OSNOVI SHELL PROGRAMIRANJA

Hello, World!

1. Script `ss1` (upisan iz `cat`-a u fajl `ss1`)

```
$ cat >ss1
#
# ss1: jednostavan shell program
#
clear
echo "Hello, World!"
<CTRL-D>
```

2. Davanje korisnicima x (execute) dozvole nad datotekom

```
$ chmod +x ss1
```

Kojim korisničkim kategorijama je data dozvola x?

3. Pokretanje

```
$ ./ss1
```

Skript briše ekran (komanda clear),
a zatim na ekranu ispisuje poruku Hello, World!
Sav tekst u liniji iza znaka # se smatra komentarom.
Kako izbeći `./` ?

- Prilikom pokretanja script-a može se specificirati komandni interpreter u kome će se program izvršavati. Potrebno je u prvu liniju script-a upisati sledeće:
  `#!/bin/bash`
- Ukoliko se komandni interpreter ne specificira na ovaj način, program se izvršava u tekućem interpreteru.
- Script se može pokrenuti i na drugi način, bez eksplicitne dodele x dozvole - dovoljno je pozvati komandni interpreter da izvrši shell program:
  `$ bash ss1`
  ili
  `$ /bin/sh ss1`
  Ukoliko se shell program ne nalazi u tekućem direktorijumu, potrebno je specificirati putanju do programa.
  `$ bash /home/jsmith/ss1`

Za razvoj i korišćenje shell skriptova preporučuje se sledeća procedura:
- script treba razviti na svom direktorijumu,
- zatim ga istestirati: `$ bash imeskripta`,
- i na kraju iskopirati u neki direktorijum koji je podrazumevano uključen u sistemsku putanju.

Program se može kopirati u `bin` poddirektorijum home direktorijuma autora. Ukoliko veći broj korisnika želi da koristi program datoteku treba kopirati u direktorijume `/bin` ili `/usr/bin` kojima mogu pristupati svi korisnici. Dodatno, korisnicima treba dati dozvolu execute da bi mogli da pokreću program pomoću imena datoteke.

Komande koje se mogu zadavati u skriptovima su:
- standardne Linux komande (poput `cp` ili `mv`)
- komande specifične za shell programiranje. Neke od komandi specifičnih za shell programiranje su gotovo kompletni programski jezici (na primer `awk`).

Jedna od često koričćenih:
- **echo**

prikazuje tekst ili vrednost promenljive na ekranu. Sintaksa komande echo je:

```
$ echo [opcije] [string, promenljive...]
```

Opcije:

- -n ova opcija ne prebacuje kursor u novi red,nakon izvršenja echo komande
- -e omogućava interpretaciju sledećih karaktera u kombinaciji sa obrnutom kosom crtom:
  - \a upozorenje (*alert bell*)
  - \b povratak unazad (*backspace*)
  - \c ne prelaziti u novi red (*suppress trailing new line*)
  - \n novi red (*new line*)
  - \r povratak na početak reda (*carriage return*)
  - \t horizontalni tabulator (*horizontal tab*)
  - \\ obrnuta kosa crta (*backslash*)

Primer.

```
$ echo -e "Linux\n\t\tRulez !\n"
Linux
Rulez !
```

NAVODNICI

Linux prepoznaje tri tipa navodnika.

- Dvostruki navodnici - "Double Quotes". Sve što se nalazi u ovim navodnicima gubi originalno značenje (osim \ i $).
- Jednostruki navodnici - 'Single quotes'. Sve što je zatvoreno jednostrukim navodnicima ostaje nepromenjeno.
- Obrnuti navodnici `Back quote`. Izraz zatvoren obrnutim navodnicima tretira se kao komanda koju treba izvršavati.

Primer.

```
$ echo "Danasnji datum : date" # tretira date kao string
Danasnji datum : date
$ echo "Danasnji datum : `date`" # tretira date kao komandu
Danasnji datum : Fri Apr 2 16:30:35 CEST 2004
```

Za dalje nam je potrebno još malo alata

Advanced Tools

This chapter tackles advanced commands and regular expressions (formulas for matching strings that follow specific patterns).

Regular Expressions and Metacharacters

A regular expression is a syntactical set or phrase that represents a pattern of text or strings. Regular expressions enable you to represent a varying array of characters with a much smaller set of predefined characters. They often include metacharacters—characters that represent another set or group of characters or commands.

No discussion of regular expressions and metacharacters is useful without examples, so create a file called **`/tmp/testfile`** that you can use as you work your way through this chapter:
Here's what to put in the file (note the capitalization and punctuation):

```
Juliet Capulet
The model identifier is DEn5c89zt.
Sarcastic was what he was.
No, he was just sarcastic.
Simplicity
The quick brown fox jumps over the lazy dog
It's a Cello? Not a Violin?
This character is (*) is the splat in Unix.
activity
apricot
capulet
cat
celebration
corporation
cot
cut
cutting
dc9tg4
eclectic
housecat
persnickety
The punctuation and capitalization is important in this example.
simplicity
undiscriminating
Two made up words below:
c?t
C?*.t
cccot
cccccot
```

Metacharacters

Metacharacters are useful in reducing the amount of text used with commands and for representing groups of text with a minimal set of characters.
The following list shows some of the more common metacharacters.

- **. —Dot or period. Represents one character.**
  Example:

Find any instances of the letter c and the letter t with exactly one character between them:

```
c.t
```

Results from testfile:

```
Simplicity cut simplicity
apricot cutting c?t
cat dc9tg4 cccot
cot housecat cccccot
```

- **[] — Square brackets. Result will match any one of the characters inside them.**

  Example: Find any instances of the letter c and the letter t with only one of the letters in the square brackets between them:

  ```
  c[aeiou]t
  ```

  Results from testfile:

  ```
  Simplicity cot simplicity
  apricot cut ccot
  cat housecat ccccot
  ```

- *** — Asterisk (splat). Represents zero or more occurrences of other characters.**

  Example: Find any instances of the letter c and the letter t with zero or more characters between them.

  ```
  c*t
  ```

  Results from testfile:

  ```
  Juliet Capulet
  The model identifier is DEn5c89zt.
  Sarcastic, was what he was.
  No, he was just sarcastic.
  Simplicity
  The quick brown fox jumps over the lazy dog
  It’s a Cello? Not a Violin?
  This character is (*) is the splat in Unix.
  activity
  apricot
  capulet
  cat
  celebration
  corporation
  cot
  cut
  cutting
  dc9tg4
  eclectic (also eclectic; same word so only one instance shows)
  housecat
  persnickety
  The punctuation and capitalization is important in this example.
  simplicity
  undiscriminating
  c?t
  c?*.t
  cccot
  cccccot
  ```

- **[^ *insert_character(s)*] — Square brackets with a caret between them. Do not match any of the characters following the caret.**

Example: Find any instances of the letter c and the letter 5 with none of the characters inside the brackets between them.

c[^aeiou]t

Results from testfile:

d**c9t**g4
c?t

- ^*insert_character* — **Match the sequence only if it is at the beginning of the line.**
  Example: Find any instances of the exact string ca at the beginning a line:

  ^ca

  Results from testfile:

  capulet
  cat

  Without the ^, the output would be instances of ca anywhere on the line:

  Results from testfile:
  Sar**ca**stic was what he was.
  No, he was just sar**ca**stic.
  capulet
  cat
  house**ca**t

- ^[*insert_character(s)*] — **Caret preceding a bracketed sequence. Match any one character inside brackets; match sequence at the beginning of the line.**
  Example: Find all instances of the letter c at the beginning of the line, any one of characters in the brackets, and the character t.

