Uvod u programski jezik Perl

Programski jezik Perl

- autor: Larry Wall, 1987
- *Practical Extraction and Report Language*
- *Pathologically Eclectic Rubbish Lister*
- Perl je nastao kao alat kojeg je razvio Larry Wall dok je nastojao generirati neka izvješća, a primjena skripti u Unix ljusci i alata poput `sed`-a, i `grep`-a mu nije bila dovoljna
- kao “lijeni” programer, odlučio je oblikovati alat koji će mu olakšati posao
  - brzina kodiranja poput programiranja u Unix ljusci
  - mogućnosti naprednijih alata kao što su `grep, sort, sed,...`

Programski jezik Perl (2)

- Perl nastoji ispuniti prazninu između programiranja niske razine (`C, C++, asembler`) i programiranja visoke razine (ljuska OS-a)
  - programiranje niske razine – “teško”, ali bez ograničenja, uz dobro programiranje maksimalna brzina izvođenja na danom računalu
  - programiranje visoke razine – relativno lako i brzo pisanje koda, ali programi su najčešće spori, a mogućnosti su ograničene na naredbe koje su nam na raspolaganju
  - u Perlu je relativno lako programirati, malo je ograničenja, izvođenje je uglavnom brzo
- zahvaljujući svojoj snazi i fleksibilnosti Perl se nametnuo kao jedan od najpopularnijih programskih jezika, posebno u domeni WWW programiranja, obrade teksta i administracije sustava

Programski jezik Perl (3)

- Perl ima potpunu podršku regularnim izrazima
- podržava objektno-orijentirano programiranje
- ima podršku za mrežno programiranje
- omogućava upravljanje procesima
- proširiv je i podržava razvoj prenosivih programa
- moto:
  "There's More Than One Way To Do It,"

Perl danas

- Larry Wall i dalje vodi razvoj Perl-a
- razvojni tim danas broji 30-tak ključnih ljudi, uz više stotina drugih koji doprinose razvoju, sa svih strana svijeta
- Perl je besplatan i otvorenog koda

Za što je Perl prikladan ?

- za kratke programe koje treba napisati u nekoliko minuta
- može se primijeniti i za velike programe
- optimiran za probleme koji su 90% rada s tekstom i 10% ostalog – čini se da to odgovara većini programskih zadataka danas
- Perl je vrlo prikladan za generiranje HTML dokumenata, vrlo je čest za pisanje CGI skripti

Za što Perl nije prikladan ?

- izgradnja izvršnih programa u binarnom obliku, koji se prodaju uz želju da korisnik ne može doći do algoritama -> Perl programe tipično dajemo u izvornom obliku

Literatura i Internet resursi

- *Perl homepage* (http://www.perl.com/)
- *Comprehensive Perl Archive Network* (http://www.perl.com/CPAN/)
- Perl dokumentacija – može se pregledavati korištenjem naredbe `perldoc`
- stranice priručnika (*man pages*) `man perl`
- Larry Wall, Tom Christiansen, and Jon Orwant, *Programming Perl* ("Camel Book"), 3rd edition, O'Reilly, 2000
- Tom Phoenix, Randal L. Schwartz, *Learning Perl* ("Llama Book"), 3rd Edition, O'Reilly, 2001
- *Robert's Perl Tutorial* (link)

Kako napisati i pokrenuti Perl program ?

- Perl program je obična tekst datoteka – za pisanje je dovoljan tekst editor.

```
#!/usr/bin/perl
print "Hello, world!\n";
```

```
$ chmod a+x my_program.pl
```

```
$ ./my_program.pl
```

- jezik “slobodne forme” – prazni znakovi mogu se slobodno koristiti
- komentari – od znaka `"#"` do kraja retka

```
#!/usr/bin/perl
   print  # Ovo je komentar
   "Hello, world!\n"
   ;   # Nemojte ovako pisati svoje programe :-)
```

Pisanje i pokretanje Perl programa

- `#!/usr/bin/perl` – najmanje portabilan dio Perl programa, jer smještaj Perl interpretera može varirati
- i na *non-Unix* sustavima, tradicionalno se koristi prvi redak istog tipa: `#!perl`
  - ako ništa drugo – čini očiglednim da se radi o Perl programu
  - neće pomoći pokretanju programa pod Windowsima :-)
    `perl my_program.pl`

Skalarni podaci

- *skalar* je najjednostavnija vrsta podataka u Perlu
  - broj – npr. `255` ili `3.25e20`
  - niz znakova (*string*)
  - Perl brojeve i nizove znakova koristi na vrlo sličan način – kao pojedinačne, skalarne vrijednosti !
- na skalarne vrijednosti mogu se primijeniti operatori (zbrajanje, ulančavanje i sl.), pri čemu se obično dobija skalarna vrijednost

Brojevi

- svi brojevi se interno pohranjuju u istom formatu !
  - cijeli brojevi (`255, 2008`)
  - realni brojevi (`3.14159`)
  - interno, Perl računa s brojevima u zapisu s pomičnim zarezom dvostruke točnosti !
  - točna preciznost (*double-precision floating-point*) ovisi o prevoditelju kojim je Perl preveden, tj. ovisi o računalu
  - tipično – IEEE format s 15 znamenaka mantise i rasponom od `1e-100` do `1e100`

Brojevi (2)

- literali – vrijednosti zapisane u izvornom kodu (konstante)
- brojevi s pomičnom točkom
  - primjeri:

```
1.25                    -12e-24
255.000                 -1.2E-23
7.25e45
```

- cijeli brojevi
  - primjeri:

```
0
2008
-40
61298040283768
```

  - može i ovako (čitljivost):

```
61_298_040_283_768
```

Brojevi (3)

- nedekadski literali:
  - oktalni literali započinju vodećom nulom
    `0377` = 255_{10}
  - heksadekadski započinju s `0x`, koriste se i znamenke `A-F` ili `a-f`
    `0xff` = 255_{10}
  - binarni započinju s `0b`
    `0b11111111` = 255_{10}
    (raspoloživo od verzije 5.6)
  - od verzije 5.6 mogu se koristiti znakovi `"_"` za grupiranje znamenaka
    `0x50_65_72_7C`

Aritmetički operatori

- Perl koristi uobičajene aritmetičke operatore (`+, -, *, /,...`)

```
2 + 3
5.1 - 2.4
3 * 12
14 / 2
10 / 3    # FP aritmetika! => 3.3333333...
10 % 3    # modulo
10.5 % 3.2 # operandi se reduciraju na cjelobrojne
2**3      # 2^3 = 8
```

Znakovni nizovi

- niz *proizvoljnih* znakova (*string*) => moguće baratanje sa “sirovim” binarnim podacima
- znakovni nizovi u Perlu ne zaključuju se nul-znakom (`"\0"`)
- znakovni nizovi uokviruju se *jednostrukim* ili *dvostrukim* navodnicima – različita interpretacija !
- znakovni niz uokviren jednostrukim navodnicima:
  - svaki znak osim jednostrukog navodnika (`"'"`) ili kose crte (`"\"`), uključujući prelaske u novi red, predstavlja sam sebe
  - kosa crta uključuje se u niz kao `"\\"`, jednostruki navodnik kao `"\'"`
  - unutar jednostrukih navodnika *ne interpretiraju se* posebne sekvence poput `"\n"` !

Znakovni nizovi (2)

- znakovni niz uokviren dvostrukim navodnicima:
  - interpolacija varijabli (zamjena imena vrijednošću)
  - kosa crta (`"\"`) koristi se za označavanje posebnih znakova (*backslash escapes*) poput `"\n"`
  - značenja *nekih* posebnih znakova:

`\n`	novi red
`\t`	tabulator
`\007`	oktalna ASCII vrijednost
`\x7f`	heksadekadska ASCII vrijednost
`\l`	sljedeći znak prebaci u malo slovo
`\L`	prebaci u mala slova sve znakove do `\E`
`\u`	sljedeći znak prebaci u veliko slovo
`\U`	prebaci u velika slova sve znakove do `\E`
`\E`	završetak slijeda nakon `\L` i `\U`

Operatori nad znakovnim nizovima

- nadovezivanje: operator `"."`

```
"hello" . "world" # "helloworld"
"hello" . ' ' . "world" # 'hello world'
'hello world' . "\n" # "hello world\n"
```

- ponavljanje niza: operator `"x"`

```
"fred" x 3   # => "fredfredfred"
"barney" x (3+1) # => "barney" x 4, tj.
                   # "barneybarneybarneybarney"
5 x 4      # zapravo "5" x 4, tj. "5555"
```

  - uočiti: implicitna pretvorba tipova
  - necijeli broj ponavljanja se reducira na cijeli, ako je broj ponavljanja < 1, dobiva se prazni niz

Automatska pretvorba između brojeva i nizova

- ovisno o *operatoru* koji se koristi, Perl obavlja pretvorbu brojeva u nizove znakova ili obrnuto
- ako se nad skalarnim vrijednostima primjenuje numerički operator (+, -, *,...) vrijednost se koristi kao numerička
- za operatore koji su definirani za znakovne nizove (`.`, `x`) brojevi se koriste kao nizovi znakova

  npr. `"12" * "3"` daje vrijednost `36`
- nenumerički nastavak niza se odbacuje, kao i početne praznine

