zbir je", zbir)
```

Ako korisnik unese 5, petlja prolazi kroz vrednosti od 1 do 5. Svaka od njih dodaje se trenutnoj vrednosti promenljive `zbir`, pa ona redom dobija vrednosti 1, 3, 6, 10 i 15. Na kraju se prikazuje:

```
Unesi pozitivan ceo broj: 5
Zbir je 15
```

U izrazu `range(1, n + 1)` koristi se `n + 1` zato što se krajnja vrednost funkcije `range` ne uključuje. Na taj način poslednja vrednost kroz koju petlja prolazi jeste upravo `n`.

Još jedan jednostavan primer je prikazivanje tablice množenja za broj koji korisnik unese:

```
broj = int(input("Unesi broj: "))

for faktor in range(1, 11):
    proizvod = broj * faktor
    print(broj, "*", faktor, "=", proizvod)
```

Ako korisnik unese 3, početak ispisa izgleda ovako:

```
Unesi broj: 3
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
```

Petlja se nastavlja sve do vrednosti 10, pa se isti postupak ponavlja za svaki faktor od 1 do 10.

Petlje `while` i `for` omogućavaju ponavljanje naredbi, ali se obično koriste u različitim situacijama. Petlja `while` pogodna je kada ponavljanje zavisi od uslova i unapred ne znamo koliko će puta biti potrebno. Petlja `for` je prirodnija kada želimo da redom prolazimo kroz unapred određene vrednosti.

Petlja `for` može se koristiti i za prolazak kroz druge vrste podataka, a ne samo kroz vrednosti dobijene pomoću funkcije `range`. Takve primere upoznaćemo kasnije pri radu sa kolekcijama.

3.4 Definisanje i pozivanje funkcija

U prethodnim primerima već smo koristili različite funkcije koje Python pruža, kao što su `print`, `input`, `type` i `range`. Pored postojećih funkcija, možemo definisati i sopstvene funkcije koje obavljaju određeni zadatak.

Funkcija omogućava da grupu naredbi izdvojimo u celinu kojoj dodeljujemo ime. Kada kasnije želimo da se te naredbe izvrše, dovoljno je da je pozovemo.

Nova funkcija definiše se pomoću ključne reči `def`. Na primer:

```
def pozdrav():
    print("Zdravo!")
```

Ovim kodom definisana je funkcija koja se zove **pozdrav**. Posle imena funkcije navode se zagrade, a zatim dvotačka. Naredbe koje čine telo funkcije pišu se uvučeno.

Samo definisanje funkcije ne izvršava naredbe iz njenog tela. Da bi se funkcija izvršila, potrebno je da je pozovemo:

```
def pozdrav():  
    print("Zdravo!")  
  
pozdrav()
```

Izvršavanjem ovog programa prikazuje se:

```
Zdravo!
```

Prve dve linije određuju šta funkcija **pozdrav** radi, dok se naredbom **pozdrav()** funkcija poziva. Tek tada se izvršava njeno telo i prikazuje odgovarajuća poruka.

Telo funkcije može sadržati više naredbi:

```
def prikazi_pozdrav():  
    print("Zdravo!")  
    print("Dobrodosli na kurs.")  
    print("Pocinjemo sa radom.")  
  
prikazi_pozdrav()
```

Sve tri naredbe **print** pripadaju funkciji jer su jednako uvučene. Jednim pozivom funkcije izvršavaju se redom sve naredbe iz njenog tela.

Jedna od prednosti funkcija jeste to što isti deo programa možemo izvršiti više puta bez ponovnog zapisivanja svih njegovih naredbi. Na primer:

```
def prikazi_poruku():  
    print("Program se izvrsava.")  
  
prikazi_poruku()  
prikazi_poruku()  
prikazi_poruku()
```

Rezultat je:

```
Program se izvrsava.  
Program se izvrsava.  
Program se izvrsava.
```

Telo funkcije zapisano je samo jednom, a zatim je funkcija pozvana tri puta. Ako kasnije želimo da promenimo poruku, dovoljno je izmeniti njenu definiciju umesto svakog mesta na kome se poruka koristi.

U telu funkcije mogu se koristiti i druge konstrukcije koje smo već upoznali. Na primer, funkcija može sadržati uslovnu naredbu ili petlju. Sledeća funkcija prikazuje brojeve od 1 do 5:

```
def prikazi_brojeve():  
    for broj in range(1, 6):  
        print(broj)
```

```
prikazi_brojeve()
```

Pozivanjem funkcije izvršava se petlja koja se nalazi u njenom telu, pa se dobija:

```
1
2
3
4
5
```

U ovom primeru mogu se uočiti dva nivoa uvlačenja. Petlja `for` uvučena je zato što pripada funkciji, dok je naredba `print` dodatno uvučena zato što pripada petlji.

Funkcije mogu pomoći i pri organizovanju programa na manje i preglednije celine. Na primer, različite delove jednostavnog programa možemo izdvojiti u posebne funkcije:

```
def prikazi_naslov():
    print("Program za vezbanje Pythona")

def prikazi_kraj():
    print("Kraj programa")

prikazi_naslov()
print("Glavni deo programa")
prikazi_kraj()
```

Izvršavanjem programa dobija se:

```
Program za vezbanje Pythona
Glavni deo programa
Kraj programa
```

Svaka funkcija u ovom primeru ima jasno određen zadatak, dok pozivi funkcija pokazuju redosled kojim se ti zadaci izvršavaju.

Imena funkcija podležu istim osnovnim pravilima koja smo upoznali za imena promenljivih. Poželjno je birati jasna imena koja opisuju zadatak funkcije, a kada se ime sastoji od više reči, one se uobičajeno razdvajaju donjom crtom, kao u imenima `prikazi_poruku` i `prikazi_brojeve`.

3.5 Parametri i povratne vrednosti

Funkcije koje smo do sada definisali uvek su izvršavale isti unapred određeni postupak. Njihova primena postaje znatno šira kada im možemo proslediti podatke nad kojima treba da rade.

Podatak koji funkcija prima opisuje se pomoću parametra. Parametar se navodi između zagrada pri definisanju funkcije. Na primer:

```
def pozdrav(ime):
    print("Zdravo, ", ime)

pozdrav("Ana")
```

U definiciji funkcije `pozdrav` navedeno je ime `ime`. Ono predstavlja parametar funkcije. Pri pozivu `pozdrav("Ana")` funkciji se prosleđuje vrednost `"Ana"`, pa se ona tokom tog poziva koristi kao vrednost parametra `ime`.

Rezultat je:

```
Zdravo, Ana
```

Vrednost koja se funkciji prosleđuje pri pozivu naziva se argument. U prethodnom primeru `ime` je parametar, dok je `"Ana"` argument.

Ista funkcija može se pozvati više puta sa različitim argumentima:

```
def pozdrav(ime):  
    print("Zdravo,", ime)  
  
pozdrav("Ana")  
pozdrav("Marko")  
pozdrav("Jelena")
```

Dobija se:

```
Zdravo, Ana  
Zdravo, Marko  
Zdravo, Jelena
```

Pri svakom pozivu izvršava se isto telo funkcije, ali parametar `ime` dobija drugu vrednost. Na taj način jedna definicija funkcije može se koristiti za različite podatke.

Funkcija može imati i više parametara. Oni se pri definisanju razdvajaju zarezima:

```
def prikazi_zbir(a, b):  
    zbir = a + b  
    print("Zbir je", zbir)  
  
prikazi_zbir(4, 7)  
prikazi_zbir(10, 20)
```

Pri prvom pozivu parametri `a` i `b` dobijaju vrednosti 4 i 7, dok pri drugom dobijaju vrednosti 10 i 20. Zato se prikazuju vrednosti 11 i 30.

Funkcija ne mora samo da prikazuje rezultat. Često je korisnije da izračunatu vrednost vrati delu programa koji ju je pozvao. Za to se koristi naredba `return`.

Na primer, funkcija koja računa kvadrat broja može se definisati ovako:

```
def kvadrat(broj):  
    return broj * broj
```

Poziv funkcije tada ima vrednost koju je funkcija vratila:

```
rezultat = kvadrat(5)  
print(rezultat)
```

Funkcija se poziva sa argumentom 5. Parametar `broj` zato dobija tu vrednost, izraz `broj * broj` ima vrednost 25, a naredba `return` vraća tu vrednost. Ona se zatim dodeljuje promenljivoj `rezultat`.

Vraćena vrednost ne mora najpre da se sačuva u promenljivoj. Poziv funkcije može se neposredno koristiti tamo gde je potrebna njegova vrednost:

```
print(kvadrat(6))
print(kvadrat(3) + 10)
```

Prvi poziv vraća vrednost 36, dok drugi vraća 9, pa se zatim izračunava izraz $9 + 10$. Zato se prikazuju vrednosti 36 i 19.

Važno je razlikovati prikazivanje vrednosti pomoću funkcije `print` od vraćanja vrednosti pomoću naredbe `return`. Na primer:

```
def prikazi_kvadrat(broj):
    print(broj * broj)

def izracunaj_kvadrat(broj):
    return broj * broj
```

Funkcija `prikazi_kvadrat` prikazuje izračunatu vrednost korisniku. Sa druge strane, funkcija `izracunaj_kvadrat` vraća vrednost programu koji ju je pozvao. Zato rezultat druge funkcije možemo dalje koristiti u izračunavanjima:

```
rezultat = izracunaj_kvadrat(4)
ukupno = rezultat + 5

print(ukupno)
```

Dobija se vrednost 21. Ovakav pristup je koristan kada rezultat funkcije treba dalje obrađivati, a ne samo prikazati.

Naredba `return` može se koristiti i zajedno sa uslovnim izvršavanjem. Na primer, sledeća funkcija vraća veći od dva broja:

```
def veci(a, b):
    if a > b:
        return a
    else:
        return b

print(veci(8, 5))
print(veci(3, 10))
```

Pri prvom pozivu uslov $a > b$ je ispunjen, pa funkcija vraća vrednost 8. Pri drugom pozivu uslov nije ispunjen, pa se izvršava deo `else` i vraća se vrednost 10.

Promenljive definisane unutar funkcije koriste se za njen unutrašnji rad. Na primer:

```
def povrsina_pravougaonika(sirina, visina):
    povrsina = sirina * visina
    return povrsina

rezultat = povrsina_pravougaonika(4, 3)
print(rezultat)
```

Parametri `sirina` i `visina`, kao i promenljiva `povrsina`, koriste se unutar funkcije. Pro-

menljiva `rezultat` nalazi se izvan funkcije i dobija vrednost koju funkcija vraća. Promenljive koje se definišu i koriste unutar funkcije nazivaju se lokalne promenljive.

Funkcije mogu sadržati i petlje. Na primer, možemo definisati funkciju koja računa zbir svih celih brojeva od 1 do zadanog broja:

```
def zbir_do(n):
    zbir = 0

    for broj in range(1, n + 1):
        zbir += broj

    return zbir

rezultat = zbir_do(5)
print(rezultat)
```

Parametar `n` dobija vrednost 5. Petlja zatim prolazi kroz brojeve od 1 do 5 i postepeno ih dodaje promenljivoj `zbir`. Nakon završetka petlje naredba `return` vraća konačnu vrednost 15.

Možemo povezati funkciju i sa unosom podataka:

```
def zbir_do(n):
    zbir = 0

    for broj in range(1, n + 1):
        zbir += broj

    return zbir

n = int(input("Unesi pozitivan ceo broj: "))
rezultat = zbir_do(n)

print("Zbir je", rezultat)
```

Na ovaj način glavni deo programa određuje koje podatke funkcija treba da obradi, dok je sam postupak računanja izdvojen u funkciju. Funkcije tako omogućavaju da se složeniji program podeli na manje celine sa jasno određenim zadacima, što je jedna od važnih ideja proceduralnog organizovanja programa.

3.6 Rekurzivne funkcije

Funkcija može u svojoj definiciji pozivati i samu sebe. Takva funkcija naziva se rekurzivna funkcija, a ovakav način rešavanja problema rekurzija.

Da bi se rekurzivno izračunavanje završilo, potrebno je odrediti bazni slučaj u kome se rezultat dobija neposredno, bez novog poziva iste funkcije. U ostalim slučajevima problem se svodi na jednostavniji problem iste vrste, sve dok se ne dođe do baznog slučaja.

Jedan od najpoznatijih primera jeste faktorijel nenegativnog celog broja. Faktorijel broja 0 jednak je 1, dok se za $n > 0$ može izračunati kao

$$n! = n \cdot (n - 1)!$$

Ovakva definicija može se neposredno zapisati pomoću rekurzivne funkcije:

```
def faktorijel(n):
    if n == 0:
        return 1
    return n * faktorijel(n - 1)
```

U ovom primeru slučaj `n == 0` predstavlja bazni slučaj. Tada funkcija neposredno vraća vrednost 1. Za pozitivan broj `n` funkcija poziva samu sebe sa manjim argumentom `n - 1`.

Na primer:

```
print(faktorijel(4))
```

Dobija se:

```
24
```

Izračunavanje možemo pratiti korak po korak:

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

Pri svakom rekurzivnom pozivu argument se smanjuje za 1. Na taj način se posle konačnog broja koraka dolazi do baznog slučaja `faktorijel(0)`.

Važno je da rekurzivni pozivi zaista vode ka baznom slučaju. Na primer, sledeća funkcija nije ispravno definisana:

```
def funkcija(n):
    return funkcija(n + 1)
```

Ona nema bazni slučaj, a svaki novi poziv vodi do još većeg argumenta. Zbog toga se izračunavanje ne završava na uobičajen način i Python će nakon velikog broja ugnježenih rekurzivnih poziva prijaviti grešku.

Još jedan jednostavan primer jeste zbir svih celih brojeva od 1 do `n`. Za `n = 0` zbir je 0, dok se za pozitivan broj `n` rezultat može dobiti tako što se broj `n` doda zbiru brojeva od 1 do `n - 1`:

```
def zbir_do(n):
    if n == 0:
        return 0
    return n + zbir_do(n - 1)
```

Na primer:

```
print(zbir_do(5))
```

Dobija se 15. Isti problem može se rešiti i pomoću petlje:

```
def zbir_do(n):
```

```
zbir = 0

for broj in range(1, n + 1):
    zbir += broj

return zbir
```

U prvoj definiciji ponavljanje se ostvaruje rekurzivnim pozivima funkcije, dok se u drugoj koristi petlja. Dakle, rekurzija i petlje mogu predstavljati različite načine rešavanja istog problema.

U Pythonu se za mnoge probleme koji zahtevaju ponavljanje prirodno koriste petlje, ali postoje i problemi čija se struktura pregledno može izraziti rekurzijom. U Haskellu ćemo videti da rekurzija ima još značajniju ulogu, naročito pri definisanju funkcija koje obrađuju liste. To će nam omogućiti i da uporedimo različite načine organizovanja izračunavanja u proceduralnoj i funkcionalnoj paradigmi.

4 Kolekcije u Pythonu

4.1 Liste

U programima je često potrebno čuvati više međusobno povezanih vrednosti. Na primer, možemo imati više ocena jednog studenta, temperature izmerene tokom nekoliko dana ili imena više učenika. Umesto da za svaku vrednost koristimo posebnu promenljivu, podatke možemo objediniti u kolekciju.

Jedna od najčešće korišćenih kolekcija u Pythonu jeste lista. Lista predstavlja uređenu kolekciju vrednosti koje se zapisuju između uglastih zagrada i razdvajaju zarezima.

Na primer, nekoliko ocena možemo zapisati ovako:

```
ocene = [8, 10, 7, 9, 10]
```

Vrednosti 8, 10, 7, 9 i 10 predstavljaju elemente liste. Njihov redosled je sačuvan, pa je prvi element vrednost 8, drugi 10, treći 7 i tako dalje.

Lista može sadržati i tekstualne vrednosti:

```
imena = ["Ana", "Marko", "Jelena"]
```

U ovom primeru lista sadrži tri imena. Ceo sadržaj liste može se prikazati pomoću funkcije `print`:

```
imena = ["Ana", "Marko", "Jelena"]  
  
print(imena)
```

Dobija se:

```
['Ana', 'Marko', 'Jelena']
```

Python pri prikazivanju liste prikazuje njene elemente između uglastih zagrada. Tekstualne vrednosti pritom se prikazuju sa navodnicima kako bi bilo jasno da su u pitanju stringovi.

Lista može biti i prazna:

```
rezultati = []
```

Takva lista trenutno nema nijedan element. Elementi joj se mogu dodavati tokom izvršavanja programa, što ćemo razmotriti u narednoj podsekciji.

Broj elemenata liste može se dobiti pomoću funkcije `len`. Na primer:

```
ocene = [8, 10, 7, 9, 10]  
  
print(len(ocene))
```

Lista `ocene` sadrži pet elemenata, pa se prikazuje vrednost 5.

Funkcija `len` ne računa koliko se različitih vrednosti nalazi u listi, već koliko lista ukupno ima elemenata. Zato ponovljene vrednosti predstavljaju posebne elemente. U listi

```
ocene = [10, 8, 10, 9]
```

vrednost 10 pojavljuje se dva puta, a lista ima ukupno četiri elementa.

Python dozvoljava da elementi iste liste budu različitih tipova. Na primer, moguć je zapis:

```
podaci = ["Ana", 20, 8.75, True]
```

Ova lista sadrži tekst, ceo broj, decimalan broj i logičku vrednost. Ipak, liste se često koriste za čuvanje međusobno srodnih podataka, pa ćemo u većini primera koristiti elemente koji predstavljaju istu vrstu podataka.

Promenljivoj se može dodeliti cela lista, a zatim se sa njom može raditi kao sa jednom celinom. Na primer:

```
temperature = [18.5, 20.0, 21.5, 19.0]

print("Temperature:", temperature)
print("Broj merenja:", len(temperature))
```

Izvršavanjem programa dobija se:

```
Temperature: [18.5, 20.0, 21.5, 19.0]
Broj merenja: 4
```

Lista nam tako omogućava da više povezanih vrednosti čuvamo pod jednim imenom. U nastavku ćemo videti kako se pristupa pojedinačnim elementima liste i kako se njen sadržaj može menjati.

4.2 Pristup elementima i izmena listi

Elementi liste imaju određen redosled, pa svakom od njih možemo pristupiti pomoću njegovog položaja. Položaj elementa određuje se indeksom.

U Pythonu indeksiranje počinje od nule. Zato prvi element liste ima indeks 0, drugi indeks 1 i tako dalje. Na primer:

```
ocene = [8, 10, 7, 9]

print(ocene[0])
print(ocene[1])
print(ocene[3])
```

Dobija se:

```
8
10
9
```

Izraz `ocene[0]` predstavlja prvi element liste, `ocene[1]` drugi, dok `ocene[3]` predstavlja četvrti element.

Python omogućava i korišćenje negativnih indeksa. Indeks `-1` označava poslednji element liste, `-2` preposlednji i tako dalje:

```
imena = ["Ana", "Marko", "Jelena", "Petar"]

print(imena[-1])
print(imena[-2])
```

Rezultat je:

```
Petar  
Jelena
```

Negativni indeksi su naročito korisni kada želimo da pristupimo elementima sa kraja liste, a da prethodno ne računamo njihov položaj.

Indeks mora odgovarati postojećem elementu liste. Na primer, lista sa četiri elementa ima indekse od 0 do 3. Pokušaj pristupa elementu `lista[4]` u takvoj listi dovodi do greške.

Vrednost postojećeg elementa može se promeniti dodelom nove vrednosti odgovarajućem indeksu:

```
ocene = [8, 10, 7, 9]  
  
ocene[2] = 10  
  
print(ocene)
```

Treći element, koji je prethodno imao vrednost 7, sada dobija vrednost 10. Zato se prikazuje:

```
[8, 10, 10, 9]
```

Na ovaj način sadržaj liste može se menjati tokom izvršavanja programa.

Pored pojedinačnih elemenata, moguće je izdvojiti i deo liste. Za to se navode početni i krajnji indeks razdvojeni dvotačkom:

```
brojevi = [10, 20, 30, 40, 50, 60]  
  
deo = brojevi[1:4]  
  
print(deo)
```

Dobija se:

```
[20, 30, 40]
```

Početni indeks 1 je uključen, dok krajnji indeks 4 nije. Zato se izdvajaju elementi sa indeksima 1, 2 i 3.

Početni ili krajnji indeks mogu se i izostaviti. Ako se izostavi početni indeks, uzimaju se elementi od početka liste:

```
brojevi = [10, 20, 30, 40, 50]  
  
print(brojevi[:3])
```

Rezultat je:

```
[10, 20, 30]
```

Ako se izostavi krajnji indeks, uzimaju se elementi od zadatog položaja do kraja liste:

```
brojevi = [10, 20, 30, 40, 50]
```

```
print(brojevi[2:])
```

Dobija se:

```
[30, 40, 50]
```

Sadržaj liste može se menjati i dodavanjem novih elemenata. Metoda `append` dodaje novi element na kraj liste:

```
imena = ["Ana", "Marko"]  
  
imena.append("Jelena")  
  
print(imena)
```

Rezultat je:

```
['Ana', 'Marko', 'Jelena']
```

Prazna lista se zato može postepeno popunjavati:

```
brojevi = []  
  
brojevi.append(5)  
brojevi.append(8)  
brojevi.append(12)  
  
print(brojevi)
```

Dobija se:

```
[5, 8, 12]
```

Element se može ukloniti pomoću metode `remove`. Navodi se vrednost koju želimo da uklonimo:

```
imena = ["Ana", "Marko", "Jelena"]  
  
imena.remove("Marko")  
  
print(imena)
```

Nakon uklanjanja dobija se:

```
['Ana', 'Jelena']
```

Ako se ista vrednost u listi pojavljuje više puta, metoda `remove` uklanja samo njeno prvo pojavljivanje.

Kada želimo da uklonimo element sa određenog položaja, možemo koristiti metodu `pop`. Na primer:

```
brojevi = [10, 20, 30, 40]
```

```
brojevi.pop(1)

print(brojevi)
```

Uklanja se element sa indeksom 1, odnosno vrednost 20, pa se dobija:

```
[10, 30, 40]
```

Ako se metod `pop` pozove bez indeksa, uklanja se poslednji element liste:

```
brojevi = [10, 20, 30]

brojevi.pop()

print(brojevi)
```

Rezultat je:

```
[10, 20]
```

Na jednom jednostavnom primeru možemo povezati nekoliko prethodnih operacija:

```
ocene = [8, 9, 7]

ocene[2] = 10
ocene.append(9)
ocene.remove(8)

print(ocene)
```

Lista na početku sadrži vrednosti 8, 9 i 7. Zatim se poslednja ocena menja u 10, na kraj se dodaje ocena 9, a vrednost 8 se uklanja. Konačan sadržaj liste je:

```
[9, 10, 9]
```

Liste se tako mogu menjati tokom izvršavanja programa. Pojedinačnim elementima pristupa se pomoću indeksa, delovi liste mogu se izdvajati pomoću odgovarajućih granica, a elementi se mogu menjati, dodavati i uklanjati. U narednoj podsekciji videćemo kako se kroz elemente liste prolazi pomoću petlje i kako se nad čitavom listom mogu obavljati različite operacije.

4.3 Prolazak kroz liste i osnovne operacije

Kada želimo da obradimo sve elemente liste, najčešće koristimo petlju `for`. Promenljiva petlje tada redom dobija vrednost svakog elementa liste.

Na primer:

```
ocene = [8, 10, 7, 9]

for ocena in ocene:
    print(ocena)
```

Izvršavanjem programa dobija se:

```
8
10
7
9
```

Pri prvom prolasku kroz petlju promenljiva `ocena` dobija vrednost 8, zatim 10, pa 7 i na kraju 9. Petlja se završava kada se obrade svi elementi liste.

Na isti način možemo prolaziti i kroz listu tekstualnih vrednosti:

```
imena = ["Ana", "Marko", "Jelena"]

for ime in imena:
    print("Zdravo,", ime)
```

Dobija se:

```
Zdravo, Ana
Zdravo, Marko
Zdravo, Jelena
```

Kada nam je potrebna samo vrednost svakog elementa, ovakav način prolaska kroz listu je najjednostavniji. Ponekad nam je, međutim, potreban i položaj elementa. Tada možemo prolaziti kroz indekse pomoću funkcija `range` i `len`:

```
imena = ["Ana", "Marko", "Jelena"]

for indeks in range(len(imena)):
    print(indeks, imena[indeks])
```

Lista ima tri elementa, pa `len(imena)` ima vrednost 3. Funkcija `range(3)` određuje indekse 0, 1 i 2. Zato se dobija:

```
0 Ana
1 Marko
2 Jelena
```

Ovakav prolazak je koristan kada nam je, pored samog elementa, potreban i njegov indeks. Ako nam indeks nije potreban, jednostavnije je neposredno prolaziti kroz elemente liste.

Često želimo samo da proverimo da li se određena vrednost nalazi u listi. Za to se koristi operator `in`:

```
ocene = [8, 10, 7, 9]

print(10 in ocene)
print(6 in ocene)
```

Dobija se:

```
True
False
```

Izraz `10 in ocene` ima vrednost `True`, jer se broj 10 nalazi u listi. Izraz `6 in ocene` ima

vrednost `False`.

Suprotna provera može se izvršiti pomoću operatora `not in`:

```
imena = ["Ana", "Marko", "Jelena"]

if "Petar" not in imena:
    print("Petar nije u listi.")
```

Pošto se vrednost `"Petar"` ne nalazi u listi, poruka se prikazuje.

Za neke česte operacije nad listama Python pruža već gotove funkcije. Funkciju `len` već smo koristili za određivanje broja elemenata. Za liste brojeva mogu se koristiti i funkcije `sum`, `min` i `max`:

```
brojevi = [12, 5, 18, 9]

print(len(brojevi))
print(sum(brojevi))
print(min(brojevi))
print(max(brojevi))
```

Dobija se:

```
4
44
5
18
```

Funkcija `sum` računa zbir elemenata, `min` određuje najmanju, a `max` najveću vrednost. Pomoću ovih funkcija lako možemo izračunati i prosečnu vrednost elemenata liste:

```
ocene = [8, 10, 7, 9]

prosek = sum(ocene) / len(ocene)

print(prosek)
```

Zbir elemenata je 34, a lista ima četiri elementa, pa se dobija:

```
8.5
```

Kada želimo da utvrdimo koliko elemenata ispunjava određeni uslov, možemo koristiti petlju i brojač. Na primer, možemo prebrojati koliko je ocena najmanje 6:

```
ocene = [5, 8, 10, 4, 7, 6]

broj_polozenih = 0

for ocena in ocene:
    if ocena >= 6:
        broj_polozenih += 1

print("Broj polozenih ocena je", broj_polozenih)
```

Ocene 8, 10, 7 i 6 ispunjavaju uslov, pa program prikazuje:

```
Broj polozenih ocena je 4
```

Petlja može poslužiti i za pretragu liste. Na primer, možemo proveriti da li lista sadrži negativan broj:

```
brojevi = [4, 7, -2, 10]

postoji_negativan = False

for broj in brojevi:
    if broj < 0:
        postoji_negativan = True

print(postoji_negativan)
```

Promenljiva `postoji_negativan` na početku ima vrednost `False`. Kada petlja nađe na vrednost `-2`, ona se menja u `True`. Zato se na kraju prikazuje:

```
True
```

Za jednostavnu proveru da li se tačno određena vrednost nalazi u listi obično je pogodnije koristiti operator `in`. Petlja je korisna kada tražimo element koji ispunjava neki složeniji uslov.

Više operacija možemo povezati u jednom programu. Na primer, za listu ocena možemo izračunati prosečnu ocenu, pronaći najmanju i najveću ocenu i prebrojati koliko je ocena prolaznih:

```
ocene = [8, 5, 10, 7, 9, 6]

broj_polozenih = 0

for ocena in ocene:
    if ocena >= 6:
        broj_polozenih += 1

prosek = sum(ocene) / len(ocene)

print("Najmanja ocena je", min(ocene))
print("Najveca ocena je", max(ocene))
print("Prosecna ocena je", prosek)
print("Broj polozenih ocena je", broj_polozenih)
```

Za ovu listu dobija se:

```
Najmanja ocena je 5
Najveca ocena je 10
Prosecna ocena je 7.5
Broj polozenih ocena je 5
```

Na ovaj način liste se mogu povezati sa petljama i uslovnim izvršavanjem kako bi se obradilo više podataka istim postupkom.

4.4 Torke i skupovi

Pored liste, Python pruža i druge vrste kolekcija. Torke i skupovi na prvi pogled mogu ličiti na liste, ali imaju drugačija svojstva i koriste se u drugačijim situacijama.

Torka je uređena kolekcija elemenata koja se najčešće zapisuje između običnih zagrada. Na primer, koordinate tačke u ravni možemo zapisati ovako:

```
tacka = (3, 5)

print(tacka)
```

Dobija se:

```
(3, 5)
```

Kao i kod liste, elementi torke imaju određen redosled i može im se pristupati pomoću indeksa:

```
tacka = (3, 5)

print(tacka[0])
print(tacka[1])
```

Rezultat je:

```
3
5
```

Indeksiranje počinje od nule, pa izraz `tacka[0]` predstavlja prvi, a `tacka[1]` drugi element. Mogu se koristiti i negativni indeksi, na isti način kao kod liste.

Broj elemenata torke može se dobiti pomoću funkcije `len`:

```
podaci = ("Ana", 20, 8.75)

print(len(podaci))
```

Torka ima tri elementa, pa se prikazuje vrednost 3.

Kroz elemente torke može se prolaziti i pomoću petlje `for`:

```
dani = ("ponedeljak", "utorak", "sreda")

for dan in dani:
    print(dan)
```

Torka se od liste razlikuje po tome što se njeni elementi nakon formiranja ne mogu menjati, dodavati ili uklanjati. Na primer, sledeća naredba nije dozvoljena:

```
tacka = (3, 5)
tacka[0] = 10
```

Pokušaj promene prvog elementa dovodi do greške. Torke su zato pogodne kada želimo da grupišemo nekoliko vrednosti koje tokom rada programa ne treba menjati. Koordinate tačke predstavljaju jednostavan primer takvih podataka.

Posebnu pažnju zahteva torka sa samo jednim elementom. Ona se zapisuje sa zarezom iza tog elementa:

```
broj = (5,)
```

Zapis (5) predstavlja samo broj 5 između zagrada, dok (5,) predstavlja torku sa jednim elementom.

Druga vrsta kolekcije je skup. Skup sadrži jedinstvene elemente i zapisuje se pomoću vitičastih zagrada:

```
brojevi = {2, 4, 6, 8}
```

Za razliku od liste i torke, kod skupa se ne oslanjamo na položaj elemenata. Elementima zato ne pristupamo pomoću indeksa.

Jedna od važnih osobina skupa jeste da ista vrednost ne može biti prisutna više puta. Na primer:

```
brojevi = {2, 4, 2, 6, 4}

print(len(brojevi))
```

Skup sadrži samo vrednosti 2, 4 i 6, pa funkcija `len` vraća vrednost 3. Ponovljena pojavljivanja iste vrednosti ne predstavljaju nove elemente skupa.

Operator `in` može se koristiti za proveru pripadnosti skupu:

```
brojevi = {2, 4, 6, 8}

print(4 in brojevi)
print(5 in brojevi)
```

Dobija se:

```
True
False
```

Novi element može se dodati metodom `add`:

```
brojevi = {2, 4, 6}

brojevi.add(8)

print(8 in brojevi)
```

Rezultat je `True`, jer je vrednost 8 dodata skupu.

Postojeći element može se ukloniti metodom `remove`:

```
brojevi = {2, 4, 6, 8}

brojevi.remove(4)

print(4 in brojevi)
```

Nakon uklanjanja rezultat je `False`.

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

```
a = {1, 2, 3}
b = {3, 4, 5}

unija = a | b
presek = a & b
razlika = a - b
```

Operator `|` određuje uniju skupova, pa `unija` sadrži vrednosti 1, 2, 3, 4 i 5. Operator `&` određuje presek, pa `presek` sadrži samo vrednost 3. Operator `-` određuje razliku, pa `razlika` sadrži vrednosti 1 i 2.

Pri prikazivanju skupa ne treba se oslanjati na određeni redosled njegovih elemenata. Skup se koristi kada je važno koje vrednosti pripadaju kolekciji, a njihov položaj nije značajan.

Prazan skup ne zapisuje se kao `{}`, jer taj zapis u Pythonu predstavlja prazan rečnik. Prazan skup formira se pomoću funkcije `set`:

```
brojevi = set()

print(len(brojevi))
```

Dobija se vrednost 0.

Liste, torke i skupovi zato imaju različita svojstva. Lista čuva uređene elemente i može se menjati, torka čuva uređene elemente koji se ne menjaju, dok skup čuva jedinstvene elemente bez oslanjanja na njihov položaj. Izbor odgovarajuće kolekcije zavisi od načina na koji podatke želimo da čuvamo i koristimo.

4.5 Rečnici

Rečnik je kolekcija u kojoj se podaci čuvaju u obliku parova ključ–vrednost. Ključ služi za pronalaženje odgovarajuće vrednosti, pa elementima rečnika ne pristupamo pomoću njihovog položaja kao kod listi i torki.

Rečnik se zapisuje pomoću vitičastih zagrada. Na primer, podatke o studentu možemo zapisati ovako:

```
student = {
    "ime": "Ana",
    "godine": 20,
    "prosek": 8.75
}
```

U ovom rečniku ključevi su `"ime"`, `"godine"` i `"prosek"`, dok su njihove odgovarajuće vrednosti `"Ana"`, 20 i 8.75.

Vrednosti se dobijaju navođenjem odgovarajućeg ključa između uglastih zagrada:

```
student = {
    "ime": "Ana",
    "godine": 20,
    "prosek": 8.75
}
```

```
print(student["ime"])
print(student["prosek"])
```

Dobija se:

```
Ana
8.75
```

Izraz `student["ime"]` daje vrednost povezanu sa ključem "ime", dok `student["prosek"]` daje vrednost povezanu sa ključem "prosek".

Ključevi u jednom rečniku moraju biti jedinstveni. Svaki ključ određuje jednu odgovarajuću vrednost. Ako želimo da promenimo postojeću vrednost, možemo je jednostavno ponovo dodeliti istom ključu:

```
student = {
    "ime": "Ana",
    "godine": 20,
    "prosek": 8.75
}

student["prosek"] = 9.1

print(student["prosek"])
```

Nova vrednost povezana sa ključem "prosek" jeste 9.1, pa se ona i prikazuje.

Na sličan način rečniku možemo dodati novi par ključ-vrednost:

```
student = {
    "ime": "Ana",
    "godine": 20
}

student["grad"] = "Beograd"

print(student)
```

Dobija se rečnik koji sada sadrži i novi podatak:

```
{'ime': 'Ana', 'godine': 20, 'grad': 'Beograd'}
```

Element se može ukloniti pomoću metode `pop`. Navodi se ključ elementa koji želimo da uklonimo:

```
student = {
    "ime": "Ana",
    "godine": 20,
    "prosek": 8.75
}

student.pop("godine")

print(student)
```

Nakon uklanjanja dobija se:

```
{'ime': 'Ana', 'prosek': 8.75}
```

Operator `in` može se koristiti za proveru da li određeni ključ postoji u rečniku:

```
student = {  
    "ime": "Ana",  
    "godine": 20  
}  
  
print("ime" in student)  
print("prosek" in student)
```

Dobija se:

```
True  
False
```

Važno je primetiti da se operator `in` kod rečnika odnosi na ključeve. U prethodnom primeru proverava se da li postoje ključevi `"ime"` i `"prosek"`, a ne da li se te vrednosti pojavljuju među podacima.

Broj parova ključ-vrednost može se dobiti pomoću funkcije `len`:

```
student = {  
    "ime": "Ana",  
    "godine": 20,  
    "prosek": 8.75  
}  
  
print(len(student))
```

Rečnik sadrži tri para, pa se prikazuje vrednost 3.

Kroz rečnik možemo prolaziti pomoću petlje `for`. Ako neposredno prolazimo kroz rečnik, promenljiva petlje redom dobija njegove ključeve:

```
student = {  
    "ime": "Ana",  
    "godine": 20,  
    "prosek": 8.75  
}  
  
for kljuc in student:  
    print(kljuc)
```

Na taj način možemo koristiti ključ da dobijemo i odgovarajuću vrednost:

```
for kljuc in student:  
    print(kljuc, student[kljuc])
```

Dobija se ispis parova ključeva i vrednosti, na primer:

```
ime Ana
```

```
godine 20
prosek 8.75
```

Još pogodniji način za istovremeni pristup ključu i vrednosti omogućava metoda `items`:

```
student = {
    "ime": "Ana",
    "godine": 20,
    "prosek": 8.75
}

for kljuc, vrednost in student.items():
    print(kljuc, vrednost)
```

Pri svakom prolasku kroz petlju dobijaju se jedan ključ i njegova odgovarajuća vrednost. Zato se svi podaci iz rečnika mogu obraditi bez dodatnog pristupa pomoću uglastih zagrada.

Rečnici su naročito pogodni kada podatke prirodno prepoznamo po nekom imenu ili drugom ključu. Na primer, cene nekoliko proizvoda možemo zapisati ovako:

```
cene = {
    "hleb": 80,
    "mleko": 130,
    "sok": 150
}

for proizvod, cena in cene.items():
    print(proizvod, cena)
```

Dobija se ispis svakog proizvoda i njegove cene:

```
hleb 80
mleko 130
sok 150
```

Možemo, na primer, pristupiti ceni konkretnog proizvoda i koristiti je u izračunavanju:

```
cene = {
    "hleb": 80,
    "mleko": 130,
    "sok": 150
}

kolicina = 3
ukupno = cene["mleko"] * kolicina

print("Ukupna cena je", ukupno)
```

Pošto jedno mleko košta 130, za tri komada dobija se:

```
Ukupna cena je 390
```

Liste, torke, skupovi i rečnici omogućavaju različite načine organizovanja više podataka.

Liste i torke čuvaju elemente određenim redosledom, skupovi su pogodni kada su važne jedinstvene vrednosti, dok rečnici omogućavaju pristup podacima pomoću ključeva.

U dosadašnjim Python programima podatke smo uglavnom čuvali u promenljivama i kolekcijama, dok smo postupke nad njima izdvajali u naredbe i funkcije. Takva organizacija prirodno odgovara proceduralnom načinu programiranja. Python, međutim, podržava i objektno-orijentisanu paradigmu, u kojoj se podaci i postupci povezani sa njima organizuju pomoću objekata. U narednoj sekciji upoznaćemo osnovne pojmove tog pristupa i način njihovog korišćenja u Pythonu.

5 Objektно-orijentisano programiranje u Pythonu

5.1 Klase i objekti

U dosadašnjim programima podatke smo čuvali u promenljivama i kolekcijama, dok smo postupke nad njima zapisivali pomoću naredbi i funkcija. Objektно-orijentisana paradigma nudi drugačiji način organizovanja programa, zasnovan na objektima.

Objekat predstavlja celinu koja može da sadrži određene podatke i da obavlja postupke povezane sa tim podacima. Na primer, u programu za evidenciju studenata jedan objekat može predstavljati konkretnog studenta. Sa njim mogu biti povezani podaci kao što su ime, broj indeksa i prosečna ocena, kao i postupci koji se nad tim podacima izvršavaju.

Objekti iste vrste obično imaju sličnu strukturu i ponašanje. Njihove zajedničke osobine opisuju se pomoću klase. Klasa zato predstavlja opis na osnovu koga se mogu formirati pojedinačni objekti.

Na primer, možemo imati klasu `Student`, dok konkretni studenti Ana i Marko predstavljaju dva različita objekta te klase. Oni pripadaju istoj vrsti objekata, ali mogu imati različite podatke.

U Pythonu se nova klasa definiše pomoću ključne reči `class`. Najjednostavnija klasa može izgledati ovako:

```
class Student:
    pass
```

Ovim kodom definisana je klasa `Student`. Posle njenog imena navodi se dvotačka, a sadržaj klase piše se uvučeno. Naredba `pass` ne obavlja nikakav postupak. Ona se može koristiti kada je određeni deo programa sintaksno potreban, ali za sada ne želimo da u njemu navedemo druge naredbe.

Na osnovu klase možemo napraviti objekat:

```
class Student:
    pass

student = Student()
```

Izraz `Student()` pravi novi objekat klase `Student`, a promenljiva `student` zatim se koristi za pristup tom objektu.

Na osnovu iste klase može se napraviti više objekata:

```
class Student:
    pass

student1 = Student()
student2 = Student()
```

Promenljive `student1` i `student2` odnose se na dva različita objekta, iako su oba napravljena na osnovu iste klase `Student`. Svaki od njih kasnije može sadržati sopstvene podatke.

Za objekat napravljen na osnovu određene klase kaže se i da je instanca te klase. Zato su objekti `student1` i `student2` dve instance klase `Student`.

Klase možemo koristiti za predstavljanje različitih vrsta objekata iz problema koji rešavamo. Na primer, možemo definisati klase `Student`, `Proizvod` ili `Pravougaonik`, a zatim napraviti više konkretnih objekata svake od njih.

Imena klasa se u Pythonu uobičajeno zapisuju tako da svaka reč počinje velikim slovom. Zato se, na primer, koriste imena `Student`, `Proizvod` ili `BankovniRacun`, za razliku od imena promenljivih i funkcija kod kojih smo reči razdvajali donjom crtom.

Sama prazna klasa još ne opisuje mnogo o svojim objektima. Da bi objekti mogli da čuvaju konkretne podatke i izvršavaju odgovarajuće postupke, klasi se dodaju atributi i metode. Njih, zajedno sa načinom postavljanja početnih vrednosti objekta, razmotrićemo u narednoj podsekciji.

5.2 Atributi, metode i konstruktor

Prazna klasa određuje vrstu objekata koje možemo napraviti, ali takvi objekti još ne sadrže podatke koji ih međusobno razlikuju. Podaci koji pripadaju konkretnom objektu nazivaju se atributi.

Početne vrednosti atributa najčešće se postavljaju pomoću posebne metode `__init__`. Ona se automatski poziva nakon formiranja novog objekta i služi za postavljanje njegovog početnog stanja. Metoda `__init__` često se naziva konstruktorom, pa ćemo taj naziv koristiti i u nastavku, iako se, strogo govoreći, sam objekat formira pre njenog poziva.

Na primer, klasu `Student` možemo definisati ovako:

```
class Student:
    def __init__(self, ime, prosek):
        self.ime = ime
        self.prosek = prosek
```

Metoda `__init__` u ovom primeru prima podatke o imenu i prosečnoj oceni studenta. Izrazi `self.ime` i `self.prosek` predstavljaju attribute objekta.

Parametar `self` označava konkretan objekat nad kojim se metoda izvršava. Zahvaljujući njemu, svakom objektu možemo pridružiti njegove sopstvene vrednosti atributa.

Sada možemo napraviti dva različita objekta:

```
student1 = Student("Ana", 9.2)
student2 = Student("Marko", 7.8)
```

Pri prvom pozivu konstruktora parametri `ime` i `prosek` dobijaju vrednosti "Ana" i 9.2. Te vrednosti se zatim čuvaju u atributima objekta `student1`. Pri drugom pozivu na isti način se formira objekat `student2` sa drugačijim podacima.

Parametar `self` ne navodimo posebno pri pravljenju objekta. Python ga automatski povezuje sa objektom koji se trenutno formira.

Atributima konkretnog objekta pristupa se pomoću tačke:

```
print(student1.ime)
print(student1.prosek)

print(student2.ime)
print(student2.prosek)
```

Dobija se:

```
Ana
9.2
Marko
```

7.8

Iako su oba objekta napravljena na osnovu iste klase, svaki od njih čuva sopstvene vrednosti atributa.

Vrednost atributa može se i promeniti:

```
student1.prosek = 9.5

print(student1.prosek)
print(student2.prosek)
```

Rezultat je:

```
9.5
7.8
```

Promenjena je samo vrednost atributa objekta `student1`. Objekat `student2` i dalje čuva svoju prethodnu vrednost.

Pored podataka, klasa može opisivati i postupke koje njeni objekti mogu da izvršavaju. Funkcije definisane unutar klase nazivaju se metode.

Na primer, klasi `Student` možemo dodati metodu za prikazivanje podataka:

```
class Student:
    def __init__(self, ime, prosek):
        self.ime = ime
        self.prosek = prosek

    def prikazi_podatke(self):
        print("Ime:", self.ime)
        print("Prosek:", self.prosek)
```

Metoda `prikazi_podatke` pripada klasi `Student`. Kao prvi parametar ima `self`, pa može da pristupi atributima konkretnog objekta.

Metoda se poziva pomoću objekta i njenog imena:

```
student = Student("Ana", 9.2)

student.prikazi_podatke()
```

Dobija se:

```
Ime: Ana
Prosek: 9.2
```

Pri pozivu `student.prikazi_podatke()` parametar `self` odnosi se na objekat `student`. Zato izrazi `self.ime` i `self.prosek` predstavljaju upravo njegove podatke.

Metoda može i da izračuna i vrati neku vrednost. Na primer, možemo odrediti da li je prosečna ocena studenta najmanje 8.5:

```
class Student:
    def __init__(self, ime, prosek):
        self.ime = ime
```

```

        self.prosek = prosek

    def uspesan(self):
        return self.prosek >= 8.5

student = Student("Ana", 9.2)

print(student.uspesan())

```

Pošto je prosečna ocena objekta `student` jednaka 9.2, metoda vraća logičku vrednost `True`.

Metode mogu imati i dodatne parametre. Na primer, možemo definisati metodu kojom se menja prosečna ocena:

```

class Student:
    def __init__(self, ime, prosek):
        self.ime = ime
        self.prosek = prosek

    def promeni_prosek(self, novi_prosek):
        self.prosek = novi_prosek

student = Student("Ana", 9.2)

student.promeni_prosek(9.6)

print(student.prosek)

```

Argument 9.6 prosleđuje se parametru `novi_prosek`, dok se `self` automatski odnosi na objekat `student`. Metoda zatim menja njegov atribut `prosek`, pa se prikazuje vrednost 9.6.

Klasa može objediniti više atributa i metoda koje zajedno opisuju određenu vrstu objekata. Na primer:

```

class Student:
    def __init__(self, ime, broj_indeksa, prosek):
        self.ime = ime
        self.broj_indeksa = broj_indeksa
        self.prosek = prosek

    def prikazi_podatke(self):
        print("Ime:", self.ime)
        print("Broj indeksa:", self.broj_indeksa)
        print("Prosek:", self.prosek)

    def uspesan(self):
        return self.prosek >= 8.5

    def promeni_prosek(self, novi_prosek):
        self.prosek = novi_prosek

student = Student("Ana", "123/2025", 9.2)

student.prikazi_podatke()

```

```
print("Uspesan:", student.uspesan())

student.promeni_prosek(9.6)
print("Novi prosek:", student.prosek)
```

Izvršavanjem programa dobija se:

```
Ime: Ana
Broj indeksa: 123/2025
Prosek: 9.2
Uspesan: True
Novi prosek: 9.6
```

U ovom primeru objekat objedinjuje svoje podatke, kao što su ime, broj indeksa i prosek, sa postupcima koji se nad tim podacima izvršavaju. Atributi opisuju stanje objekta, dok metode određuju njegovo ponašanje. Upravo povezivanje podataka i postupaka nad njima predstavlja jednu od osnovnih ideja objektno-orijentisanog programiranja.

5.3 Nasleđivanje

Pri definisanju više klasa može se dogoditi da one imaju zajedničke osobine i ponašanje. Umesto da isti kôd ponavljamo u svakoj klasi, možemo definisati jednu opštiju klasu, a zatim na njenoj osnovi praviti posebne klase. Takav odnos naziva se nasleđivanje.

Klasa čije se osobine nasleđuju naziva se bazna klasa, dok se klasa koja ih nasleđuje naziva izvedena klasa.

Na primer, možemo definisati klasu `Osoba`:

```
class Osoba:
    def __init__(self, ime):
        self.ime = ime

    def predstavi_se(self):
        print("Ja sam", self.ime)
```

Ako želimo da definišemo klasu `Student` kao posebnu vrstu osobe, možemo napisati:

```
class Student(Osoba):
    pass
```

Navođenjem klase `Osoba` između zagrada označava se da klasa `Student` nasleđuje njene osobine. Zato objekat klase `Student` može koristiti konstruktor i metodu definisane u njoj:

```
student = Student("Ana")

print(student.ime)
student.predstavi_se()
```

Dobija se:

```
Ana
Ja sam Ana
```

Iako u klasi `Student` nismo posebno definisali atribut `ime` niti metodu `predstavi_se`, ona ih nasleđuje od klase `Osoba`.

Izvedena klasa može da sadrži i sopstvene metode. Na primer:

```
class Osoba:
    def __init__(self, ime):
        self.ime = ime

    def predstavi_se(self):
        print("Ja sam", self.ime)

class Student(Osoba):
    def uci(self):
        print(self.ime, "uci.")
```

Sada objekat klase `Student` može koristiti i nasleđenu i novu metodu:

```
student = Student("Ana")
student.predstavi_se()
student.uci()
```

Dobija se:

```
Ja sam Ana
Ana uci.
```

Izvedena klasa često treba da čuva i dodatne podatke. Tada može imati sopstveni konstruktor. Ako želimo da iskoristimo konstruktor bazne klase, možemo ga pozvati pomoću funkcije `super`:

```
class Osoba:
    def __init__(self, ime):
        self.ime = ime

class Student(Osoba):
    def __init__(self, ime, broj_indeksa):
        super().__init__(ime)
        self.broj_indeksa = broj_indeksa
```

Naredba `super().__init__(ime)` poziva konstruktor bazne klase `Osoba`, koji postavlja atribut `ime`. Nakon toga konstruktor klase `Student` dodaje atribut `broj_indeksa`.

Objekat se zatim može napraviti na uobičajen način:

```
student = Student("Ana", "123/2025")
print(student.ime)
print(student.broj_indeksa)
```

Dobija se:

```
Ana
123/2025
```

Izvedena klasa može i da promeni ponašanje metode koju je nasledila. Na primer:

```

class Osoba:
    def __init__(self, ime):
        self.ime = ime

    def predstavi_se(self):
        print("Ja sam", self.ime)

class Student(Osoba):
    def __init__(self, ime, broj_indeksa):
        super().__init__(ime)
        self.broj_indeksa = broj_indeksa

    def predstavi_se(self):
        print("Ja sam", self.ime)
        print("Broj indeksa:", self.broj_indeksa)

```

Sada poziv metode zavisi od objekta nad kojim se ona izvršava:

```

osoba = Osoba("Marko")
student = Student("Ana", "123/2025")

osoba.predstavi_se()
student.predstavi_se()

```

Dobija se:

```

Ja sam Marko
Ja sam Ana
Broj indeksa: 123/2025

```

Klasa `Student` nasleđuje klasu `Osoba`, ali daje sopstvenu definiciju metode `predstavi_se`. To omogućava da objekti srodnih klasa na isti poziv metode odgovore na način koji odgovara njihovoj konkretnoj klasi.

Nasleđivanje tako omogućava da se zajednički podaci i postupci izdvoje u opštiju klasu, dok se izvedene klase mogu dopunjavati ili prilagođavati konkretnim potrebama. Zajedno sa klasama, objektima, atributima i metodama, ono predstavlja jednu od važnih mogućnosti objektno-orijentisanog načina organizovanja programa.

5.4 Polimorfizam

U prethodnoj podsekciji videli smo da izvedena klasa može da redefiniše metodu koju je nasledila od bazne klase. To nas dovodi do još jednog važnog pojma objektno-orijentisanog programiranja, koji se naziva polimorfizam.

Polimorfizam omogućava da isti poziv metode dovede do različitog ponašanja u zavisnosti od konkretnog objekta nad kojim se metoda izvršava. Jedan čest oblik polimorfizma javlja se kada srodne klase imaju metode istog imena, ali ih definišu na različite načine.

Na primer, možemo definisati baznu klasu `Zivotinja`:

```

class Zivotinja:
    def __init__(self, ime):

```

```

        self.ime = ime

    def oglasi_se(self):
        print(self.ime, "se oglasava.")

```

Zatim možemo definisati klase `Pas` i `Macka` koje nasleđuju klasu `Zivotinja`, ali na različite načine definišu metodu `oglas_i_se`:

```

class Pas(Zivotinja):
    def oglasi_se(self):
        print(self.ime, "kaze av.")

class Macka(Zivotinja):
    def oglasi_se(self):
        print(self.ime, "kaze mjau.")

```

Sada možemo napraviti objekte obe izvedene klase i pozvati istoimenu metodu:

```

pas = Pas("Lesi")
macka = Macka("Maza")

pas.oglas_i_se()
macka.oglas_i_se()

```

Dobija se:

```

Lesi kaze av.
Maza kaze mjau.

```

U oba slučaja poziva se metoda `oglas_i_se`, ali se izvršava ona definicija koja odgovara konkretnoj klasi objekta. Objekat klase `Pas` zato se ponaša drugačije od objekta klase `Macka`.

Prednost ovakvog pristupa posebno se vidi kada više različitih objekata obrađujemo na isti način. Na primer, objekte možemo smestiti u jednu listu:

```

zivotinje = [Pas("Lesi"), Macka("Maza"), Pas("Bobi")]

for zivotinja in zivotinje:
    zivotinja.oglas_i_se()

```

Izvršavanjem programa dobija se:

```

Lesi kaze av.
Maza kaze mjau.
Bobi kaze av.

```

Petlja u svakom prolasku izvršava isti poziv `zivotinja.oglas_i_se()`. Ipak, ponašanje zavisi od konkretnog objekta koji se u tom trenutku nalazi u promenljivoj `zivotinja`. Za objekat klase `Pas` izvršava se metoda definisana u klasi `Pas`, dok se za objekat klase `Macka` izvršava metoda definisana u klasi `Macka`.

Bez ovakvog pristupa bili bismo primorani da posebno proveravamo kojoj vrsti pripada svaki objekat i da za različite vrste pišemo različite naredbe. Polimorfizam omogućava da različite objekte koristimo na jedinstven način, dok svaka klasa određuje sopstveno ponašanje

odgovarajuće metode.

Nasleđivanje i polimorfizam zato se prirodno dopunjuju. Nasleđivanje omogućava da srodne klase dele zajedničke osobine i ponašanje, dok polimorfizam omogućava da njihove zajedničke operacije imaju različito ponašanje u zavisnosti od konkretnog objekta.

Python smo do sada koristili i proceduralno, kroz naredbe, petlje i funkcije, i objektno-orijentisano, kroz klase i objekte. U okviru objektno-orijentisanog pristupa upoznali smo klase, objekte, attribute, metode, nasleđivanje i polimorfizam. Time smo na istom jeziku videli kako se program može organizovati prema različitim programskim paradigmama. U nastavku prelazimo na Haskell, jezik u kome ćemo upoznati drugačiji način razmišljanja o programima, zasnovan prvenstveno na funkcionalnoj paradigmi.

6 Uvod u Haskell

6.1 Osnovne osobine Haskell

Haskell je programski jezik opšte namene koji je posebno vezan za funkcionalnu paradigmu. Za razliku od Pythona, koji smo koristili i proceduralno i objektno-orijentisano, Haskell ćemo koristiti prvenstveno da bismo upoznali funkcionalni način organizovanja programa.

Razvoj Haskell započet je krajem osamdesetih godina XX veka sa ciljem da se postojeće ideje funkcionalnog programiranja objedine u jedan programski jezik. Od tada je on postao jedan od najpoznatijih predstavnika funkcionalne paradigme.

Programi u Haskellu u velikoj meri se grade pomoću definicija vrednosti i funkcija. Na primer, funkciju koja udvostručuje zadati broj možemo definisati ovako:

```
udvostruci broj = broj + broj
```

Ime funkcije je `udvostruci`, dok `broj` predstavlja njen argument. Izraz sa desne strane znaka `=` određuje vrednost koju funkcija vraća.

Nova funkcija može koristiti već definisanu funkciju. Na primer:

```
udvostruci broj = broj + broj
```

```
ucetvorostruci broj = udvostruci (udvostruci broj)
```

Funkcija `ucetvorostruci` dva puta primenjuje funkciju `udvostruci`, pa tako vraća četvorostruku vrednost svog argumenta.

Već se na ovom jednostavnom primeru vidi da se zapis funkcija razlikuje od Pythona. Ne koriste se ključna reč `def`, zagrade oko parametara ni naredba `return`. Definicija se zapisuje kao jednakost između primene funkcije i izraza koji određuje njen rezultat. Ovakav zapis detaljnije ćemo upoznati u narednim sekcijama.

Jedna od važnih osobina Haskell jeste statički sistem tipova. Svaki izraz ima određen tip, a sistem tipova proverava da li se vrednosti i operacije koriste na odgovarajući način. Mnoge greške zato mogu biti otkrivene pre izvršavanja programa.

Pri tome tipove često nije potrebno eksplicitno navoditi. Haskell u mnogim situacijama može sam da zaključi tip izraza ili funkcije na osnovu načina na koji su definisani. O tipovima i načinu njihovog zapisivanja detaljnije ćemo govoriti u narednoj sekciji.

Važna razlika u odnosu na dosadašnje Python programe jeste i način rada sa vrednostima. U Pythonu smo često menjali vrednost neke promenljive tokom izvršavanja programa. U Haskellu se imena vezuju za vrednosti koje se zatim ne menjaju na takav način. Zbog toga se izračunavanje mnogo više zasniva na građenju i izračunavanju izraza nego na nizu naredbi koje menjaju stanje programa.

Haskell koristi i lenjo izračunavanje. To znači da se vrednost nekog izraza ne mora izračunati odmah, već se izračunavanje odlaže dok ta vrednost zaista ne postane potrebna. Ova osobina će kasnije omogućiti neke karakteristične načine rada sa funkcijama i listama.

Haskell nam zato pruža drugačiji pogled na organizovanje programa od onog koji smo prethodno upoznali u Pythonu. Pre nego što pređemo na konkretnu sintaksu jezika, detaljnije ćemo razmotriti kako izgleda samo izračunavanje u funkcionalnom pristupu.

6.2 Funkcionalni način izračunavanja

U proceduralnim Python programima često smo opisivali niz koraka koji se izvršavaju redom. Pri tome su se vrednosti promenljivih mogle menjati tokom izvršavanja. U funkcionalnom pristupu pažnja je više usmerena na izraze i funkcije kojima se određuje rezultat izračunavanja.

Na primer, definišimo dve jednostavne funkcije:

```
povecaj broj = broj + 1
kvadriraj broj = broj * broj
```

Ako želimo da broj 3 najpre povećamo za 1, a zatim kvadriramo, možemo napisati:

```
kvadriraj (povecaj 3)
```

Vrednost ovog izraza može se odrediti postepenim pojednostavljivanjem:

```
kvadriraj (povecaj 3)
= kvadriraj 4
= 4 * 4
= 16
```

Vrednost ovog izraza može se odrediti postepenim pojednostavljivanjem. Izraz `povecaj 3` ima vrednost 4, pa se zatim dobija primena funkcije `kvadriraj` na tu vrednost i konačan rezultat 16.

Ovakav način posmatranja programa važan je u funkcionalnom programiranju. Umesto da pratimo kako se stanje programa menja kroz niz naredbi, pratimo kako se složeniji izraz svodi na jednostavnije izraze dok se ne dobije njegova vrednost.

Znak `=` u Haskellu takođe ima drugačiju ulogu od dodele vrednosti promenljivoj u Pythonu. U Pythonu smo, na primer, mogli napisati:

```
broj = 5
broj = broj + 1
```

Vrednost promenljive `broj` najpre je 5, a zatim se menja u 6.

U Haskellu zapis

```
broj = 5
```

predstavlja definiciju, odnosno vezivanje imena `broj` za vrednost 5. U običnim definicijama koje ćemo koristiti to ime se zatim ne menja tako što mu se dodeljuje nova vrednost.

Ako nam je potrebna druga vrednost, možemo definisati novo ime:

```
broj = 5
sledeciBroj = broj + 1
```

Ovde ime `broj` i dalje označava vrednost 5, dok ime `sledeciBroj` označava vrednost 6. Nije bilo potrebno menjati prethodnu vrednost da bi se dobio novi rezultat.

Funkcionalno programiranje se u velikoj meri oslanja na čiste funkcije. Njihov rezultat zavisi samo od njihovih argumenata, a njihovo izvršavanje ne izaziva sporedne efekte. Ako čistu funkciju više puta primenimo na iste argumente, dobićemo isti rezultat.

Na primer:

```
kvadriraj broj = broj * broj
```

Za argument 5 rezultat funkcije `kvadriraj` uvek je 25. Funkcija pri tome ne menja neku spoljašnju promenljivu niti zavisi od prethodnog poziva.

Zbog toga se pri razumevanju izraza primena funkcije može zameniti njenom vrednošću. Na primer:

```
povecaj 5
= 5 + 1
= 6
```

Isto važi i kada se funkcije kombinuju:

```
kvadriraj (povecaj 5)
= kvadriraj 6
= 6 * 6
= 36
```

Na taj način složeniji programi mogu se graditi kombinovanjem jednostavnijih funkcija i izraza. Umesto promene stanja korak po korak, rezultat se opisuje preko vrednosti drugih izraza.

To ne znači da Haskell ne može da komunicira sa korisnikom ili spoljnim svetom, već da su čista izračunavanja jasno odvojena od takvih postupaka. U nastavku ćemo se baviti prvenstveno čistim funkcijama, jer one najbolje pokazuju osnovne ideje funkcionalne paradigme.

Da bismo ovakve izraze mogli neposredno da isprobavamo i proveravamo njihove vrednosti, u narednoj podsekciji upoznaćemo interaktivno okruženje `ghci`.

6.3 Rad u interaktivnom okruženju

Za rad sa programskim jezikom Haskell najčešće se koristi Glasgow Haskell Compiler (GHC). GHC obuhvata kompajler `ghc`, kojim se Haskell programi mogu prevoditi, kao i interaktivno okruženje `ghci`, namenjeno neposrednom unosu i izračunavanju izraza.

Za instalaciju potrebnih alata može se koristiti `GHCup`¹. Pomoću njega mogu se instalirati GHC i prateći alati potrebni za rad sa Haskellom. Za početno upoznavanje sa jezikom korišćićemo prvenstveno interaktivno okruženje `ghci`.

Okruženje `ghci` pokreće se iz terminala komandom

```
ghci
```

Nakon pokretanja pojavljuje se prompt `ghci>`, koji označava da okruženje čeka unos izraza. Izlazak iz okruženja može se izvršiti komandom `:quit`, odnosno njenim kraćim oblikom `:q`.

U interaktivnom okruženju možemo neposredno izračunavati aritmetičke izraze:

```
ghci> 2 + 3
5

ghci> 5 * 10
50
```

¹<https://www.haskell.org/ghcup/>

```
ghci> 10 - 4
6
```

```
ghci> 5 / 2
2.5
```

Kao i u matematici, različiti operatori imaju različite prioritete. Množenje i deljenje, na primer, imaju viši prioritet od sabiranja i oduzimanja:

```
ghci> 2 + 3 * 4
14
```

Ako želimo drugačiji redosled izračunavanja, koristimo zagrade:

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

Haskell podržava i logičke vrednosti `True` i `False`. Osnovne logičke operacije mogu se isprobati neposredno u interaktivnom okruženju:

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

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

```
ghci> not True
False
```

Operator `&&` predstavlja logičko i, operator `||` logičko ili, dok funkcija `not` predstavlja negaciju.

Vrednosti se mogu i porediti. Operator `==` proverava jednakost, dok operator `/=` proverava nejednakost. Koriste se i uobičajeni operatori `<`, `<=`, `>` i `>=`:

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

```
ghci> 5 /= 3
True
```

```
ghci> 10 > 4
True
```

```
ghci> 3 <= 2
False
```

Pojedinačni karakteri zapisuju se između jednostrukih navodnika, dok se tekstualne vrednosti zapisuju između dvostrukih navodnika:

```
ghci> 'a'
'a'
```

```
ghci> "Zdravo"  
"Zdravo"  
  
ghci> "Ana" == "Ana"  
True
```

U Haskellu se funkcije pozivaju drugačije nego u Pythonu. Ime funkcije i njeni argumenti navode se jedan za drugim, razdvojeni razmacima. Zgrade oko argumenata se ne koriste.

Na primer, funkcija `abs` vraća apsolutnu vrednost broja:

```
ghci> abs (-7)  
7
```

Ovde su zgrade potrebne zato što je argument negativan broj. Znak `-` se u ovom slučaju tumači kao negacija, pa se negativan argument funkcije zapisuje u zagradi.

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

```
ghci> min 8 5  
5  
  
ghci> max 8 5  
8
```

Primena funkcije ima viši prioritet od aritmetičkih operatora. Zato se izraz

```
ghci> abs (-5) + max 3 7  
12
```

tumači kao

```
ghci> (abs (-5)) + (max 3 7)  
12
```

Neke funkcije sa dva argumenta mogu se zapisati i između svojih argumenata. Tada se ime funkcije okružuje obrnutim apostrofima. Na primer, funkcija `div` vrši celobrojno deljenje:

```
ghci> div 10 3  
3  
  
ghci> 10 `div` 3  
3
```

Oba zapisa imaju isto značenje. Drugi oblik omogućava da se obična funkcija sa dva argumenta koristi slično operatoru.

U interaktivnom okruženju možemo privremeno definisati i sopstvene vrednosti. Za to možemo koristiti ključnu reč `let`:

```
ghci> let broj = 5  
ghci> broj  
5
```

```
ghci> let sledeciBroj = broj + 1
ghci> sledeciBroj
6
```

Na isti način možemo definisati i jednostavnu funkciju:

```
ghci> let kvadriraj broj = broj * broj
ghci> kvadriraj 6
36
```

Ovakve definicije pogodne su za brzo isprobavanje ideja. One postoje samo tokom rada u trenutnoj sesiji interaktivnog okruženja i nisu same po sebi sačuvane kao program.

Interaktivno okruženje `ghci` zato je pogodno za isprobavanje izraza, funkcija i kratkih delova koda. Kada želimo da definicije sačuvamo, organizujemo u duži program i kasnije ponovo koristimo, Haskell kôd zapisujemo u datotekama. Način rada sa njima razmotrićemo u narednoj podsekciji.

6.4 Rad sa datotekama

Interaktivno okruženje `ghci` pogodno je za brzo isprobavanje izraza i kraćih definicija. Kada želimo da napisani kôd sačuvamo i kasnije ponovo koristimo, Haskell definicije zapisujemo u tekstualnim datotekama sa ekstenzijom `.hs`.

Za pisanje Haskell koda može se koristiti bilo koji tekstualni editor. Jedan od pogodnih izbora je Visual Studio Code. U njemu se može instalirati dodatak `Haskell`, koji pruža podršku za rad sa Haskell kodom, kao što su označavanje sintakse, prikazivanje grešaka i dopunjavanje koda.²

Za rad ovog dodatka koristi se Haskell Language Server (HLS). Alati potrebni za rad sa Haskellom mogu se instalirati pomoću programa `GHCup`, koji smo pomenuli u prethodnoj podsekciji. Za jednostavne primere koje ćemo koristiti dovoljno je da možemo da pokrenemo `GHC` i interaktivno okruženje `ghci`.

Napravimo, na primer, datoteku `primer.hs` sa sledećim sadržajem:

```
broj = 5

kvadriraj broj = broj * broj

povecaj broj = broj + 1
```

Datoteka sadrži jednu definiciju vrednosti i dve definicije funkcija. U ovoj fazi nije potrebno da Haskell datoteka predstavlja kompletan samostalni program. Koristićemo je prvenstveno za čuvanje definicija koje zatim možemo učitati i isprobavati u interaktivnom okruženju.

Datoteku možemo učitati u `ghci` pomoću komande `:load`, odnosno njenog kraćeg oblika `:l`:

```
ghci> :load primer.hs
Ok, one module loaded.
```

Nakon uspešnog učitavanja, definicije iz datoteke postaju dostupne u interaktivnom okruženju:

²<https://marketplace.visualstudio.com/items?itemName=haskell.haskell>

```
ghci> broj
5

ghci> kvadriraj 6
36

ghci> povecaj 10
11
```

Na ovaj način kôd možemo pisati i čuvati u editoru, a zatim njegove definicije neposredno isprobavati u `ghci`.

Ako datoteku izmenimo i ponovo sačuvamo, nije potrebno ponovo navoditi njeno ime. Dovoljno je koristiti komandu `:reload`, odnosno njen kraći oblik `:r`:

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

Na primer, ako u datoteku dodamo funkciju

```
utrostruci broj = 3 * broj
```

i zatim izvršimo komandu `:reload`, nova funkcija postaje dostupna:

```
ghci> utrostruci 4
12
```

Pri pisanju Haskell koda važno je voditi računa o uvlačenju. Delovi definicija koji pripadaju istom nivou treba da budu pravilno poravnati. Ovo će postati posebno važno kada budemo koristili složenije definicije koje se prostiru u više redova.

Imena vrednosti i funkcija u Haskellu uobičajeno počinju malim slovom. Kada se ime sastoji od više reči, obično se koristi zapis u kome svaka naredna reč počinje velikim slovom. Na primer:

```
brojStudenata = 25

izracunajProsek zbir broj = zbir / broj
```

U primerima ćemo koristiti srpske nazive zapisane bez dijakritičkih znakova, kako bi kôd ostao jednostavan za unos. Tako ćemo, na primer, koristiti imena `brojStudenata`, `sledeciBroj` i `izracunajProsek`.

Haskell podržava i komentare. Jednolinijski komentar počinje sa `--` i traje do kraja reda:

```
-- Kvadrira zadati broj
kvadriraj broj = broj * broj
```

Komentar koji zauzima više redova zapisuje se između oznaka `{-` i `-}`:

```
{-
Ova funkcija udvostrucuje broj.
-}
udvostruci broj = broj + broj
```

Za naše potrebe najčešći način rada biće jednostavan. Haskell kôd pisaćemo i čuvati u datotekama sa ekstenzijom `.hs`, a njihove definicije učitavati i isprobavati u okruženju `ghci`. Time smo upoznali osnovno okruženje za rad sa Haskellom, pa sada možemo detaljnije razmotriti tipove i način definisanja funkcija u ovom jeziku.

7 Tipovi i funkcije u Haskellu

7.1 Osnovni tipovi i zaključivanje tipova

U prethodnoj sekciji pomenuli smo da svaki izraz u Haskellu ima određen tip. Tip određuje kakvu vrstu vrednosti izraz predstavlja i koje operacije nad tom vrednošću mogu da se izvršavaju.

Ako izraz `izraz` ima tip `Tip`, to se u Haskellu zapisuje u obliku

```
izraz :: Tip
```

pri čemu se simbol `::` može čitati kao „ima tip”. Na primer, logička vrednost `True` ima tip `Bool`, što možemo zapisati kao

```
True :: Bool
```

Tip nekog izraza možemo proveriti i u interaktivnom okruženju pomoću komande `:type`, odnosno njenog kraćeg oblika `:t`:

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

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

Haskell ima statički sistem tipova. To znači da se ispravnost tipova proverava pre nego što se izraz izvrši. Ako pokušamo da primenimo neku operaciju na vrednost neodgovarajućeg tipa, Haskell će prijaviti tipsku grešku.

Na primer, izraz

```
True + 1
```

nije ispravan, jer se logička vrednost `True` ne može koristiti kao broj nad kojim se izvršava sabiranje.

Iako svaki izraz ima tip, u Haskellu najčešće nije potrebno da ga sami navodimo. Haskell može da zaključi tip na osnovu vrednosti i operacija koje se u izrazu koriste. Ova osobina naziva se zaključivanje tipova.

Na primer, ako u interaktivnom okruženju definišemo

```
ghci> let uspesan = True
```

Haskell može sam da zaključi tip ove vrednosti:

```
ghci> :t uspesan
uspesan :: Bool
```

Slično tome:

```
ghci> let slovo = 'A'
ghci> :t slovo
slovo :: Char

ghci> let pozdrav = "Zdravo"
```

```
ghci> :t pozdrav
pozdrav :: String
```

Na taj način sistem tipova omogućava proveru ispravnosti programa, a zaključivanje tipova istovremeno smanjuje potrebu da tip navodimo uz svaku vrednost.

Haskell poseduje više osnovnih tipova. Tip `Bool` predstavlja logičke vrednosti `True` i `False`. Nad njima smo već koristili logičke operacije `&&`, `||` i `not`:

```
ghci> True && False
False

ghci> not False
True
```

Za predstavljanje celih brojeva često se koriste tipovi `Int` i `Integer`. Tip `Int` predstavlja cele brojeve iz ograničenog opsega, dok tip `Integer` može predstavljati proizvoljno velike cele brojeve, uz praktično ograničenje raspoložive memorije.

Tip neke vrednosti možemo i eksplicitno zadati. Na primer:

```
brojStudenata :: Int
brojStudenata = 25

velikiBroj :: Integer
velikiBroj = 12345678901234567890
```

U prvom primeru ime `brojStudenata` vezano je za vrednost tipa `Int`, dok je u drugom primeru vrednost `velikiBroj` tipa `Integer`.

Eksplicitno navođenje tipa nije korisno samo radi jasnijeg koda. U nekim situacijama raspoloživi kontekst nije dovoljan da se odredi jedan konkretan tip, pa može biti potrebno da željeni tip dodatno navedemo. Takve primere ćemo razmotriti kasnije, kada uvedemo polimorfne tipove i tipske klase.

Nad celim brojevima mogu se koristiti uobičajene aritmetičke operacije, kao što su sabiranje, oduzimanje i množenje:

```
ghci> 7 + 5
12

ghci> 7 - 5
2

ghci> 7 * 5
35
```

Za celobrojno deljenje mogu se koristiti funkcije `div` i `mod`. Funkcija `div` vraća celobrojni količnik, dok `mod` vraća ostatak pri deljenju:

```
ghci> 7 `div` 2
3

ghci> 7 `mod` 2
1
```

Za predstavljanje brojeva sa pokretnim zarezom koriste se tipovi `Float` i `Double`. Oba tipa predstavljaju realne brojeve približno, pri čemu `Double` obično omogućava veću preciznost i zato se često koristi kada je preciznost značajna.

Na primer:

```
prosek :: Double
prosek = 9.25

temperatura :: Float
temperatura = 23.5
```

Nad ovim vrednostima mogu se koristiti uobičajene aritmetičke operacije, uključujući deljenje:

```
ghci> 5.0 / 2.0
2.5
```

Tip `Char` predstavlja jedan karakter. Karakter se zapisuje između jednostrukih navodnika:

```
slovo :: Char
slovo = 'A'
```

Tip `String` koristi se za tekstualne vrednosti, koje se zapisuju između dvostrukih navodnika:

```
ime :: String
ime = "Ana"
```

Zato `'A'` i `"A"` nisu vrednosti istog tipa. Prva predstavlja jedan karakter tipa `Char`, dok druga predstavlja tekstualnu vrednost tipa `String`.

Kasnije ćemo videti da je `String` u Haskellu zapravo lista karaktera. Za sada je dovoljno da ga koristimo kao tip za tekstualne vrednosti.

Vrednosti mnogih tipova možemo porediti odgovarajućim operatorima. Na primer:

```
ghci> 5 < 8
True

ghci> 'a' == 'a'
True

ghci> "Ana" == "Ana"
True
```

Tipovi predstavljaju jedan od osnovnih mehanizama Haskell. Sistem tipova omogućava da se mnoge greške otkriju pre izvršavanja programa, dok zaključivanje tipova omogućava da tipove često ne moramo eksplicitno navoditi. Ipak, eksplicitno zapisivanje tipova naročito je korisno pri definisanju funkcija, čime ćemo se baviti u narednoj podsekciji.

7.2 Tipovi i definisanje funkcija

Kao i druge vrednosti, i funkcije u Haskellu imaju svoje tipove. Tip funkcije određuje kakvu vrednost funkcija prima kao argument i kakvu vrednost vraća kao rezultat.

Ako funkcija prima argument tipa `A` i vraća rezultat tipa `B`, njen tip zapisuje se kao

```
A -> B
```

pri čemu strelica `->` razdvaja tip argumenta od tipa rezultata.

Na primer, funkcija koja prima ceo broj i vraća ceo broj može imati tip

```
Int -> Int
```

Jednu takvu funkciju možemo definisati ovako:

```
kvadriraj :: Int -> Int
kvadriraj broj = broj * broj
```

Prvi red predstavlja tipski potpis funkcije. Zapis `kvadriraj :: Int -> Int` govori da funkcija `kvadriraj` prima vrednost tipa `Int` i vraća vrednost istog tipa.

Drugi red predstavlja samu definiciju funkcije. Ime `broj` predstavlja parametar funkcije, dok izraz `broj * broj` određuje njen rezultat.

Funkciju zatim možemo primeniti na konkretnu vrednost:

```
ghci> kvadriraj 5
25
```

Pri ovom pozivu vrednost `5` predstavlja argument funkcije. Parametar `broj` u definiciji funkcije dobija tu vrednost, pa se izračunava izraz `5 * 5`.

Tip funkcije možemo proveriti pomoću komande `:t`:

```
ghci> :t kvadriraj
kvadriraj :: Int -> Int
```

Tip rezultata ne mora biti isti kao tip argumenta. Na primer, funkcija može primiti ceo broj i vratiti logičku vrednost:

```
jePozitivan :: Int -> Bool
jePozitivan broj = broj > 0
```

Njen tip pokazuje da funkcija prima vrednost tipa `Int`, a vraća vrednost tipa `Bool`:

```
ghci> jePozitivan 7
True

ghci> jePozitivan (-3)
False
```

Slično tome, funkcija može primiti i vraćati logičke vrednosti:

```
negacija :: Bool -> Bool
negacija vrednost = not vrednost
```

Na primer:

```
ghci> negacija True
False
```

```
ghci> negacija False
True
```

Tipski potpis nije uvek neophodno napisati. Haskell često može da zaključi tip funkcije na osnovu njene definicije. Ako, na primer, definišemo funkciju bez tipskog potpisa

```
negacija vrednost = not vrednost
```

Haskell na osnovu funkcije `not` može da zaključi njen tip:

```
ghci> :t negacija
negacija :: Bool -> Bool
```

Ipak, pri pisanju funkcija u datotekama korisno je eksplicitno navoditi njihove tipove. Tipski potpis jasno pokazuje kakve argumente funkcija očekuje i kakav rezultat vraća, a može pomoći i u ranijem otkrivanju grešaka.

Sistem tipova proverava da li se funkcija primenjuje na argument odgovarajućeg tipa. Pošto funkcija `kvadriraj` ima tip `Int -> Int`, izraz

```
kvadriraj 5
```

je ispravan, dok izraz

```
kvadriraj True
```

nije ispravan. Funkcija očekuje argument tipa `Int`, dok je vrednost `True` tipa `Bool`, pa Haskell prijavljuje tipsku grešku.

Tipski potpis zato možemo posmatrati kao opis načina na koji se funkcija može koristiti. Na primer, već iz potpisa

```
jePozitivan :: Int -> Bool
```

vidimo da funkciji treba proslediti ceo broj i da će njen rezultat biti logička vrednost, čak i ako još nismo pogledali samu definiciju funkcije.

Do sada smo razmatrali funkcije koje primaju po jedan argument. U Haskellu se veoma često koriste i funkcije sa više argumenata, a način na koji se njihovi tipovi zapisuju predstavlja jednu od važnih osobina jezika. Njih ćemo razmotriti u narednoj podsekciji.

7.3 Funkcije sa više argumenata

U prethodnoj podsekciji razmatrali smo funkcije koje primaju jedan argument. U Haskellu se veoma često koriste i funkcije kojima pri pozivu prosleđujemo više argumenata.

Na primer, funkciju koja sabira dva cela broja možemo definisati ovako:

```
saberi :: Int -> Int -> Int
saberi prvi drugi = prvi + drugi
```

Funkcija se može pozvati navođenjem oba argumenta:

```
ghci> saberi 2 3
5
```

```
ghci> saberi 10 7
17
```

Tipski potpis

```
saberi :: Int -> Int -> Int
```

možemo za početak čitati tako da funkcija prima dva argumenta tipa `Int` i vraća rezultat tipa `Int`. Međutim, način na koji Haskell tumači ovaj zapis ima dodatno značenje koje je važno za funkcionalno programiranje.

Strelica -> u tipskim potpisima grupiše se zdesna nalevo. Zato se tip

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

tumači kao

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

To znači da se funkcija `saberi` može posmatrati kao funkcija koja najpre prima jedan argument tipa `Int`, a kao rezultat vraća novu funkciju tipa `Int -> Int`. Ta nova funkcija zatim prima drugi argument i vraća konačan rezultat.

Sa druge strane, primena funkcije na argumente grupiše se sleva nadesno. Zato se izraz

```
saberi 2 3
```

tumači kao

(saberi 2) 3

Najpre se, dakle, funkcija `saberi` primenjuje na argument 2. Dobija se nova funkcija koja očekuje još jedan ceo broj. Zatim se ta funkcija primenjuje na argument 3, čime se dobija rezultat 5.

Ovo možemo proveriti i u interaktivnom okruženju:

```
ghci> :t saberi
saber1 :: Int -> Int -> Int
```

```
ghci> :t saberi 2
saberi 2 :: Int -> Int
```

Izraz **saberi 2** zato nije greška niti nepotpun poziv. On predstavlja novu funkciju koja očekuje još jedan argument tipa **Int**. Na primer:

```
ghci> saberi 2 5
7
```

```
ghci> saberi 2 10
12
```

Ako neku funkciju primenimo na manji broj argumenata nego što može da primi, govorimo o delimičnoj primeni funkcije. Dobijeni rezultat je nova funkcija kojoj su neki argumenti već određeni.

Na primer, možemo definisati funkciju koja svakom broju dodaje 10:

```
dodajDeset :: Int -> Int
dodajDeset = saberi 10
```

Funkcija `saberi 10` već ima određen prvi argument i očekuje još samo jedan ceo broj. Zato funkcija `dodajDeset` ima tip `Int -> Int`.

Na primer:

```
ghci> dodajDeset 5
15

ghci> dodajDeset 20
30
```

Ovakav način predstavljanja funkcija sa više argumenata, pri kome funkcija prima argumente jedan po jedan, naziva se kariranje (engl. *currying*). Ova osobina Haskell-a kasnije će biti posebno korisna pri radu sa funkcijama višeg reda.

Isti princip važi i kada funkcija ima više od dva argumenta. Na primer, funkcija koja računa proizvod tri cela broja može se definisati ovako:

```
pomnoziTri :: Int -> Int -> Int -> Int
pomnoziTri prvi drugi treci = prvi * drugi * treci
```

Na primer:

```
ghci> pomnoziTri 2 3 4
24
```

Njen tip

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

tumači se kao

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

Funkcija zato prima prvi argument i vraća funkciju koja očekuje drugi argument, a ona zatim vraća funkciju koja očekuje treći argument. Tek nakon prosleđivanja sva tri argumenta dobija se konačan rezultat tipa `Int`.

Delimična primena moguća je i u ovom slučaju. Na primer:

```
ghci> :t pomnoziTri 2
pomnoziTri 2 :: Int -> Int -> Int

ghci> :t pomnoziTri 2 3
pomnoziTri 2 3 :: Int -> Int
```

Argumenti funkcije ne moraju svi imati isti tip. Na primer, možemo definisati funkciju koja na osnovu celog broja i logičke vrednosti vraća ceo broj:

```
izaberiBroj :: Int -> Bool -> Int
izaberiBroj broj True = broj
```

```
izaberiBroj broj False = 0
```

Njen tipski potpis pokazuje da je prvi argument tipa `Int`, drugi tipa `Bool`, a konačan rezultat tipa `Int`.

Na primer:

```
ghci> izaberiBroj 7 True
7

ghci> izaberiBroj 7 False
0
```

U ovom poslednjem primeru funkciju smo definisali različito u zavisnosti od vrednosti jednog argumenta. U narednim podsekcijama detaljnije ćemo upoznati načine na koje rezultat funkcije može zavisiti od uslova i oblika njenih argumenata.

7.4 Uslovni izrazi, ograđene i lokalne definicije

Rezultat funkcije često zavisi od nekog uslova. Najjednostavniji način da se u Haskellu izabere između dve vrednosti jeste uslovni izraz `if then else`.

Njegov opšti oblik je

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

Ako je `uslov` tačan, vrednost celog izraza određuje `izraz1`, a u suprotnom `izraz2`.

Na primer, funkciju koja vraća apsolutnu vrednost celog broja možemo definisati ovako:

```
apsolutna :: Int -> Int
apsolutna broj =
  if broj >= 0 then broj else -broj
```

Na primer:

```
ghci> apsolutna 7
7

ghci> apsolutna (-7)
7
```

Za razliku od uslovne naredbe koju smo koristili u Pythonu, `if then else` u Haskellu predstavlja izraz. To znači da on uvek ima neku vrednost. Zbog toga se grana `else` ne može izostaviti.

Obe moguće grane moraju dati vrednosti odgovarajućih tipova, tako da i ceo uslovni izraz ima jednoznačno određen tip. Na primer, sledeći izraz nije ispravan:

```
if True then 5 else "pet"
```

Jedna grana daje broj, a druga tekstualnu vrednost, pa nije moguće odrediti jedinstven tip celog izraza.

Uslovni izrazi mogu se i ugnježđavati. Na primer, funkcija koja određuje znak celog broja može se definisati ovako:

```
znak :: Int -> Int
znak broj =
  if broj < 0 then -1
  else if broj == 0 then 0
  else 1
```

Ako funkcija ima više mogućih slučajeva, ovakav zapis može postati nepregledan. Haskell zato omogućava i ograđene definicije (engl. *guards*), kod kojih se uz definiciju funkcije navodi više logičkih uslova.

Istu funkciju možemo zapisati preglednije:

```
znak :: Int -> Int
znak broj
  | broj < 0    = -1
  | broj == 0  = 0
  | otherwise  = 1
```

Svaka grana počinje znakom `|`, nakon koga se navodi uslov. Uslovi se proveravaju redom, a rezultat funkcije određuje prva grana čiji je uslov tačan.

Ime `otherwise` koristi se kada želimo da poslednja grana obuhvati sve preostale slučajeve. Ono je unapred definisano tako da ima logičku vrednost `True`.

Na primer:

```
ocena :: Int -> String
ocena poeni
  | poeni >= 90 = "Odlican"
  | poeni >= 75 = "Vrlo dobar"
  | poeni >= 60 = "Dobar"
  | poeni >= 50 = "Dovoljan"
  | otherwise  = "Nedovoljan"
```

Funkciju možemo isprobati u interaktivnom okruženju:

```
ghci> ocena 82
"Vrlo dobar"

ghci> ocena 45
"Nedovoljan"
```

Redosled uslova je važan. U prethodnom primeru najpre se proveravaju veće granice, jer se vrednost određuje čim se naiđe na prvi zadovoljen uslov.

Poželjno je da grane obuhvate sve moguće slučajeve. Ako nijedan uslov nije zadovoljen, funkcija za takav argument nema određenu vrednost i pri pokušaju njenog izračunavanja može doći do greške. Zbog toga se u mnogim definicijama poslednja grana završava sa `otherwise`.

Ograđene definicije mogu se koristiti i kod funkcija sa više argumenata. Na primer:

```
najveciOdTri :: Int -> Int -> Int -> Int

najveciOdTri prvi drugi treci
  | prvi >= drugi && prvi >= treci = prvi
  | drugi >= prvi && drugi >= treci = drugi
```

```
| otherwise = treci
```

Na primer:

```
ghci> najveciOdTri 4 9 6
9
```

Pri pisanju složenijih funkcija često se javlja potreba da neki pomoćni izraz izračunamo i imenujemo, kako ga ne bismo više puta ponavljali. Za to se mogu koristiti lokalne definicije.

Jedan način jeste konstrukcija **where**. Lokalne vrednosti navode se nakon glavne definicije funkcije i dostupne su unutar te definicije.

Na primer:

```
opisBroja :: Int -> String
opisBroja broj
  | apsolutnaVrednost == 0 = "Nula"
  | apsolutnaVrednost <= 9 = "Jednocifren"
  | otherwise = "Visecifren"
where
  apsolutnaVrednost = abs broj
```

Izraz `abs broj` izračunava se preko lokalnog imena `apsolutnaVrednost`, koje zatim možemo koristiti u svim granama definicije. Ovo je preglednije nego da isti izraz ponavljamo u svakom uslovu.

Pomoću **where** možemo definisati i više lokalnih vrednosti. Na primer:

```
polozaj :: Int -> String
polozaj broj
  | broj < donjaGranica = "Ispod"
  | broj > gornjaGranica = "Iznad"
  | otherwise = "Unutar"
where
  donjaGranica = 10
  gornjaGranica = 20
```

Lokalna imena `donjaGranica` i `gornjaGranica` koriste se samo u okviru ove definicije.

Drugi način lokalnog definisanja jeste konstrukcija **let ... in**. Ključnu reč **let** već smo koristili u interaktivnom okruženju za uvođenje privremenih definicija. U Haskell izrazima ona se može koristiti zajedno sa ključnom rečju **in** da bi se uvele vrednosti koje važe samo u određenom izrazu.

Opšti oblik je

```
let ime = vrednost
in izraz
```

Na primer:

```
cenaSaPopustom :: Double -> Double -> Double
cenaSaPopustom cena procenat =
  let popust = cena * procenat / 100
  in cena - popust
```

Na primer:

```
ghci> cenaSaPopustom 2000 10
1800.0
```

Ime `popust` postoji samo u izrazu koji sledi nakon ključne reči `in`. Ono omogućava da pomoćnom izračunavanju damo jasno ime i time definiciju učinimo preglednijom.

Konstrukcije `where` i `let ... in` imaju sličnu namenu, jer obe omogućavaju lokalno definisanje vrednosti. Konstrukcija `where` navodi pomoćne definicije nakon glavne definicije funkcije, dok ih `let ... in` uvodi neposredno unutar izraza.

Za sada je dovoljno koristiti onaj oblik koji konkretnu definiciju čini jasnijom. U narednoj podsekciji upoznaćemo još jedan veoma važan način definisanja funkcija u Haskellu, zasnovan na podudaranju oblika njihovih argumenata.

7.5 Podudaranje oblika

U prethodnoj podsekciji videli smo da se različiti slučajevi mogu opisati pomoću uslovnih izraza i ogradenih definicija. Haskell omogućava i drugi veoma važan način definisanja funkcija, kod koga se različite definicije biraju prema vrednosti ili obliku argumenta. Ovaj postupak naziva se podudaranje oblika (engl. *pattern matching*).

Najjednostavniji primer možemo videti kod logičkih vrednosti. Funkciju koja predstavlja logičku negaciju možemo definisati pomoću dve jednačine:

```
negacija :: Bool -> Bool
negacija True = False
negacija False = True
```

Kada se funkcija pozove, Haskell proverava kojoj definiciji odgovara prosleđeni argument. Na primer:

```
ghci> negacija True
False

ghci> negacija False
True
```

U pozivu `negacija True` argument se podudara sa obrascem `True`, pa se koristi prva definicija. U pozivu `negacija False` koristi se druga definicija.

Obrazac može sadržati i konkretnu brojnu vrednost. Na primer, možemo posebno izdvojiti slučaj kada je broj jednak nuli:

```
jeNula :: Int -> Bool
jeNula 0 = True
jeNula broj = False
```

Prva definicija primenjuje se samo kada je argument jednak 0. U drugoj definiciji ime `broj` predstavlja obrazac koji može da odgovara bilo kojoj vrednosti tipa `Int`.

Na primer:

```
ghci> jeNula 0
True
```

```
ghci> jeNula 7
False

ghci> jeNula (-3)
False
```

U drugoj definiciji funkcije `jeNula` konkretna vrednost argumenta nam zapravo nije potrebna. Kada želimo da kažemo da obrazac može odgovarati bilo kojoj vrednosti, ali tu vrednost nećemo koristiti, možemo upotrebiti znak `_`. Takav obrazac naziva se džoker (engl. *wildcard*).

Prethodnu funkciju zato možemo zapisati jednostavnije:

```
jeNula :: Int -> Bool
jeNula 0 = True
jeNula _ = False
```

Znak `_` odgovara bilo kojoj vrednosti, ali joj ne daje ime koje bismo kasnije mogli da koristimo u izrazu.

Jednačine funkcije proveravaju se redom kojim su napisane. Koristi se prva definicija čiji se obrazac podudara sa prosleđenim argumentom. Zbog toga specifičnije obrasce obično navodimo pre opštijih.

Na primer, funkciju koja za brojeve od 1 do 3 vraća odgovarajući tekst možemo definisati ovako:

```
brojRecima :: Int -> String
brojRecima 1 = "Jedan"
brojRecima 2 = "Dva"
brojRecima 3 = "Tri"
brojRecima _ = "Drugi broj"
```

Funkciju možemo isprobati u interaktivnom okruženju:

```
ghci> brojRecima 2
"Dva"

ghci> brojRecima 8
"Drugi broj"
```

Poslednji obrazac `_` treba da se nalazi nakon posebnih slučajeva. Kada bi bio naveden prvi, odgovarao bi svakom argumentu, pa se naredne definicije nikada ne bi koristile.

Važno je i da definicija funkcije obuhvati sve vrednosti za koje želimo da je koristimo. Na primer:

```
opis :: Int -> String
opis 0 = "Nula"
opis 1 = "Jedan"
```

Ova definicija nije potpuna. Funkcija ima definisan rezultat za argumente 0 i 1, ali ne i za ostale cele brojeve. Ako pokušamo da je primenimo na vrednost za koju nijedan obrazac ne odgovara, pri izračunavanju će doći do greške.

U ovom slučaju definiciju možemo učiniti potpunom dodavanjem opšteg obrasca:

```
opis :: Int -> String
opis 0 = "Nula"
opis 1 = "Jedan"
opis _ = "Ostalo"
```

Podudaranje oblika i ograđene definicije imaju sličnu namenu, ali se koriste na različite načine. Podudaranje oblika je naročito pogodno kada želimo da izdvojimo određene vrednosti ili oblike argumenata, dok su ograđene definicije pogodne kada izbor rezultata zavisi od logičkih uslova kao što su poređenja.

Na primer, za poseban slučaj vrednosti 0 prirodno je koristiti podudaranje oblika:

```
jeNula :: Int -> Bool
jeNula 0 = True
jeNula _ = False
```

Sa druge strane, kada želimo da razlikujemo negativne, nulu i pozitivne brojeve, ograđena definicija je prirodnija jer se slučajevi opisuju uslovima.

Podudaranje oblika predstavlja jednu od važnih mogućnosti Haskell-a. Za sada smo ga koristili samo sa jednostavnim vrednostima. Kasnije ćemo videti da ono postaje posebno korisno pri radu sa listama i korisnički definisanim tipovima, gde se funkcije mogu definisati neposredno prema strukturi podataka koje obrađuju.

7.6 Polimorfni tipovi i tipske klase

Do sada smo u tipskim potpisima uglavnom koristili konkretne tipove, kao što su `Int`, `Bool` i `String`. Međutim, neke funkcije mogu da rade na isti način sa vrednostima različitih tipova.

Najjednostavniji primer jeste funkcija koja samo vraća vrednost koju je dobila kao argument:

```
identitet :: a -> a
identitet vrednost = vrednost
```

U ovom tipskom potpisu `a` nije ime konkretnog tipa. Ono predstavlja tipsku promenljivu koja može označavati proizvoljan tip.

Imena konkretnih tipova, kao što su `Int`, `Bool` i `String`, počinju velikim slovom, dok se za tipske promenljive uobičajeno koriste mala slova, kao što su `a`, `b` i `c`.

Funkciju `identitet` zato možemo koristiti sa vrednostima različitih tipova:

```
ghci> identitet 5
5

ghci> identitet True
True

ghci> identitet "Ana"
"Ana"
```

U svakom slučaju tip rezultata jednak je tipu argumenta. Ako je argument tipa `Bool`, rezultat je tipa `Bool`. Ako je argument tipa `String`, rezultat je tipa `String`.

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

U tipskom potpisu može se pojaviti i više različitih tipskih promenljivih. Na primer:

```
prvi :: a -> b -> a
prvi prva druga = prva
```

Funkcija `prvi` prima dva argumenta, koji ne moraju biti istog tipa, i vraća prvi argument. Na primer:

```
ghci> prvi 5 True
5

ghci> prvi "Ana" 10
"Ana"
```

Tipske promenljive `a` i `b` mogu u jednom pozivu predstavljati različite tipove. Jedino je određeno da rezultat mora imati isti tip kao prvi argument.

Međutim, nije svaka funkcija primenljiva na potpuno proizvoljne tipove. Ako funkcija koristi neku određenu operaciju, argumenti moraju pripadati tipovima za koje je ta operacija definisana.

Za opis takvih zahteva u Haskellu se koriste tipske klase. Tipska klasa predstavlja grupu tipova koji podržavaju određene operacije. Za konkretan tip koji pripada nekoj tipskoj klasi kaže se da je instanca te klase.

Tipske klase u Haskellu ne treba poistovećivati sa klasama koje smo koristili u objektno-orijentisanom programiranju u Pythonu. Haskellova tipska klasa prvenstveno opisuje koje operacije neki tip podržava.

Jedna od osnovnih tipskih klasa jeste `Eq`. Njoj pripadaju tipovi čije vrednosti možemo porediti na jednakost i nejednakost pomoću operatora `==` i `/=`.

Na primer, možemo definisati polimorfnu funkciju koja proverava da li su dva argumenta jednaka:

```
jednaki :: Eq a => a -> a -> Bool
jednaki prva druga = prva == druga
```

Zapis

```
Eq a => a -> a -> Bool
```

znači da funkcija prima dve vrednosti istog tipa `a` i vraća logičku vrednost, pri čemu tip `a` mora pripadati klasi `Eq`. Deo `Eq a` naziva se klasno ograničenje, dok simbol `=>` odvaja ograničenje od ostatka tipa.

Funkcija se zato može koristiti nad različitim tipovima čije se vrednosti mogu porediti:

```
ghci> jednaki 5 5
True

ghci> jednaki 'a' 'b'
False

ghci> jednaki "Ana" "Ana"
True
```

Klasa `Ord` obuhvata tipove čije se vrednosti mogu uređivati i porediti operatorima kao što su `<`, `<=`, `>` i `>=`.

Na primer:

```
veci :: Ord a => a -> a -> a
veci prva druga =
    if prva >= druga then prva else druga
```

Funkcija `veci` može da se koristi sa različitim tipovima koji pripadaju klasi `Ord`:

```
ghci> veci 5 8
8

ghci> veci 'a' 'z'
'z'
```

Klasa `Num` obuhvata osnovne numeričke tipove i operacije kao što su sabiranje, oduzimanje i množenje. Zato funkcija koja koristi takve operacije ne mora biti ograničena samo na tip `Int`.

Na primer:

```
udvostruci :: Num a => a -> a
udvostruci broj = broj + broj
```

Ista funkcija može se koristiti sa različitim numeričkim tipovima:

```
ghci> udvostruci (5 :: Int)
10

ghci> udvostruci (2.5 :: Double)
5.0
```

Sada možemo preciznije razumeti i tip numeričkih literala. Ako u interaktivnom okruženju proverimo tip broja 5, dobićemo:

```
ghci> :t 5
5 :: Num a => a
```

Broj 5, dakle, nije unapred vezan samo za jedan konkretan numerički tip. Može se koristiti kao vrednost različitih numeričkih tipova, u zavisnosti od konteksta.

Ako želimo da navedemo tačno određeni tip, možemo ga eksplicitno zadati:

```
ghci> :t (5 :: Int)
(5 :: Int) :: Int

ghci> :t (5 :: Double)
(5 :: Double) :: Double
```

To je jedan od razloga zbog kojih eksplicitno navođenje tipa ponekad može biti korisno kada želimo da odredimo koji konkretan tip treba koristiti.

Klasa `Show` obuhvata tipove čije se vrednosti mogu pretvoriti u tekstualni prikaz pomoću funkcije `show`.

Na primer, možemo definisati:

```
prikazi :: Show a => a -> String
prikazi vrednost = show vrednost
```

Funkcija se može primeniti na različite vrednosti čiji tipovi pripadaju klasi `Show`:

```
ghci> prikazi True
"True"

ghci> prikazi (25 :: Int)
"25"
```

Klasa `Show` biće nam korisna i kasnije, kada budemo definisali sopstvene tipove podataka i želeli da njihove vrednosti možemo jednostavno prikazivati.

Ponekad funkcija zahteva više različitih svojstava istog tipa. Tada se klasna ograničenja navode zajedno u zagradama.

Na primer:

```
pozitivniKvadrat :: (Num a, Ord a) => a -> a
pozitivniKvadrat broj
| broj > 0 = broj * broj
| otherwise = 0
```

Ograničenje `Num a` potrebno je zato što koristimo množenje i broj 0, dok je `Ord a` potrebno zato što vrednost poredimo sa nulom.

Na primer:

```
ghci> pozitivniKvadrat (4 :: Int)
16

ghci> pozitivniKvadrat (-3 :: Int)
0
```

Polimorfni tipovi omogućavaju da ista funkcija radi sa vrednostima različitih tipova. Kada funkcija zahteva određene operacije, tipske klase omogućavaju da precizno navedemo koje osobine ti tipovi moraju imati. Na taj način Haskell istovremeno omogućava opšte definicije funkcija i strogu proveru njihovih tipova.

Time smo upoznali osnovne načine definisanja funkcija i najvažnije elemente sistema tipova u Haskellu. U narednoj sekciji preći ćemo na liste i rekurziju, gde će polimorfni tipovi i podudaranje oblika imati posebno važnu ulogu.

8 Liste i rekurzija u Haskellu

8.1 Liste i osnovne operacije

Lista je uređena kolekcija vrednosti istog tipa. Elementi liste zapisuju se između uglastih zagrada i razdvajaju zarezima.

Na primer:

```
[2, 4, 6, 8]
[True, False, True]
['a', 'b', 'c']
["Ana", "Marko", "Jovan"]
```

Prazna lista zapisuje se kao

```
[]
```

Za razliku od nekih kolekcija koje smo koristili u Pythonu, svi elementi jedne Haskell liste moraju imati isti tip. Na primer, možemo definisati:

```
brojevi :: [Int]
brojevi = [2, 4, 6, 8]

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

slova :: [Char]
slova = ['a', 'b', 'c']
```

Zapis `[Int]` označava listu čiji su elementi tipa `Int`, dok `[Bool]` označava listu logičkih vrednosti.

Tip liste može biti i polimorfan. Zapis

```
[a]
```

označava listu elemenata nekog tipa `a`. Svi elementi konkretne liste i dalje moraju imati isti tip, ali taj tip ne mora unapred biti određen.

Zato, na primer, sledeća lista nije ispravna, jer njeni elementi pripadaju različitim tipovima:

```
[5, True, 'a']
```

Element liste može i sam biti lista. Na primer:

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

Tip `[[Int]]` možemo čitati kao lista listi celih brojeva.

Ranije smo pomenuli da je tip `String` u Haskellu zapravo lista karaktera. Zbog toga su zapisi

```
"Ana"
['A', 'n', 'a']
```

dva načina predstavljanja iste tekstualne vrednosti. To možemo proveriti i u interaktivnom okruženju:

```
ghci> "Ana" == ['A', 'n', 'a']
True
```

Jedna od najvažnijih operacija nad listama jeste operator `:`, kojim se od jednog elementa i postojeće liste konstruiše nova lista.

Na primer:

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

ghci> 'A' : ['n', 'a']
"Ana"
```

Sa leve strane operatora `:` nalazi se jedan element, a sa desne lista elemenata odgovarajućeg tipa.

Svaka neprazna lista može se posmatrati kao prvi element dodat na početak ostatka liste. Tako je lista

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

ekvivalentna zapisu

```
1 : 2 : 3 : []
```

Ovaj način posmatranja liste kasnije će biti posebno važan pri podudaranju oblika i pisanju rekurzivnih funkcija.

Za spajanje dve liste koristi se operator `++`:

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

ghci> "Dobar " ++ "dan"
"Dobar dan"
```

Važno je razlikovati operatore `:` i `++`. Operator `:` dodaje jedan element na početak liste, dok operator `++` spaja dve liste.

Prvom elementu neprazne liste možemo pristupiti pomoću funkcije `head`:

```
ghci> head [10, 20, 30]
10

ghci> head "Ana"
'A'
```

Funkcija `tail` vraća listu bez njenog prvog elementa:

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

ghci> tail "Ana"
```

```
"na"
```

Funkcije `head` i `tail` nisu definisane za praznu listu. Zato bi pokušaj izračunavanja izraza

```
head []
```

doveo do greške. Kasnije ćemo videti kako se pomoću podudaranja oblika prazna i neprazna lista mogu posebno obraditi.

Broj elemenata liste možemo dobiti pomoću funkcije `length`:

```
ghci> length [10, 20, 30, 40]
4
```

```
ghci> length "Haskell"
7
```

Funkcija `null` proverava da li je lista prazna:

```
ghci> null []
True
```

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

Redosled elemenata možemo obrnuti pomoću funkcije `reverse`:

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

```
ghci> reverse "Ana"
"anA"
```

Funkcija `elem` proverava da li se određena vrednost nalazi u listi:

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

```
ghci> elem 7 [1, 2, 3, 4]
False
```

```
ghci> elem 'a' "Ana"
True
```

Kod numeričkih listi često su korisne i funkcije `sum` i `product`. Prva računa zbir, a druga proizvod elemenata:

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

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

Pojedinačnom elementu liste može se pristupiti i prema njegovom indeksu pomoću operatora

!!. Indeksi počinju od nule:

```
ghci> [10, 20, 30, 40] !! 0
10

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

Indeks mora odgovarati postojećem elementu liste. Pokušaj pristupa elementu izvan granica liste dovodi do greške.

Liste ćemo u nastavku koristiti kao jednu od osnovnih struktura podataka u Haskellu. Posebno će nam biti važno da svaku listu možemo posmatrati kao praznu listu ili kao prvi element povezan sa ostatkom liste. Pre nego što to iskoristimo za podudaranje oblika i rekurziju, upoznaćemo još nekoliko pogodnih načina za konstruisanje i obradu listi.

8.2 Rasponi i izlistavanje

Liste čiji elementi prate pravilan redosled često možemo zapisati pomoću raspona. Najjednostavniji oblik raspona određuje početnu i krajnju vrednost.

Na primer:

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

ghci> [5..10]
[5,6,7,8,9,10]
```

U ovim primerima elementi se povećavaju za 1, a obe navedene granice pripadaju listi kada se mogu dostići odgovarajućim korakom.

Ako želimo drugačiji korak, navodimo prva dva elementa raspona. Razlika između njih određuje način na koji se formiraju naredni elementi.