  ^c[aeiou]t

  Results from testfile:

  cat
  cot
  cut
  cutting

  If the ^ were not in the syntax, the output would be instances of c[aeiou]t anywhere on the line:

  Simpli**cit**y
  apri**cot**
  cat
  cot
  cut
  cutting
  house**cat**
  simpli**cit**y
  cc**cot**
  cccc**cot**

- **$ — Dollar sign. Match the occurrence at the end of the line only.**
  Example: Find the character c and the character t at the end of the line, with zero to any combination of characters in between:

  c*t$

  Results from testfile:

  Capulet cat c?t
  DEn5**C89zt cot c?*.t**
  apri**cot cut** cc**cot**
  capulet house**cat** ccc**cot**

- **\ — Blackslash. Removes special meaning from the character immediately following, so that a ? is taken literally rather than as a metacharacter.**
  Example: Find all instances of the character c, a literal ?, and the character t:

c\?t

Results from testfile:

c?t

- **? — Question mark. Represents zero or one character (not to be confused with *, which matches zero, one, or many characters). Not available with all programs in Unix.**
  Example: Find all instances of the character c and the character t with zero or one character between them:
  `c?t`
  Results from testfile:
  ```
  Simplicity
  eclectic
  activity
  housecat
  apricot
  The punctuation and capitalization is important in this example.
  cat
  simplicity
  cot
  c?t
  cut
  cccot
  cutting
  cccccot
  dc9tg4
  ```

- **[a-z] — Full lowercase alphabet designation inside brackets. Match all occurrences of any single letter.**
  Example: Match all instances of the character c and the character t with a single instance of a letter a through z between them.
  `c[a-z]t`
  Results from testfile:
  ```
  Simplicity cut simplicity
  apricot cutting cccot
  cat housecat cccccot
  cot
  ```

- **[0-9] — Matches single instances of all numbers 0-9.**
  Example: Find all instances of the letter c and the letter 5 with any single instance of a number between 0 and 9 between them.
  `c[0-9]t`
  Results from testfile:
  ```
  dc9tg4
  ```

✡

- **[d-m7-9] — Matches a single occurrence of any character d through m or 7 through 9. This illustrates how these commands can be grouped.**
  Example: Find all instances of the letter c and the letter t with a single instance of any of the letters c through t or a single instance of any number 0 through 4.
  `c[c-t0-4]t`
  Results from testfile:
  ```
  Simplicity dc9tg4
  apricot cccot
  cot cccccot
  simplicity
  ```
  The metacharacters listed here are commonly used and generally available with most commands. Not all metacharacters or regular expressions work with every program, and

sometimes only a subset of metacharacters is supported within a program. Read the man page for the command you are going to use to determine its support for the metacharacters.

Metacharacters and regular expressions are typically used with the following commands although there are others): awk, ed, emacs, expr, grep, fgrep, egrep, less, more, sed, vi.

Regular Expressions

Regular expressions can be extremely simple or very complex, depending on the program they are used with and what you are looking for. They can include metacharacters or regular characters. A regular expression is the syntax used to match something, and metacharacters enable you to expand a more complex group of regular characters with a smaller subset of predefined characters.

```
c[a-n]n
[0-9][0-9][0-9]\.[0-9][0-9][0-9]\.[0-9][0-9][0-9][0-9]
```

TOOLS

grep

```
$ grep only myfile
$ grep 'w.r' myfile
```

Grep stands for global regular expression print. The command structure for grep is:

```
grep string_to_search_for file_to_search
```

`grep` supports the use of most of the metacharacters.

- A simple grep would be a search for the word `root` in the `/etc` directory and its subdirectories:
  ```
  grep root /etc/*
  ```
  This would result in significant output because, as you might guess, there are many files that contain the word `root`.

- `grep` also has a `-v` argument, which enables you to search for everything but a named string. To search `/etc/passwd` for all accounts except the root string, you'd type:
  ```
  grep -v root /etc/passwd
  ```
  The output would show the contents of the `/etc/passwd` file that didn't contain the string root.

- `cat /etc/passwd | grep root`
  This produces the same output as the preceding command would without the `-v` option, but it demonstrates how you can use grep to search for specific characters from the output of another command.

sort

The sort command is a powerful little utility that enables you to sort the output of a command or file in a specified order. The options for sort are described in the following table:

`-d`	Sorts via dictionary order, ignoring non-alphanumerics or blanks.
`-f`	Ignores case when sorting.
`-g`	Sorts by numerical value.
`-M`	Sorts by month (i.e., January before December).
`-r`	Provides the results in reverse order.
`-m`	Merges sorted files.

`-u` Sorts, considering unique values only.

1. Create a file called `/tmp/outoforder` with the following text:

```
Zebra
Quebec
hosts
Alpha
Romeo
juliet
unix
XRay
xray
Sierra
Charlie
horse
horse
horse
Bravo
1
11
2
23
```

2. Sort the file by dictionary order:

```
sort -d /tmp/outoforder
```

The results are:

```
1
11
2
23
Alpha
Bravo
Charlie
Quebec
Rome
Sierra
XRay
Zebra
horse
horse
horse
hosts
juliet
unix
xtra
```

Notice that the strings beginning with an uppercase letter have come before any of the lowercase words.

3. The word horse is in the file three times. To remove the extra instances of it in your sort, you can use:

```
sort -du /tmp/outoforder
```

tee

The tee command enables you to split the output of a command to multiple locations. For instance, if you need to see the output of a command on the screen, but you also need to have the output written to a file for later use, you can use tee. To run this command, you simply need to define a command and then identify where you want the output file to go. Here's an example:

```
      ps -ef | tee /tmp/troubleshooting_file
```

shows the output of the `ps` command on the screen and also writes it to `/tmp/troubleshooting_file`, where it can be viewed later.

To append to the file instead of overwriting, you use the `-a` option:

```
      ps -ef | tee -a /tmp/troubleshooting_file
```

You can specify as many files as you want after the tee, and the output will go to each file.

script

The script command enables you to record your entire interactive login session. It captures and places in a file every keystroke you make (and its output) from the time you start it until you end it. **script** is especially useful for troubleshooting problems or using the contents of a session for later review. To use script, you simply need to type it with the **-a** option and a filename:

```
      script -a /tmp/script_session
```

Without the **-a** option, the specified file, if it already exists, will be overwritten.

Everything you type after the `script` command is recorded in the file you indicate, `/tmp/script` in this example. If you don't indicate a filename, the command creates a file called `typescript` in the directory you start the command in. When you have completed your scripting session, **type exit to end it. Be careful not to leave the script session running**, because the file you create can begin taking up a significant amount of space, to the point of filling up your file system.

awk

Atypical example of an AWK program is one that transforms data into a formatted report. The data might be a log file generated by a Unix program such as traceroute, and the report might summarize the data in a format useful to a system administrator. Or the data might be extracted from a text file with a specific format, such as the following example. In other words, **AWK is a pattern-matching program**.

AWK takes two inputs:

- a command, set of commands, or a command file and it contains pattern-matching instructions for which AWK is to use as a guideline for processing the data or data file
- a data or data file.