  `"12fred34" * " 3"` također daje `36`
- niz koji nije broj pretvara se u vrijednost `0`
- brojevi se prema potrebi pretvaraju u nizove:

  `"Z" . 5 * 7 # kao "Z" . 35` daje vrijednost `"Z35"`

Ugrađena upozorenja

- generiranje upozorenja (*warnings*) može se uključiti pri pozivu Perl interpretera korištenjem opcije `-w`

  ```
  $ perl -w my_program
  ```

  ili pri svakom pokretanju programa:

  ```
  #!/usr/bin/perl -w
  ```

- više detalja i objašnjenja pojedinih upozorenja:

  ```
  $ man perldiag
  ```

Skalarne varijable

- imena skalarnih varijabli započinju znakom `"$"`, a slijedi *Perl identifikator*: niz slova, znamenaka i podvlaka (`"_"`), pri čemu prvi znak ne može biti znamenka
- velika i mala slova se razlikuju, a svi su znakovi značajni: `$Count` i `$count` su različite varijable `$a_very_long_variable_that_ends_in_1` i `$a_very_long_variable_that_ends_in_2` su različite varijable
- skalarne varijable u Perlu se *uvijek* referenciraju predznačene znakom `"$"` !

Skalarno dodjeljivanje vrijednosti

- dodjela vrijednosti varijabli, operator "="

```
$fred = 17;
$barney = 'hello';
$barney = $fred + 3;
$barney = $barney * 2
```

- izrazi u kojima se ista varijabla pojavljuje i na lijevoj i na desnoj strani su vrlo česti, pa Perl poput jezika C i Java uvodi kraći zapis

```
$fred += 5;
$barney *= 2;
$str .= " "; # vrijedi i za ulancavanje
$x **= 3; # x=x^3
```

Ispis pomoću operatora `print()`

- operator `print( )` ispisuje skalarnu vrijednost navedenu kao argument na standardni izlaz
  - zagrade nije nužno pisati

```
print "hello world\n";
print 6 * 7;
print (".\n");
```

  - može se navesti i niz argumenata

```
print "Rezultat je ", 6 * 7, ".\n";
   # niz vrijednosti odvojenih zarezima
```

Interpolacija skalarnih varijabli u nizovima

- kada je znakovni niz naveden u dvostrukim navodnicima, podložan je interpolaciji varijabli
  - ime varijable navedeno u znakovnom nizu zamjenjuje se vrijednošću varijable

```
$meal = "brontosaurus steak";
$barney = "fred ate a $meal";
$barney = 'fred ate a ' . $meal; # isti rezultat
```

- ako skalarnoj varijabli nije dodijeljena vrijednost (odnosno vrijednost joj *nije definirana*), ispisuje se *prazni niz*
  - ako su uključena upozorenja, Perl će se "požaliti"
- da bi u niz uključili znak `"$"` u doslovnom značenju, treba ga predznačiti silaznom kosom crtom

Interpolacija skalarnih varijabli (2)

- primjeri

```
$fred = 'hello';
print "Ime je \$fred.\n"; # ispisuje znak dollar
print 'Ime je $fred' . "\n"; # isto
```

- kao ime varijable upotrijebit će se *najdulji* mogući podniz koji ima smisla
  - to može biti problem

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n"; # varijabla je $whats
print "fred ate $n ${what}s.\n"; # sad je ime $what
print "fred ate $n $what" . "s.\n"; # moze i ovako
```

Operatori i redoslijed primjene

- za uobičajene operatore koji se koriste i u C-u vrijede i jednaki redoslijedi primjene
- zagrade mijenjaju redoslijed primjene operatora
- tablica – prednost opada prema dnu tablice

asocijativnost	operatori
lijeva	zagrade i argumenti operatora listi
lijeva	`->`
	`++ --`
desna	`**`
desna	`\ ! ~ + -` (unarni operatori)
lijeva	`=~ !~`
lijeva	`* / % x`
lijeva	`+ - .` (binarni operatori)
lijeva	`<< >>`

Operatori i redoslijed primjene (2)

- nastavak tablice

asocijativnost	operatori
	`< <= > >= lt le gt ge`
	`== != <=> eq ne cmp`
lijeva	`&`
lijeva	`| ^`
lijeva	`&&`
lijeva	`||`
lijeva	`.. ...`
desna	`?:` (ternarni operator)
desna	`= += -= .=` (i slični operatori pridruživanja)
lijeva	`, =>`
desna	`not`
lijeva	`and`
lijeva	`or xor`

Operatori i redoslijed primjene (3)

- redoslijed primjene kod operatora iste razine razrješava se pravilima asocijativnosti

```
4 ** 3 ** 2 # 4 ** (3 ** 2) = 4 ** 9 (desna asoc.)
72 / 12 / 3 # (72 / 12) / 3 = 6/3 = 2 (l. asoc.)
36 / 6 * 3 # (36/6)*3 = 18
```

- u slučaju nedoumice o redoslijedu primjene operatora dobro je koristiti zagrade

Operatori usporedbe

- operatori za usporedbu brojeva: `<` `<=` `==` `>=` `>` `!=` vraćaju vrijednost *true* ili *false*
- za usporedbu znakovnih nizova Perl ima slične operatore:
  `lt le eq ge gt ne`
  - ispituje se jesu li dva znakovna niza jednaka, odnosno koji od njih je prije u standardnom redoslijedu sortiranja
  - velika slova se u ASCII kodu nalaza prije malih

usporedba	brojevi	znakovni nizovi
jednako	==	eq
različito	!=	ne
manje od	<	lt
veće od	>	gt
manje ili jednako	<=	le
veće ili jednako	>=	ge

Operatori usporedbe (2)

- primjeri

```
35 != 30 + 5    # false
35 == 35.0      # true
'35' eq '35.0'  # false (usporedba nizova)
'fred' lt 'barney' # false
'fred' lt 'free' # true
'fred' eq "fred" # true
'fred' eq 'Fred' # false
' ' gt ''       # true
```

Ispitivanje uvjeta

- naredba **`if`**

```
if ($name gt 'fred') {
  print "'$name' se sortira nakon 'fred'.\n";
}

if ($name gt 'fred') {
  print "'$name' se sortira nakon 'fred' .\n";
 } else {
  print "'$name' se ne sortira nakon 'fred'.\n";
  print "a mozda je to isti niz.\n";
 }
```

- za razliku od C-a, *vitičaste zagrade se zahtijevaju* !

Logičke vrijednosti

- pri ispitivanju uvjeta može se koristiti bilo koja skalarna vrijednost
- to je prikladno ako želimo pohraniti vrijednost istinitosti u varijablu

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

- Perl nema poseban logički tip podatka (*boolean*), već se koristi nekoliko jednostavnih pravila:
  - nedefinirana vrijednost odgovara logičkoj vrijednosti *false*
  - nula odgovara logičkoj vrijednosti *false*, ostali brojevi su *true*
  - prazan niz (`''`) je *false*, ostali nizovi su *true*
    - iznimka je niz `'0'` – njegova vrijednost istinitosti je *false*
- negacija logičkog izraza postiže se operatorom `!`

```
if (! $is_bigger) { ... }
```

Učitavanje podataka

- podaci s tipkovnice mogu se učitati operatorom `<STDIN>`
- kada se `<STDIN>` upotrijebi na mjestu gdje se očekuje skalarna vrijednost, Perl učitava cijeli redak teksta sa standardnog ulaza, sve do prve oznake novog reda
- tipično, vrijednost `<STDIN>` završava znakom `\n`

```
$line = <STDIN>;
if ($line eq "\n") {
  print "To je samo prazni redak!\n";
} else {
  print "Ucitani redak je: $line";
}
```

- u praksi ne želimo zadržati `\n` na kraju učitanog znakovnog niza

Funkcija chomp

- operator `chomp` uklanja oznaku kraja reda iz znakovnog niza

```
$text = "redak teksta\n"; # ili niz ucitan sa <STDIN>
chomp($text);        # skida \n
```

- često se koristi pri učitavanju teksta

```
chomp($text = <STDIN>); # ucitavanje teksta bez \n
```

- `chomp` je *funkcija* – ima povratnu vrijednost, koja predstavlja broj uklonjenih znakova (nije baš korisno :-)

```
$redak = <STDIN>;
$broj = chomp $redak # $broj je 1, to smo vec znali
```

- `chomp` možemo upotrijebiti i *bez zagrada*
  - zagrade nisu obavezne osim ako njihovo uklanjanje može promijeniti značenje izraza !
- u starijim Perl programima možemo naići na funkciju `chop` – ona uklanja proizvoljan završni znak niza, a ne samo `\n`

Petlja `while`

- `while` petlja ponavlja blok naredbi dok god je uvjet ispunjen (*true*)