Na primer:

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

ghci> [1,3..10]
[1,3,5,7,9]
```

U prvom primeru korak je 2. U drugom primeru poslednja vrednost koja ne prelazi gornju granicu jeste 9, pa se broj 10 ne pojavljuje u listi.

Raspon može biti i opadajući. Tada drugi navedeni element mora biti manji od prvog:

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

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

Rasponi se mogu koristiti i sa karakterima:

```
ghci> ['a'..'f']
```

```
"abcdef"
```

```
ghci> ['A'..'E']  
"ABCDE"
```

Pošto je `String` lista karaktera, rezultat se u interaktivnom okruženju prikazuje kao tekstualna vrednost.

Raspone ćemo uglavnom koristiti sa celim brojevima i karakterima. Kod brojeva sa pokretnim zarezom mogu se pojaviti posledice njihovog približnog predstavljanja, pa takve raspone za sada nećemo koristiti.

Haskell omogućava i raspone kojima nije zadat krajnji element. Na primer:

```
[1..]
```

predstavlja beskonačnu listu prirodnih brojeva počev od 1. Ovakva lista može da postoji zahvaljujući lenjom izračunavanju, koje smo pomenuli prilikom upoznavanja sa osnovnim osobinama Haskell. Elementi se ne moraju svi izračunati unapred, već se dobijaju kada postanu potrebni.

Naravno, nema smisla zahtevati prikaz cele beskonačne liste. Umesto toga možemo uzeti samo određeni broj početnih elemenata pomoću funkcije `take`:

```
ghci> take 5 [1..]  
[1,2,3,4,5]  
  
ghci> take 4 [10,20..]  
[10,20,30,40]
```

Funkcija `take` prima broj elemenata i listu, a vraća početni deo te liste.

Slična funkcija `drop` odbacuje određeni broj početnih elemenata:

```
ghci> drop 2 [10,20,30,40,50]  
[30,40,50]  
  
ghci> take 5 (drop 10 [1..])  
[11,12,13,14,15]
```

U drugom primeru najpre se odbacuje prvih deset elemenata beskonačne liste, a zatim se uzima narednih pet.

Još jedan karakterističan način konstruisanja listi u Haskellu jeste izlistavanje (engl. *list comprehension*). Njime možemo opisati kako se elementi nove liste dobijaju iz elemenata neke postojeće liste.

Osnovni oblik izlistavanja je

```
[izraz | element <- lista]
```

Zapis `element <- lista` naziva se generatorom. On redom uzima elemente zadate liste, dok izraz sa leve strane znaka `|` određuje šta će se dodati u novu listu.

Na primer, kvadrate brojeva od 1 do 5 možemo dobiti ovako:

```
ghci> [broj * broj | broj <- [1..5]]  
[1,4,9,16,25]
```

Za svaki element liste `[1..5]` ime `broj` dobija odgovarajuću vrednost, a u novu listu dodaje se vrednost izraza `broj * broj`.

Izlistavanju možemo dodati i uslov kojim određujemo koje elemente želimo da zadržimo. Na primer:

```
ghci> [broj | broj <- [1..10], broj > 5]
[6,7,8,9,10]
```

U novu listu ulaze samo oni elementi za koje je navedeni uslov tačan.

Na sličan način možemo izdvojiti parne brojeve:

```
ghci> [broj | broj <- [1..10], broj `mod` 2 == 0]
[2,4,6,8,10]
```

Izbor elemenata i njihovo transformisanje mogu se koristiti istovremeno. Na primer, možemo izračunati kvadrate samo parnih brojeva:

```
ghci> [broj * broj | broj <- [1..10], broj `mod` 2 == 0]
[4,16,36,64,100]
```

Izlistavanje tako omogućava da na sažet način opišemo od kojih elemenata želimo da napravimo novu listu i kako svaki od njih treba transformisati. Umesto da eksplicitno opisujemo prolazak kroz kolekciju pomoću petlje, navodimo svojstva elemenata koji treba da se pojave u rezultatu.

Rasponi i izlistavanja predstavljaju pogodne načine za konstruisanje listi. U nastavku ćemo detaljnije razmotriti samu strukturu liste i videti kako se podudaranjem oblika mogu posebno obraditi prazna lista i lista koja sadrži bar jedan element.

8.3 Podudaranje oblika nad listama

Podudaranje oblika ranije smo koristili da bismo različite definicije funkcije vezali za različite vrednosti argumenata. Na primer, posebno smo mogli obraditi vrednosti `True` i `False` ili izdvojiti slučaj kada je ceo broj jednak nuli.

Podudaranje oblika posebno je korisno pri radu sa listama. Svaka lista može biti prazna ili neprazna. Prazna lista zapisuje se kao `[]`.

Dok se neprazna lista može posmatrati kao prvi element dodat na početak ostatka liste:

```
prvi : ostatak
```

Na primer, listu

```
[10, 20, 30]
```

možemo posmatrati kao

```
10 : [20, 30]
```

Zato pri podudaranju oblika možemo koristiti obrazac

```
(prvi:ostatak)
```

Ako se ovaj obrazac podudara sa nepraznom listom, ime `prvi` dobija vrednost prvog elementa, dok ime `ostatak` predstavlja listu svih preostalih elemenata.

Na primer, za listu `[10, 20, 30]` važi

```
prvi = 10
ostatak = [20, 30]
```

Imena `prvi` i `ostatak` nisu posebne reči jezika Haskell. To su obična imena koja smo izabrali kako bi njihova uloga bila jasna. U Haskell programima često se koriste kraća imena `x` i `xs`, ali ćemo u primerima uglavnom koristiti opisnije nazive.

Praznu i nepraznu listu zato možemo jednostavno razlikovati pomoću dve definicije funkcije:

```
prazna :: [a] -> Bool
prazna [] = True
prazna (_:_) = False
```

Prva definicija odgovara samo praznoj listi. Druga odgovara svakoj nepraznoj listi.

U obrascu `(_:_)` ne koristimo ni prvi element ni ostatak liste, pa su oba predstavljena džokerom `_`.

Na primer:

```
ghci> prazna []
True

ghci> prazna [1, 2, 3]
False

ghci> prazna "Ana"
False
```

Ako nam je prvi element liste potreban, možemo mu dati ime. Na primer, definišimo funkciju koja vraća prvi element liste celih brojeva, a za praznu listu vraća 0:

```
prviIliNula :: [Int] -> Int
prviIliNula [] = 0
prviIliNula (prvi:_) = prvi
```

Na primer:

```
ghci> prviIliNula []
0

ghci> prviIliNula [10, 20, 30]
10
```

U drugoj definiciji ime `prvi` vezuje se za prvi element liste. Ostatak liste nam nije potreban, pa umesto njegovog imena koristimo `_`.

Možemo razlikovati i listu koja sadrži tačno jedan element. Takav obrazac možemo zapisati kao

```
[element]
```

Na primer:

```
opisListe :: [a] -> String
opisListe [] = "Prazna"
opisListe [_] = "Jedan element"
opisListe _ = "Vise elemenata"
```

Funkciju možemo isprobati u interaktivnom okruženju:

```
ghci> opisListe []
"Prazna"

ghci> opisListe [5]
"Jedan element"

ghci> opisListe [5, 10]
"Vise elemenata"
```

Prvi obrazac odgovara praznoj listi, drugi listi sa tačno jednim elementom, dok poslednji obrazac `_` obuhvata sve preostale slučajeve.

Možemo opisati i listu koja sadrži najmanje dva elementa. Na primer, obrazac

```
(prvi:drugi:ostatak)
```

odgovara svakoj listi koja sadrži najmanje dva elementa. Ime `prvi` dobija prvi element, `drugi` drugi element, dok `ostatak` predstavlja listu svih preostalih elemenata.

Operator `:` grupiše se zdesna, pa se ovaj obrazac može posmatrati kao

```
prvi : (drugi : ostatak)
```

Na primer, možemo proveriti da li lista sadrži najmanje dva elementa:

```
imaBarDva :: [a] -> Bool
imaBarDva (_:_:_) = True
imaBarDva _ = False
```

Na primer:

```
ghci> imaBarDva []
False

ghci> imaBarDva [5]
False

ghci> imaBarDva [5, 10]
True

ghci> imaBarDva [5, 10, 15]
True
```

Kao i kod drugih oblika podudaranja, redosled definicija je važan. Specifičnije obrasce treba navesti pre opštijih, jer se koristi prva definicija čiji se obrazac podudara sa argumentom.

Na primer, u funkciji `opisListe` obrazac `_` mora biti naveden nakon obrazaca `[]` i `[_]`.

Kada bi bio naveden prvi, odgovarao bi svakoj listi i ostale definicije se ne bi koristile.

Važno je i da obuhvatimo sve oblike liste za koje želimo da funkcija bude definisana. Na primer:

```
prviElement :: [Int] -> Int
prviElement (element:_) = element
```

ne obuhvata slučaj prazne liste. Pokušaj da se funkcija `prviElement` primeni na praznu listu doveo bi do greške.

Podudaranje oblika omogućava da se struktura liste neposredno vidi u definiciji funkcije. Posebno će nam biti važna podela na dva osnovna slučaja:

```
funkcija [] = ...
funkcija (prvi:ostatak) = ...
```

Prva definicija obrađuje praznu listu, dok druga odvajava prvi element od ostatka neprazne liste. Upravo će ovaj obrazac predstavljati osnovu za mnoge rekurzivne funkcije nad listama koje ćemo upoznati u nastavku.

8.4 Rekurzivne funkcije

Rekurzivne funkcije upoznali smo već pri radu sa Pythonom. Funkcija je rekurzivna ako u svojoj definiciji poziva samu sebe. Da bi se izračunavanje završilo, potrebno je odrediti bazni slučaj u kome se rezultat dobija neposredno, bez novog rekurzivnog poziva.

U ostalim slučajevima problem se svodi na jednostavniji problem iste vrste. Rekurzivni pozivi treba da vode ka baznom slučaju, kako bi se posle konačnog broja koraka dobio rezultat.

U Haskellu se bazni i rekurzivni slučaj često veoma prirodno zapisuju pomoću podudaranja oblika. Jedan od najjednostavnijih primera jeste faktorijel nenegativnog celog broja.

Faktorijel broja 0 jednak je 1, dok za pozitivan broj važi

$$n! = n \cdot (n - 1)!$$

Ovu definiciju možemo neposredno zapisati u Haskellu:

```
faktorijel :: Integer -> Integer
faktorijel 0 = 1
faktorijel n = n * faktorijel (n - 1)
```

Prva jednačina predstavlja bazni slučaj. Kada je argument jednak 0, rezultat se neposredno dobija kao 1.

Druga jednačina predstavlja rekurzivni slučaj. Za pozitivan argument `n` funkcija ponovo poziva samu sebe, ali sa manjim argumentom `n - 1`.

Na primer:

```
ghci> faktorijel 4
24

ghci> faktorijel 6
720
```

Izračunavanje vrednosti `faktorijel 4` možemo pratiti korak po korak:

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

Pri svakom rekurzivnom pozivu argument se smanjuje za 1. Na taj način se posle konačnog broja koraka dolazi do baznog slučaja **faktorijel 0**.

U izrazu

```
faktorijel (n - 1)
```

zagrade su važne zato što želimo da rezultat izraza $n - 1$ prosledimo kao argument funkciji. Bez zagrada bi zapis

```
faktorijel n - 1
```

označavao oduzimanje broja 1 od rezultata izraza **faktorijel n**, što nije isto.

Još jedan jednostavan primer jeste zbir svih celih brojeva od 1 do zadatog nenegativnog broja. Za argument 0 zbir je jednak 0. Za pozitivan broj n zbir možemo dobiti tako što broju n dodamo zbir svih brojeva do $n - 1$.

Rekurzivna definicija može se zapisati ovako:

```
zbirDo :: Integer -> Integer
zbirDo 0 = 0
zbirDo n = n + zbirDo (n - 1)
```

Na primer:

```
ghci> zbirDo 5
15

ghci> zbirDo 10
55
```

Izračunavanje prvog primera izgleda ovako:

```
zbirDo 5
= 5 + zbirDo 4
= 5 + (4 + zbirDo 3)
= 5 + (4 + (3 + zbirDo 2))
= 5 + (4 + (3 + (2 + zbirDo 1)))
= 5 + (4 + (3 + (2 + (1 + zbirDo 0))))
= 15
```

I funkcija **faktorijel** i funkcija **zbirDo** u ovim definicijama namenjene su nenegativnim argumentima. Za pozitivan argument svaki rekurzivni poziv smanjuje njegovu vrednost, pa se na kraju dolazi do baznog slučaja.

Ako rekurzivni pozivi ne vode ka baznom slučaju, izračunavanje se neće završiti na željeni

način. Zato je pri pisanju rekurzivne funkcije važno odrediti ne samo bazni slučaj, već i način na koji se svaki naredni poziv približava tom slučaju.

U Pythonu smo za veliki broj ponavljanja koristili petlje `for` i `while`, a rekurziju smo upoznali kao još jedan način rešavanja problema. U čistom Haskell kodu ponavljanje se često izražava rekurzijom, a kasnije ćemo videti i da se mnogi česti oblici ponavljanja mogu izraziti pomoću funkcija višeg reda.

Posebno važna primena rekurzije u Haskellu jeste obrada listi. Pošto se svaka lista može posmatrati kao prazna lista ili kao prvi element povezan sa ostatkom liste, njena struktura prirodno odgovara strukturi rekurzivne funkcije. To ćemo detaljno razmotriti u narednoj podsekciji.

8.5 Rekurzivne funkcije nad listama

Rekurzija je posebno pogodna za obradu listi. U prethodnim podsekcijama videli smo da se svaka lista može posmatrati kao prazna lista ili kao prvi element povezan sa ostatkom liste:

```
[]  
(prvi:ostatak)
```

Ova podela prirodno odgovara strukturi rekurzivne funkcije. Prazna lista obično predstavlja bazni slučaj, dok se kod neprazne liste obrađuje prvi element, a ista funkcija rekurzivno primenjuje na ostatak liste.

Kao prvi primer, definišimo funkciju koja računa zbir elemenata numeričke liste:

```
zbirListe :: Num a => [a] -> a  
zbirListe [] = 0  
zbirListe (prvi:ostatak) =  
    prvi + zbirListe ostatak
```

Prva definicija predstavlja bazni slučaj. Zbir elemenata prazne liste jednak je 0.

Druga definicija predstavlja rekurzivni slučaj. Zbir elemenata neprazne liste dobija se tako što se njen prvi element doda zbiru elemenata ostatka liste.

Na primer:

```
ghci> zbirListe []  
0  
  
ghci> zbirListe [2, 3, 4]  
9  
  
ghci> zbirListe [10, 20, 30, 40]  
100
```

Izračunavanje vrednosti `zbirListe [2,3,4]` možemo pratiti korak po korak:

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

Pri svakom rekurzivnom pozivu funkcija dobija kraću listu. Od liste [2,3,4] prelazimo na [3,4], zatim na [4] i konačno na praznu listu. Na taj način se posle konačnog broja koraka dolazi do baznog slučaja.

Isti princip možemo primeniti na različite probleme. Bazni slučaj određujemo za praznu listu, dok u rekurzivnom slučaju obrađujemo prvi element i nastavljamo obradu nad ostatkom liste.

Na primer, možemo sami definisati funkciju koja određuje broj elemenata liste:

```
duzina :: [a] -> Int
duzina [] = 0
duzina (_:ostatak) =
    1 + duzina ostatak
```

Vrednost prvog elementa nam ovde nije važna, pa koristimo džoker `_`. Važno je samo da neprazna lista sadrži jedan prvi element i određeni broj elemenata u ostatku.

Na primer:

```
ghci> duzina []
0

ghci> duzina [10, 20, 30]
3

ghci> duzina "Haskell"
7
```

Izračunavanje za listu sa tri elementa može se predstaviti ovako:

```
duzina [10,20,30]
= 1 + duzina [20,30]
= 1 + (1 + duzina [30])
= 1 + (1 + (1 + duzina []))
= 1 + (1 + (1 + 0))
= 3
```

Rekurzijom možemo i proveravati svojstva elemenata liste. Na primer, definišimo funkciju koja proverava da li se data vrednost nalazi u listi:

```
sadrzi :: Eq a => a -> [a] -> Bool
sadrzi _ [] = False
sadrzi trazeni (prvi:ostatak)
    | trazeni == prvi = True
    | otherwise = sadrzi trazeni ostatak
```

Ako je lista prazna, tražena vrednost se u njoj ne nalazi. Kod neprazne liste najpre proveravamo prvi element. Ako je jednak traženoj vrednosti, rezultat je `True`. U suprotnom nastavljamo pretragu u ostatku liste.

Na primer:

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

```
ghci> sadrzi 7 [1, 2, 3, 4]
False

ghci> sadrzi 'a' "Ana"
True
```

Klasno ograničenje `Eq` a potrebno je zato što elemente poredimo pomoću operatora `==`.

Rekurzivna funkcija ne mora da izračunava samo jednu vrednost. Njome možemo konstruisati i novu listu. Na primer, možemo napraviti listu u kojoj je svaki broj iz početne liste udvostručen:

```
udvostruciSve :: Num a => [a] -> [a]
udvostruciSve [] = []
udvostruciSve (prvi:ostatak) =
  (2 * prvi) : udvostruciSve ostatak
```

Bazni slučaj kaže da od prazne liste dobijamo praznu listu. Kod neprazne liste udvostručujemo prvi element i pomoću operatora `:` dodajemo ga na početak rezultata koji dobijamo rekurzivnom obradom ostatka.

Na primer:

```
ghci> udvostruciSve []
[]

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

Izračunavanje možemo posmatrati i ovako:

```
udvostruciSve [1,2,3]
= 2 : udvostruciSve [2,3]
= 2 : (4 : udvostruciSve [3])
= 2 : (4 : (6 : udvostruciSve []))
= 2 : (4 : (6 : []))
= [2,4,6]
```

Na ovim primerima vidi se koliko se podudaranje oblika i rekurzija prirodno dopunjuju pri radu sa listama. Obrazac `[]` određuje bazni slučaj, dok obrazac `(prvi:ostatak)` omogućava da obradimo jedan element i zatim isti postupak primenimo na kraću listu.

U svakom od prikazanih primera rekurzivni poziv dobija `ostatak`, koji sadrži jedan element manje od početne liste. Zbog toga se za konačne liste na kraju dolazi do prazne liste i baznog slučaja.

U Pythonu smo slične obrade kolekcija često zapisivali pomoću petlji. U Haskellu se struktura liste može neposredno odraziti na strukturu rekurzivne funkcije, što predstavlja karakterističan način organizovanja ovakvih izračunavanja u funkcionalnom pristupu.

Ipak, mnogi obrasci rekurzivne obrade listi veoma se često ponavljaju. U narednoj sekciji videćemo kako se takvi obrasci mogu izraziti opštijim funkcijama koje primaju druge funkcije kao argumente. Time ćemo doći do jedne od najvažnijih ideja funkcionalnog programiranja, funkcija višeg reda.

9 Funkcije višeg reda u Haskellu

9.1 Funkcije kao argumenti i lambda izrazi

U prethodnoj sekciji videli smo da se pri obradi listi često ponavljaju slični obrasci. Na primer, rekursivna funkcija može redom obrađivati sve elemente liste, pri čemu se samo konkretna operacija nad svakim elementom razlikuje. Jedan način da se takvi obrasci izdvoje jeste korišćenje funkcija višeg reda.

Funkcija višeg reda jeste funkcija koja prima drugu funkciju kao argument ili vraća funkciju kao rezultat. Sa drugim slučajem već smo se susreli kod delimične primene funkcija. Na primer, ako funkcija `saberi` ima tip

```
saberi :: Int -> Int -> Int
```

onda izraz `saberi 10` predstavlja novu funkciju tipa `Int -> Int`.

Sada ćemo razmotriti drugi važan slučaj, kada se funkcija prosleđuje drugoj funkciji kao argument.

Jednostavan primer predstavlja funkcija koja zadatu funkciju primenjuje dva puta:

```
primeniDvaput :: (a -> a) -> a -> a
primeniDvaput funkcija vrednost =
    funkcija (funkcija vrednost)
```

Prvi argument funkcije `primeniDvaput` jeste funkcija tipa `a -> a`. Ona, dakle, prima vrednost nekog tipa `a` i vraća vrednost istog tipa. Drugi argument je vrednost tipa `a`, a konačan rezultat takođe je istog tipa.

Zagrade u delu

```
(a -> a)
```

važne su zato što pokazuju da je ceo funkcijski tip `a -> a` tip prvog argumenta funkcije `primeniDvaput`.

Definišimo, na primer, funkciju koja ceo broj povećava za 2:

```
povecajZaDva :: Int -> Int
povecajZaDva broj = broj + 2
```

Ovu funkciju možemo proslediti kao argument funkciji `primeniDvaput`:

```
ghci> primeniDvaput povecajZaDva 5
9
```

Izračunavanje možemo pratiti korak po korak:

```
primeniDvaput povecajZaDva 5
= povecajZaDva (povecajZaDva 5)
= povecajZaDva 7
= 9
```

Funkcija `povecajZaDva` ovde se ne poziva u trenutku kada je navodimo kao prvi argument. Prosleđujemo samu funkciju, a funkcija `primeniDvaput` određuje gde će je i koliko puta primeniti.

Funkciju koju želimo da prosledimo ne moramo uvek prethodno posebno imenovati. Haskell omogućava zapis funkcije pomoću lambda izraza. Lambda izraz predstavlja funkciju bez posebnog imena, pa se često naziva i anonimna funkcija.

Opšti oblik lambda izraza je

```
\argument -> izraz
```

Znak `\` predstavlja zapis grčkog slova λ , `argument` predstavlja argument funkcije, a izraz sa desne strane strelice određuje njen rezultat.

Na primer, funkcija koja broj povećava za 3 može se zapisati kao

```
\broj -> broj + 3
```

Lambda izraz možemo neposredno primeniti na argument:

```
ghci> (\broj -> broj + 3) 10  
13
```

Zagrade oko lambda izraza potrebne su zato što želimo da ceo lambda izraz predstavlja funkciju koja se zatim primenjuje na vrednost 10.