Try out this one awk command at the command line:

```
            $ awk '{ print $0 }' /etc/passwd
```

Results will look something like the following:

```
            root:x:0:0:root:/root:/bin/bash
            bin:x:1:1:bin:/bin:/sbin/nologin
            sync:x:5:0:sync:/sbin:/bin/sync
            shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
            halt:x:7:0:halt:/sbin:/sbin/halt
            mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
            …
```

In this example, AWK isn't processing any data but is simply reading the `/etc/passwd` file's contents and sending the data unfiltered to standard out, much like the `cat` command. When AWK was invoked, it was provided with the two pieces of information it needs: an editing command and data to edit. The example specifies `/etc/passwd` as input file for data, and the edit command simply directs AWK to print each line in the file in order. All output is sent to standard out (which can be directed elsewhere), a file, or another command.

The real working power of AWK is in extracting parts of data from a larger formatted body. Using the /etc/passwd file again, the following command takes two of the fields from each entry in the /etc/passwd file and creates a more human-friendly output:

```
    % awk -F”:” ‘{ print “username: “ $1 “\t\t\t user id:” $3 }’ /etc/passwd
```

The results will be something similar to this:

```
    username: root          user id:0
    username: bin           user id:1
    username: sync          user id:5
    username: shutdown      user id:6
    username: halt          user id:7
    username: mail          user id:8
    username: nobody        user id:99
    username: sshd          user id:74
    username: apache        user id:48
    username: webalizer     user id:67
    username: ldap          user id:55
    username: mysql         user id:27
    username: pdw           user id:500
    %
```

By default AWK associates a **blank space as a delimiter** for the incoming data; to change this association the **-F** switch is used to denote a different field separator—the colon (:).
This example also uses a print command to provide a structure to the output:

```
print “username: “ $1 “\t\t\t user id:” $3
```

All the text to be printed as noted is wrapped in double quotation marks. The **$1** and **$3** are variables that contain the data AWK is filtering. AWK initializes a set of variables every time it process a file; **$1** is the variable that contains the text up to the first delimiter; **$2** is the second field, and so on, so this command calls for the contents of the first (**$1**) and third (**$3**) fields. Variable **$0** would be the whole line.

The AWK editing command consists of two parts:
- patterns and
- commands.

Patterns are matched with the line from the data file. If no pattern is provided, AWK matches any line and the command is always executed. The preceding example has no pattern to match before determining whether the command is to be executed, so AWK executes the command for each and every line.

Working with Patterns
Patterns in AWK consist of a string of text or one or more regular expressions contained within slashes (/). Here are a couple of example patterns:

```
    # String example
    /text pattern/
    # Reg Ex example match any lowercase chars
    /[a-z]/
```

The commands that follow the pattern rules provide instructions on what AWK is to do when a pattern match evaluates as true. Commands may contain several instructions, each separated by a semicolon (;). Common AWK instructions include =, print, printf, if, while, and for. These instructions behave like similar instructions in other programming languages, providing the capability to assign values to variables (=), print output (print and printf), or execute segments of code given a certain set of conditions (if, while, and for). Conditional statements provide for the capability to refine pattern matching.

In the preceding example, the first and third fields are printed in between text that identifies the values for the user reading the output. The statement printf denotes that the output is going to have its own format and allows for the use of escape sequences such as the \t, which simply denotes spacing of the output, specifically that the output should be spaced by a tab; the example called for two tabs.

Programming with AWK
In the following "Try It Out," you place commands from the previous "Try It Out" example into a file that AWK can then use as a collection of steps for processing data.

1. Use vi to enter the following and save the file as **`print.awk`**:
```
BEGIN {
FS=":"
}
{ printf "username: " $1 "\t\t\t user id: " $3 }
```
2. Execute awk as follows:
```
% awk -f print.awk /etc/passwd
```
   The resulting output is just as the previous example
```
username: root user id:0
username: bin user id:1
…
```

The script as executed performs the same function as the previous example; the difference here is the commands reside within a file, with a slightly different format. Because AWK is a structured programming language, there is a general format to the layout of the file:
1. Beginning commands, which are executed **only once at the beginning of the file**, are set into a **block starting with the word `BEGIN`**. The block is contained in braces exactly as the example shows:
```
BEGIN {
FS=":"
}
```
2. **Pattern-matching commands are blocks of commands that are executed once for each and every line in the data file**.
```
{ printf "username: " $1 "\t\t\t user id: $3 }
```
3. Ending commands, a block of commands first denoted by the word **`END`**, are executed only once, when the end of file is reached. While the "Try It Out" example contains no commands with an **`END`** block, a possible **`END`** block for this example might look something like this:
```
END {
Printf "All done processing /etc/passwd"
}
```
The only code in the example's `BEGIN` block is the definition of the delimiter, the colon. This is akin to the `-F` switch used at the command line in the previous example. In this example, the delimiter is assigned to a variable, `FS`, which AWK will check when executing the main pattern-matching block of code.

`FS` (field separator) is one of several standard variables that AWK uses. Others include:

- `NF`— Variable for providing a count as to the number of words on a specific line.
- `NR`— Variable for the record being processed. That is, the value in `NR` is the current line in a file awk is working on.
- `FILENAME`—Variable for providing the name of the input file.
- `RS`—Variable for denoting what the separator for each line in a file is.

AWK variables are typeless.

The main section of the program, the pattern-matching commands for each data line, is executed in order from the top line down. Lines from the data file are read and evaluated one by one, from the top of the file down as well.

When the AWK program is reading and evaluating a data file, the commands see only the current single line from the data file at a time and all AWK program variables. The whole line is subject to pattern matching and is automatically loaded into special variables such as $0, $1, and so on. The END block, as with the BEGIN, provides a set of commands that are to be executed only once, when the end of the incoming data has been reached. The last example did not include an END segment. BEGIN and END segments are not necessary for all AWK programs. They are helpful, however, in setting up and cleaning up a working environment from which the bulk of the programming can work within.

The versions of AWK available on some systems do not allow BEGIN or END commands, but set all variables to zero or space when the execution of the program starts. Make sure you consult the documentation on your system before attempting anything ambitious.

In any case, the point for BEGIN and END segments is that AWK is a stateless programming environment.

That is, AWK treats each new input line in a similar way. Beginning and ending blocks as well as variables and conditional instructions enable the programmer to create a set of states that allows some lines of data to be treated in different ways than other lines of data. This is important because most real tasks need a set of states to filter data in a useful manner.

awk

Osnovna funkcija komande awk je pronalaženje uzorka teksta u datoteci i izvršavanje neke akcije, najčešće obrade pronađenog teksta. Generalno, awk je tekst procesor realizovan putem jednostavnog programskog jezika, a najpozntiji interpreteri su GNU awk (gawk) i mawk.

Primer 1.

```
$ cat /tmp/data
123abc
Wile E.
aabcc
Coyote
$ awk '/abc/ {print}' /tmp/data
123abc
aabcc
```

U ovom slučaj awk traži uzorak ‘abc’ u datoteci /tmp/data, a akcija koja se pri tom obavlja nad nađenim uzorcima je prikazivanje teksta na ekranu (print).

Primer 2.