```
$count = 0;
while ($count < 10) {
  $count += 1;
  print "broj je sada $count\n";
                # ispisuje brojeve 1 do 10
}
```

- uvjet se ponaša kao i u naredbi `if`
- vitičaste zagrade su *obavezne*
- uvjet se ispituje *prije* prvog ulaska u petlju, pa petlja može biti preskočena ako uvjet od početka nije ispunjen

Nedefinirana vrijednost

- prije nego se varijabli prvi put dodijeli vrijednost, ona ima posebnu vrijednost `undef`
- ako se takva varijabla upotrijebi na mjestu gdje se očekuje znakovni niz, ona se ponaša kao prazni niz
- ako se nedefinirana varijabla upotrijebi na mjestu broja, njena je vrijednost 0, što se može upotrijebiti umjesto inicijalizacije:

```
# zbrajanje neparnih brojeva
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # sljedeci neparni broj
}
print "Suma je $sum.\n";
```

- slično se mogu koristiti i nizovi:

```
$string .= "more text\n";
```

Funkcija `defined`

- ako želimo ispitati je li neka varijabla definirana ili ne možemo primjeniti funkciju `defined`
- npr. `<STDIN>` može vratiti vrijednost `undef`, ako se upis retka prekine oznakom kraja datoteke (`EOF`, `ctrl-D`)

```
$redak = <STDIN>;
if ( defined($redak) ) {
  print "Ucitani redak je: $redak";
} else {
  print "Redak nije ucitan!\n";
}
```

Liste i polja

- lista je uređeni niz skalarnih vrijednosti
- polje je varijabla koja sadrži listu
- u Perlu se ta dva pojma koriste kao jednakoznačni, no ako želimo biti precizni – lista je skup podataka, a polje je varijabla
- lista ne mora biti spremljena u polje, ali polje uvijek sadrži listu (koja može biti i prazna)
- svaki element polja ili liste je odvojena skalarna varijabla s nezavisnom vrijednošću – lista može sadržavati brojeve, znakovne nizove ili mješavinu različitih skalarnih vrijednosti
- elementi liste su uređeni (poredani) i indeksirani
- prvi element liste ima indeks 0, broj elemenata je ograničen samo raspoloživim memorijskim prostorom

Pristup elementima polja

- elementi polja su indeksirani slijednim cijelim brojevima počevši od 0

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

- ime polja nalazi se u potpuno odvojenom *prostoru imena* (*namespace*) u odnosu na skalarne varijable !
  - možemo imati skalarnu varijablu jednakog imena (`$fred`)
  - sintaksa je uvijek nedvosmislena, mada ponekad zbunjujuća
- svaki element polja može se koristiti gdje god se može koristiti skalarna varijabla
  - ima i izuzetaka – npr. upravljačka varijabla `foreach` petlje mora biti jednostavan skalar

Pristup elementima polja (2)

- indeks može biti bilo koji izraz koji daje numeričku vrijednost
  - ako vrijednost nije cjelobrojna, reducira se na cjelobrojnu

```
$broj = 2.71828;
print $fred[$broj - 1]; # isto kao print $fred[1]
```

- ako se indeksira "element" iza kraja polja, njegova je vrijednost `undef`
- ako se pohrani vrijednost u element polja iza njegovog kraja, polje se automatski proširuje – nema ograničenja osim raspoloživog memorijskog prostora
- prazni elementi polja koji pritom mogu nastati imaju vrijednost `undef`

```
$polje[0] = 'nula'; # prvi element
$polje[1] = 'jedan'; # drugi element
$polje[33] = 'tri_tri'; # nastaje 31 undef element
```

Pristup elementima polja (3)

- ponekad je potrebno saznati indeks posljednjeg elementa u polju
- u Perl-u se taj indeks može dobiti s `$#polje`
  - to je indeks posljednjeg elementa a ne broj elemenata
  - manipulacijom ove vrijednosti mijenja se i veličina polja

```
$end = $#polje; # u ovom slucaju 33
$broj_elemenata = $end + 1;
$#polje = 5; # polje se smanjilo,
             # ostali elementi su izgubljeni
```

- korištenje posljednjeg indeksa u polju je često, pa je u Perl uvedena pokrata – negativni indeksi odbrojavaju se od *kraja* polja
  - indeks `-1` označava posljednji element
  - prekoračenje veličine negativnim indeksom generira pogrešku

```
$polje[ -1 ] = 'zadnji'; # posljednji element
$prvi = $polje[-6]; # 0-ti element u polju od 6 el.
$polje[ -50 ] = 'podbacaj'; # fatal error!
```

Liste kao literali

- liste se kao vrijednosti u programu (literali) navode kao niz vrijednosti odvojenih zarezima, unutar oblih zagrada

```
(1, 2, 3)
(1, 2, 3,) # ista lista, zarez na kraju se zanemaruje
("fred", 4.5)
( )    # prazna lista - 0 elemenata
(1..5) # isto kao (1, 2, 3, 4, 5) - range operator
(1.7..5.7) # ista lista - int
(5..1)  # prazna lista - nema odbrojavanja unazad
(0, 2..6, 10, 12) # lista (0, 2, 3, 4, 5, 6, 10, 12)
```

- elementi liste ne moraju nužno biti konstante – mogu biti i izrazi koji se evaluiraju svaki put kada se literal koristi

```
($a..$b) # raspon odredjuju vrijednosti $a i $b
(0..$#polje) # lista svih indeksa polja
($a, 17) # dva elementa: vrijednost $a i 17
($b+$c, $d+$e) # dva elementa
```

Kratica qw

- liste riječi su vrlo česte

```
("fred", "barney", "betty", "wilma", "dino")
```

- kratica `qw` omogućuje zapisivanje liste riječi bez potrebe za brojnim navodnicima (*quoted words*)

```
qw/ fred barney betty wilma dino / # ista lista
```

- riječi se koriste kao da su u *jednostrukim* navodnicima
  - nema tumačenja specijalnih sekvenci ni interpolacije varijabli !
  - praznine (nizovi razmaka, tabulatori i oznake novog reda) se zanemaruju

```
qw/fred
  barney   betty
wilma dino/ # ista lista, neobican stil
```

Kratica qw (2)

- u prethodnim primjerima se kao graničnik koristi znak "/", no Perl dozvoljava izbor proizvoljnog znaka interpunkcije kao graničnika:

```
qw! fred barney betty wilma dino !
qw# fred barney betty wilma dino #
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >

qw! yahoo\! google excite lycos ! # \ escape
```

- čemu to ? – npr. lista Unix datoteka

```
qw{
  /usr/dict/words
  /home/rootbeer/.ispell_english
}
```

Pridruživanje listi

- pridruživanje vrijednosti listama slično je pridruživanju vrijednosti varijablama

```
($fred, $barney, $dino) = ("jedan", "dva", undef);
```

- varijable u listi s lijeve strane znaka pridruživanja dobivaju nove vrijednosti kao u 3 odvojena pridruživanja
- jednostavna zamjena vrijednosti dviju varijabli:

```
($fred, $barney) = ($barney, $fred); # swap
```

- ako broj varijabli u listi s lijeve i vrijednosti u listi s desne strane operatora pridruživanja nije jednak
  - višak vrijednosti (desna strana) se zanemaruje
  - varijable viška dobivaju vrijednost `undef`

```
($fred, $barney) = (qw< jedan dva tri cetiri >);
                        # dvije vrijednosti viska
($wilma, $dino) = qw[flintstone]; # $dino je undef
```

Referenciranje polja

- za referenciranje cijelog polja koristi se jednostavnija notacija: ime polja predznači se znakom `"@"` i ne koriste se indeksne zagrade:

```
@polje = qw/ jedan dva tri /;
@tiny = ( ); # prazna lista
@giant = 1..1e5; # lista od 100.000 elemenata
@stuff = (@giant, undef, @giant); # 200.001 element
```

- polje može sadržavati samo skalare, a ne i polja – ime polja *zamjenjuje se* listom koju sadrži !
- polje kojem još nisu dodijeljene vrijednosti sadrži praznu listu
- kada se polju pridruži vrijednost drugog polja, polje se zapravo kopira

```
@kopija = @polje; # polje se kopira
```

Operatori push i pop

- novi elementi mogu se dodavati na kraj polja korištenjem sve većih indeksa, ali Perl nudi i alternativne načine za rad s poljima – bez korištenja indeksa
  - obično brži kod
  - izbjegavanje pogrešaka u pristupu elementima
- često nam je potrebna stožna struktura (LIFO)
- operator `pop` uzima zadnji element iz polja

```
@polje = 5..9;
$fred = pop(@polje);
        # $fred je 9, @polje je (5, 6, 7, 8)
$barney = pop @polje;
        # $barney je 8, @polje je (5, 6, 7)
pop @polje; # skida se 7, @polje je (5, 6)
```

- ako je polje prazno, `pop` ne utječe na polje, a vraća `undef`
- uočiti – zagrade *nisu obavezne* ako je kontekst *nedvosmislen*

Operatori `push` i `pop` (2)

- suprotna operacija je `push` – element se dodaje na kraj polja