Lambda izrazi su naročito korisni kada kratku funkciju želimo neposredno da prosledimo drugoj funkciji. Na primer:

```
ghci> primeniDvaput (\broj -> broj + 3) 10  
16
```

Lambda funkcija najpre preslikava broj 10 u 13, a zatim se ista funkcija još jednom primenjuje na dobijenu vrednost i daje 16.

Lambda izraz može imati i više argumenata. Na primer:

```
ghci> (\prvi drugi -> prvi + drugi) 4 5  
9
```

Ovaj zapis predstavlja kraći oblik izraza

```
\prvi -> (\drugi -> prvi + drugi)
```

što odgovara načinu na koji smo ranije tumačili karirane funkcije. Funkcija najpre prima prvi argument, a zatim vraća funkciju koja prima drugi argument.

Lambda izrazu možemo i dodeliti ime:

```
udvostruci :: Int -> Int  
udvostruci = \broj -> 2 * broj
```

Ova definicija ima isto značenje kao

```
udvostruci :: Int -> Int  
udvostruci broj = 2 * broj
```

Za obične imenovane funkcije drugi zapis je najčešće jednostavniji i čitljiviji. Lambda izrazi su posebno pogodni kada je funkcija kratka i potrebna samo na mestu na kome je koristimo.

Kod operatora postoji još kraći način da dobijemo jednostavnu funkciju. Ako operatoru unapred zadamo jedan od argumenata, možemo dobiti funkciju koja očekuje preostali argument.

Na primer, izraz

```
(+1)
```

predstavlja funkciju koja svom argumentu dodaje 1:

```
ghci> (+1) 5  
6
```

Slično tome:

```
ghci> (*2) 5  
10
```

Ovakve funkcije možemo prosledivati drugim funkcijama na isti način kao imenovane funkcije i lambda izraze. Na primer:

```
ghci> primeniDvaput (+1) 5  
7  
  
ghci> primeniDvaput (*2) 3  
12
```

U prvom primeru broj se dva puta povećava za 1. U drugom se dva puta množi brojem 2.

Funkcije višeg reda omogućavaju da odvojimo opšti način izračunavanja od konkretne operacije koju treba izvršiti. Funkciju koja predstavlja tu operaciju možemo proslediti kao imenovanu funkciju, lambda izraz ili odgovarajuću delimičnu primenu operatora.

Ova ideja posebno je korisna pri radu sa listama. U narednoj podsekciji upoznaćemo funkcije `map`, `filter` i `zipWith`, kojima ćemo prosledivati druge funkcije da bismo na sažet način obrađivali elemente listi.

9.2 Funkcije `map`, `filter` i `zipWith`

Pri obradi listi često možemo koristiti funkcije višeg reda, kojima prosleđujemo funkciju koju želimo da primenimo.

Jedna od najvažnijih takvih funkcija jeste `map`. Njen tip je

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

Funkcija `map` kao prvi argument prima funkciju tipa `a -> b`, a kao drugi argument listu elemenata tipa `a`. Zadatu funkciju primenjuje na svaki element liste i vraća novu listu elemenata tipa `b`.

Na primer:

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

Funkcija `(*2)` primenjuje se redom na svaki element početne liste. Rezultat je nova lista u kojoj je svaki element udvostručen.

Istu operaciju smo u prethodnoj sekciji zapisali pomoću rekurzivne funkcije:

```
udvostruciSve :: Num a => [a] -> [a]
```

```
udvostruciSve [] = []
udvostruciSve (prvi:ostatak) =
  (2 * prvi) : udvostruciSve ostatak
```

Pomoću funkcije `map` istu ideju možemo zapisati kraće:

```
udvostruciSve :: Num a => [a] -> [a]
udvostruciSve lista = map (*2) lista
```

Funkcija prosleđena funkciji `map` ne mora da vraća vrednost istog tipa koji prima. Na primer:

```
ghci> map length ["Ana", "Marko", "Jovan"]
[3,5,5]
```

Ovde funkcija `length` prima tekstualnu vrednost, odnosno listu karaktera, a vraća broj njenih elemenata. Zato rezultat više nije lista tekstualnih vrednosti, već lista brojeva.

Možemo proslediti i lambda izraz:

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

Funkcija `map` uvek obrađuje svaki element početne liste i za svaki od njih proizvodi jedan element rezultujuće liste.

Druga važna funkcija višeg reda jeste `filter`. Njen tip je

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

Prvi argument funkcije `filter` jeste funkcija koja za jedan element vraća logičku vrednost. Takva funkcija naziva se predikat. Funkcija `filter` zadržava samo one elemente liste za koje predikat vraća `True`.

Na primer, definišimo funkciju koja proverava da li je ceo broj paran:

```
jeParan :: Int -> Bool
jeParan broj = broj `mod` 2 == 0
```

Sada pomoću nje možemo izdvojiti parne brojeve:

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

Za svaki element početne liste funkcija `filter` proverava vrednost predikata `jeParan`. Element se pojavljuje u rezultatu samo ako je ta vrednost `True`.

Kao predikat možemo koristiti i kratku funkciju zadatu neposredno:

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

Zapis `(>5)` predstavlja funkciju koja proverava da li je njen argument veći od 5.

Isti rezultat mogli smo ranije dobiti pomoću izlistavanja:

```
[broj | broj <- [1..10], broj > 5]
```

Funkcije `filter` i izlistavanje tako mogu izraziti slične načine izbora elemenata liste.

Važno je razlikovati uloge funkcija `map` i `filter`. Funkcija `map` transformiše svaki element početne liste, dok `filter` ne menja elemente već bira koje od njih treba zadržati.

Na primer:

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

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

Još jedna korisna funkcija višeg reda jeste `zipWith`. Ona istovremeno obrađuje dve liste. Njen tip je

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

Prvi argument funkcije `zipWith` jeste funkcija koja prima po jedan element iz svake od dve liste i od njih pravi jedan rezultat. Funkcija `zipWith` tu funkciju primenjuje na odgovarajuće elemente dve liste.

Na primer:

```
ghci> zipWith (+) [1, 2, 3] [10, 20, 30]
[11,22,33]
```

Prvo se sabiraju elementi 1 i 10, zatim 2 i 20, a potom 3 i 30.

Na isti način možemo množiti odgovarajuće elemente:

```
ghci> zipWith (*) [1, 2, 3] [4, 5, 6]
[4,10,18]
```

Funkcija koju prosleđujemo funkciji `zipWith` može biti i lambda izraz. Na primer:

```
ghci> zipWith (\broj tekst -> tekst ++ show broj) [1, 2, 3] ["A", "B", "C"]
["A1", "B2", "C3"]
```

U ovom primeru elementi dve početne liste nisu istog tipa. Prva lista sadrži brojeve, a druga tekstualne vrednosti. Lambda funkcija od jednog broja i jedne tekstualne vrednosti pravi novu tekstualnu vrednost.

Ako dve liste nemaju isti broj elemenata, funkcija `zipWith` završava obradu kada se potroši kraća lista:

```
ghci> zipWith (+) [1, 2, 3] [10, 20]
[11,22]
```

Zato rezultujuća lista ima onoliko elemenata koliko ih ima kraća od dve početne liste.

Funkcije `map`, `filter` i `zipWith` predstavljaju različite načine obrade listi. Funkcija `map` transformiše svaki element liste, `filter` bira elemente koji zadovoljavaju određeni uslov, dok `zipWith` kombinuje odgovarajuće elemente dve liste. Zajedničko im je da kao rezultat ponovo dobijamo listu.

Postoje, međutim, i obrade kod kojih od svih elemenata liste želimo da dobijemo jedan rezultat. Na primer, možemo želeći da izračunamo zbir ili proizvod svih elemenata. Takve probleme smo već rešavali pomoću rekurzivnih funkcija. U nastavku ćemo upoznati funkcije `foldr` i `foldl`, koje omogućavaju da se ovakve obrade zapišu na opštiji način.

9.3 Funkcije foldr i foldl

U prethodnoj sekciji definisali smo rekurzivnu funkciju koja računa zbir elemenata liste:

```
zbirListe :: Num a => [a] -> a
zbirListe [] = 0
zbirListe (prvi:ostatak) =
    prvi + zbirListe ostatak
```

Ova funkcija praznoj listi pridružuje vrednost 0, dok kod neprazne liste prvi element sabira sa rezultatom dobijenim obradom ostatka liste.

Ovakav način obrade liste često se pojavljuje i može se izdvojiti u opštiji oblik pomoću funkcije `foldr`. Na primer:

```
ghci> foldr (+) 0 [1, 2, 3]
6
```

Ovaj izraz odgovara izrazu

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

Funkcija `foldr` prima funkciju kojom se elementi kombinuju, početnu vrednost i listu. Primene zadate funkcije grupišu se zdesna.

Njen opšti tip je

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

Prvi argument je funkcija koja kombinuje jedan element liste tipa `a` sa već dobijenim rezultatom tipa `b`. Drugi argument predstavlja početnu vrednost tipa `b`, a treći argument je lista elemenata tipa `a`. Konačan rezultat takođe je tipa `b`.

Prethodnu funkciju za računanje zbira zato možemo zapisati i pomoću funkcije `foldr`:

```
zbirDesno :: Num a => [a] -> a
zbirDesno lista = foldr (+) 0 lista
```

Na primer:

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

Početna vrednost zavisi od operacije koju koristimo. Kod sabiranja prirodna početna vrednost je 0, dok je kod množenja to 1:

```
ghci> foldr (*) 1 [2, 3, 4]
24
```

Početna vrednost određuje i rezultat za praznu listu:

```
ghci> foldr (+) 0 []
0

ghci> foldr (*) 1 []
1
```

Sličnu ulogu ima funkcija `foldl`, ali ona elemente kombinuje na drugačiji način. Na primer:

```
ghci> foldl (+) 0 [1, 2, 3]
6
```

Ovaj izraz odgovara izrazu

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

Kod funkcije `foldl` počinjemo od početne vrednosti i redom je kombinujemo sa elementima liste.

Opšti tip funkcije `foldl` je

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

Prvi argument je funkcija koja kombinuje trenutni rezultat tipa `b` sa narednim elementom liste tipa `a` i vraća novi rezultat tipa `b`. Zatim se navode početna vrednost i lista.

Zbir elemenata liste možemo zato zapisati i pomoću funkcije `foldl`:

```
zbirLevo :: Num a => [a] -> a
zbirLevo lista = foldl (+) 0 lista
```

Na primer:

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

Kod sabiranja funkcije `foldr` i `foldl` daju isti rezultat:

```
foldr (+) 0 [1, 2, 3]
= 1 + (2 + (3 + 0))
= 6

foldl (+) 0 [1, 2, 3]
= ((0 + 1) + 2) + 3
= 6
```

Međutim, isti rezultat ne moramo dobiti za svaku operaciju. To se jasno vidi kod oduzimanja:

```
ghci> foldr (-) 0 [5, 2, 1]
4

ghci> foldl (-) 0 [5, 2, 1]
-8
```

Kod funkcije `foldr` dobijamo

```
foldr (-) 0 [5, 2, 1]
= 5 - (2 - (1 - 0))
= 4
```

Dok kod funkcije `foldl` dobijamo

```
foldl (-) 0 [5, 2, 1]
```

```
= ((0 - 5) - 2) - 1
= -8
```

Razlika nastaje zato što se primene funkcije drugačije grupišu. Kod sabiranja to ne menja rezultat, dok kod oduzimanja menja.

Funkcije `foldr` i `foldl` tako predstavljaju opšte načine za postepeno kombinovanje elemenata liste u jedan rezultat. Kao i funkcije `map` i `filter`, omogućavaju da česte obrasce obrade listi koristimo bez ponovnog pisanja odgovarajuće rekurzije.

U poslednjoj podsekciji razmotrićemo još jedan važan način rada sa funkcijama. Videćemo kako se postojeće funkcije mogu povezivati u nove funkcije pomoću kompozicije.

9.4 Kompozicija funkcija

Funkcije možemo povezivati tako da rezultat jedne funkcije postane argument druge. Na primer, definišimo dve jednostavne funkcije:

```
udvostruci :: Int -> Int
udvostruci broj = 2 * broj

povecajZaJedan :: Int -> Int
povecajZaJedan broj = broj + 1
```

Ako želimo da broj najpre udvostručimo, a zatim povećamo za 1, možemo napisati:

```
ghci> povecajZaJedan (udvostruci 5)
11
```

Izraz `udvostruci 5` ima vrednost 10, pa je ceo izraz ekvivalentan izrazu `povecajZaJedan 10`, čija je vrednost 11.

Haskell omogućava da ovakvo povezivanje funkcija zapišemo pomoću operatora kompozicije `..`. Prethodni primer možemo zapisati i ovako:

```
ghci> (povecajZaJedan . udvostruci) 5
11
```

Izraz

```
povecajZaJedan . udvostruci
```

predstavlja novu funkciju. Ona najpre primenjuje funkciju `udvostruci`, a zatim na dobijeni rezultat primenjuje funkciju `povecajZaJedan`.

Opšta ideja kompozicije može se zapisati kao

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

Kod kompozicije `f . g` najpre se primenjuje funkcija `g`, a zatim funkcija `f`. Zbog toga je korisno kompoziciju čitati zdesna nalevo.

Da bi dve funkcije mogle da se povežu kompozicijom, rezultat funkcije koja se primenjuje prva mora biti odgovarajući argument funkcije koja se primenjuje posle nje.

Pomoću kompozicije možemo definisati i novu funkciju:

```
obradi :: Int -> Int
```

```
obradi broj = (povecajZaJedan . udvostruci) broj
```

Na primer:

```
ghci> obradi 5
11

ghci> obradi 10
21
```

Kompozicija je posebno korisna kada povezujemo više koraka obrade listi. Definišimo najpre funkciju za kvadriranje broja:

```
kvadriraj :: Int -> Int
kvadriraj broj = broj * broj
```

Pretpostavimo da iz liste želimo da izdvojimo parne brojeve, zatim ih kvadiramo i na kraju izračunamo njihov zbir. To možemo zapisati ovako:

```
zbirKvadrataParnih :: [Int] -> Int
zbirKvadrataParnih lista = sum (map kvadriraj (filter even lista))
```

Na primer, za listu [1,2,3,4,5] redom dobijamo:

```
filter even [1,2,3,4,5]
= [2,4]

map kvadriraj [2,4]
= [4,16]

sum [4,16]
= 20
```

Istu obradu možemo zapisati pomoću kompozicije:

```
zbirKvadrataParnih :: [Int] -> Int
zbirKvadrataParnih lista = (sum . map kvadriraj . filter even) lista
```

Funkcije ponovo čitamo zdesna nalevo. Najpre se pomoću `filter even` izdvajaju parni brojevi, zatim se pomoću `map kvadriraj` svaki od njih kvadrira, a funkcija `sum` na kraju računa njihov zbir.

Pri pisanju izraza u Haskellu često se koristi i operator `$`. On omogućava da se neke zagrade izostave. Na primer, izrazi

```
sum (map (*2) [1, 2, 3])
```

i

```
sum $ map (*2) [1, 2, 3]
```

imaju isto značenje.

Na primer:

```
ghci> sum $ map (*2) [1, 2, 3]
12
```

Operator `$` nije operator kompozicije. On samo omogućava da izraz sa njegove desne strane prosledimo funkciji sa leve strane bez dodatnih zagrada.

Kompozicija omogućava da složeniju obradu izgradimo povezivanjem jednostavnijih funkcija. Zajedno sa funkcijama `map`, `filter`, `zipWith`, `foldr` i `foldl`, ona pokazuje kako se jednostavne funkcije mogu povezivati i koristiti za izražavanje složenijih obrada.

Time se jasno vidi jedna od važnih osobina funkcionalnog programiranja. Funkcije se ne koriste samo za neposredno izračunavanje rezultata, već se mogu prosleđivati drugim funkcijama i međusobno povezivati.

10 Definisanje novih tipova u Haskellu

10.1 Tipski sinonimi

Tipovi koje koristimo u programu ponekad mogu imati složeniji zapis ili njihovo značenje iz samog zapisa nije odmah očigledno. U takvim slučajevima postojećem tipu možemo dati novo, opisnije ime.

Za to se u Haskellu koristi ključna reč `type`. Opšti oblik definicije je

```
type NovoIme = PostojeciTip
```

Novo ime tipa počinje velikim slovom.

Na primer, možemo uvesti posebno ime za tekstualnu vrednost koja predstavlja ime osobe:

```
type Ime = String
```

Nakon toga ime `Ime` možemo koristiti u potpisima funkcija:

```
pozdrav :: Ime -> String
pozdrav ime = "Zdravo, " ++ ime
```

Na primer:

```
ghci> pozdrav "Ana"
"Zdravo, Ana"
```

Tip `Ime` ovde nije novi tip različit od tipa `String`. To je samo drugo ime za isti tip.

Tipski sinonim možemo koristiti i za složeniji postojeći tip. Na primer, listi celih brojeva možemo dati opisnije ime:

```
type Brojevi = [Int]
```

Takav naziv zatim možemo koristiti u potpisima funkcija:

```
zbirBrojeva :: Brojevi -> Int
zbirBrojeva brojevi = sum brojevi
```

Na primer:

```
ghci> zbirBrojeva [2, 4, 6]
12
```

Tip `Brojevi` ovde nije novi tip. On je samo drugo ime za tip `[Int]`, pa funkciji možemo neposredno proslediti običnu listu celih brojeva.

Sa jednim poznatim tipskim sinonimom već smo se susreli. U Haskellu je `String` definisan kao sinonim za listu karaktera, odnosno za tip `[Char]`. Zbog toga tekstualne vrednosti možemo obrađivati kao liste karaktera.

Tipski sinonim, dakle, ne uvodi nove moguće vrednosti i ne razlikuje se od tipa na koji se odnosi. Njegova osnovna uloga jeste da kod učini čitljivijim i jasnije izrazi značenje podataka.

Kada želimo da zaista definišemo novi tip sa sopstvenim mogućim vrednostima, tipski sinonim nije dovoljan. Za to se u Haskellu koristi deklaracija `data`, koju ćemo upoznati u narednoj podsekciji.

10.2 Novi tipovi i konstruktori

Tipski sinonim omogućava da postojećem tipu damo drugo ime, ali time ne nastaje novi tip. Kada želimo da definišemo zaista novi tip i odredimo koje vrednosti tog tipa mogu da postoje, koristimo deklaraciju `data`.

Najjednostavniji primer može biti tip koji predstavlja dane u nedelji:

```
data Dan =  
    Ponedeljak  
  | Utorak  
  | Sreda  
  | Cetvrtak  
  | Petak  
  | Subota  
  | Nedelja  
deriving (Show)
```

Ime novog tipa je `Dan`. Kao i druga imena tipova u Haskellu, ono počinje velikim slovom.

Vrednosti ovog tipa predstavljene su konstruktorima `Ponedeljak`, `Utorak`, `Sreda` i ostalim navedenim imenima. Konstruktori takođe počinju velikim slovom.

Znak `|` razdvaja različite moguće oblike vrednosti novog tipa. Tako vrednost tipa `Dan` može biti jedna od sedam navedenih vrednosti.

Dodatak

```
deriving (Show)
```

omogućava da se vrednosti novog tipa prikazuju u interaktivnom okruženju.

Na primer:

```
ghci> Ponedeljak  
Ponedeljak  
  
ghci> Subota  
Subota
```

Pošto su konstruktori različiti mogući oblici vrednosti, možemo ih koristiti u podudaranju oblika. Na primer, možemo definisati funkciju koja proverava da li određeni dan pripada vikendu:

```
jeVikend :: Dan -> Bool  
jeVikend Subota = True  
jeVikend Nedelja = True  
jeVikend _ = False
```

Na primer:

```
ghci> jeVikend Subota  
True  
  
ghci> jeVikend Sreda  
False
```

U prve dve definicije funkcije navodimo konkretne konstruktore, dok poslednji obrazac obuhvata sve ostale dane.

Konstruktori ne moraju predstavljati samo pojedinačne vrednosti. Oni mogu sadržati i dodatne podatke.

Na primer, možemo definisati tip kojim predstavljamo različite geometrijske oblike:

```
data Oblik =  
    Krug Double  
  | Pravougaonik Double Double  
  deriving (Show)
```

Tip `Oblik` ima dva konstruktora. Konstruktor `Krug` sadrži jednu vrednost tipa `Double`, koja može predstavljati poluprečnik. Konstruktor `Pravougaonik` sadrži dve vrednosti tipa `Double`, koje mogu predstavljati dužine njegovih stranica.

Na primer:

```
ghci> Krug 2.0  
Krug 2.0  
  
ghci> Pravougaonik 3.0 4.0  
Pravougaonik 3.0 4.0
```

Izrazi `Krug 2.0` i `Pravougaonik 3.0 4.0` predstavljaju vrednosti tipa `Oblik`. Imena `Krug` i `Pravougaonik` nisu posebni tipovi, već konstruktori vrednosti tipa `Oblik`.

Konstruktoe možemo koristiti i u podudaranju oblika. Na primer, površinu oblika možemo izračunati ovako:

```
povrsina :: Oblik -> Double  
povrsina (Krug r) = pi * r * r  
povrsina (Pravougaonik a b) = a * b
```

U obrascu

```
Krug r
```

ime `r` dobija vrednost sačuvanu u konstruktoru `Krug`. Slično tome, u obrascu

```
Pravougaonik a b
```

imena `a` i `b` dobijaju dve vrednosti sačuvane u konstruktoru `Pravougaonik`.

Na primer:

```
ghci> povrsina (Krug 2.0)  
12.566370614359172  
  
ghci> povrsina (Pravougaonik 3.0 4.0)  
12.0
```

Na ovaj način jedna funkcija može različito da obradi različite oblike vrednosti istog tipa.