Drugi primer je štampanje rednog broja prve linije teksta iza koje se traženi uzorak ne pojavljuje:

```
$ awk '/abc/ {i=i+1} END {print i}' /tmp/data
3
```

Ukoliko se u datoteci traži više uzoraka i ukoliko se vrši više obrada, potrebno je prvo napraviti datoteku u kojoj su opisane akcije (na primer actionfile.awk). Prilikom zadavanja komande awk potrebno je zameniti tekst između navodnika, kojim su opisani uzorak i akcija, imenom datoteke: '-f actionfile.awk'.

SHELL PROGRAMIRANJE nastavak

PROMENLJIVE

Na Linux sistemima postoje dva tipa promenljivih:

- **sistemske promenljive**, koje kreira i održava sam operativni sistem. Ne preporučuje se promena njihovog sadržaja (zašto?). Ovaj tip promenljivih definiše se strogo velikim slovima,
- **korisnički definisane promenljive** (*User defined variables* - UDV), koje kreiraju i održavaju korisnici. Ovaj tip promenljivih se obično definiše malim slovima;

U shell programiranju promenljive se ne deklarišu za specifični tip podataka - dovoljno je dodeliti vrednost promenljivoj i ona će biti alocirana prema toj vrednosti. U Bourne Again Shellu, promenljive mogu sadržavati brojeve, karaktere ili nizove karaktera.

VAŽNIJE SISTEMSKE PROMENLJIVE

Sistemske promenljive mogu se videti pozivom komande `set`:

```
$ set
BASH=/bin/bash
HOME=/home/jsmith
PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
PS1=[\u@\h \W]\$
PWD=/tmp/junk
SHELL=/bin/bash
USERNAME=jsmith
...
```

Neke od njih su:

`BASH`	lokacija komandnog interpretera
`HOME`	home directorijum korisnika
`PATH`	putanja u kojoj se traže izvršne datoteke
`PS1`	podešavanje prompta
`PWD`	tekući direktorijum
`SHELL`	ime komandnog interpretera
`USERNAME`	ime korisnika koji je u ovom režimu trenutno prijavljen na sistem.

Pojedinačno, sadržaj promenljive može se videti pozivom:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

DEFINISANJE KORISNIČKIH PROMENLJIVIH

Svaka promenljiva je univerzalna i nema nikakvu deklaraciju tipa (integer, float, string) i definiše se na sledeći način:

```
variablename = value
```

Primer.

```
$ br=10
```

Prilikom definisanja, odnosno dodele vrednosti, potrebno je primeniti sledeće konvencije o imenima promenljivih:

- Ime promenljive mora početi alfanumeričkim karakterom ili donjom crtom '_' (underscore character) praćenim jednim ili više alfanumeričkih karaktera.
  Primer.
  Korektne promenljive su: `HOME`, `SYSTEM_VERSION`, `br`, `_ime`;

- Prazne karaktere ne treba stavljati ni sa jedne strane znaka jednakosti prilikom dodele vrednosti promenljivim.
  Primer.
  ```
  $ br =10 # neispravno - prazni karakteri
  $ br= 10 # neispravno - prazni karakteri
  $ br = 10 # neispravno - prazni karakteri
  $ br=10 # ispravno
  ```
- case sensitive
  ```
  $ bR=20
  $ Br=30
  $ echo $bR
  20
  ```
- Može se definisati promenljiva nulte dužine (NULL variable), odnosno promenljiva koja nema vrednost u trenutku definisanja. Vrednosti ovih promenljivih se ne mogu prikazati komandom `echo`, sve dok sim se ne dodeli vrednost;
  ```
  $ br=
  $ ime=""
  ```
- Imena promenljivih ne smeju sadržati specijalne znake (poput ? i *).

Prikazivanje i korišćenje vrednosti UDV promenljivih

Primer 1.
```
$ echo $ime # prikazuje vrednost promenljive ime
johnny
$ echo ime # prikazuje string ime
ime
```

Primer 2.
```
$ x=10
$ xn=abc
$ echo $x $abc
10 abc
```

Primer 3. Definisati dve promenljive, x i y, sa vrednostima 20 i 5, respektivno, i promenljivu z kao njihov količnik. Rezultat prikazati na ekranu, u jednom redu.
```
$ x=20
$ y=5
$ z=`expr $x / $y`
$ echo x/y=$z
x/y=4
```

Komanda expr
```
$ expr 6 + 3    # expr posmatra 6 + 3 kao matematički izraz
9
```
Komanda `expr` određuje rezultat neke matematičke operacije. Sintaksa komande `expr` je:
```
expr op1 operacija op2
```
gde su op1 i op2 celi brojevi, a operator `+, -, \*, /, &`. Rezultat operacije je ceo broj. Argumenti op1, op2 i operator se moraju razdvojiti praznim karakterom.
```
$ expr 6 + 3    # ispravno
9
```

SPECIJALNE PROMENLJIVE

These variables are reserved for specific functions. For example, the $ character represents the process ID number, or PID, of the current shell.
If you were to type

```
$ echo $?
```

at a shell prompt, you'd get the exit status of the last command as the output.

Special variables that you can use in your bash scripts.

Variable	Function
?	The previous command's exit status.
$	The PID of the current shell process.
-	Options invoked at start-up of the current shell.
!	The PID of the last command that was run in the background.
0	The filename of the current script.
1-9	The first through ninth command-line arguments given when the current script was invoked: **$1** is the value of the first command-line argument, **$2** the value of the second, and so forth.
_	The last argument given to the most recently invoked command before this one.

ČITANJE PODATAKA SA ULAZA - read

Komanda `read` se koristi za čitanje ulaznih podataka sa tastature i memorisanje unete vrednosti u promenljivu. Sintaksa komande je:

```
$ read varible1, varible2,...varibleN
```

Primer.

```
#
# ss2: upotreba komande read
#
echo "Unesite podatak:"
read var1
echo "Uneli ste: $var1"
```

Pokretanje

```
# bash ss2
Unesite podatak: 123
Uneli ste: 123
```

KOMANDE, ARGUMENTI I IZLAZNI STATUS

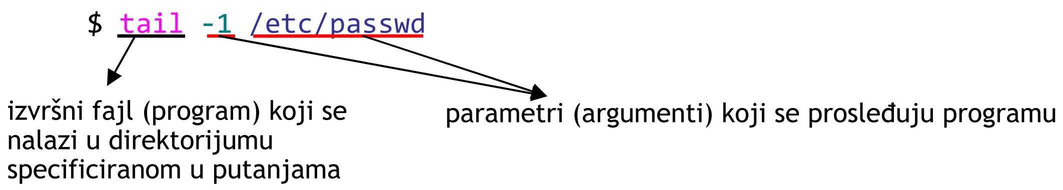
IZLAZNI STATUS KOMANDI

Nakon izvršenja Linux komande vraćaju vrednost na osnovu koje se može odrediti da li je komanda izvršena uspešno ili ne. Ako je povratna vrednost 0, komanda je izvršena uspešno. Ako je povratna vrednost različita od 0 (veća od 0), komanda se nije uspešno završila, a taj broj predstavlja neku vrstu dijagnostičkog statusa koja se naziva izlazni status.

```
$ rm plumph
rm: cannot remove `plumph': No such file or directory
$ echo $?
1 # izlazni status 1 -> komanda izvršena s greškom
$ date
$ echo $?
0 # izlazni status 0 -> komanda izvršena bez greške
```

ARGUMENTI

Komanda može biti zadata bez parametara (npr. date, clear, who), kao i sa jednim ili više parametara (`ls -l`, `ls -l /etc`, `mount -t ntfs /dev/hda1 /mnt/winc`).
Promenljiva $# memoriše broj argumenata specifirane komandne linije, a $* ili $@ upućuju na sve argumente koji se prosleđuju shell programu.


