DODATAK A

Rešenja vežbi

U ovom dodatku nalaze se rešenja vežbi iz poglavlja ove knjige.

Rešenja vežbi iz drugog poglavlja

1. Evo jednog načina da se to uradi:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
$obim = 2 * $pi * 12.5;
print "Obim kruga poluprečnika 12.5 je $obim.\n";
```

Program smo započeli uobičajenim redom `#!`; vaša putanja do Perla može biti drugačija. Isključili smo prikazivanje upozorenja.

U prvom pravom redu koda, promenljiva `$pi` dobija vrednost broja π. Postoji nekoliko razloga zbog kojih svaki dobar programer voli da koristi konstantne* vrednosti: potrebno je mnogo vremena da se napiše `3.141952654` na više mesta u kodu. Pored toga, možete dobiti netačan rezultat ako na jednom mestu slučajno upotrebite vrednost `3.141592654`, a na drugom `3.14159`. U samo jednom redu koda proverava se da li je vrednost ispravno uneta. Lakše je unositi `$pi` nego π, naročito ako ne koristite format Unicode. Održavanje programa takođe će biti lakše ako se vrednost broja π promeni.† Dalje, kada se obim kruga izračuna, smešta se u promenljivu `$obim`, čiji se sadržaj ispisuje u poruci. Poruka se završava oznakom za novi red jer bi tako trebalo da se završava svaki izlazni red dobrog programa. Bez toga bi poruka mogla da izgleda ovako (u zavisnosti od odzivnika vašeg komandnog okruženja):

```
Obim kruga poluprečnika 12.5 je
78.53981635.bash-2.01$[]
```

* Ako više volite formalno unošenje konstanti, pragma `constant` je ono što tražite.

† Zamalo da se promeni pre više od jednog veka, zahvaljujući predlogu zakona u Indijani. Pogledajte House Bill No. 246, Indiana State Legislature, 1897, *http://db.uwaterloo.ca/~alopez-o/math-faq/node45.html*.

Uglaste zagrade predstavljaju ulazni pokazivač, koji trepće na kraju reda, a to je ujedno i odzivnik komandnog okruženja na kraju poruke.* Budući da obim kruga nije `78.53981635.bash-2.01$`, izlaz bi verovatno bio protumačen kao greška. Zato na kraju svakog izlaznog reda treba koristiti oznaku `\n`.

Evo jednog od načina da se to uradi:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "Koliki je poluprečnik? ";
chomp($poluprecnik = <STDIN>);
$obim = 2 * $pi * $poluprecnik;
print "Obim kruga poluprečnika $poluprecnik je $obim.\n";
```

Ovaj kôd je isti kao prethodni, samo što sada od korisnika tražimo da zada poluprečnik, a promenljivu `$poluprecnik` koristimo na svim mestima na kojima smo u prethodnom kodu upotrebljavali vrednost `12.5`. Da smo prilikom pisanja prethodnog programa više razmišljali unapred, i u njemu bismo imali promenljivu `$poluprecnik`. Primetite da smo na ulazni red primenili operator `chomp`. Da nismo to uradili, matematička formula bi i dalje radila jer se znakovni niz `"12.5\n"` bez problema može pretvoriti u broj `12.5`. Ipak, poruka bi izgledala ovako:

```
Obim kruga poluprečnika 12.5
 je 78.53981635.
```

Oznaka za novi red i dalje se nalazi u okviru promenljive `$poluprecnik`, čak i ako je koristimo kao broj. Budući da u naredbi `print` postoji razmak između sadržaja promenljive `$poluprecnik` i reči „`je`“, postojaće i razmak na početku drugog izlaznog reda. Poruka priče je sledeća: na sve ulazne redove treba primeniti operator `chomp`, osim ako imate dobar razlog da to ne radite.

2. Evo jednog od načina da se to uradi:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "Koliki je poluprečnik? ";
chomp($poluprecnik = <STDIN>);
$obim = 2 * $pi * $poluprecnik;
if ($poluprecnik < 0) {
  $obim = 0;
}
print "Obim kruga poluprečnika $poluprecnik je $obim.\n";
```

Ovde smo dodali proveru za neispravan poluprečnik. Čak i ako je poluprečnik neispravan, obim kruga neće biti negativan. Vrednost neispravnog poluprečnika mogli ste postaviti i na nulu i tako izračunati obim kruga. Postoji više načina da se ovaj zadatak reši. U stvari, to je Perlovo geslo – postoji više načina za rešenje problema i zato rešenje svakog zadatka počinje rečima: „Evo jednog od načina da se to uradi“.

* Tražili smo od izdavačke kuće O'Reilly da ulože dodatni novac u štampanje pokazivača pomoću trepćućeg mastila, ali njeni predstavnici to nisu hteli da urade.

Evo jednog od načina da se to uradi:

```
print "Unesite prvi broj: ";
chomp($prvi = <STDIN>);
print "Unesite drugi broj: ";
chomp($drugi = <STDIN>);
$rezultat = $prvi * $drugi;
print "Rezultat je $rezultat.\n";
```

Primetite da smo u ovom rešenju izostavili red `#!`. Od sada ćemo podrazumevati da on već postoji pa nećete morati da ga čitate svaki put.

Možda imena promenljivih nisu dobro izabrana. U velikom programu, programer koji održava kôd mogao bi pomisliti da promenljiva `$druga` sadrži vrednost `2`. To u ovom kratkom programu verovatno nije važno; u velikom programu, promenljive bi verovatno imale opisna imena poput `$prvi_odgovor`.

Nije važno što smo u ovom programu zaboravili da na promenljive `$prva` i `$druga` primenimo operator `chomp` jer ih nikada ne koristimo kao znakovne nizove. Ipak, ako bi programer za održavanje koda izmenio program tako da prikazuje poruku `Rezultat množenja $prva i $druga je $rezultat.\n`, znakovi za novi red bi nam došli glave. Zato ponavljamo: uvek koristite operator `chomp`, osim ako imate dobar razlog da to ne radite* – kao što je slučaj u sledećem zadatku.

3. Evo jednog od načina da se to uradi:

```
print "Unesite znakovni niz: ";
$niz = <STDIN>;
print "Unesite broj ponavljanja: ";
chomp($br_ponavljanja = <STDIN>);
$rezultat = $niz x $br_ponavljanja;
print "Rezultat je:\n$rezultat";
```

Ovaj program je skoro isti kao prethodni. U njemu se znakovni niz „množi“ brojem ponavljanja. Zato smo zadržali strukturu programa iz prethodnog zadatka. U ovom slučaju nismo hteli da primenimo operator `chomp` na prvu ulaznu stavku (znakovni niz) jer se u zadatku traži da se znakovni nizovi pojavljuju u zasebnim redovima. Kada bi korisnik uneo `fred` i oznaku za novi red kao znakovni niz, i broj tri, svaka od reči `fred` ispisivala bi se u zasebnom redu.

U naredbi `print` na kraju programa koristimo oznaku za novi red ispred promenljive `$rezultat` zato što smo želeli da se prva reč `fred` ispiše u zasebnom redu, tj. nismo želeli da dobijemo izlaz sličan ovom (poravnate su samo dve od tri reči `fred`):

```
Rezultat je: fred
fred
fred
```

U isto vreme, nije nam trebala još jedna oznaka za novi red na kraju izlaza jer ga promenljiva `$rezultat` već sadrži.

* Primena operatora `chomp` je kao žvakanje: nije uvek neophodno, ali uglavnom neće škoditi.

U većini slučajeva Perlu je svejedno na kojim mestima se nalaze razmaci u programu; možete ih čak i izostaviti. Ipak, važno je da ne pravite greške prilikom unosa programa. Ako bi se operator `x` iz prethodnog primera nalazio odmah pored promenljive `$niz`, Perl bi taj izraz protumačio kao `$nizx`, a to je pogrešno.

Rešenja vežbi iz trećeg poglavlja

1. Evo jednog od načina da se to uradi:

```
print "Unesite redove, a zatim pritisnite Ctrl-D:\n"; # ili možda Ctrl-Z
@redovi = <STDIN>;
@obrnuti_redovi = reverse @redovi;
print @obrnuti_redovi;
```

   ili možda čak jednostavnije:

```
print "Unesite redove, a zatim pritisnite Ctrl-D:\n";
print reverse <STDIN>;
```

   Većina programera na Perlu više voli da koristi drugi način, pod uslovom da spisak redova ne mora da se čuva za kasniju upotrebu.

2. Evo jednog od načina da se to uradi:

```
@imena = qw/ fred beti barni dino vilma kamicak bam-bam /;
print "Unesite brojeve od 1 do 7, po jedan u svakom redu, a zatim pritisnite
Ctrl-D:\n";
chomp(@brojevi = <STDIN>);
foreach (@brojevi) {
  print "$imena[ $_ - 1 ]\n";
}
```

   Moramo oduzeti jedan od vrednosti indeksa tako da korisnik može da broji od jedan do sedam čak i ako se vrednosti indeksa kreću od nula do šest. Isti rezultat postići ćete i korišćenjem lažne stavke u nizu `@imena`:

```
@imena = qw/ lažna_stavka fred beti barni dino vilma kamicak bam-bam /;
```

   Nagradite sebe dodatnim poenima ako ste proverili da li je korisnik uneo brojeve između jedan i sedam.

   Evo jednog od načina da se to uradi ako želite da izlazni podaci budu u jednom redu:

```
chomp(@redovi = <STDIN>);
@sortirani = sort @redovi;
print "@sortirani\n";
```

   Ako želite da izlazne podatke ispišete u zasebnim redovima, uradite to ovako:

```
print sort <STDIN>;
```

Rešenja vežbi iz četvrtog poglavlja

1. Evo jednog od načina da se to uradi:

```
sub ukupno {
  my $zbir;  # privatna promenljiva
  foreach (@_) {
    $zbir += $_;
  }
  $zbir;
}
```

   Ovaj potprogram koristi promenljivu `$zbir` za čuvanje ukupne vrednosti. Na početku potprograma, promenljiva `$zbir` ima vrednost `undef` pošto je reč o novoj promenljivoj. Potom petlja `foreach` prolazi kroz listu parametara (iz promenljive `@_`) koristeći `$_` kao kontrolnu promenljivu. (Ne postoji automatska veza između promenljive `@_`, niza parametara i promenljive `$_`, podrazumevane promenljive petlje `foreach`.)

   U prvom prolazu kroz petlju `foreach`, prvi broj (iz promenljive `$_`) dodaje se promenljivoj `$zbir`. Promenljiva `$zbir` sadrži vrednost `undef` jer joj prethodno nije dodeljena vrednost. Budući da je koristimo kao broj (Perl to zna jer se koristi numerički operator `+=`), Perl će se ponašati kao da joj je dodeljena vrednost nula. Zato će Perl sabrati nulu i vrednost prvog parametra i ukupnu vrednost ponovo smestiti u promenljivu `$zbir`.

   U sledećem prolazu kroz petlju, promenljivoj `$zbir` (više ne sadrži vrednost `undef`) dodaje se naredni parametar. Ukupna vrednost se ponovo smešta u promenljivu `$zbir`, a postupak se ponavlja za sve preostale parametre. Na kraju, u poslednjem redu koda, vrednost promenljive `$zbir` vraća se pozivaocu.

   U ovom potprogramu postoji potencijalna greška, zavisno od toga kako posmatrate problem. Pretpostavimo da je potprogram pozvan s praznom listom parametara kao što smo pretpostavili za potprogram `&max` u tekstu poglavlja. U tom slučaju, promenljiva `$zbir` imala bi vrednost `undef`, a to bi bila i povratna vrednost potprograma. U ovom programu bolje bi bilo da povratna vrednost bude nula, a ne `undef`. Ukoliko želite da napravite razliku između zbira elemenata prazne liste i zbira elemenata liste `(3, -5, 2)`, bilo bi u redu da povratna vrednost bude `undef`.

   Ukoliko ne želite da povratna vrednost bude nedefinisana, nije teško ispraviti program: promenljivoj `$zbir` dodelite početnu vrednost nula, a ne `undef`:

```
my $zbir = 0;
```

   Sada će potprogram uvek vraćati brojčanu vrednost, čak i ako je lista parametara prazna.

2. Evo jednog od načina da se to uradi:

```
# Ne zaboravite da uključite potprogram &ukupno iz prethodnog zadatka!
print "Zbir brojeva od 1 do 1000 je ", &ukupno(1..1000), ".\n";
```

Ovaj potprogram se ne može pozvati iz znakovnog niza pod dvostrukim navodnicima,* tako da je poziv potprograma još jedna od stavki koje se prosleđuju naredbi `print`. Zbir bi trebalo da iznosi `500500`, što je lep okrugao broj. Program ne bi trebalo dugo da se izvršava; prosleđivanje liste sa 1000 vrednosti svakodnevni je posao za Perl.

3. Evo jednog od načina da se to uradi:

```
sub prosek {
  if (@_ == 0) { return }
  my $brojac = @_;
  my $zbir = &ukupno(@_);              # iz prethodnog zadatka
  $zbir/$brojac;
}
sub iznad_proseka {
  my $prosek = &prosek(@_);
  my @lista;
  foreach $element (@_) {
    if ($element > $prosek) {
      push @lista, $element;
    }
  }
  @lista;
}
```

Potprogram `prosek` završava se bez vraćanja vrednosti ako je lista parametara prazna. Na taj način se pozivaocu vraća vrednost `undef`† kao znak da se za praznu listu ne može izračunati prosek. Ako lista nije prazna, potprogram `&ukupno` olakšava računanje proseka. Nismo morali da koristimo privremene promenljive `$zbir` i `$brojac`, ali se uz njih kôd lakše čita.

Drugi potprogram `iznad_proseka` pravi i vraća listu željenih stavki. (Zašto je kontrolna promenljiva `$element`, a ne Perlova podrazumevana promenljiva `$_`?) Ovaj potprogram koristi drugačiju tehniku za obradu prazne liste parametara.

Rešenja vežbi iz petog poglavlja

1. Evo jednog od načina da se to uradi:

```
print reverse <>;
```

Prilično je jednostavno! Prethodni kôd će raditi jer naredba `print` prihvata listu znakovnih nizova, a ona se dobija pozivanjem funkcije `reverse` u kontekstu liste. Funkcija `reverse` takođe prihvata listu znakovnih nizova, a dobija je kada se operator dijamant upotrebljava u kontekstu liste. Znači, operator dijamant vraća listu svih redova iz svih datoteka koje je korisnik zadao. Ta lista redova se može odštampati pomoću operatora `cat`. Funkcija `reverse` obrće listu redova, a naredba `print` ih ispisuje.

* Ovo se ne može uraditi bez korišćenja naprednih tehnika. Teško je naći nešto što se u Perlu *uopšte* ne može uraditi.

† Ili praznu listu ako se potprogram `&prosek` koristi u kontekstu liste.

2. Evo jednog od načina da se to uradi:

```
print "Unesite redove, a zatim pritisnite Ctrl-D:\n";  # ili Ctrl-Z
chomp(my @redovi = <STDIN>);
print "1234567890" x 7, "12345\n";  # redovi se ravnaju po sedamdeset petom
                                    # stupcu
foreach (@redovi) {
  printf "%20s\n", $_;
}
```

Ovaj kôd započinjemo čitanjem svih redova teksta i primenom operatora `chomp`. Potom se ispisuje red za poravnavanje. Budući da nam je to samo pomoć prilikom otkrivanja grešaka, navedeni red treba komentarisati kada se završi pisanje programa. Mogli smo da unosimo znakovni niz `"1234567890"` ili da ga kopiramo da bi red za poravnavanje bilo dovoljno dugačak, ali smo odlučili da to uradimo na ovaj način.

Petlja `foreach` prolazi kroz listu redova i štampa svaki od njih pomoću konverzije `%20s`. Ako ste i vi to uradili, mogli ste napraviti odgovarajući format da biste listu odštampali bez korišćenja petlje:

```
my $format = "%20s\n" x @redovi;
printf $format, @redovi;
```

Jedna od čestih grešaka je korišćenje stubaca od devetnaest znakova. To se dešava kada sebi* kažete sledeće: „Zašto bih koristio operator `chomp` kada ću znakove za novi red dodati kasnije?“ Ipak, kada izostavite operator `chomp` i iskoristite format `"20s"` (bez oznake za novi red)†, izlazni red će biti kraći za jedan znak. Šta nije u redu?

Problem je u načinu na koji Perl broji razmake potrebne za pravljenje odgovarajućeg broja stubaca. Ako korisnik unese reč **`vilma`** i oznaku za novi red, Perl će videti *šest* znakova, a ne pet, jer je i oznaka za novi red znak. Zato će Perl ispisati četrnaest razmaka i znakovni niz od šest znakova, što je ukupno dvadeset znakova, koje ste zahtevali primenom formata `"%20s"`.

Perl ne gleda sadržaj znakovnog niza prilikom utvrđivanja njegove dužine; on vidi samo broj znakova. Oznaka za novi red (ili neki drugi specijalan znak poput tabulatora ili znaka null) napraviće pometnju.‡

Evo jednog od načina da se to uradi:

```
print "Koliku širinu stupca želite? ";
chomp(my $sirina = <STDIN>);
print "Unesite redove, a zatim pritisnite Ctrl-D:\n";  # ili Ctrl-Z
chomp(my @redovi = <STDIN>);
print "1234567890" x (($sirina+9)/10), "\n";    # poravnajte redove po potrebi
foreach (@redovi) {
  printf "%${sirina}s\n", $_;
}
```

* Ili Lariju, ako je tu negde.

† Osim ako vam je Lari rekao da to ne radite.

‡ Kao što je Lari trebalo da vam dosad objasni.

Ovaj kôd je prilično sličan prethodnom, samo što ovde zahtevamo da korisnik zada širinu stupca. To radimo zato što ne možemo zahtevati unošenje podataka *posle* indikatora za kraj datoteke (bar u nekim sistemima). U praksi ćete verovatno koristiti bolji indikator za kraj unosa, kao što ćete videti u rešenjima drugih zadataka.

Još jedna izmena u odnosu na rešenje prethodnog zadatka jeste i red za poravnavanje. Iskoristili smo malo matematike da bismo napravili red koji je dugačak onoliko koliko nam je potrebno. Dokazivanje da smo matematičke proračune tačno obavili dodatan je izazov. (Savet: razmislite o širinama `50` i `51` i setite se da se desni operand operacije `x` odseca, a ne zaokružuje.)

Da bismo dobili odgovarajući format, iskoristili smo izraz `"%${sirina}s\n"`, koji interpolira sadržaj promenljive `$sirina`. Vitičaste zagrade su neophodne da bi odvojile ime promenljive od slova `s`; kada se vitičaste zagrade ne bi koristile, interpolirao bi se sadržaj promenljive `$sirinas`, koja ne postoji. Ako ste zaboravili kako se koriste vitičaste zagrade, mogli ste napisati i izraz `'%' . $sirina . "s\n"` da biste dobili isti format.

Programeri koji se nikada nisu susreli s naredbom `printf`, mogli su se setiti još jednog rešenja. Budući da je naredba `printf` deo programskog jezika C, koji ne koristi interpolaciju znakovnih nizova, mogli smo iskoristiti trik koji upotrebljavaju programeri na jeziku C. Ako se umesto numeričke širine polja u konverziji pojavi zvezdica (`*`), koristiće se vrednost iz liste parametara:

```
printf "%*s\n", $sirina, $_;
```

Rešenja vežbi iz šestog poglavlja

1. Evo jednog od načina da se to uradi:

```
my %prezime = qw{
  fred kremenko
  barni kamenko
  vilma kremenko
};
print "Unesite ime: ";
chomp(my $ime = <STDIN>);
print "$ime $prezime{$ime}.\n";
```

Ovde koristimo listu `qw//` (s vitičastim zagradama za razdvajanje) za inicijalizaciju heša. To ima smisla za ovaj jednostavan skup podataka, a i kôd se lako održava jer svaki podatak sadrži samo ime i prezime. Kada bi podaci sadržali razmake (primera radi, kada bi Bedrok posetili `robert de niro` ili `meri džej blajdž`), ovo rešenje ne bi bilo tako dobro.

Svaki par ključ–vrednost mogli ste dodeliti zasebno, kao u ovom slučaju:

```
my %prezime;
$prezime{"fred"} = "kremenko";
$prezime{"barni"} = "kamenko";
$prezime{"vilma"} = "kremenko";
```

Ako heš deklarišete pomoću operatora `my` (recimo zbog pragme `use strict`), morate ga deklarisati pre nego što mu dodelite elemente. Operator `my` ne možete koristiti samo u jednom delu promenljive:

```
my $prezime{"fred"} = "kremenko";  # Ups!
```

Operator `my` možete koristiti samo s *celim* promenljivama, a nikada samo s jednim elementom niza ili heša. Kada govorimo o leksičkim promenljivama, verovatno ste primetili da se leksička promenljiva `$ime` deklariše unutar poziva funkcije `chomp`; prilično je uobičajeno da se sve leksičke promenljive tako deklarišu.

Ovo je još jedan slučaj kada je upotreba funkcije `chomp` vrlo važna. Kada bi korisnik uneo znakovni niz `"fred\n"`, a mi zaboravili da primenimo funkciju `chomp`, tražili bismo nepostojeću reč `"fred\n"` kao deo heša. Ipak, funkcija `chomp` nije svemoguća; ako bi korisnik uneo reč `"fred \n"` (s razmakom), ne bi postojao način da znamo da je on u stvari hteo da unese reč `"fred"`.

Dobićete dodatne poene ako ste napravili proveru da li dati ključ postoji u hešu i ako korisniku prikazujete poruku koja ga obaveštava da je uneo nepostojeće ime.

2. Evo jednog od načina da se to uradi:

```
my(@reci, %brojac, $rec);     # (opciono) deklarišemo naše promenljive
chomp(@reci = <STDIN>);
foreach $rec (@reci) {
  $brojac{$rec} += 1;          # ili $brojac{$rec} = $brojac{$rec} + 1;
}
foreach $rec (keys %brojac) {  # ili sort keys %brojac
  print "$rec je ponovljena $brojac{$rec} puta.\n";
}
```

U ovom primeru, sve promenljive smo deklarisali na početku. Programeri koji su ranije koristili jezike poput Pascala (gde se promenljive deklarišu na početku programa) prepoznaće ovaj način rada. Promenljive deklarišemo jer mislimo da postoji pragma `use strict`; Perl podrazumevano ne zahteva ovakve deklaracije.

Potom se koristi operator standardnog ulaza `<STDIN>` u kontekstu liste, koji čita sve ulazne redove u niz `@reci`, a zatim na njih primenjuje funkciju `chomp`. Tako će niz `@reci` sadržati listu ulaznih reči ako se sve nalaze u zasebnim redovima kao što bi trebalo da bude.

Prva petlja `foreach` prolazi kroza sve reči. Ona sadrži najvažniju naredbu u celom programu: onu koja inkrementira vrednost izraza `$brojac{$rec}`. Iako ste mogli koristiti i kraći (pomoću operatora `+=`) i duži zapis, kraći zapis je neznatno efikasniji jer Perl sadržaj promenljive `$rec` mora da traži u hešu samo jednom.* Za svaku reč u prvoj petlji `foreach`, vrednost izraza `$brojac{$rec}` uvećava se za jedan. Ako je prva reč `fred`, vrednost izraza `$brojac{"fred"}` uvećaće se za jedan.

* U nekim verzijama Perla, korišćenjem kraćeg zapisa sprečićete prikazivanje upozorenja o upotrebi nedefinisane vrednosti. Prikazivanje upozorenja može se izbeći i korišćenjem operatora `++` za inkrementiranje vrednosti promenljive, mada još nismo objasnili kako se on koristi.

Budući da se izraz `$brojac{"fred"}` prvi put pojavljuje, njegova vrednost je `undef`. Nedefinisana vrednost se tretira kao broj (zbog operatora `+=` odnosno `+` ako ste koristili duži zapis), pa će Perl automatski vrednost `undef` pretvoriti u nulu. Krajnja vrednost je jedan i ona se smešta u izraz `$brojac{"fred"}`.

Pretpostavimo da se u sledećem prolazu kroz petlju `foreach` koristi reč `barni`. Vrednost izraza `$brojac{"barni"}` uvećaće se za jedan.

Neka je sledeća reč opet `fred`. Sada će se za jedan uvećati vrednost izraza `$brojac{"fred"}`, koja je bila jedan, a sada je dva, što znači da smo reč `fred` videli dvaput.

Na kraju prve petlje `foreach`, znamo koliko se puta koja reč pojavila. Heš ima ključ za svaku (jedinstvenu) ulaznu reč, a odgovarajuća vrednost je broj njenih ponavljanja.

Druga petlja `foreach` prolazi kroz sve ključeve heša (tj. jedinstvene ulazne reči). U ovoj petlji, svaka *različita* reč pojavljuje se samo jednom i ispisuje se poruka poput „`reč fred je viđena 3 puta`“.

Ukoliko želite dodatne poene, trebalo bi da iskoristite operator `sort` ispred ključeva da biste ih sortirali. Ako lista sadrži veliki broj stavki, dobro je da ih sortirate kako bi programer koji održava kôd mogao brzo da pronađe ono što mu treba.

Rešenja vežbi iz sedmog poglavlja

1. Evo jednog od načina da se to uradi:

```
while (<>) {
  if (/fred/) {
    print;
  }
}
```

   Ovo je prilično jednostavno. Važnije je ovaj zadatak isprobati s raznim znakovnim nizovima. Ne uparuje se reč `Fred`, što pokazuje da se u regularnim izrazima razlikuju velika i mala slova. (Videćemo kasnije kako se to može izmeniti.) Uparuju se reči `frederik` i `Alfred`, jer oba znakovna niza sadrže reč `fred`. (Uparivanje celih reči, tako da se reči `frederik` i `Alfred` ne mogu upariti, druga je mogućnost, koju ćemo videti kasnije.)

2. Evo jednog od načina da se to uradi: izmenite šablon koji se koristi u rešenju prvog zadatka tako da sada glasi `/[fF]red/`. Možete isprobati i šablone `/(f|F)red/` ili `/fred|Fred/`, ali je klasa znakova efikasnija.

3. Evo jednog od načina da se to uradi: izmenite šablon koji se koristi u rešenju prvog zadatka tako da sada glasi `/\./`. Obrnuta kosa crta je neophodna zato što je tačka metaznak, a možete koristiti i klasu znakova `/[.]/`.

4. Evo jednog od načina da se to uradi: izmenite šablon koji se koristi u rešenju prvog zadatka tako da sada glasi `/[A-Z][a-z]+/`.
5. Evo jednog od načina da se to uradi:

```
while (<>) {
  if (/vilma/) {
    if (/fred/) {
      print;
    }
  }
}
```

Posle uparivanja šablona `/vilma/`, ovaj kôd uparuje šablon `/fred/`, ali se reč `fred` može pojaviti ispred ili iza reči `vilma` u redu; uparivanja su nezavisna jedno od drugog.

Ako ste želeli da izbegnete ugnežđivanje strukture `if`, mogli ste napisati kôd nalik na sledeći:*

```
while (<>) {
  if (/vilma.*fred|fred.*vilma/) {
    print;
  }
}
```

Ovaj kôd će raditi jer se reč `vilma` nalazi ispred reči `fred` ili se reč `fred` nalazi ispred reči `vilma`. Da smo koristili šablon `/vilma.*fred/`, upario bi se i red `fred i vilma kremenko` iako se u njemu pominju obe reči.

Za rešenje ovog zadatka dajemo dodatne poene jer većina programera u ovoj situaciji ima mentalnu blokadu. Pokazali smo vam operaciju „ili“ (pomoću vertikalne crte |), ali vam nismo pokazali operaciju „i“. To je zato što se ona ne koristi u regularnim izrazima.† Ako želite da saznate da li je uparivanje pomoću oba šablona uspelo, iskoristite oba.

Rešenja vežbi iz osmog poglavlja

1. Postoji jednostavan način da se to uradi i mi smo ga pokazali u tekstu poglavlja. Ukoliko kao rezultat niste dobili `pre<match>posle` kao što bi trebalo, onda ste izabrali teži način za rešavanje ovog zadatka.
2. Evo jednog od načina da se to uradi:

```
/a\b/
```

(Ovo je šablon koji treba koristiti u programu.) Ako vaš šablon greškom uparuje reč `barni`, verovatno ste zaboravili da iskoristite sidro za ograničavanje reči.

* Oni koji poznaju logički operator i (pročitajte deseto poglavlje) mogli bi da koriste šablone `/fred/` i `/vilma/` u istoj strukturi `if`. To je efikasniji, prenosiviji i generalno bolji način. Ipak, još uvek nismo videli logičko i.

† Postoje neki napredni načini da se obavi ono što neki nazivaju operacija „i“. Oni su po pravilu manje efikasni od upotrebe Perlovog operatora logičko i, u zavisnosti od toga koje optimizacije Perl i njegov deo za rad s regularnim izrazima mogu da obave.

3. Evo jednog od načina da se to uradi:

```
#!/usr/bin/perl
while (<STDIN>) {
  chomp;
  if (/(\b\w*a\b)/) {
    print "Upareno: |$`<$&>$'|\n";
    print "\$1 sadrži '$1'\n";       # Novi izlazni red
  } else {
    print "Nije upareno: |$_|\n";
  }
}
```

   Ovo je isti program (s novim šablonom), osim što je dodat jedan red za ispisivanje sadržaja promenljive `$1`.

   U šablonu se unutar zagrada koristi par `\b` sidara za ograničavanje reči*, mada će šablon raditi i ako se sidra koriste izvan zagrada. To je zato što sidra odgovaraju mestu, a ne znakovima u znakovnom nizu: sidra imaju „nultu dužinu“.

4. Evo jednog od načina da se to uradi:

```
m!
  (\b\v*a\b)       # $1: reč koja se završava slovom a
  (.{0,5})         # $2: iza koje sledi do pet znakova
!xs                # modifikatori /x i /s
```

   (Ne zaboravite da dodate kôd za prikazivanje sadržaja promenljive `$2`. Ukoliko izmenite šablon tako da se ponovo koristi samo jedna promenljiva, samo komentarišite dodatni red.) Ukoliko vaš šablon više ne uparuje reč `vilma`, verovatno broj znakova treba da bude jedan ili više a ne nula ili više. Možda ste izostavili i modifikator `/s` jer u okviru podataka ne bi trebalo da bude oznaka za novi red. (Ako ih ipak ima, modifikator `/s` mogao bi da utiče na rezultat rada programa.)

5. Evo jednog od načina da se to uradi:

```
while (<>) {
  chomp;
  if (/\s+$/) {
    print "$_#\n";
  }
}
```

   Ovde smo za označavanje koristili znak `#`.

Rešenja vežbi iz devetog poglavlja

1. Evo jednog od načina da se to uradi:

```
/($sta){3}/
```

   Kada se promenljiva `$sta` interpolira, dobija se šablon `/(fred|barni){3}/`. Kada se zagrade ne bi koristile, šablon bi izgledao ovako: `/fred|barni{3}/`, a to je isto što i `/fred|barniii/`. Zato su zagrade neophodne.

* Priznajemo, prvo sidro nije potrebno iz razloga koje ovde nećemo pominjati. Ipak, ono doprinosi efikasnosti i jasnoći programa.

Evo jednog od načina da se to uradi:

```
my $ulaz = $ARGV;
unless (defined $ulaz) {
  die "Uputstvo za korišćenje: $0 ime_datoteke";
}
my $izlaz = $ulaz;
$izlaz =~ s/(\.\w+)?$/.izlaz/;
unless (open IN, "<$ulaz") {
  die "Ne mogu da otvorim '$ulaz': $!";
}
unless (open OUT, ">$izlaz") {
  die "Ne mogu da upišem podatke u '$izlaz': $!";
}
while (<IN>) {
  s/Fred/Lari/gi;
  print OUT $_;
}
```

Na početku ovog programa imenuje se jedini parametar sa komandne linije i proverava se njegovo postojanje. Njegova vrednost se kopira u promenljivu `$izlaz`, a nastavak imena datoteke (ako postoji) menja se u `.izlaz`. (Dovoljno je imenu datoteke dodati nastavak `.izlaz`.)

Kada se otvore identifikatori datoteka `IN` i `OUT`, program može početi da obavlja koristan posao. Ukoliko niste koristili opcije `/g` i `/i`, oduzmite sebi pola poena, jer se moraju izmeniti *sve* reči `fred` i `Fred`.

2. Evo jednog od načina da se to uradi:

```
while (<IN>) {
  chomp;
  s/Fred/\n/gi;        # Menjaju se sve reči Fred
  s/Vilma/Fred/gi;     # Menjaju se sve reči Vilma
  s/\n/Vilma/g;        # Menja se mesto za argument
  print OUT "$_\n";
}
```

Ovaj program zamenjuje petlju iz rešenja prethodnog zadatka. Da bi se to moglo uraditi, mora postojati neki znakovni niz koji se ne pojavljuje u podacima. Korišćenjem funkcije `chomp` (i dodavanjem oznaka za novi red), omogućavamo da taj znakovni niz bude oznaka za novi red. (Trebalo bi da izaberete neki drugi znakovni niz, koji se retko pojavljuje. Jedan od dobrih kandidata je znak NUL, `\0`.)

3. Evo jednog od načina da se to uradi:

```
$^I = ".bak";          # rezervna kopija
while (<>) {
  if (/^#!/) {         # da li je početni red?
    $_ .= "## Copyright (C) 20XX Iskreno vaš\n";
  }
}
```

Pozovite ovaj program s imenima datoteka koje želite da ažurirate. Na primer, ako ste rešenjima zadataka davali imena *zad01-1, zad01-2* itd., tako da počinju rečju `zad...`, mogli biste iskoristiti sledeću naredbu:

```
./fix_my_copyright zad*
```

4. Da podatke o autorskim pravima ne bismo unosili dvaput, moramo dvaput proći kroz datoteke. Prvo pravimo heš u kome su imena datoteka ključevi, a pošto vrednosti nisu važne, koristimo broj jedan jer nam je tako zgodno:

```
my %uradi_ovo;
foreach (@ARGV) {
  $uradi_ovo{$_} = 1;
}
```

Potom ispitujemo datoteke i uklanjamo sve one koje imaju podatke o autorskim pravima. Trenutno ime datoteke nalazi se u promenljivoj `$ARGV`, pa ga možemo koristiti kao ključ heša:

```
while (<>) {
  if (/^## Copyright/) {
    delete $uradi_ovo{$ARGV};
  }
}
```

Na kraju, kada smo dobili skraćenu listu imena u nizu `@ARGV`, program je postao isti kao i onaj iz prethodnog zadatka:

```
@ARGV = sort keys %uradi_ovo;
$^I = ".bak";          # rezervna kopija
while (<>) {
  if (/^#!/) {         # da li je početni red?
    $_ .= "## Copyright (c) 20XX Iskreno vaš\n";
  }
}
```

Rešenja vežbi iz desetog poglavlja

1. Evo jednog od načina da se to uradi:

```
my $tajni_broj = int(1 + rand 100);
# Sledeći red se može komentarisati tokom testiranja programa
# print "Nemojte reći nikome, ali tajni broj je $tajni_broj.\n";
while (1) {
  print "Unesite broj od 1 do 100: ";
  chomp(my $broj = <STDIN>);
  if ($broj =~ /kraj|izlaz|^\s*$/i) {
    print "Žao nam je što ste odustali. Broj je $tajni_broj.\n";
    last;
  } elsif ($broj < $tajni_broj) {
    print "Premali broj. Pokušajte ponovo!\n";
  } elsif ($broj == $tajni_broj) {
    print "Pogodak!\n";
    last;
  } else {
    print "Preveliki broj. Pokušajte ponovo!\n";
  }
}
```

U prvom redu se prihvata tajni broj između jedan i sto, a program dalje radi ovako: `rand` je Perlova funkcija za generisanje slučajnih brojeva, što znači da će naredba `rand 100` generisati broj između 0 i 100 (ne uključujući sto), tj. najveća

moguća vrednost izraza može biti `99.999`.* Kada dodamo jedan, dobićemo vrednost u opsegu od `1` do `100.999`, koju će funkcija `int` zaokružiti, pa ćemo tako dobiti rezultat između `1` i `100`, kao što nam je i potrebno.

Komentarisani red pomaže prilikom razvoja i testiranja programa, a može poslužiti i ako volite da varate. Glavni deo programa nalazi se u petlji `while`. U njoj će se tražiti unos brojeva sve dok se ne izvrši naredba `last`.

Važno je proveriti sve znakovne nizove pre brojeva. Kada to ne bismo uradili, vidite li šta bi se desilo ako bi korisnik uneo naredbu `quit`? Ona bi se interpretirala kao broj (uzrokujući verovatno pojavljivanje upozorenja ako je njihovo prikazivanje uključeno); budući da bi vrednost tog broja bila nula, korisnici bi dobili poruku da je broj premali. U tom slučaju verovatno nikada ne bismo došli do provere znakovnih nizova.

Beskonačna petlja se može napraviti i pomoću golog bloka s naredbom `redo`. Taj način nije ni više ni manje efikasan; to je samo još jedan od načina da rešimo ovaj zadatak. Ako očekujete da će petlja raditi sve vreme, iskoristite petlju `while`. Ukoliko će prolazak kroz petlju biti izuzetak, goli blok je bolji izbor.

Rešenja vežbi iz jedanaestog poglavlja

1. Evo jednog od načina da se to uradi:

```
foreach my $datoteka (@ARGV) {
  my $atributi = &atributi($datoteka);
  print "'$datoteka' $atributi.\n";
}
sub atributi {
  # ispisuju se atributi datoteke
  my $datoteka = shift @_;
  return "ne postoji" unless -e $datoteka;
  my @atributi;
  push @atributi, "može se čitati" if -r $datoteka;
  push @atributi, "mogu se upisivati podaci" if -w $datoteka;
  push @atributi, "izvršna" if -x $datoteka;
  return "postoji" unless @atributi;
  'je ' . join " i ", @atributi;  # povratna vrednost
}
```

U rešenju ovog zadatka pogodno je koristiti potprogram. U glavnoj petlji se ispisuje po jedan red atributa za svaku od datoteka, pa tako možemo saznati da je recimo `'pahuljice'` izvršna datoteka ili da datoteka `'brontosaurus'` ne postoji.

Potprogram prikazuje atribute datoteke. Ako datoteka ne postoji, nema potrebe ispitivati atribute, pa se zato ta provera nalazi na početku potprograma. Znači, ukoliko nema datoteke, potprogram će rad završiti ranije.

* Stvarna najveća moguća vrednost zavisi od sistema. Ako želite da znate njenu tačnu vrednost, posetite adresu *http://www.cpan.org/doc/FMTEYEWTK/random*.

Ako datoteka postoji, napravićemo listu atributa. (Dajte sebi dodatne poene ako ste koristili specijalni identifikator datoteke _ umesto promenljive `$datoteka` da biste sprečili upućivanje poziva sistemu za svaki novi atribut.) Nije bilo teško dodati provere postojanja atributa, ali šta će se desiti ako datoteka nema nijedan atribut? Pošto ne možemo da kažemo ništa drugo, reći ćemo samo da datoteka postoji. Naredba `unless` koristi činjenicu da će niz `@atributi` imati istinitu vrednost (u logičkom kontekstu, što je poseban slučaj skalarnog konteksta) samo ako nije prazan.

Ako atributi postoje, spojićemo ih rečju `"and"` i staviti reč `"is"` ispred da bismo napravili opis poput `is readable and writable`. To nije idealno rešenje; ako postoje tri atributa, možete dobiti poruku poput `is readable and writable and executable`, koja ima previše reči `"and"`, ali nećemo se više truditi. Ukoliko želite da proveravate postoji li više atributa, program napravite tako da prikazuje poruke poput `is readable`, `writable`, `executable and nonempty`. Uradite tako ako vam je to važno.

Ako komandna linija ne sadrži nijedno ime datoteke, neće biti izlaznih podataka. To ima smisla – ako tražite podatake za nula datoteka, dobićete isto toliko izlaznih podataka. Ipak, uporedimo to sa sledećim programom.

2. Evo jednog od načina da se to uradi:

```
die "Nije navedena nijedna datoteka!\n" unless @ARGV;
my $najstarije_ime = shift @ARGV;
my $najstarija = -M $najstarije_ime;
foreach (@ARGV) {
  my $starost = -M;
  ($najstarije_ime, $najstarija) = ($_, $starost)
    if $starost > $najstarija;
}
printf "Najstarija datoteka je %s, i ona je %.1f dana stara.\n",
  $najstarije_ime, $najstarija;
```

Ovaj program se prvo žali ako nema zadatih datoteka na komandnoj liniji. To je zato što bi on trebalo da nam saopšti ime najstarije datoteke, a to ne može da uradi ako nema datoteka za proveru.

Još jednom koristimo algoritam „oznake visokog vodostaja“. Prva datoteka je sigurno trenutno najstarija. Moramo pratiti njenu starost, pa je zato pamtimo u promenljivoj `$najstarija`.

Za svaku od preostalih datoteka, starost se utvrđuje korišćenjem parametra `-M`, kao što smo to uradili i u slučaju prve datoteke (s tim što ćemo ovde koristiti podrazumevani argument `$_`). Vreme poslednje izmene je ono što korisnici najčešće podrazumevaju pod „starošću“ datoteke, mada biste vi mogli koristiti i neku drugu vrednost. Ako je starost datoteke veća od vrednosti koja se nalazi u promenljivoj `$najstarija`, ažuriraćemo njenu vrednost. Nismo morali da koristimo dodeljivanje liste, ali je to zgodan način da se ažurira nekoliko promenljivih odjednom.

Starost datoteke dobijenu korišćenjem parametra `-M` smeštamo u privremenu promenljivu `$starost`. Šta bi se desilo da smo umesto promenljive svaki put koristili parametar `-M`? Ako ne bismo koristili specijalni identifikator datoteke _, svaki put bismo pozivali funkciju operativnog sistema za računanje starosti datoteke, što je potencijalno spora operacija (to ćete verovatno primetiti samo ako imate stotine hiljada datoteka, a možda čak ni tada). Vrlo je važno da razmislimo o tome šta bi se desilo kada bi se datoteka ažurirala u toku provere. Znači, saznali smo da je neka datoteka trenutno najstarija, ali pre nego što po drugi put iskoristimo parametar `-M`, datoteka se ažurira. Tako će promenljiva `$najstarija` u stvari sadržati *najmlađu* datoteku, a mi bismo saznali koja je najstarija datoteka od tog trenutka, a ne najstarija datoteka od svih zadatih; taj problem bi se teško rešio.

Na kraju programa koristimo naredbu `printf` da bismo prikazali ime i starost datoteke, pri čemu je starost zaokružena na najbližu deseticu dana. Dodelite sebi dodatne poene ako ste se pomučili da starost pretvorite u broj dana, sati i minuta.

Rešenja vežbi iz dvanaestog poglavlja

1. Evo jednog od načina da se to uradi pomoću operatora `glob`:

```
print "Zadajte direktorijum: (matični direktorijum je podrazumevan) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {         # Prazan red
  chdir or die "Ne mogu da pronađem matični direktorijum: $!";
} else {
  chdir $dir or die "Ne mogu da pronađem direktorijum '$dir': $!";
}
my @datoteke = <*>;
foreach (@datoteke) {
  print "$_\n";
}
```

   Prvo se prikazuje odzivnik, prihvata se željeni direktorijum i primenjuje se funkcija `chomp`. (Kada ne bismo koristili funkciju `chomp`, tražili bismo direktorijum koji se završava oznakom za novi red, što je dozvoljeno u Unixu.)

   Ako je ime direktorijuma zadato, program će pokušati da ga pronađe, a ako ne uspe, prekinuće rad. Ukoliko direktorijum nije zadat, koristiće se matični direktorijum.

   Na kraju, operator `glob` primenjen na zvezdicu izvlači sva imena u nov radni direktorijum, automatski sortirana po abecednom redu; imena se štampaju jedno po jedno.

2. Evo jednog od načina da se to uradi:

```
print "Zadajte direktorijum (matični direktorijum je podrazumevan) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {        ## Prazan red
  chdir or die "Ne mogu da pronađem matični direktorijum:
$!";
} else {
```

```
  chdir $dir or die "Ne mogu da pronađem direktorijum '$dir': $!";
}
my @datoteke = <.* *>;       ## sada sadrži i datoteke čija imena počinju
                             ## tačkom
foreach (sort @datoteke) {   ## sada se lista sortira
  print "$_\n";
}
```

Dve su razlike u odnosu na rešenje prethodnog zadatka: prvo, operator `glob` sada sadrži znakovni niz „tačka zvezdica“, čime se uparuju sva imena datoteka koja *počinju* tačkom. Drugo, rezultujuća lista se mora sortirati, jer se neka od imena koja počinju tačkom moraju smestiti na odgovarajuća mesta pre ili posle liste imena koja ne počinju tačkom.

3. Evo jednog od načina da se to uradi:

```
print "Zadajte direktorijum (matični direktorijum je podrazumevan) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {         # Prazan red
  chdir or die "Ne mogu da pronađem matični direktorijum:
$!";
} else {
  chdir $dir or die "Ne mogu da pronađem direktorijum '$dir': $!";
}
opendir DOT, "." or die "Ne mogu da pronađem direktorijum dot: $!";
foreach (sort readdir DOT) {
  # next if /^\./; ## ukoliko preskačemo datoteke iz direktorijuma dot
  print "$_\n";
}
```

Ovaj program ima istu strukturu kao i prethodna dva, ali sada otvaramo identifikator direktorijuma. Kada promenimo tekući direktorijum, želimo da ga otvorimo, a to je identifikator direktorijuma `DOT`.

Zašto `DOT`? Ako korisnik zada apsolutno ime direktorijuma, poput `/etc`, otvorićemo ga bez problema, ali ako je ime relativno (primera radi, `fred`), pogledajte šta će se desiti. Iskoristićemo komandu `chdir` da bismo prešli u direktorijum `fred` i komandu `opendir` da bismo ga otvorili. Na taj način bismo otvorili direktorijum `fred` u novom direktorijumu, a ne u izvornom. Jedino ime za koje možemo biti sigurni da označava „tekući direktorijum“ jeste `.`, koje uvek ima to značenje (bar u Unixu i sličnim sistemima).

Funkcija `readdir` vraća spisak svih datoteka iz direktorijuma, a on se zatim sortira i prikazuje. Da smo na ovaj način rešili prvi zadatak, preskočili bismo datoteke čija imena počinju tačkom. Taj problem se rešava tako što ćete ukloniti komentar iz odgovarajućeg reda u petlji `foreach`.

Možda se pitate sledeće: „Zašto smo prvo koristili funkciju `chdir`? Funkcija `readdir` može se koristiti s bilo kojim direktorijumom, a ne samo s tekućim.“ Želeli smo da omogućimo korisnicima izbor tekućeg direktorijuma samo jednim pritiskom na taster. Ovaj program bi mogao da bude početak uslužnog programa za upravljanje datotekama. U sledećem koraku korisnici bi mogli da biraju datoteke iz direktorijuma koje žele da snime na trake.

4. Evo jednog od načina da se to uradi:

```
unlink @ARGV;
```

ili ako želite da upozorite korisnika na probleme:

```
foreach (@ARGV) {
  unlink $_ or warn "Ne mogu da obavim operaciju unlink '$_': $!, nastavljam...
\n";
}
```

Ovde se svaka stavka sa komandne linije smešta u promenljivu `$_`, koja se koristi kao argument funkcije `unlink`. Ako nešto krene naopako, razloge ćete saznati u upozorenju.

5. Evo jednog od načina da se to uradi:

```
use File::Basename;
use File::Spec;
my($izvor, $cilj) = @ARGV;
if (-d $cilj) {
  my $osnovno_ime = basename $izvor;
  $cilj = File::Spec->catfile($cilj, $osnovno_ime);
}
rename $izvor, $cilj
  or die "Ne mogu da preimenujem '$izvor' u '$cilj': $!\n";
```

Najvažniji deo ovog programa je poslednji izraz, ali je ostatak programa neophodan zbog preimenovanja u direktorijumu. Posle deklarisanja korišćenih modula, zadajemo imena argumenata u komandnoj liniji. Ukoliko je promenljiva `$cilj` direktorijum, moramo saznati osnovno ime iz sadržaja promenljive `$izvor` i nadovezati ga na ime direktorijuma (nalazi se u promenljivoj `$cilj`). Kada se završi rad s promenljivom `$cilj` (ako je potrebno), na scenu stupa funkcija `rename`.

6. Evo jednog od načina da se to uradi:

```
use File::Basename;
use File::Spec;
my($izvor, $cilj) = @ARGV;
if (-d $cilj) {
  my $osnovno_ime = basename $izvor;
  $cilj = File::Spec->catfile($cilj, $osnovno_ime);
}
link $izvor, $cilj
  or die "Ne mogu da povežem '$izvor' i '$cilj': $!\n";
```

Kao što je nagovešteno u postavci zadatka, ovaj program treba da liči na prethodni. Razlika je u tome što se umesto funkcije `rename` koristi funkcija `link`. Ako vaš sistem ne podržava čvrste veze, umesto poslednjeg izraza mogli ste napisati sledeće:

```
print "Trebalo bi pozvati funkciju za povezivanje '$izvor' i '$cilj'.\n";
```

7. Evo jednog od načina da se to uradi:

```
use File::Basename;
use File::Spec;
my $simbolicka_veza = $ARGV eq '-s';
```

```
shift @ARGV if $simbolicka_veza;
my($izvor, $cilj) = @ARGV;
if (-d $cilj) {
  my $osnovno_ime = basename $izvor;
  $cilj = File::Spec->catfile($cilj, $osnovno_ime);
}
if ($simbolicka_veza) {
  symlink $izvor, $cilj
    or die "Ne mogu da napravim simboličku vezu od '$izvor' do '$cilj': $!\n";
} else {
  link $izvor, $cilj
    or die "Ne mogu da napravim čvrstu vezu od '$izvor' do '$cilj': $!\n";
}
```

Prvih nekoliko redova koda (posle dve deklaracije `use`) ispituju prvi argument komandne linije: ako je on `-s`, pravi se simbolička veza i dodeljuje se promenljivoj `$simbolicka_veza`. Ukoliko je prvi argument `-s`, moramo ga se osloboditi u narednom redu. Sledećih nekoliko redova kopirani su iz rešenja prethodnih zadataka. Na kraju, u zavisnosti od sadržaja promenljive `$simbolicka_veza`, pravi se ili simbolička ili čvrsta veza. Ažurirali smo upozorenje o neuspešno obavljenoj operaciji da bismo jasno stavili do znanja o kojoj vezi se radi.

8. Evo jednog od načina da se to uradi:

```
foreach (<.* *>) {
  my $cilj = readlink $_;
  print "$_ -> $cilj\n" if defined $cilj;
}
```

Svi rezultati rada operatora `glob` smeštaju se u promenljivu `$_`. Ako je rezultat simbolička veza, funkcija `readlink` vraća definisanu vrednost i prikazuje se lokacija. U suprotnom, uslov nije ispunjen, pa se izvršavanje naredbe preskače.

Rešenja vežbi iz trinaestog poglavlja

1. Evo jednog od načina da se to uradi:

```
my @brojevi;
push @brojevi, split while <>;
foreach (sort { $a <=> $b } @brojevi) {
  printf "%20g\n", $_;
}
```

Drugi red ovog koda vas zbunjuje, zar ne? To smo namerno uradili. Iako vam preporučujemo da pišete razumljiv kôd, neki programeri vole da pišu kôd koji se vrlo teško razume,* pa smo hteli da vas pripremimo na najgore. Ponekad ćete morati da održavate zbunjujući kôd kao što je ovaj.

* Ne preporučujemo ovakav način pisanja koda za *uobičajene* potrebe, ali je svakako zanimljivo pisati zbunjujući kôd, a može biti i poučno kada provedete ceo vikend pokušavajući da protumačite nečiji nerazumljiv kod. Ako vas zanimaju primeri nerazumljivog koda, raspitajte se na sastanku grupe Perl Mongers, potražite JAPH-ove na Webu ili pogledajte primer zbunjujućeg koda pri kraju rešenja zadataka iz ovog poglavlja.

Budući da se u tom redu koristi modifikator `while`, on se može napisati i pomoću petlje na sledeći način:

```
while (<>) {
  push @brojevi, split;
}
```

Ovo je bolje, mada možda još uvek nije potpuno razumljivo. (Ipak, ne smeta nam ovakav način pisanja koda, jer se nalazi sa ispravne strane linije „previše složeno da bi se razumelo na prvi pogled“.) Petlja `while` čita po jedan red (sa ulaza koji je korisnik izabrao, kao što je prikazano pomoću operatora dijamant). Funkcija `split` deli ulazni red po belinama da bi se dobila lista reči ili – u ovom slučaju – brojeva. Ulazni podaci su brojevi odvojeni belinama. Kako god da ih napišete, petlja `while` će ih sve smestiti u niz `@brojevi`.

Petlja `foreach` prihvata sortiranu listu i ispisuje njene elemente u zasebnim redovima koristeći format `%20g` da bi ih poravnala udesno. Mogli ste koristiti i format `%20s`. Kakva je razlika? To je format za znakovne nizove, tako da bi oni ostali neizmenjeni. Format `%20g` je numerički format, pa će jednaki brojevi biti prikazani identično. Oba formata su potencijalno tačna, u zavisnosti od toga šta želite da postignete.

2. Evo jednog od načina da se to uradi:

```
# ne zaboravite da iskoristite heš %prezime (tj. %last_name),
# iz teksta zadatka ili iz preuzete datoteke
my @kljucevi = sort {
  "\L$prezime{$a}" cmp "\L$prezime{$b}"  # po prezimenu
   or
  "\L$a" cmp "\L$b"                      # po imenu
} keys %prezime;
foreach (@kljucevi) {
  print "$prezime{$_}, $_\n";            # Rubble, Bamm-Bamm
}
```

Nema se mnogo šta reći o ovom programu. Ključevi se sortiraju po traženom redosledu, a zatim se ispisuju. Mi smo odlučili da ih ispišemo po redosledu „prezime, ime“, mada u tekstu zadatka nije navedeno koji redosled treba koristiti.

3. Evo jednog od načina da se to uradi:

```
print "Unesite znakovni niz: ";
chomp(my $znakovni_niz = <STDIN>);
print "Unesite deo znakovnog niza: ";
chomp(my $podniz = <STDIN>);
my @mesta;
for (my $pozicija = -1; ; ) {             # čudno korišćenje petlje iz tri dela
  $pozicija = index($znakovni_niz, $podniz, $pozicija + 1); # pronalaženje
                                                            # sledeče pozicije
  last if $pozicija == -1;
  push @mesta, $pozicija;
}
print "Pozicije '$podniz' u '$znakovni_niz' su: @mesta\n";
```

Ovaj program zahteva od korisnika da unese znakovni niz i jedan njegov podniz, a zatim deklariše niz koji će sadržati pozicije podniza. Kao što vidimo u petlji `for`, kôd je „optimizovan za pametnjakoviće“, što treba raditi samo iz zabave, a nikako u kodu koji ćete negde koristiti. Ovde prikazujemo ispravnu tehniku, koja može biti korisna u nekim slučajevima, pa da vidimo kako ona radi.

Promenljiva `$pozicija` deklariše se kao privatna za oblast važenja petlje i njena početna vrednost je `-1`. Da vas ne bismo držali u neizvesnosti, odmah ćemo vam reći da će ona sadržati poziciju podniza u velikom znakovnom nizu. Odeljci za proveru i inkrementiranje petlje `for` prazni su, što znači da je u pitanju beskonačna petlja. (Na kraju ćemo je ipak napustiti pomoću operatora `last`.)

Prva naredba u petlji `for` pronalazi prvo pojavljivanje podniza na poziciji `$pozicija + 1` (ili posle nje). To znači da će u prvom prolazu kroz petlju (kada je vrednost promenljive `$pozicija` još uvek `-1`) pretraživanje početi od pozicije nula, tj. od početka znakovnog niza. Pozicija podniza se ponovo smešta u promenljivu `$pozicija`. Ukoliko je pozicija `-1`, petlja se završava i iz nje se izlazi pomoću operatora `last`. Ako pozicija nije `-1`, smešta se u niz `@mesta` i ponovo se prolazi kroz petlju. U tom slučaju pozicija `$pozicija + 1` znači da se pretraživanje nastavlja iza prethodno zapamćene pozicije. Tako ćemo pronaći sve pozicije, a svet će opet biti srećnije mesto.

Ukoliko vam se ne sviđa čudna upotreba petlje `for`, isti rezultat ste mogli postići i pomoću sledećeg koda:

```
{
  my $pozicija = -1;
  while (1) {
    ... # Isto telo petlje kao u prethodnom primeru
  }
}
```

Spoljašnji goli blok ograničava oblast važenja promenljive `$pozicija`. To ne morate da radite, ali je najčešće pametno deklarisati svaku promenljivu za najmanju moguću oblast važenja. Na taj način će manji broj promenljivih biti „živ“ u svakom trenutku izvršavanja programa, a smanjuju se i šanse da se promenljiva `$pozicija` greškom upotrebi u druge svrhe. Iz istog razloga, ako promenljive ne deklarišete za malu oblast važenja, trebalo bi da im date duža imena da ih ne biste iskoristili u pogrešne svrhe. U ovom slučaju dobar izbor bilo bi ime `$pozicija_podniza`.

S druge strane, ako želite da otežate razumevanje koda (sram vas bilo), možete napraviti čudovište slično ovom (sram nas bilo):

```
for (my $pozicija = -1; -1 !=
  ($pozicija = indeks
    +$znakovni_niz,
    +$podniz,
    +$pozicija
    +1
  );
```

```
push @mesta, ((((+$pozicija))))) {
    'for ($pozicija != 1; # ;$pozicija++) {
      print "pozicija $pozicija\n";#;';#' } pop @mesta;
}
```

Ovaj kôd može zameniti izvornu čudnu petlju `for`. Trebalo bi da budete u stanju da ga protumačite ili svoj kôd učinite nerazumljivim da biste zadivili prijatelje i zbunili neprijatelje. Koristite ove moći samo za dobra dela, nikako za nanošenje zla.

Šta ste dobili kao rezultat kada ste tražili slovo `t` u znakovnom nizu `Test je test`? Dobili ste pozicije `8` i `11`, ali ne i poziciju `0`. Budući da se velika slova ne uparuju, neće se upariti ni slovo `T`.

Rešenja vežbi iz četrnaestog poglavlja

1. Evo jednog od načina da se to uradi:

```
chdir "/" or die "Ne mogu da pređem u glavni direktorijum: $!";
exec "ls", "-l" or die "Ne mogu da izvršim ls: $!";
```

   Prvom naredbom se za tekući direktorijum bira glavni direktorijum. U drugom redu se koristi funkcija `exec` s više argumenata, koja šalje podatke na standardni izlaz. Mogli smo da koristimo i samo jedan argument, ali i ovako je dobro.

2. Evo jednog od načina da se to uradi:

```
open STDOUT, ">ls.out" or die "Ne mogu da ispišem podatke u ls.out: $!";
open STDERR, ">ls.err" or die "Ne mogu da ispišem podatke u ls.err: $!";
chdir "/" or die "Ne mogu da pređem u glavni direktorijum: $!";
exec "ls", "-l" or die "Ne mogu da izvršim ls: $!";
```

   Prva i druga naredba pre prelaska u novi direktorijum ponovo otvaraju identifikatore `STDOUT` i `STDERR`, koji ukazuju na datoteku iz tekućeg direktorijuma. Posle prelaska u novi direktorijum, izvršava se komanda za prikazivanje njegovog sadržaja, koja šalje podatke u datoteke otvorene u izvornom direktorijumu.

   Gde bi se pojavila poruka ako bi se izvršila poslednja naredba `die`? Pojavila bi se u datoteci *ls.err* jer na nju pokazuje identifikator `STDERR`. Na istom mestu bi se pojavila i poruka naredbe `die` komande `chdir` iz drugog reda. Gde bi se poruka pojavila ako ne bismo mogli ponovo da otvorimo identifikator `STDERR`? Pojavila bi se u datoteci na koju pokazuje stari identifikator `STDERR`. Ako ne uspe ponovno otvaranje tri standardna identifikatora datoteka – `STDIN`, `STDOUT` i `STDERR` – stari identifikatori će i dalje biti otvoreni.

3. Evo jednog od načina da se to uradi:

```
if (`date` =~ /^S/) {
  print "idi da se igraš!\n";
} else {
  print "idi na posao!\n";
}
```

Budući da i subota (engl. *Saturday*) i nedelja (engl. *Sunday*) počinju slovom `S`, a dan u nedelji je prvi deo rezultata komande `date`, ovo rešenje je prilično jednostavno. Proverite rezultat komande `date` kako biste se uverili da dani počinju slovom `S`. Postoji mnogo težih načina da se reši ovaj zadatak i većinu njih smo videli na našim kursevima.

Da smo ovaj program morali da koristimo u praksi, verovatno bismo upotrebili šablon `/^(Sat|Sun)/`. On je neznatno efikasniji, ali to nije važno. Važnije je to što ga programer koji održava kôd lakše može razumeti.

Rešenja vežbi iz petnaestog poglavlja

1. Evo jednog od načina da se to uradi:

```
#!/usr/bin/perl
use Module::CoreList;
my %moduli = %{ $Module::CoreList::version{5.006} };
print join "\n", keys %moduli;
```

   Rešenje koristi referencu heša (informacije o njoj moraćete da potražite u knjizi sa alpakom, a mi smo vam objasnili ono najpotrebnije. Ne morate da znate kako ovaj program radi, bitno je samo da znate da radi. Obavite posao, a detalje proučite kasnije.)

2. Postoji nekoliko načina da se pristupi ovom problemu. Mi smo koristili modul `C<Cwd>` (Current Working Module) da bismo saznali gde se nalazimo u okviru sistema datoteka. Upotrebili smo operator `glob` da bismo dobili spisak svih datoteka u tekućem direktorijumu; imena ne moraju da sadrže informacije o direktorijumima, pa smo zato morali da iskoristimo operator. Mogli ste iskoristiti i naredbu `C<opendir>g`, ali je operator `C<glob>g` kraći. Šablon za operator `glob` je `C<.*>` da bi se mogle pročitati i Unixove skrivene datoteke, koje se ne mogu dobiti pomoću šablona `C<*>`.

   Pošto smo dobili spisak datoteka, prolazimo kroz njega pomoću petlje `C<foreach>`. Za svako ime datoteke poziva se funkcija `C<File::Spec->catfile()>`, kao što je prikazano u dokumentaciji. Rezultat se smešta u promenljivu `C<$putanja>`, čiji se sadržaj ispisuje na standardnom izlazu:

```
#!/usr/bin/perl

use Cwd;        # Current Working Directory
use File::Spec;

my $cwd = getcwd;
my @datoteke = glob ".* *";

foreach my $datoteka ( @datoteke )
{
my $putanja = File::Spec->catfile( $cwd, $datoteka );
print "$putanja\n";
}
```

3. Ovo rešenje je jednostavnije od prethodnog, mada prvo morate napisati prethodni program da biste mogli koristiti ovaj. Ceo posao se obavlja u petlji `C<while>`. Za svaki ulazni red poziva se funkcija `C<basename>` iz modula `C<File::Basename>`. Detalje o njenom korišćenju saznali smo iz dokumentacije za modul `C<File::Basename>`. Ulazni red se direktno prosleđuje funkciji `C<basename>`, a rezultat se smešta u promenljivu `C<$ime>`. Budući da na ulazne redove ne primenjujemo funkciju `C<chomp>`, promenljiva `C<$ime>` sadržaće oznaku za novi red, tako da slobodno možemo ispisati imena datoteka u zasebnim redovima:

```
#!/usr/bin/perl

use File::Basename;

while( <STDIN> )
{
my $ime = basename( $_ );

print $ime;
}
```

Rešenja vežbe iz šesnaestog poglavlja

1. Evo jednog od načina da se to uradi:

```
my $ime_datoteke = 'putanja/do/teksta';
open FILE, $ime_datoteke
  or die "Ne mogu da otvorim '$ime_datoteke': $!";
chomp(my @znakovni_nizovi = <FILE>);
while (1) {
  print "Unesite šablon: ";
  chomp(my $sablon = <STDIN>);
  last if $sablon =~ /^\s*$/;
  my @upareni = eval {
    grep /$sablon/, @znakovni_nizovi;
  };
  if ($@) {
    print "Error: $@";
  } else {
    my $brojac = @upareni;
    print "Upareno je $brojac znakovnih nizova:\n",
      map "$_\n", @upareni;
  }
  print "\n";
}
```

Ovaj program koristi blok `eval` za otkrivanje grešaka koje bi se mogle pojaviti prilikom korišćenja regularnih izraza. Unutar tog bloka, funkcija `grep` prihvata uparene znakovne nizove iz liste.

Kada blok `eval` obavi posao, prikazuje se poruka o grešci ili upareni znakovni niz. Svakom znakovnom nizu se pre ispisivanja dodaje oznaka za novi red.