```
push(@polje, 0); # @polje je (5, 6, 0)
push @polje, 8; # @polje je (5, 6, 0, 8)
push @polje, 1..10; # dodaje se jos 10 elemenata
@others = qw/ 9 0 2 1 0 /;
push @polje, @others; # jos 5 -> ukupno 19
```

- prvi argument za `push` i argument za `pop` moraju biti varijable – dodavanje ili skidanje elementa iz literala nema smisla

Operatori `shift` i `unshift`

- slično, operatori `shift` i `unshift` barataju s početkom polja

```
@polje = qw# dino fred barney #;
$a = shift(@polje);
     # $a je "dino", @polje je ("fred", "barney")
$b = shift @polje;
     # $b je "fred", @polje je ("barney")
shift @polje;  # @polje je prazno, ()
$c = shift @polje; # $c je undef, @polje je ()
unshift(@polje, 5); # @polje je (5)
unshift @polje, 4; # @polje je (4, 5);
@others = 1..3;
unshift @polje, @others; # @polje je (1, 2, 3, 4, 5)
```

- kao i `pop`, `shift` primijenjen na prazno polje vraća vrijednost `undef`

Interpolacija polja

- kao i skalarne vrijednosti, vrijednosti polja mogu se *interpolirati* u znakovne nizove unutar dvostrukih navodnika
- elementi polja se (automatski) razdvajaju prazninama (*zapravo je separator vrijednost specijalne varijable* `$"`, *čija je uobičajena vrijednost* `' '`)
- primjeri

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";
```

- oko interpoliranog polja ne umeću se praznine – dodajemo ih prema potrebi

```
print "Three rocks are: @rocks.\n";
print "Zagrade (@empty) su prazne.\n";
```

Interpolacija polja (2)

- paziti na e-mail adrese

```
$email = "fred@bedrock.edu"; # WRONG!
           # Tries to interpolate @bedrock
$email = "fred\@bedrock.edu"; # Correct
$email = 'fred@bedrock.edu'; # Another way to do that
```

- pojedinačni element polja zamjenjuje se svojom vrijednošću

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";
                  # "This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # same thing
```

- indeksni izraz se izračunava prije interpolacije varijabli
- zadatak: ako je `$y = "2*4";` koji će element polja biti interpoliran u gornjem primjeru ?
  - ne bi bilo loše uključiti upozorenja (`-w`)

Interpolacija polja (3)

- ako želimo ispisati lijevu uglatu zagradu neposredno iza jednostavne skalarne varijable, moramo je odijeliti

```
@fred = qw(eating rocks is wrong);
$fred = "right";
     # we are trying to say "this is right[3]"
print "this is $fred[3]\n";
     # prints "wrong" using $fred[3]
print "this is ${fred}[3]\n"; # prints "right"
print "this is $fred"."[3]\n";
     # right again (different string)
print "this is $fred\[3]\n";
     # right again (backslash hides it)
```

Petlja `foreach`

- Perl nudi upravljačku strukturu prikladnu za obradu cijelog polja ili liste
- petlja `foreach` prolazi kroz sve vrijednosti u listi i izvršava blok naredbi za svaku od vrijednosti

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";
       # Prints names of three rocks
}
```

- upravljačka varijabla (ovdje `$rock`) u svakoj iteraciji preuzima novu vrijednost iz liste
  - upravljačka varijabla *nije kopija* elementa liste, već se preko nje barata samim elementom

Petlja `foreach` (2)

- ako se unutar petlje mijenja upravljačka varijabla, mijenja se element liste

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
  $rock = "\t$rock"; # tab prije svake rijeci
  $rock .= "\n"; # novi red iza rijeci
}
print "The rocks are:\n", @rocks;
```

- vrijednost upravljačke varijable po završetku petlje je *ista* kao i prije ulaska u petlju (o tome se brine Perl)

Podrazumijevana varijabla $_

- ako se izostavi upravljačka varijabla na početku petlje, Perl koristi podrazumijevanu (*default*) varijablu `$_`

```
foreach (1..10) { # Uses $_ by default
  print "I can count to $_\n";
}
```

- ovo je najčešća pretpostavljena varijabla u Perl-u, koristi se i u nekim drugim prilikama kada se ne navede druga varijabla

```
$_ = "Yabba dabba doo\n";
print; # prints $_ by default
```

Operator `reverse`

- operator `reverse` preuzima listu vrijednosti i vraća listu poredanu obrnutim redom

```
@fred = 6..10;
@barney = reverse(@fred); # 10, 9, 8, 7, 6
@wilma = reverse 6..10; # isto, ali bez polja
@fred = reverse @fred;
            # rezultat se vraca u isto polje
```

- Perl izračunava vrijednost koju treba pridijeliti *prije* samog pridjeljivanja
- operator `reverse` ne utječe na svoj argument (listu) – ako se rezultat ne pohrani u neku varijablu, gubi se

```
reverse @fred; # pogresno - @fred se ne mijenja
```

Operator `sort`

- operator `sort` preuzima listu vrijednosti i vraća listu poredanu prema internom redoslijedu sortiranja
- za ASCII nizove to je “ASCIIbetical” redoslijed
  - velika slova dolaze prije malih, brojevi prije slova,...
  - ovaj se redoslijed može izmijeniti, ali o tome nećemo sada,...
- primjeri

```
@rocks = qw/ bedrock slate granite /;
@sorted = sort(@rocks); # bedrock, granite, slate
@back = reverse sort @rocks;
                    # slate, granite, bedrock
@rocks = sort @rocks;
@numbers = sort 97..102; # 100, 101, 102, 97, 98, 99
```

- operator `sort` ne utječe na svoj argument (listu) – ako se rezultat ne pohrani u neku varijablu, gubi se

```
sort @rocks; # pogresno - @rocks se ne mijenja
```

Skalari i liste ovisno o kontekstu

- Perl tumači izraz ovisno o kontekstu u kojem se nalazi

```
42 + something # The something must be a scalar
sort something # The something must be a list
```

- izrazi u Perlu vraćaju vrijednost koja odgovara kontekstu u kojem se nalaze
- kada se ime polja nađe u kontekstu u kojem se očekuje lista, vraća elemente polja, dok se u skalarnom kontekstu vraća *broj elemenata* polja

```
@people = qw( fred barney betty );
@sorted = sort @people; # lista: barney, betty, fred
$number = 42 + @people; # skalar: 42 + 3 = 45
```

- dodjeljivanje vrijednosti također može imati različite kontekste

```
@list = @people; # a list of three people
$n = @people; # the number 3
```

Liste u skalarnom kontekstu

- razni izrazi mogu se koristiti za generiranje listi – pitanje je što ćemo dobiti ako te izraze upotrijebimo u skalarnom kontekstu
- neki izrazi u skalarnom kontekstu nemaju vrijednost – npr. funkcija `sort` – u skalarnom kontekstu vraća vrijednost `undef`
  (zašto bismo sortirali listu da bismo dobili broj elemenata liste ?)
- funkcija `reverse` u kontekstu liste vraća elemente liste obrnutim redom, dok u skalarnom kontekstu vraća niz znakova koji se dobije promjenom redoslijeda znakova dobivenih ulančavanjem svih elemenata liste

```
@backwards = reverse qw/ yabba dabba doo /;
  # gives doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
  # gives oodabbadabbay
```

- nije uvijek očigledno o kakvom se kontekstu radi !

Liste u skalarnom kontekstu (2)

- primjeri nekih uobičajenih konteksta

```
$fred = something;      # scalar context
@pebbles = something;   # list context
($wilma, $betty) = something; # list context
($dino) = something;    # still list context!
```

  - `($dino)` je lista s jednim elementom

- još neki primjeri skalarnog konteksta

```
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

Liste u skalarnom kontekstu (3)

- još neki primjeri konteksta liste

```
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

Skalari u kontekstu liste

- pravilo je vrlo jednostavno: ako neki izraz obično daje skalarnu vrijednost, ona se automatski pretvara u *listu s jednim elementom*

```
@fred = 6 * 7; # lista (42)
@barney = "hello" . ' ' . "world";
        # lista ("hello world")
@wilma = undef; # lista (undef), nije isto kao
@betty = ( ); # prazna lista - polje se brise
```

Forsiranje skalarnog konteksta

- u nekim slučajevima želimo skalarni kontekst na mjestu gdje Perl očekuje listu
- može se upotrijebiti funkciju `scalar`
  - nije prava funkcija – samo daje uputu Perlu da upotrijebi skalarni kontekst
  - nema funkcije za forsiranje konteksta liste

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks.\n";
                # ispisuje listu a ne broj
print "I have ", scalar @rocks, " rocks.\n";
                          # daje broj
```

<STDIN> u kontekstu liste

- operator <STDIN> u skalarnom kontekstu vraća redak ulaznih znakova
- u kontekstu liste, <STDIN> vraća sve retke do kraja datoteke (EOF)
  - svaki redak vraća se kao poseban element liste

```
@lines = <STDIN>;
      # read standard input in list context
```