$ tail -1 /etc/passwd
izvršni fajl (program) koji se nalazi u direktorijumu specificiranom u putanjama
parametri (argumenti) koji se prosleđuju programu

Komandni argumenti se na isti način mogu zadati i shell script-u.
Na primer:

```
$ ss3 arg1 arg2 arg3
```

Argumente ovako pozvanog script-a se može pamte sledeće promenljive:

- $0 je ime programa - ss3
- $1 je prvi komandni argument - arg1
- $2 je drugi komandni argument - arg2
- $3 je treći komandni argument - arg3
- $# je broj komandnih argumenta - 3
- $* su svi komandni argumenti. $* se proširuje u \`$1,$2...$9\` - arg1 arg2 arg3.

Primer.

```
$ df
$ less /etc/passwd
$ ls -l /etc
$ mount -r /dev/hda2 /mnt/winc
```

ime programa	broj argumenata	argumenti		
$0	$#	$1	$2	$3
df	0			
less	1	/etc/passwd		
ls	2	-l	/etc	
mount	3	-r	/dev/hda2	/mnt/winc

Primer.

Script:

```
#
# ss3: korišcenje argumenata komandne linije
#
echo "Ukupan broj argumenata komandne linije: $#"
echo "$0 je ime programa, a $1 je prvi argument."
echo "Svi argumenti su redom: $*"
```

Pokretanje:

```
$ bash ss3 arg1 arg2 arg3
```

Flow Control

Sure, variables and other required programming constructs are interesting, but on the face of it, they may not seem to be particularly useful. One of the main reasons for variables in a shell script, however, is that they permit flow control. Flow control is crucial because it allows the program to evaluate conditions and take actions contingent on those conditions.

Flow control can generally be broken down into two types:

- conditional and
- iterative.

CONDITIONAL FLOW CONTROL

The if-then Statement

In general terms, an if-then statement looks like this:

```
if some_condition then
            something happens
fi
```

The if-then block is terminated with the word `fi`.

Primer.

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR=”purple” ]
then
echo “You are correct.”
fi
```

Such constructs can quickly become quite complex. You can specify multiple conditions by adding an `else` clause to the if-then construct, as in this example:
Primer.

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR = ”purple” ]
    then
        echo “You are correct.”
    else
        echo “Your guess was incorrect.”
fi
```

obratite pažnju na razmake i nove linije

NAPOMENA
`$COLOR = “purple”` - poredi vrednosti
`$COLOR=“purple”` - dodeljuje vrednost

In addition, you can use an `elif` clause to specify a second condition:

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR = ”purple” ]
    then
        echo “You are correct.”
    elif [ $COLOR = ”blue” ]
        then  echo “You’re close.”
        else
            echo “Your guess was incorrect.”
fi
```

You can add as many `elif` clauses as you want. They are particularly useful for finely shaded responses from the script, or when you know that there are a limited number of possible input values, each of which requires a particular response.

Multiple Conditions:

```
      if [ condition1 ]
                then
                      if [ condition2 ]
                                then
                                some action
                      fi
      fi
```

Or you could simplify it by using a logical and operator

```
      if [ condition1 && condition2 ]
                then
                some action
      fi
```

There is also a logical or operator (||):

```
      if [ condition1 || condition2 ]
                then
                some action
      fi
```

which is equivalent to:

```
      if [ condition1 ]
                then
                some action
                elif [ condition2 ]
                then
                      the same action
      fi
```

The `test` Command

The test command is used to evaluate conditions. You'll notice that the preceding examples include square brackets around the conditions to be evaluated. The square brackets are syntactically equal to the test command. For example, the preceding script could have been written:

```
            if ( test $COLOR = ”purple” )
```

This concept is important because the `test` command has a number of options that can be used to evaluate all sorts of conditions, not just simple equality. For example, use test to see whether a particular file exists:

```
            if ( test -e filename )
```

In this case, the test command would return a value of true, or 0, if the file exists, and false, or 1, if it doesn't.

You can get the same effect by using square brackets:

```
            if [ -e filename ]
```

The following table shows other options you can use with test or with square brackets:

Option	**Test Condition**
`-d`	The specified file exists and is a directory.
`-e`	The specified file exists.
`-f`	The specified file exists and is a regular file (not a directory or other special file)
`-G`	The file owner's group ID matches the file's ID.
`-nt`	The file is newer than another specified file (takes the syntax `file1 -nt file2`).

-ot	The file is older than another specified file (takes the syntax `file1 -ot file2`).
-O	The user issuing the command is the file's owner.
-r	The user issuing the command has read permission for the file.
-s	The specified file exists and is not empty.
-w	The user issuing the command has write permission for the file.
-x	The user issuing the command has execute permission for the file.

With all these options, the command will return a value of either 1 or 0, depending on whether the implied statement is true or false.

Comparison Operators
Comparison operators, shown in the following table, work in the same way that they do in simple arithmetic.

=	is the same
!=	is not the same
>	is greater than
<	is lesser than

You might wonder how a text string can have a greater or lesser value than another text string, because letters don't normally have numerical values. In this case, however, they do have a value. **Textual comparison is strictly alphabetical: a is greater than b, b is greater than c, and so forth**.

The `case` Statement

General format:
```
case expression in
pattern1)
      action1
      ;;
pattern2)
      action2
      ;;
pattern3)
      action3
      ;;
esac
```
Note also that each option's section is concluded with the double semicolon (`;;`).
case statements are great for evaluating command-line arguments. Take a look at this script excerpt:

```
# See how we were called.
case “$1” in
start)
      # Start daemons.
      action $”Starting NFS services: “ /usr/sbin/exportfs -r
      echo -n $”Starting NFS quotas: “
      daemon rpc.rquotad
      echo
      echo -n $”Starting NFS mountd: “
      daemon rpc.mountd $RPCMOUNTDOPTS
      echo
      echo -n $”Starting NFS daemon: “
      daemon rpc.nfsd $RPCNFSDCOUNT
```

```
            echo
            touch /var/lock/subsys/nfs
            ;;
      stop)
            # Stop daemons.
            echo -n $”Shutting down NFS mountd: “
            killproc rpc.mountd
            echo
            echo -n $”Shutting down NFS daemon: “
            killproc nfsd
            echo
            action $”Shutting down NFS services: “ /usr/sbin/exportfs -au
            echo -n $”Shutting down NFS quotas: “
            killproc rpc.rquotad
            echo
            rm -f /var/lock/subsys/nfs
            ;;
      status)
            status rpc.mountd
            status nfsd
            status rpc.rquotad
            ;;
            restart)
            echo -n $”Restarting NFS services: “
            echo -n $”rpc.mountd “
            killproc rpc.mountd
            daemon rpc.mountd $RPCMOUNTDOPTS
            /usr/sbin/exportfs -r
            touch /var/lock/subsys/nfs
            echo
            ;;
      reload)
            /usr/sbin/exportfs -r
            touch /var/lock/subsys/nfs
            ;;
            probe)
            if [ ! -f /var/lock/subsys/nfs ] ; then
            echo start; exit 0
            fi
            /sbin/pidof rpc.mountd >/dev/null 2>&1; MOUNTD=”$?”
            /sbin/pidof nfsd >/dev/null 2>&1; NFSD=”$?”
            if [ $MOUNTD = 1 -o $NFSD = 1 ] ; then
            echo restart; exit 0
            fi
            if [ /etc/exports -nt /var/lock/subsys/nfs ] ; then
            echo reload; exit 0
            fi
            ;;
      *)
            echo $”Usage: $0 {start|stop|status|restart|reload}”
            exit 1
      esac
```

This sample is taken from the `/etc/rc.d/init.d/nfs` script on Red Hat Linux. It is the script that controls the NFS (Network File System) service.

Iterative Flow Control

The `while` Statement

The while statement causes a block of code to repeat as long as a certain condition is met. For example:

```
#!/bin/bash
echo “Guess the secret color: red, blue, yellow, purple, or orange \n”
read COLOR
while [ $COLOR != “purple” ]
do
      echo “Incorrect. Guess again. \n”
      read COLOR
done
echo “Correct.”
```

The `until` Statement

The condition in the until statement is the opposite of that in the while statement. For example, you could rewrite the previous example this way:

```
#!/bin/bash
echo “Guess the secret color: red, blue, yellow, purple, or orange \n”
read COLOR
until [ $COLOR = “purple” ]
do
      echo “Incorrect. Guess again. \n”
      read COLOR
done
echo “Correct.”
```

for petlja

Sintaksa for petlje ne liči mnogo na sintaksu C jezika:

```
for name [in words ...];
do
commands;
done
```

Promenljiva `name` dobija vrednost tekućeg člana liste `word`. Ako se `in words` izostavi u naredbi select, ili ako se specificira `in "$@"', tada će `name` uzimati vrednost pozicionih parametara. **Izlazni status for petlje jednak je izlaznom statusu zadnje izvršene komande u grupi commands.** Ako je lista words prazna nijedna komanda se neće izvršiti i tada će izlazni status biti 0.

```
#
# ss11: upotreba for petlje
#
if [ $# -eq 0 ]
      then
      echo "Greška - numericki argument nije naveden"
      echo "Sintaksa : $0 broj"
      echo "Program prikazuje tablicu množenja za dati broj"
      exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
```

```
do
      echo "$n * $i = `expr $i \* $n`"
done
```

Sledeći primer ilustruje upotrebu alternativne for petlje:

```
for i in `seq 1 10`
do
      echo $i
done
```

For petlja najpre kreira promenljivu i, a zatim joj redom dodeljuje vrednosti iz liste (u ovom slučaju numeričke vrednosti od 1 do 10). Shell izvršava echo naredbu za svaku vrednost promenljive i.
Alternativna forma for petlje podse¤a na sintaksu for petlje programskog jezika C:

```
for (( expr1 ; expr2 ; expr3 )) ;
do
commands ;
done
```

Naredba select

```
select name [in words ...];
do
commands;
done
```

Lista reči se proširuje generišući listu stavki (item). Skup proširenih reči prikazuje se na standardnom izlazu za greške, pri čemu svakoj prethodi redni broj. Ako se `` `in words' `` izostavi u naredbi select, ili ako se specificira `in "$@"', tada se prikazuju pozicioni parametri. U slučaju `` `in "$@"' `` PS3 prompt se prikazuje i linije se čitaju sa standardnog ulaza. Ako se linija sastoji od broja koji odgovara jednoj od prikazanih reči tada se vrednost promenljive name postavlja u tu reč. Ukoliko je linija prazna reči i prompt se prikazuju ponovo. Ako se pročita EOF select komanda završava rad. Svaka druga pročitana vrednost uzrokuje da promenljiva name bude postavljena na nulu. Pročitana linija se čuva u promenljivoj REPLAY.
Komande se izvršavaju posle svake selekcije sve dok se ne izvrši break komanda, čime se komanda select završava.
Primer ilustruje upotrebu naredbe select: program dozvoljava korisniku da sa tekućeg direktorijuma izabere datoteku čije će ime i indeks nakon toga biti prikazani.

```
select fname in *;
do
      echo Datoteka: $fname \($REPLY\)
break;
done
```

Sledeći primer ilustruje kreiranje prostog menija:

```
opcije="Pozdrav Kraj"
select op in $opcije;
do
if [ "$op" = "Kraj" ];
then
echo OK.
exit
elif [ "$op" = "Pozdrav" ];
then
echo Linux Rulez !
else
clear
echo Opcija ne postoji.
fi
```

```
	done
```

STDIN, STDOUT, and STDERR

Every time you open up a shell, Unix opens up three files for use by your program:

- STDIN (standard in)—This is generally your terminal’s keyboard.
- STDOUT (standard out)—This is generally your terminal’s monitor.
- STERR (standard error)—This also generally points to your terminal’s monitor.

The important thing to remember here is that, by default, input comes from your keyboard and is printed to your screen. While this is the way most interaction is achieved, it is by no means the only way in which programs and files can interact. Redirection often involves associating a file with one of the standard input or output (IO) files and sending the desired information to the required file.
Redirection is pretty simple; Unix provides you with a few simple operators that handle the associations for you.

Redirection Operators

Operator	Action
>	Redirects STDOUT to a file
<	Redirects STDIN to a file
>>	Appends STDOUT to a file.
|	Takes output from one program, or process, and sends it to another
<< delimiter	Associates the current input stream with STDIN until the specified delimiter is reached

Primeri.

```
$ ls > fileList
```

The output is redirected to `fileList`.
Of course, you might not want `fileList` to be written over every time you send output to it. In that case, you could use >> to seek to the end of the target file and append the results there:

```
$ ls /home/student >> fileList
```

Sometimes you will want to send output from one process directly to another process.

```
$ ls | wc
```

In this example, the STDOUT of ls is sent directly to the STDIN of `wc` utility, which dutifully prints out its results to the screen (because its output isn’t redirected elsewhere).

The final method of redirection examined here is << delimiter, which is used in a special type of document called a HERE file. Basically, this uses redirection to read from the given input until the delimiter is reached. One simple use of this would be to print multiple lines of output to the screen without using echo over and over:

```
Cat <<END
The cat
Sat on the
Mat.
END
```

```
$ ls >& fileList
```

Using > with & causes both `STDOUT` and `STDERR` to be redirected. In this case, any error messages would not print to the screen but would be sent to `fileList`.

Standardni ulaz (`STDIN`), standardni izlaz (`STDOUT`) i standardni izlaz za greške (`STDERR`) su deskriptori datoteke kojima su dodeljeni brojevi po sledećim pravilima:

0	predstavlja `STDIN`,
1	predstavlja `STDOUT` i
2	predstavlja `STDERR`.

Primeri redirekcije:
Sledeći primer demonstrira kreiranje datoteke `greperr.txt` i upis poruka o greškama koje proizvodi komanda grep u datoteku;

```
$ grep kyuss * 2> greperr.txt
```

Redirekcija `STDERR` u `STDOUT`, koja je demonstrirana sledećim primerom. Rezultat izvršenja komande `grep` smešta se u bafer i može se naknadno videti, a poruke o greškama koje komanda `grep` proizvodi prikazuju se na standardnom izlazu, a to je u podrazumevanom stanju ekran;