Važno je razlikovati deklaracije `type` i `data`. Deklaracija `type` samo uvodi novo ime za već postojeći tip. Deklaracija `data` definiše novi tip i određuje konstruktore pomoću kojih se njegove vrednosti grade.

Konstruktori mogu predstavljati pojedinačne vrednosti, kao kod tipa `Dan`, ali mogu i sadržati dodatne podatke, kao kod tipa `Oblik`. U narednoj podsekciji videćemo da novi tip može biti definisan i tako da ne zavisi od jednog konkretnog tipa podataka, već da sadrži tipski parametar.

10.3 Parametrizovani tipovi i tip `Maybe`

Tipovi koje smo do sada sami definisali bili su vezani za konkretne vrste podataka. Na primer, konstruktori tipa `Oblik` sadržali su vrednosti tipa `Double`.

Novi tip, međutim, ne mora unapred biti vezan za jedan konkretan tip. Možemo definisati parametrizovani tip, koji sadrži tipski parametar.

Na primer:

```
data Par a = NapraviPar a a
  deriving (Show)
```

Slovo `a` predstavlja tipski parametar. Ono označava da konstruktor `NapraviPar` može sadržati dve vrednosti nekog tipa, pri čemu obe vrednosti moraju imati isti tip.

Na primer:

```
brojevi :: Par Int
brojevi = NapraviPar 3 5

imena :: Par String
imena = NapraviPar "Ana" "Marko"

odgovori :: Par Bool
odgovori = NapraviPar True False
```

U prvom primeru tipski parametar `a` predstavlja tip `Int`, pa dobijamo tip `Par Int`. U drugom primeru dobijamo `Par String`, a u trećem `Par Bool`.

Vrednosti možemo prikazati i u interaktivnom okruženju:

```
ghci> brojevi
NapraviPar 3 5

ghci> imena
NapraviPar "Ana" "Marko"
```

Na ovaj način jednom definicijom određujemo oblik podataka koji možemo koristiti sa različitim tipovima vrednosti.

Jedan od najvažnijih parametrizovanih tipova koji su već definisani u Haskellu jeste `Maybe`. Njegovu definiciju možemo posmatrati ovako:

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

Ovu deklaraciju ne moramo sami pisati, jer je tip `Maybe` već dostupan. Prikazujemo je da bismo razumeli njegovu strukturu.

Tip `Maybe a` predstavlja rezultat koji može da postoji, ali i ne mora. Konstruktor `Nothing` označava odsustvo vrednosti, dok konstruktor `Just` sadrži jednu vrednost tipa `a`.

Tako vrednost tipa `Maybe Int` može biti `Nothing` ili, na primer, `Just 5`.

Slično tome, vrednost tipa `Maybe String` može biti `Nothing` ili vrednost kao što je `Just "Ana"`.

Tip `Maybe` posebno je koristan kada funkcija za neke argumente možda nema rezultat.

Ranije smo, pri upoznavanju podudaranja oblika nad listama, definisali funkciju koja za praznu listu vraća 0:

```
prviIliNula :: [Int] -> Int
prviIliNula [] = 0
prviIliNula (prvi:_) = prvi
```

Takva funkcija bila je pogodna za objašnjenje podudaranja oblika, ali vrednost 0 ne govori da li je lista bila prazna ili je njen prvi element zaista bio jednak nuli.

Pomoću tipa `Maybe` možemo jasnije izraziti mogućnost da prvi element ne postoji:

```
prviMozda :: [a] -> Maybe a
prviMozda [] = Nothing
prviMozda (prvi:_) = Just prvi
```

Na primer:

```
ghci> prviMozda []
Nothing

ghci> prviMozda [10, 20, 30]
Just 10

ghci> prviMozda "Ana"
Just 'A'
```

Rezultat `Nothing` jasno pokazuje da vrednost ne postoji. Rezultat `Just 10` pokazuje da vrednost postoji i da je jednaka 10.

Konstruktor `Nothing` i `Just` možemo koristiti i u podudaranju oblika. Na primer:

```
vrednostIliNula :: Maybe Int -> Int
vrednostIliNula Nothing = 0
vrednostIliNula (Just broj) = broj
```

Na primer:

```
ghci> vrednostIliNula Nothing
0

ghci> vrednostIliNula (Just 7)
7
```

U prvom obrascu posebno obrađujemo odsustvo vrednosti. U drugom konstruktor `Just` omogućava da izdvojimo sadržanu vrednost i damo joj ime `broj`.

Još jedan jednostavan primer jeste deljenje celih brojeva. Deljenje sa nulom nema rezultat, pa funkcija može vratiti vrednost tipa `Maybe Int`:

```
bezbednoDeljenje :: Int -> Int -> Maybe Int
bezbednoDeljenje _ 0 = Nothing
```

```
bezbednoDeljenje a b = Just (a `div` b)
```

Na primer:

```
ghci> bezbednoDeljenje 20 4
Just 5
```

```
ghci> bezbednoDeljenje 20 0
Nothing
```

Umesto da za poseban slučaj biramo neku proizvoljnu broječanu vrednost, tipom `Maybe` neposredno izražavamo da rezultat možda ne postoji.

Parametrizovani tipovi omogućavaju da isti oblik podataka koristimo sa različitim tipovima vrednosti. Tip `Maybe` predstavlja posebno važan primer, jer na jasan način razlikuje postojanje vrednosti od njenog odsustva.

U poslednjoj podsekciji videćemo još jedan pogodan način definisanja sopstvenih tipova, kod koga podaci unutar vrednosti imaju svoja imena.

10.4 Tipovi sa imenovanim poljima

Konstruktori novih tipova mogu sadržati više podataka. Kada takvih podataka ima nekoliko, nije uvek dovoljno jasno šta svaki od njih predstavlja.

Na primer, podatke o studentu mogli bismo predstaviti konstruktorom koji sadrži ime i prosek. Haskell omogućava da tim podacima damo posebna imena pomoću zapisa sa imenovanim poljima (engl. *record*).

Na primer:

```
data Student = Student
  { ime :: String
  , prosek :: Double
  }
  deriving (Show)
```

Ovde definišemo novi tip `Student`. Njegov konstruktor takođe se zove `Student`, a sadrži dva imenovana polja.

Polje `ime` ima tip `String`, dok polje `prosek` ima tip `Double`.

Konkretnu vrednost možemo napraviti navođenjem vrednosti odgovarajućih polja:

```
student1 :: Student
student1 = Student
  { ime = "Ana"
  , prosek = 9.2
  }
```

Vrednost možemo prikazati u interaktivnom okruženju:

```
ghci> student1
Student {ime = "Ana", prosek = 9.2}
```

Dodatak `deriving (Show)` omogućava da se vrednosti tipa `Student` prikažu na ovaj način.

Za svako imenovano polje Haskell automatski pravi funkciju kojom možemo pristupiti njegovoj vrednosti. Zato možemo pisati:

```
ghci> ime student1
"Ana"

ghci> prosek student1
9.2
```

Funkcija `ime` prima vrednost tipa `Student` i vraća njeno polje tipa `String`. Slično tome, funkcija `prosek` vraća vrednost odgovarajućeg polja tipa `Double`.

Imenovana polja možemo koristiti i u drugim funkcijama. Na primer:

```
uspesan :: Student -> Bool
uspesan student = prosek student >= 8.5
```

Na primer:

```
ghci> uspesan student1
True
```

Ako želimo vrednost koja se od postojeće razlikuje samo u jednom polju, možemo koristiti zapis za izmenu polja.

Na primer:

```
student2 :: Student
student2 = student1 { prosek = 9.5 }
```

Na ovaj način ne menjamo vrednost `student1`. Umesto toga formiramo novu vrednost `student2`, kod koje je promenjeno samo polje `prosek`.

To možemo proveriti u interaktivnom okruženju:

```
ghci> student1
Student {ime = "Ana", prosek = 9.2}

ghci> student2
Student {ime = "Ana", prosek = 9.5}
```

Početna vrednost ostaje nepromenjena, što je u skladu sa načinom rada sa vrednostima koji smo već upoznali u Haskellu.

Imenovana polja posebno su korisna kada jedan konstruktor sadrži više različitih podataka. Umesto da se oslanjamo samo na njihov redosled, svakom podatku možemo dati ime koje jasno opisuje njegovo značenje.

U ovoj sekciji upoznali smo nekoliko načina za opisivanje podataka u Haskellu. Deklaracija `type` daje novo ime postojećem tipu, dok deklaracija `data` omogućava definisanje zaista novih tipova i njihovih konstruktora. Parametrizovani tipovi omogućavaju da isti oblik podataka koristimo sa različitim tipovima vrednosti, a imenovana polja omogućavaju da podaci unutar konstruktora budu jasnije označeni.

11 Poređenje programskih paradigmi

11.1 Različiti pristupi rešavanju problema

Isti programski problem često se može rešiti na više različitih načina. Razlike između tih rešenja ne moraju biti samo u sintaksi programskog jezika, već i u načinu na koji organizujemo podatke, operacije i tok izračunavanja.

Programska paradigma predstavlja opšti način razmišljanja o programu i organizovanja njegovog rešenja. Različite paradigme naglašavaju različite ideje, iako se njima često mogu rešavati isti ili veoma slični problemi.

To možemo videti na jednostavnom primeru. Pretpostavimo da želimo da iz liste brojeva izdvojimo parne brojeve, izračunamo njihove kvadrate i na kraju saberemo dobijene vrednosti.

U Pythonu ovaj problem možemo rešiti proceduralnim pristupom:

```
brojevi = [1, 2, 3, 4, 5]
zbir = 0

for broj in brojevi:
    if broj % 2 == 0:
        zbir = zbir + broj * broj

print(zbir)
```

Rezultat je

20

U ovom rešenju postupak opisujemo korak po korak. Najpre postavljamo početnu vrednost promenljive `zbir`. Zatim prolazimo kroz elemente liste, proveravamo uslov i po potrebi menjamo vrednost promenljive.

Isti problem u Haskellu možemo rešiti funkcionalnim pristupom:

```
kvadriraj :: Int -> Int
kvadriraj broj = broj * broj

zbirKvadrataParnih :: [Int] -> Int
zbirKvadrataParnih lista = sum (map kvadriraj (filter even lista))
```

Na primer:

```
ghci> zbirKvadrataParnih [1, 2, 3, 4, 5]
20
```

Ovde isti postupak izražavamo kao povezivanje nekoliko operacija nad listom. Funkcija `filter` izdvaja parne brojeve, `map` ih kvadrira, a `sum` sabira dobijene vrednosti.

Oba programa rešavaju isti problem i daju isti rezultat, ali je način organizovanja rešenja različit. U Python primeru naglasak je na nizu koraka kroz koje se stanje programa menja. U Haskell primeru naglasak je na funkcijama koje postojeće podatke pretvaraju u nove vrednosti.

Ovakve razlike predstavljaju jednu od osnovnih ideja programskih paradigmi. Paradigma ne određuje samo kako se neka naredba zapisuje, već utiče na to koje strukture i načine razmišljanja prirodno koristimo pri rešavanju problema.

To ne znači da je jedan pristup uvek bolji od drugog. Različiti problemi i različiti programski jezici mogu pogodovati različitim načinima organizovanja rešenja.

U nastavku ćemo detaljnije uporediti proceduralni, objektno-orijentisani i funkcionalni pristup, koje smo upoznali kroz Python i Haskell.

11.2 Proceduralni, objektno-orijentisani i funkcionalni pristup

Različite programske paradigme stavljaju naglasak na različite načine organizovanja programa. U prethodnim delovima upoznali smo proceduralni i objektno-orijentisani pristup kroz Python, dok smo funkcionalni pristup detaljnije razmatrali kroz Haskell.

U proceduralnom pristupu problem se najčešće razlaže na postupak koji treba izvršiti korak po korak. Program se organizuje pomoću naredbi i funkcija, a redosled njihovog izvršavanja ima važnu ulogu.

U Python primerima koristili smo promenljive čije su se vrednosti menjale tokom izvršavanja programa. Za izbor između različitih mogućnosti koristili smo grananja, dok smo ponavljanje izražavali petljama `for` i `while`.

Na primer:

```
brojevi = [1, 2, 3, 4]
zbir = 0

for broj in brojevi:
    zbir = zbir + broj
```

U svakom prolasku kroz petlju menja se vrednost promenljive `zbir`. Na kraju ona sadrži rezultat čitavog postupka.

Za proceduralni pristup zato je karakteristično da jasno opisujemo koje korake program treba da izvrši i kojim redosledom.

Objektno-orijentisani pristup organizuje program na drugačiji način. U njegovom središtu nalaze se objekti koji objedinjuju podatke i operacije nad tim podacima.

U Pythonu smo objekte pravili na osnovu klasa. Atributi su predstavljali podatke objekta, dok su metode opisivale operacije koje objekat može da izvršava.

Na primer, objekat koji predstavlja račun mogao bi da sadrži podatak o stanju računa i metode kojima se nad tim stanjem izvršavaju odgovarajuće operacije.

Objektno-orijentisano programiranje omogućava da se podaci i ponašanje koje je sa njima povezano organizuju na jednom mestu. Upoznali smo i nasleđivanje, kojim jedna klasa može preuzeti osobine druge klase, kao i polimorfizam, koji omogućava da se sa objektima različitih klasa radi na zajednički način.

Kod većih programa ovakav pristup omogućava da sistem posmatramo kao skup objekata koji imaju svoje podatke, ponašanje i međusobne odnose.

U funkcionalnom pristupu centralnu ulogu imaju funkcije i izrazi. Umesto da se izračunavanje prvenstveno posmatra kao niz naredbi koje menjaju stanje programa, rezultat se često dobija povezivanjem funkcija i pretvaranjem postojećih vrednosti u nove vrednosti.

U Haskellu smo videli da jednom definisana vrednost ne menja svoje značenje. Kada želimo drugačiji rezultat, formiramo novu vrednost umesto da menjamo postojeću.

Ponavljanje smo često izražavali rekurzijom. Kasnije smo upoznali funkcije višeg reda, kojima se druge funkcije mogu prosledivati kao argumenti. Tako smo pomoću funkcija `map`, `filter`, `foldr` i `foldl` izražavali česte oblike obrade listi bez eksplicitnog pisanja petlji.

Na primer:

```
filter even [1..10]
map (*2) [1, 2, 3, 4]
foldr (+) 0 [1, 2, 3, 4]
```

Videli smo i da se jednostavnije funkcije mogu povezivati kompozicijom i tako koristiti za izgradnju složenijih obrada.

Razlike između ovih pristupa najviše se vide u tome šta se nalazi u središtu programa i kako se opisuje izračunavanje. U proceduralnom pristupu naglasak je na nizu koraka koje treba izvršiti. Vrednosti promenljivih mogu se menjati, a grananja i petlje određuju tok izvršavanja programa.

U objektno-orijentisanom pristupu program se prvenstveno organizuje oko objekata. Podaci i operacije nad njima povezuju se pomoću klasa, atributa i metoda, dok nasleđivanje i polimorfizam omogućavaju povezivanje različitih vrsta objekata.

U funkcionalnom pristupu naglasak je na funkcijama i izrazima. Postojeće vrednosti se koriste za dobijanje novih vrednosti, a funkcije se mogu prosleđivati drugim funkcijama i međusobno povezivati. Rekurzija i funkcije višeg reda često preuzimaju ulogu koju u proceduralnom kodu imaju petlje.

Ove razlike ne znače da svaki program mora pripadati samo jednoj paradigmi. One pre svega predstavljaju različite načine razmišljanja o problemu i organizovanja njegovog rešenja. Isti programski jezik može podržavati više takvih pristupa, a njihove ideje mogu se koristiti i zajedno.

Nijedan od ovih pristupa nije najbolji za svaki problem. Svaki od njih ima situacije u kojima prirodno vodi do jednostavnog i preglednog rešenja, ali i situacije u kojima može biti manje pogodan. Njihove prednosti i ograničenja razmotrićemo u narednoj podsekciji.

11.3 Prednosti i ograničenja različitih pristupa

Nijedna programska paradigma nije najbolja za sve vrste problema. Različiti pristupi imaju osobine koje mogu biti veoma korisne u jednoj situaciji, ali manje pogodne u nekoj drugoj.

Proceduralni pristup često je jednostavan i neposredan kada problem prirodno možemo opisati kao niz koraka. Kod manjih programa tok izvršavanja obično je lako pratiti, jer se jasno vidi kojim redosledom se pojedine operacije izvršavaju.

Sa povećanjem programa, međutim, može postati teže pratiti veliki broj promenljivih čije se vrednosti menjaju na različitim mestima. Ako više delova programa utiče na isto stanje, razumevanje posledica pojedinačnih promena može postati složenije.

Objektno-orijentisani pristup posebno je pogodan kada problem sadrži jasno prepoznatljive entitete koje možemo predstaviti objektima. Povezivanje podataka i operacija nad njima može pomoći da se veliki program podeli na manje i preglednije celine.

Nasleđivanje i polimorfizam mogu dodatno olakšati organizovanje srodnih vrsta objekata i ponovno korišćenje zajedničkog ponašanja. Takva organizacija često je korisna kod većih sistema u kojima postoji veliki broj međusobno povezanih objekata.

Sa druge strane, nije svaki problem prirodno objektno-orijentisan. Uvođenje velikog broja klasa za jednostavan problem može samo povećati količinu koda i otežati njegovo razumevanje. Složene hijerarhije nasleđivanja takođe mogu postati teške za praćenje.

Funkcionalni pristup naglašava izračunavanje pomoću funkcija i formiranje novih vrednosti umesto menjanja postojećih. Zbog toga je često lakše posmatrati pojedinačnu funkciju nezavisno od ostatka programa i razumeti od čega njen rezultat zavisi.

Čiste funkcije posebno su pogodne za takvo razmišljanje, jer za iste argumente daju iste rezultate i ne menjaju stanje izvan same funkcije. Funkcije višeg reda i kompozicija omogućavaju i da se složenije obrade grade povezivanjem jednostavnijih funkcija.

Ovakav pristup može biti veoma pogodan pri transformaciji i obradi podataka. Sa druge strane, rekurzija, funkcije višeg reda i apstraktniji tipovi mogu u početku biti manje intuitivni programeru koji je navikao na promenljive i petlje. I problemi koji se prirodno zamišljaju kroz niz promena stanja mogu zahtevati drugačiji način razmišljanja.

Prednosti i ograničenja pojedinih paradigmi zato ne treba posmatrati potpuno odvojeno od problema koji rešavamo. Osobina koja u jednom programu dovodi do jednostavnog i preglednog rešenja u drugom programu možda neće biti naročito važna.

U praksi se zato ne bira uvek jedna paradigma koja će se koristiti u svakoj situaciji. Važnu ulogu imaju priroda problema, programski jezik koji koristimo i mogućnosti koje taj jezik pruža. O tome ćemo govoriti u poslednjoj podsekciji.

11.4 Izbor paradigme u savremenim programskim jezicima

Izbor načina na koji ćemo organizovati program zavisi pre svega od problema koji rešavamo. Ne postoji jedna programska paradigma koja predstavlja najbolji izbor u svim situacijama.

Ako se problem prirodno može opisati kao niz koraka koje treba izvršiti određenim redosledom, proceduralni pristup može dovesti do jednostavnog i preglednog rešenja. Kada program modeluje veći broj entiteta koji imaju svoje podatke i ponašanje, objektno-orijentisani pristup može biti prirodniji. Kod problema koji se mogu izraziti kroz transformacije podataka i povezivanje funkcija, funkcionalni pristup često pruža pogodan način organizovanja rešenja.

Na izbor utiče i programski jezik koji koristimo. Programski jezik i programska paradigma, međutim, nisu isto. Jedan jezik može podržavati više različitih načina programiranja, dok neki jezici posebno podstiču određeni pristup.

Python je dobar primer jezika u kome se mogu kombinovati različiti stilovi programiranja. U njemu smo najpre koristili proceduralni pristup, sa naredbama, promenljivama, grananjima, petljama i funkcijama. Zatim smo isti jezik koristili za objektno-orijentisano programiranje, definišući klase i objekte sa atributima i metodama. Python podržava i pojedine ideje karakteristične za funkcionalno programiranje.

Haskell je, sa druge strane, snažno usmeren na funkcionalni način programiranja. Zato smo u njemu prirodno došli do nepromenljivih vrednosti, rekurzije, funkcija višeg reda, transformacija listi i kompozicije funkcija. I način definisanja i obrade podataka bio je povezan sa funkcijama i podudaranjem oblika.

U praksi granice između paradigmi često nisu stroge. Savremeni programski jezici mogu podržavati ideje iz više paradigmi, a u istom programu mogu se koristiti različiti pristupi tamo gde su najpogodniji. Važnije od toga da ceo program pripada jednoj paradigmi jeste da izabrani način organizovanja doprinosi jasnom i razumljivom rešenju problema.

Kroz Python i Haskell upoznali smo dva jezika koji na različite načine podstiču razmišljanje o programu. U Pythonu smo problem često opisivali kroz naredbe, promene stanja i objekte, dok smo u Haskellu naglasak stavljali na funkcije, izraze i formiranje novih vrednosti. Upravo to poređenje pokazuje da programske paradigme nisu samo različiti načini zapisivanja istog koda, već različiti načini razmišljanja o strukturi programa i postupku rešavanja problema.

Zato cilj upoznavanja programskih paradigmi nije samo učenje sintakse različitih programskih jezika. Važno je naučiti da isti problem možemo posmatrati na više načina, prepoznati prednosti i ograničenja različitih pristupa i izabrati način organizovanja programa koji najbolje odgovara problemu koji rešavamo.