- oznaka kraja datoteke: ctrl-D na Unix/Linux, ctrl-Z na DOS/Windows
- možemo primijeniti funkciju chomp na cijelo polje učitanih redaka

```
@lines = <STDIN>; # Read all the lines;
chomp(@lines);
```

ili elegantnije

```
chomp(@lines = <STDIN>);
```

Potprogrami

- ime potprograma (funkcije) je Perl identifikator
  - potprogram – korisnički definiran, funkcija može biti i ugrađena (*built-in*)
- ponekad se ispred imena funkcije navodi znak `&`
- potprogrami koriste odvojeni prostor imena
- definicija potprograma započinje ključnom rječju `sub`, slijedi ime potprograma (*bez* znaka `&`), te blok naredbi uokviren vitičastim zagradama

```
sub marine {
  $n += 1; # globalna varijabla $n
  print "Hello, sailor number $n!\n";
}
```

- definicija funkcije može se nalaziti *bilo gdje* u programu, definicije su *globalne*

Pozivanje potprograma

- potprogram se može pozvati u bilo kojem izrazu tako da se navede ime funkcije *uključujući* znak `&`

```
&marine; # Hello, sailor number 1!
&marine; # Hello, sailor number 2!
&marine; # Hello, sailor number 3!
&marine; # Hello, sailor number 4!
```

Povratna vrijednost

- svi Perl potprogrami imaju povratnu vrijednost
- nema posebne naredbe za vraćanje vrijednosti
- vraća se *rezultat zadnjeg izraza* u potprogramu
- primjer

```
sub sum_of_fred_and_barney {
  print "Potprogram je pozvan\n";
  $fred + $barney; # rezultat je povratna vrijednost
}

$fred = 3; $barney = 4;
$wilma = &sum_of_fred_and_barney; # $wilma = 7
print "\$wilma je $wilma.\n";
```

Povratna vrijednost (2)

- što ako dodamo još jednu naredbu u potprogram

```
sub sum_of_fred_and_barney {
  print "Potprogram je pozvan\n";
  $fred + $barney;
            # rezultat NIJE povratna vrijednost
  print "Sada vracam vrijednost\n"; # Oops!
}
```

- zadnji *evaluirani* izraz više nije ono što smo namjeravali vratiti, već poziv funkcije `print` – vraća se njena povratna vrijednost
- uključivanje upozorenja može pomoći – rezultat izraza nije nigdje pohranjen (sumnjivo)

Argumenti

- umjesto korištenja globalnih varijabli, pri pozivu potprograma mogu se navesti ulazne vrijednosti – argumenti
- prosljeđujemo listu argumenata potprogramu

```
$n = &max(10, 15); # poziv funkcije s 2 argumenta
```

- Perl pohranjuje listu argumenata u posebnu varijablu `@_`
  - preko te varijable može se pristupiti pojedinim argumentima ili saznati njihov broj
  - prvi argument pohranjen je u `$_[0]`, drugi u `$_[1]`
  - elementi ove liste *nemaju veze* s varijablom `$_` !
  - varijabla `@_` je *privatna* – ako postoji istoimena globalna varijabla, ona se automatski pohranjuje/obnavlja pri pozivu potprograma

Privatne varijable u potprogramu

- potprogram `max` mogli bismo napisati ovako

```
sub max {
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

- ovaj način baratanja argumentima je nespretan – možemo koristiti privatne varijable (deklariramo ih operatorom `my`)

```
sub max {
  my($m, $n); # nove, privatne varijable
 ($m, $n) = @_; # dajemo imena argumentima
  if ($m > $n) { $m } else { $n }
}
```

Privatne varijable u potprogramu (2)

- doseg privatne (*private, scoped*) varijable je ograničen na *blok* u kojem se nalazi

```
foreach (1..10) {
   my($square) = $_ * $_; # privatna varijabla
   print "$_ squared is $square.\n";
}
```

- operator `my` ne mijenja kontekst pridruživanja

```
my($num) = @_; # kontekst liste, kao ($num) = @_
               # $num poprima vijednost prvog elementa
my $num = @_;  # skalarni kontekst, kao $num = @_
               # u $num se pohranjuje broj elemenata
```

- bez zagrada `my` se odnosi samo na jednu varijablu

```
my $fred, $barney;  # deklarira se samo $fred
my($fred, $barney); # deklariraju se obje varijable
```

Lista argumenata promjenjive duljine

- u Perlu se često koriste liste argumenata proizvoljne duljine (“no unnecessary limits” filozofija)
  - razlika u odnosu na “tradicionalne” jezike s unaprijed definiranim brojem i tipovima parametara
  - ova fleksibilnost može donijeti probleme ukoliko je broj argumenata pri pozivu različit od onoga koji autor potprograma očekuje
  - u potprogramu je lako provjeriti broj argumenata ispitivanjem polja `@_`

```
sub max {
  if (@_ != 2) {
    print "&max mora imati 2 argumenta\n";
    # ostatak kao i prije ...
}
```

Lista argumenata promjenjive duljine (2)

- poopćenje funkcije `&max`

```
sub max {
  my($max_so_far) = shift @_; # prvi je zasad najveci
  foreach (@_) {              # za svaki od sljedecih
    if ($_ > $max_so_far) { # je li ovaj veci ?
      $max_so_far = $_;     # $_ je varijabla petlje
    }
  }
  $max_so_far; # povratna vrijednost
}

$maximum = &max(3, 5, 10, 4, 6); # poziv funkcije
```

- što ako se ova funkcija pozove bez argumenata (s praznom listom)?
  - `shift` nad praznom listom dat će `undef`, a u petlju se neće niti ući –> vraća se `undef`

Operator `return`

- operator `return` omogućuje trenutan povratak iz potprograma i generiranje povratne vrijednosti

```
my @names = qw/ fred barney betty dino wilma /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
  my($what, @array) = @_;
  foreach (0..$#array) { # svi indeksi polja @array
    if ($what eq $array[$_]) {
      return $_; # kada se pronadje - povratak
    }
  }
  -1; # povratna vrijednost - return nije potreban
}
```

Izostavljanje znaka &

- nekoliko pravila definira kada se znak `&` *može* izostaviti u pozivu potprograma – kao kod poziva ugrađenih (*built-in*) funkcija
  - kada je iz sintakse jasno da se radi o pozivu potprograma (lista parametara navedena unutar zagrada)

```
my @cards = shuffle(@deck_of_cards); # &shuffle
```

  - ako (interni) prevodilac "vidi" definiciju potprograma prije njegovog pozivanja – mogu se i izostaviti zagrade pri pozivu

```
sub podijeli {
  $_[0] / $_[1];
}

my $kvocijent = podijeli 355, 113; # &podijeli
```

  - ako potprogram ima isto ime kao ugrađena funkcija – *moramo* pri pozivu upotrijebiti znak `&`, inače će biti pozvana *ugrađena funkcija* !
- preporuka – koristiti znak `&`

Lista kao povratna vrijednost

- povratna vrijednost potprograma može biti i lista
- primjer – funkcija koja će generirati niz brojeva između dvije vrijednosti, pri čemu niz može biti i silazan

```
sub list_from_fred_to_barney {
  if ($fred < $barney) {
    # Count upwards from $fred to $barney
     $fred..$barney;
  } else {
    # Count downwards from $fred to $barney
       reverse $barney..$fred;
  }
}

$fred = 11;
$barney = 7;
@c = &list_from_fred_to_barney; # (11,10,9,8,7)
```

Učitavanje pomoću operatora <>

- operator `<>` upoznali smo kod učitavanja sa standardnog ulaza (`<STDIN>`)
- općenito se koristi za učitavanje iz datoteke čiji se identifikator navodi unutar zagrada
- poseban način korištenja je bez navođenja datoteke
  - u tom se slučaju podaci učitavaju (redak po redak) iz datoteka čija su imena navedena kao argumenti naredbenog retka (pri pozivu programa)

    ```
    $ ./my_program fred barney betty
    ```
  - ako se ne navede ni jedno ime datoteke, podaci se učitavaju sa standardnog ulaza
  - ponašanje po uzoru na standardne Unix alate
  - argument `"-"` označava `stdin`

Učitavanje pomoću operatora <> (2)

- primjer

```
while (defined($redak = <>)) {
  chomp($redak);
  print "Procitao sam: $redak \n";
}
```

  - ako ovaj program pokrenemo bez argumenata, čitaju se retci s tipkovnice
  - ako se navede više imena datoteka, čitaju se retci iz svake od njih, redom navođenja, kao da se radi o jednoj datoteci
  - ime datoteke iz koje se u nekom trenutku čita, pohranjeno je u posebnoj varijabli `$ARGV`
  - kada se dođe do kraja zadnje datoteke, operator `<>` vraća vrijednost `undef`, što uzrokuje izlazak iz petlje

Učitavanje pomoću operatora <> (3)

- često se koristi tipična Perl pokrata:

```
while (<>) { # podrazumijevana varijabla $_
  chomp; # podrazumijevana varijabla $_
  print "Procitao sam: $_ \n";
}
```

Korištenje argumenata naredbenog retka

- argumenti navedeni u naredbenom retku pri pozivu programa, u Perl programu dostupni su preko posebnog polja `@ARGV`
  - nema varijable poput `argc` u `C`-u
  - ime programa pohranjeno je u posebnoj varijabli `$0`
- pristup elementima polja `@ARGV` – kao i svakom drugom polju
  - preneseni argumenti naredbenog retka mogu se koristiti prema želji/potrebi programera
- operator `<>` koristi elemente polja `@ARGV` kao imena datoteka iz kojih čita podatke
  - moguća je manipulacija tim poljem kako bi `<>` čitao neke druge datoteke

```
@ARGV = qw/ dat1 dat2 dat3 /; # imena datoteka
while (<>) {
  chomp;
  print "Ucitao sam redak: $_ \n";
}
```

Ispis na standardni izlaz

- operator `print` dobiva listu vrijednosti koje jednu po jednu (kao znakovne nizove) ispisuje na standardni izlaz

```
$name = "Larry";
print "Hi, $name, did you know that 3+4 is ",
          3+4, "?\n";
```

- ispis polja i interpoliranog polja (u znakovnom nizu) su različiti

```
@array = qw/ fred barney betty /;
print @array; # lista vrijednosti, bez razmaka
fredbarneybetty

print "@array";
  # interpolacija polja - razmaci izmedju elemenata
fred barney betty
```

- pitanje za vježbu: što radi sljedeći Perl program ?