```
$ grep kyuss * 2>&1 greperr.txt
```

Redirekcija `STDERR` i `STDOUT` u datoteku. Ova vrsta redirekcije je korisna za programe koji rade u pozadini, tako da se od njih očekuje da poruke ne upisuju na ekran, već u neku datoteku. Dodatno, ukoliko korisnik ne želi da vidi "feedback"komande, izlaz i poruke o greškama mogu se preusmeriti na uređaj `/dev/null`, kao u sledećem primerom:

```
$ rm -f $(find / -name core) &> /dev/null
```

SHELL EXPANSIONS

Command Substitution: Back Ticks and Brace Expansion

It’s often quite important to capture the results of some command in a variable for use by your shell script. Back ticks are really useful for this sort of thing because they provide you with an inline method for executing a command and retrieving the results **before the rest of your script executes.**
For example, it is conceivable that you may wish to capture the number of lines in a given file so that your shell can then use this to determine a certain action. Using back ticks makes this a pretty simple task:

```
Lines=`wc -l textFile`
```

The variable `Lines` now contains the number of lines in the `textFile` file and can be used elsewhere in the shell script as needed.

Shells also expand the contents of double-quoted strings, so double quotation marks also play a role in command substitution. Of particular interest here is brace expansion, which uses the format: `$(command)`, where `command` is any valid Unix command.
Accordingly, you could have achieved the same result as the preceding line by saying:

```
Lines=”$(wc -l textFile)”
```

Remember to use quotation marks to help clarify what you mean in your commands.

Use single quotation marks if you don’t want any type of expansion to take place—in other words, you are enclosing a literal string. Otherwise, use double quotation marks if you do want variables to be substituted, and commands to be executed.

It’s useful to note that the `$(command)` format supports nesting without having to escape the $(and) characters, so you can perform several operations in one go if necessary. Like this:

```
$ echo “Next year will be 20$(expr $(date +%y) + 1).”
```

Shell proširenja (Shell Expansions)

Proširenje preko zagrada, tilda proširenje, proširenje parametara i promenljivih, zamena komandi, aritmetičko proširenje, razdvajanje reči, proširenje imena datoteke.
Proširenje se izvršava na komandnoj liniji. Bash prepoznaje sedam proširenja i izvršava ih

sledećim redom:

- proširenje preko zagrada (*brace expansion*) – zamena sadržaja u zagradama
- tilda proširenje (*tilde expansion*) – tilda zamena
- proširenje parametara i promenljivih (*parameter and variable expansion*) zamena parametara i promenljivih
- aritmetičko proširenje (*arithmetic expansion*)
- zamena komandi (*command substitution*), koja se obavlja sleva nadesno
- razdvajanje reči (*word splitting*)
- proširenje imena datoteke (*filename expansion*).

Samo proširenje u zagradama, razdvajanje reči i proširenje imena datoteka mogu promeniti broj reči u proširenju, ostala proširenja proširuju jednu reč u jednu reč.

- **Proširenje preko zagrada (***Brace Expansion***)**
  Proširenje preko zagrada je mehanizam kojim se mogu proširiti proizvoljni nizovi. Ovaj mehanizam je sličan proširenju imena datoteke, ali generisana imena datoteka ne moraju da postoje. Uzorci koji se preko zagrada proširuju uzimaju formu opcionog uvodnog dela, koju prati serija zapetom razdvojenih nizova između para zagrada, iza kojih ide opcioni dotatak. Uvodni deo je prefiks svakog niza koji se nalazi unutar zagrada, a dodatak se dodaje s desne strane na svaki rezultujući niz.
  Proširenja preko zagrada mogu da se se umeću jedno u drugo. Rezultati svakog proširenog niza nisu sortirani, samo se poštuje poredak sleva nadesno, odnosno prefiks, zatim niz iz zagrade, i na kraju dodatak-sufiks.
  Primer.
  Proširenje komande `echo`:

```
$ echo a{d,c,b}e
ade ace abe
```

  Proširenje preko zagrade **se izvršava pre bilo kog drugog proširenja**. Bilo koji karakter koji ima specijalno značenje za ostala proširenja čuva se u rezultatu, odnosno ne dira se. To je strogo tekstualno proširenje.
  Bash ne primenjuje interpretaciju u kontekstu proširenja ili teksta između zagrada. Da bi izbegavao konflikte sa parametarskim proširenjima niz "${" se ne smatra pogodnim za proširenje preko zagrada. Korektno formirano proširenje preko zagrada mora sadržati otvorenu i zatvorenu zagradu koje su van navodnika, i barem jednu zapetu. Svako nekorektno proširenje se ne izvršava.
  Ova konstrukcija se tipično koristi kao skraćenica kada se isti zajednički prefiks generiše više puta. Tako se:
  Primer.

```
$ mkdir /home/jsmith/{data,video,mp3}
```

  se proširuje u:

```
$ mkdir /home/jsmith/data
$ mkdir /home/jsmith/video
$ mkdir /home/jsmith/mp3
```

  Komplikovaniji slučaj je korišćenje ugnježdenih proširenja.

```
$ chown root /home/{jsmith/{ss1,ss2},nmacek/{data,ss3}}
```

  proširuje se u:

```
$ chown root /home/jsmith/ss1
$ chown root /home/jsmith/ss2
$ chown root /home/nmacek/data
$ chown root /home/nmacek/ss3
```

- **Tilda proširenje (***Tilde Expansion***)**
  Ako reč počinje tilda karakterom koji nije pod navodnicima (~), svi karakeri do prve kose crte koja je takođe van navodnika (/ slash) tretiraju se kao tilda prefiks. Ukoliko nema kose crte onda su svi karakteri tilda prefiks.

Ukoliko nema karaktera pod navodnicima unutar tilda prefiksa, tilda prefiks se tretira kao potencijalno ime korisnika za login proceduru (login-name). Tilda prefiks se zamenjuje po sledećim pravilima:
ako je login-name nulte dužine, tilda se zamenjuje vrednošću HOME promenljive, a ako je HOME promenljiva nepostavljena, tilda se zamenjuje home direktorijumom korisnika koji izvršava taj komandi interpreter.
U drugom slučaju tilda prefiks se zamenjuje home direktorijumom specificiranog korisnika (login-name). Ako je vrednost tilda prefiksa "~+", tada tilda prefiks uzima vrednost shell promenljive PWD koja predstavlja tekući radni direktorijum.
Ako je vrednost tilda prefiksa "~-", tada tilda prefiks uzima vrednost shell promenljive OLDPWD koja predstavlja prethodni tekući radni direktorijum (pod uslovom da je OLDPWD setovana).
Ako je login-name pogrešan, tilda proširenje se ne izvršava, reč s leve stane ostaje nepromenjena. Svaka dodela promenljivoj se proverava za tilda prefikse van navodnika iza kojih neposredno ide : ili =. U ovim slučajevima tilda proširenje se takođe izvršava. Prema tome, nekom mogu koristiti imena datoteka sa tildom u dodeljivanju sistemskih promenljivih kao što je PATH, MAILPATH i CDPATH, a komandni interpreter će im dodeliti proširene vrednosti.
Upotreba tilda proširenja za pozicioniranje na home direktorijum:

~	vrednost promenljive $HOME (/home/jsmith)
~	/data $HOME/data (/home/jsmith/data)
`~jim`	home directorijum korisnika jim (/home/jim).

Sledeći primer demonstrira upotrebu tilda proširenja za promenljivu `$OLDPWD`:

```
~+/misc  $PWD/misc
~-/temp  $OLDPWD/temp
```

Primer 1.