```
#!/usr/bin/perl
print sort <>;
```

Ispis na standardni izlaz (2)

- zagrade kod operatora `print` su *opcionalne* – mogu se izostaviti *ako to neće izazvati promjenu značenja naredbe*

```
print("Hello, world!\n");
print "Hello, world!\n";
```

- ako izraz koji se ispisuje daje naredbi za ispis oblik poziva funkcije, može biti problema:

```
print (2+3)*4; # Oops!
```

  - u ovom slučaju izraz unutar zagrade shvaća se kao argument funkcije `print`
  - ispisuje se `"5"`, a povratna vrijednost (1 u slučaju uspješnog ispisa, 0 inače) množi se s `4` i odbacuje (nikuda se ne pohranjuje)

```
( print(2+3) ) * 4; # Oops!
```

  - uključena upozorenja mogu pomoći

Formatirani ispis

- operator `printf` sličan je istoimenoj funkciji u programskom jeziku C

```
printf "Hi %s, your password expires in %d days!\n",
  $user, $days_to_die;
```

  - znakovi predznačeni s `%` određuju oblik ispisa odnosno *pretvorbu* (*conversion*) odgovarajućih argumenata, navedenih nakon formatnog niza
  - značenja nekih oznaka:

    `%g` : općeniti format za broj – automatski izbor formata

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;
          # 2.5 3 1.0683e+29
```

    `%d` : dekadski cijeli broj

```
printf "in %d days\n", 17.85; # in 17 days
```

    `%o` : oktalni broj

    `%x` : heksadekadski broj

    `%c` : znak

    `%s` : niz znakova

Formatirani ispis (2)

- operator `printf` najčešće se koristi za tablični ispis – u oznakama formata može se definirati širina polja koje se ispisuje

```
printf "%6d\n", 42;
printf "%2d\n", 2e3 + 1.95;
                # polje se prosiruje: 2001
```

- negativna oznaka širine polja znači da se ispis lijevo poravnava

```
printf "%-15s\n", "flintstone";
```

- format `%f` zaokružuje ispisani broj, pri čemu se može zadati i broj decimalnih mjesta

```
printf "%10f\n", 6 * 7 + 2/3; # ␣42.666667
printf "%10.3f\n", 6 * 7 + 2/3; # ␣␣␣␣42.667
printf "%10.0f\n", 6 * 7 + 2/3; # ␣␣␣␣␣␣␣␣43
```

- znak postotka može se uključiti u ispis operatorom `printf` kao `"%%"`

Datoteke

- za pristup datoteci koristi se njen identifikator (*filehandle*)
  - uobičajeni Perl identifikatori, *bez predznačavanja* (`$`, `@`, `&`)
    $\rightarrow$ postoji opasnost da se kao ime upotrijebi neka ključna riječ
  - preporuka – koristiti velika slova
  - Perl koristi 6 posebnih identifikatora datoteka:
    `STDIN, STDOUT, STDERR, DATA, ARGV, ARGVOUT`
  - prve tri datoteke (odnosno toka) već poznajemo
- otvaranje datoteke – operator `open`, navodi se identifikator i ime datoteke

```
open CONFIG, "dino"; # otvara se za citanje(default)
open CONFIG, "<dino"; # otvara se za citanje
open BEDROCK, ">fred"; # otvara se za pisanje
open LOG, ">>logfile"; # otvara se za dopisivanje
```

Datoteke (2)

- od verzije 5.6, može se koristiti i oblik operatora `open` s tri argumenta:

```
open CONFIG, "<", "dino"; # citanje
open BEDROCK, ">", $file_name; # pisanje
open LOG, ">>", &logfile_name( ); # dopisivanje
```

- prednost – jasno je odvojena oznaka načina pristupa od imena datoteke
  - primjer

    ```
    my $selected_output = "my_output";
    open LOG, ">$selected_output";
    ```

    što ako je korisnik postavio

    ```
    $selected_output = ">passwd" ?
    ```

Datoteke (3)

- ponekad se datoteka navedenog imena ne može otvoriti (dozvole, neispravno ime i sl.)
- možemo ispitati uspješnost otvaranja datoteke temeljem povratne vrijednosti funkcije `open`

```
my $success = open LOG, ">>logfile";
if ( ! $success) {
 # The open failed ...
}
```

- datoteku možemo zatvoriti:

```
close BEDROCK;
```

- Perl automatski zatvara datoteku ako je ponovno otvorimo (odnosno koristimo identifikator datoteke za novo otvaranje) ili ako izađemo iz programa → Perl programi najčešće ne brinu o zatvaranju datoteka

Prijevremeni izlazak iz programa

- funkcija `die` – ispisuje poruku na `stderr` i prekida izvođenje programa s izlaznim statusom različitim od `0`

```
if ( ! open LOG, ">>logfile")
  { die "Cannot create logfile: $!"; }
```

- posebna Perl varijabla `$!` sadrži *poruku sustava* o razlogu pogreške ("`permission denied`" ili "`file not found`") – ima smisla samo ako ispitujemo pogrešku koja nastaje pri pozivu OS-a
- funkcija `die` automatski nadodaje ime Perl programa u kojem je pozvana, te odgovarajući broj retka u programu

```
Cannot create logfile: permission denied at
your_program line 1234.
```

- ako *ne želimo* ispis imena i retka programa, niz koji se ispisuje treba zaključiti s `\n`

```
if (@ARGV < 2)
  { die "Not enough arguments\n"; }
```

Korištenje datoteke

- kad je datoteka uspješno otvorena, koristi se kao i `STDIN/STDOUT`
- učitavanje retka inicira se navođenjem identifikatora datoteke unutar trokutastih zagrada

```
if ( ! open PASSWD, "/etc/passwd") {
  die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
  chomp;
    . . .
}
```

Korištenje datoteke (2)

- ispis u datoteku – primjenom operatora `print` ili `printf`
  - datoteka mora biti otvorena za pisanje ili dopisivanje
  - identifikator datoteke navodi se neposredno prije liste argumenata (*bez zareza!*)

```
print LOG "Captain's log, stardate 3.14159\n";

printf STDERR "%d percent complete.\n",
              $done/$total * 100;

printf (STDERR "%d percent complete.\n",
              $done/$total * 100);

printf STDERR ("%d percent complete.\n",
              $done/$total * 100);
```

Promjena podrazumijevane datoteke

- podrazumijevani identifikator datoteke za ispis operatorima `print` i `printf` je `STDOUT`
- to se ponašanje može promijeniti operatorom `select`

```
select BEDROCK;
print "I hope Barney doesn't find out about this.\n";
print "Wilma!\n";
```

- da ne bi bilo zabune kasnije u programu, dobro je vratiti podrazumijevanu vrijednost na `STDOUT`

```
select STDOUT;
```

Asocijativna polja

- asocijativno polje (*hash, associative array, dictionary*) je podatkovna struktura slična polju
  - indeksiranje elemenata nije slijedno – cijelim brojevima
  - indeks pojedinog elemenata je proizvoljni ali *jedinstveni* niz znakova – ime ili *ključ* (*key*)
  - ključ je proizvoljan skalar, koji se pretvara u string
    `50/20 --> "2.5"`
- veličina asocijativnog polja nije ograničena
- asocijatovno polje može se promatrati kao jednostavna baza podataka u kojoj se podacima pristupa preko ključa

Pristup elementima asocijativnog polja

- slično kao kod polja, ali se koriste vitičaste zagrade i znakovni niz kao indeks

```
$hash{$some_key}
```

- ime asocijativnog polja je standardni Perl identifikator, koristi zasebni prostor imena
- pohranjivanje vrijednosti

```
$family_name{"fred"} = "flintstone";
$family_name{"barney"} = "rubble";
```

- ključ može biti i izraz

```
$foo = "bar";
print $family_name{ $foo . "ney" }; # "rubble"
```

- pristup asocijativnom polju s nepostojećim indeksom vraća vrijednost `undef`

```
$proba = $family_name{"larry"}; # --> undef
```

Acocijativno polje kao cjelina

- asocijativno polje kao cjelina imenuje se predznačeno znakom postotka `"%"`

```
%family_name
```

- asocijativno polje može se pretvoriti u listu i obrnuto
  - lista mora biti u obliku parova ključ–vrijednost

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
"wilma", 1.72e30, "betty", "bye\n");
```

- vrijednost asocijativnog polja u kontekstu liste – lista parova ključ–vrijednost

```
@any_array = %some_hash;
```

- parovi *nisu nužno* u istom redosljedu u kojem je bila izvorna lista ! – Perl pohranjuje parove redosljedom koji mu odgovara zbog brzine pristupa

Operacije s asocijativnim poljima

- asocijativno polje može se kopirati

```
%new_hash = %old_hash;
```

  - Perl pretvara asocijativno polje u listu (*unwind*), a zatim tom listom inicijalizira (element po element) novo asocijativno polje

- češća operacija je “okretanje” asocijativnog polja operatorom `reverse`

```
%inverse_hash = reverse %any_hash;
```

  - zamjenjuje se uloga ključa i vrijednosti
  - početno asocijativno polje se pretvara u listu, lista se okreće, a zatim se tom listom inicijalizira novo asocijativno polje
  - operacija je ispravna samo ako su vrijednosti jedinstvene
  - ako vrijednosti nisu jedinstvene, duplicirani elementi se prepisuju (ostaje zadnja zapisana vrijednost)

Još jedna notacija

- kada se obavlja inicijalizacija asocijativnog polja listom, nije uvijek očigledno koji element liste je ključ, a koji vrijednost
- Perl nudi alternativnu notaciju, u kojoj se može jasno pokazati odnos ključeva i vrijednosti
  - koristi se oznaka `"=>"` – radi se zapravo o drugom zapisu zareza
  - može se koristiti *bilo gdje* umjesto zareza

```
my %last_name = (
  "fred" => "flintstone",
  "dino" => undef,
  "barney" => "rubble",
  "betty" => "rubble",
);
```

Funkcije nad asocijativnim poljima

- Perl ima nekoliko korisnih funkcija za rad s asocijativnim poljima
- funkcija `keys` vraća listu svih ključeva, a funkcija `values` listu svih vrijednosti u asocijativnom polju

```
my %hash = ("a" => 1, "b" => 2, "c" => 3);
my @k = keys %hash; # ("a", "b", "c")
my @v = values %hash; # (1, 2, 3 )
```

  - redoslijed elemenata može biti drugačiji, ali redoslijed te dvije liste je *usklađen* – naravno, ako se između poziva funkcija `keys` i `values` asocijativno polje nije mijenjalo
  - poziv ovih funkcija u skalarnom kontekstu daje broj elemenata asocijativnog polja

```
my $count = keys %hash; # broj elemenata = 3
```

Funkcije nad asocijativnim poljima (2)

- funkcija `each` omogućava prolazak kroz sve elemente asocijativnog polja
  - svakim pozivom funkcija vraća sljedeći par ključ/vrijednost (kao listu)
  - kad se stigne do kraja asocijativnog polja, vraća se prazna lista

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

    - lista koju vraća `each %hash` pridružuje se paru `($key, $value)`
    - pridruživanje se obavlja u uvjetnom izrazu `while` petlje → skalarni kontekst, uvjet je ispunjen dok je *izvorna* lista neprazna
    - kad `each` vrati praznu listu, uvjet poprima vrijednost `false`, iako lista `($key, $value)` poprima vrijednost `(undef, undef)` – nije prazna!

Primjer primjene

- evidencija o posudbi knjiga u knjižnici

```
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

- ima li osoba posuđenu knjigu ?

```
if ($books{$osoba}) {
  print "$osoba ima bar jednu posudjenu knjigu.\n";
}
```

- ako osoba nema posuđenu knjigu, njen zapis je `0`, a ako nije nikada ni posuđivala knjige – zapis je `undef`
- ispitivanje postoji li neki ključ u asocijativnom polju postiže se funkcijom `exists`

```
if (exists $books{"dino"}) {
  print "dino je clan knjiznice.\n";
}
```

Primjer primjene (2)

- zapis iz asocijativnog polja možemo obrisati funkcijom `delete`

```
delete $books{"betty"}; # betty se isclanjuje
```

- interpolacija *elemenata* asocijativnog polja u znakovne nizove se obavlja kao i za bilo koju drugu skalarnu varijablu

```
foreach $osoba (sort keys %books) {
  if ($books{$osoba}) {
    print "$osoba je posudila $books{$osoba} knjiga.\n";
  }
}
```

- asocijativno polje kao cjelina *ne može se interpolirati*
  - ako unutar znakovnog niza navedemo `"%books"`, ne dolazi do interpolacije – ispisuje se doslovno taj niz

Regularni izrazi

- podrška regularnim izrazima je jedna od najjačih strana Perla
- regularni izrazi nazivaju se i uzorcima (*patterns*)
- upoznali smo ih kod Unix alata (`grep, sed`)
  - većina znakova predstavljaju sami sebe
  - posebni znakovi (*metaznakovi*)
    npr. `.` : podudara se s jednim znakom, izuzev `"\n"`
  - doslovno tumačenje metaznaka – predznačiti s `"\"`

Kvantifikatori

- *kvantifikatori* (broj ponavljanja podizraza)
  `?` : prethodni izraz se pojavljuje najviše jednom
  `*` : prethodni izraz se pojavljuje 0 ili više puta
  `+` : prethodni izraz se pojavljuje jednom ili više puta
  `{n}` : prethodni izraz se pojavljuje točno `n` puta
  `{n,}` : prethodni izraz se pojavljuje `n` ili više puta
  `{n,m}` : prethodni izraz se pojavljuje barem `n` ali najviše `m` puta
- podizraz koji se ponavlja (ako je dulji od jednog znaka) treba uokviriti oblim zagradama

  `/bam{2}/` se podudara s `"bamm"` a ne s `"bambam"`
  `/(bam){2}/` se podudara s `"bambam"`
- Perl kvantifikatori su “pohlepni” (*greedy*) – nastoji se postići podudaranje najduljeg mogućeg podniza

  `/[0-9]+/` se podudara s cijelim nizom `"1234567890"`

Kvantifikatori (2)

- primjer: u nizu
  `larry:x:100:10:Larry Wall:/home/larry:/bin/bash`
  želimo pronaći korisničko ime (`"larry:"`)
  - ako upotrijebimo regularni izraz `/.+:/`, dobit ćemo podniz
    `larry:x:100:10:Larry Wall:/home/larry:`
  - može pomoći negiranje klase znakova: `/[^:]+:/`, (niz znakova do *prve* dvotočke)
- još jedno svojstvo na koje treba paziti je da regularni izrazi nastoje postići podudaranje *čim ranije* – pretraživanje se obavlja s lijeva nadesno
  - ako želimo pronaći (i npr. izbrisati) niz znakova `"x"` u sredini niza `"fred xxxxxxx barney"`, regularni izraz `/x*/` neće pomoći – pronaći će *prazni niz* na početku !

Klase znakova

- klasa znakova navodi se kao lista znakova unutar uglatih zagrada – podudara se s *jednim* znakom iz liste

```
[abcwxyz]
```

- može se koristiti raspon znakova

```
[a-zA-Z]
```

- operator negacije klase znakova omogućuje da definiramo koji se znakovi *ne podudaraju* na tom mjestu

```
[^def]
```

Kratice za klase znakova

- za često korištene klase znakova Perl definira kratice:
  `\w` : alfanumerički znakovi i podvlaka `[A-Za-z0-9_]`
  `\d` : znamenke `[0-9]`
  `\s` : prazni znakovi `[\f\t\n\r ]`
- negacije ovih kratica imenovane su velikim slovima:
  `\W` : alfanumerički znakovi i podvlaka `[^\w]`
  `\D` : znamenke `[^\d]`
  `\S` : prazni znakovi `[^\s]`
- kratice se mogu koristiti kao elementi klase znakova:
  npr. `/[\dA-Fa-f]+/` : pronalazi heksadekadski broj
- ponekad je korisna kombinirana klasa znakova `[\d\D]`
  – podudara se s proizvoljnim znakom uključujući i `"\n"` (za razliku od metaznaka `" . "`)

Sidra

- sidra (*anchors*) – omogućuju ograničavanje mogućih pozicija podudaranja (početak retka, kraj retka,...)
  - podudara se s *praznim nizom*, ali definira okolinu

`\b` : granica riječi (*word boundary*), definirana kao prazni niz između znaka riječi (`\w`) i znaka koji nije znak riječi (`\W`) (proizvoljnim redosljedom)

`/\bFred\b/` se pronalazi u nizovima `"The Great Fred"` i `"Fred the Great"` ali ne u nizu `"Frederick"`

`\B` : negacija od `\b`

`/\bsearch\B/` se pronalazi u (pod)nizovima `"searches"` i `"searching"` ali ne u nizovime `"search"` i `"researching"`

`^` : početak retka (prazni niz)

`$` : kraj retka (prazni niz)

Operacije s regularnim izrazima

- uobičajene operacije koje rade s regularnim izrazima su podudaranje ili pretraživanje (*pattern-matching*) i zamjena (*substitution*)
- među ove operacije može se ubrojiti i zamjena znakova (*transliteracija*)
- ako se ne upotrijebi operator povezivanja (*binding*), operacije se izvode nad podrazumijevanom varijablom (`$_`)

Podudaranje

- operator podudaranja (*matching*):
  `m//` : unutar graničnika navodi se regularni izraz koji se traži u znakovnom nizu
  - kao i kod operatora `qw//`, kao graničnik se može koristiti i neki drugi znak interpunkcije
    `m/abc/, m(abc), m<abc>, m!abc!, m^abc^,...`
- niz za podudaranje podrazumijeva se u varijabli `$_`
- izraz `m/uzorak/` vraća vrijednost `true` ako pronađe pojavljivanje uzorka u znakovnom nizu

```
$_ = "yabba dabba doo";
if (m/abba/) {
  print "It matched!\n";
}
```

- ako se izostavi oznaka operatora, podrazumijeva se podudaranje:

```
if (/Windows 95/) { print "Time to upgrade?\n" }
```

Opcije podudaranja

- prilikom primjene operatora podudaranja mogu se specificirati neke opcije
- `/i` : zanemarivanje razlike između velikih i malih slova

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) { # case-insensitive match
  print "OK, I recommend bowling.\n";
}
```

- `/s` : podudaranje metaznaka `.` i s `"\n"`
- `/x` : zanemarivanje praznih znakova i komentara unutar regularnih izraza – omogućuje bolju preglednost

```
/
  -?  # an optional minus sign
  \d+ # one or more digits before the decimal point
  \.? # an optional decimal point
  \d* # some optional digits after the decimal point
/x    # end of pattern
```

Opcije podudaranja (2)

- `/g` : globalno podudaranje – vraćaju se svi pronađeni podnizovi

```
if (@perls = /perl/gi) {
    printf "Perl mentioned %d times.\n", scalar @perls;
}
```

- opcije se mogu kombinirati slijednim navođenjem
- ima još nekoliko mogućih opcija

Zamjena

- operator zamjene (*substitution*):
  `s///` : navodi se regularni izraz koji se traži,
  te niz znakova kojim se pronađeni podniz zamjenjuje

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/; # Replace Barney with Fred
print "$_\n";
```

- operator vraća logičku vrijednost `true` ako je zamjena uspješna (traženi uzorak je pronađen i zamijenjen)

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
  print "Successfully replaced fred with wilma!\n";
}
```

- globalno pretraživanje i zamjena: opcija `/g` – ako se ne navede, zamjenjuje se *samo jedna* pojava traženog uzorka
  - ako se navede opcija `/g`, zamjenjuju se sve *nepreklapajuće* pojave uzorka

Transliteracija

- operatorom `tr/lista_pretrazivanja/lista_zamjene/cds` obavlja se zamjena znakova liste pretraživanja znakovima liste zamjenskih znakova
  - operator ima sinonim `y///` (`sed`)
  - zamjena se obavlja znak po znak
  - vraća se broj zamijenjenih (ili obrisanih) znakova
  - mogu se postaviti opcije :
    `/c` : komplement liste pretraživanja
    `/d` : brisanje pronađenih znakova za koje nema zamjene
    `/s` : slijed znakova koji su zamijenjeni istim znakom reducira se na jedan znak
  - ako je niz zamjenskih znakova prazan, nema zamjene

```
tr/A-Z/a-z/; # $_ u mala slova
$cnt = tr/*/*/; # broji zvijezde u $_
$cnt = tr/0-9//; # broji znamenke u $_
```

Operator povezivanja

- bez upotrebe operatora povezivanja (*binding*), operacije pretraživanja ili zamjene obavljaju se na podrazumijevanoj varijabli `$_`
- operator povezivanja `=~` određuje varijablu nad kojom se obavlja operacija

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
  print "Aye, there's the rub.\n";
}
```

Interpolacija u regularnim izrazima

- regularni izrazi se interpoliraju kao znakovni nizovi u dvostrukim navodnicima → možemo generirati izraze u varijablama

#/usr/bin/perl -w!

```
my $uzorak = "larry";
while (<>) {
  if (/^($uzorak)/) { # uzorak na pocetku retka
    print "$uzorak je na pocetku niza $_";
  }
}
```

Varijable podudaranja

- zagrade služe za grupiranje dijelova regularnog izraza, no kada regularni (pod)izraz uokvirimo zagradama, aktivira se i pamćenje pronađenih uzoraka
- ako je više podizraza uokvirenih zagradama, za svaki se oblikuje varijabla koja pamti pronađene podnizove
- imena ovih varijabli su `$1`, `$2`, `$3` itd. i odgovaraju redoslijedu podizraza u zagradama

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) { # zarez van zagrada
  print "words were $1 $2 $3\n";
}
```

- pamćenje varijabli je do sljedećeg uspješnog podudaranja (resetira sve)– trebalo bi provjeriti uspješnost podudaranja (inače čitamo rezultate prethodnog)

Povezivanje unazad

- povezivanje unazad (*backreference*) vrlo je blisko varijablama podudaranja
- odnosi se na iste podnizove kao i varijable podudaranja, ali omogućuje korištenje *u samim izrazima*
  - pronađeni podniz može se koristiti za podudaranje u ostatku izraza
  - označavanje: `\1`, `\2`, `\3` itd.
- primjer – želimo pronaći ponovljene riječi u tekstu, npr. troslovne

```
/(\w\w\w)\s\1/;
```

- povezivanje unazad se koristi *unutar* uzorka za podudaranje, a varijable podudaranja *izvan* njega

```
s/(\w\w\w)\s\1/$1/g;
```

Operator `split`

- operator `split` dijeli znakovni niz prema navedenom uzorku
  - najčešće se koristi s vrlo jednostavnim regularnim izrazima
  - prikladan za dijeljenje podataka odvojenih tabovima, dvotočkama, razmacima...
  - separator se može definirati regularnim izrazom
- tipičan oblik:

```
@fields = split /separator/, $string;
```

- par primjera

```
@fields = split /:/, "ab:c:de"; # ("ab", "c", "de")
@fields = split /:/, "ab::cd"; # ("ab", "", "cd")

$ulaz = "Ovo je \t \t test.\n";
@lista = split /\s+/, $ulaz; # ("Ovo", "je", "test.")
```

- podrazumijevani oblik (*default*) – dijeljenje `$_` na prazninama :

```
my @fields = split; # kao split /\s+/, $_;
```

Funkcija `join`

- funkcija `join` djeluje suprotno od operatora `split` – povezuje nizove u jedan znakovni niz

```
my $result = join $glue, @pieces;
```

  - prvi argument mora biti znakovni niz
  - ostali argumenti predstavljaju listu dijelova
  - konačni niz dobiva se povezivanjem dijelova između kojih se umeće zadani niz (`glue`)
  - funkcija vraća spojeni niz

```
my $x = join ":", 4, 6, 8, 10, 12; # "4:6:8:10:12"
```

- primjeri

```
my $y = join "abc", "def"; # PAZI! samo "def"

my @lst = split /:/, $x; # @lst = (4, 6, 8, 10, 12)
my $z = join "-", @lst; # $z ="4-6-8-10-12"
```

Podudaranje u kontekstu liste

- kada se operator podudaranja (`m//`) koristi u kontekstu liste, povratna vrijednost je lista varijabli podudaranja

```
$_ = "Hello there, neighbor!";

my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

- na ovaj način možemo "pospremiti" rezultat podudaranja za kasniju primjenu
- ako se koristi opcija `/g`, uzorak se može naći na više mjesta u nizu – svako podudaranje vraća varijable koje odgovaraju izrazima u zagradama

```
my $text = "Fred dropped a 5 ton granite block";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
   # Result: Fred dropped a ton granite block
```

Perl programi u naredbenom retku

- Perl se može koristiti kao alat koji se poziva iz naredbenog retka, uz navođenje kratkog programa (*oneliner*)
- moguć je cijeli niz opcija pri pozivu, najjednostavniji je samo navođenje naredbe koju treba izvršiti

```
perl -e'naredba'
```

  - primjer – ispis znakova s ASCII kodovima 65 do 90:

```
perl -e'for (65..90) { print chr($_) }'
```

- može se koristiti umjesto *sed*-a

```
$ sed 's/Windows/Linux/g' OS.txt
$ perl -pe's/Windows/Linux/g' OS.txt
```

  - opcija `-p` označava da se naredba primjenjuje na svaki redak učitan sa `STDIN` ili iz datoteka navedenih kao argumenti, te da se rezultat ispisuje

```
perl -e'while (<>) {s/Windows/Linux/g;print}' OS.txt
```