```
$ whoami
jsmith
$ pwd
/etc
$ cd ~/data       # poddirektorijum data home direktorijuma
$ pwd
/home/jsmith/data
$ cd ~jim         # home direktorijum korisnika jim
$ pwd
/home/jim
```

Primer 2.

```
$ pwd
/etc
$ cd /bin
$ cd ~-/network
$ pwd
/etc/network
```

Parametarsko proširenje (*Shell Parameter Expansion*)

Znak $ uvodi parametarsko proširenje, komandnu zamenu ili aritmetičko proširenje. Ime parametra ili simbola koji se proširuje može biti zatvoreno u zagradama koje su opcione ali štite promenljivu koja se proširuje od karaktera koji je slede iza nje i koji bi se mogli pogrešno interpretirati kao deo imena. Kada se koriste zagrade, zadnja zagrada je prvi znak koji nije u sastavu obrnute kose crte ili pod navodnicima i nije u okviru aritmetičkih proširenja, komandnih zamena ili parametarskih proširenja.
Osnovna forma parametarskog proširenja je

```
            ${par}
```

Vrednost parametra (par) se zamenjuje. Zagrade se zahtevaju kada je reč o pozicionom parametru, sa više od jedne cifre, ili kada je parametar praćen karakterom koji se ne interpretira kao deo njegovog imena.
Ako je prvi karakter parametra znak uzvika "!", uvodi se nivo promenljive indirekcije. Bash koristi vrednost promenljive formirane od ostatka parametra kao ime promenljive. Ova promenljiva se zatim proširuje i ta vrednost se koristi u ostatku zamene, umesto vrednosti samog parametra. Ovo je poznato kao **indirektno proširenje**. Izuzetak ovog proširenja je slučaj

```
            ${!prefix*}.
```

${par:-word}	Ako parametar ne postoji ili je null, zamenjuje se proširenjem word. U drugom slučaju zamenuje se vrednošću parametra.
${par:=word}	Ako parametar ne postoji ili je null, proširenje word se dodeljuje parametru. Vrednost parametra se tada zamenjuje. Pozicioni i specijalni parametri ne moraju se postavljati na ovaj način.
${par:?word}	Ako parametar ne postoji ili je null, proširenje word se upusuje na standardni izlaz za greške (standard error) i komandi interpreter prekida rad (exit). Vrednost parametra se tada zamenjuje.
${par:+word}	Ako parametar ne postoji ili je null, ništa se ne zamenjuje, a u drugom slučaju proširenje word se zamenjuje.
${par:offset:lenght}	Proširuje do dužine length karaktera parametra, počevši od karaktera specificiranog pomoću polja offset. Ako se polje length izostavi, proširuje se podniz počevši od karaktera specificiranog pomoću polja offset zaključno sa zadnjim karakterom. Polja length i offset su aritmetički izrazi. Ovo se još naziva i podnizno proširenje (Substring Expansion). Polje length mora biti broj veći ili jednak 0. Ako je polje offset broj manji od 0, vrednost se koristi kao pomeraj u odnosu na kraj vrednost i parametra. Ako je parametar "@", rezultat je polje length pozicionih parametara koji po□inju u polju offset. Ako je parametar polje imena koje se indeksira pomoću "@" ili "*", rezultat je length polje članova polja koji počinju sa `${par[offset]}`. Podnizno indeksiranje je bazirano na nuli, osim u slučaju kada se koriste pozicioni parametri, kada indeksiranje startuje u 1.
${!prefix*}	Proširuje imena promenljivih čija imena počinju prefiksom prefix, razdvojenim prvim karakterom IFS specijalne promenljive.
${#par}	Dužina proširene vrednosti parametra se zamenjuje. Ako je parametar "*" ili "@", zamenjena vrednost je broj pozicionih parametara. Ako je parametar polje imena koja se indeksiraju pomoću "@" ili "*", zamenjena vrednost je broj elemenata u polju.

Komandna zamena (Command Substitution)
Komandna zamena dozvoljava da se izlaz komande zameni samom komandom, odnosno da izlaz jedne komande postaje argumenat druge. Komanda zamena se izvršava kada se komanda zatvori zagradama ili navodnicima, kao u sledećim primerima:

```
            $(command)
```

ili

```
            `command`
```

Bash izvršava proširenje izvršavanjem komande command i zamenjuje komandnu substituciju sa standardnim izlazom komande. Ugrađene nove linije se ne brišu, ali mogu da se uklone za vreme razdvajanja reči.
Kada se koristi zamena stilom forme obrnutog navodnika, karakter obrnuta kosa crta zadržava doslovno značenje osim kada je praćen sa "$", "`", ili "\". Prvi obrnuti navodnik, koji nije praćen obrnutom kosom crtom, prekida komandnu zamenu.
Kada se koristi `$(command)` forma, svi karakteri između malih zagrada tretiraju se kao komande, ništa se ne tretira specijalno.
Ako se zamena pojavljuje sa duplim navodnicima, razdvajanje reči i proširenje imena datoteka

datoteka se ne izvršava.

Primer.
Pronalažeanje svih datoteka sa ekstenzijom bak.
```
            $ find / -name ‘*.bak’ –print
```
Komprimovanje istih u jednoj komandi može se izvršiti na dva načina:
```
            $ gzip ` find / -name ‘*.bak’ –print `
```
ili
```
            $ gzip $( find / -name ‘*.bak’ –print )
```
Dodatno, pomoću komandne zamene se mogu dodeliti vrednosti promenljivim.
```
            $ x = `date`
            $ echo $x
            Thu Apr 15 09:53:44 CEST 2004
            $ y = `who am i;pwd`
            $ echo $y
            nmacek pts/0 Apr 15 09:40 (nicotine.internal.vets.edu.yu)
            /home/nmacek
```

Aritmetičko proširenje (Arithmetic Expansion)

Aritmetičko proširenje omogućava izračunavanje aritmetičkog izraza i zamenu rezultata. Format aritmetičkog izraza je:
```
            $(( expression ))
            ili
            $[ expression ]
```
Izraz se tretira kao da je bio u duplim navodnicima, ali dupli navodnici unutar zagrada se ne tretiraju specijalno. Svi simboli u izrazu podležu parametarskom proširenju, komandnoj zameni i navodničkom uklanjanju. Aritmetičke zamene mogu da se gnezde.
Izračunavanje se izvršava prema pravilima shell aritmetike. Ako je izraz pogrešan bash prikazuje poruku koja prijavljuje otkaz i zamena se ne izvršava.
Evo nekoliko primera:
```
            $ echo 1 + 1           # shell interpretira 1 + 1 kao string
            1 + 1
            $ echo $((1+1))        # $((1+1)) je aritmeti�ko proširenje
            2
            $ echo $((7/2))        # bash koristi celobrojnu aritmetiku
            3
            $ echo 3/4|bc -l       # celobrojna aritmetika
            0.75
            $ a=1
            $ b=2
            $ echo $(($a+$b))      # promenljive i aritmeti�ko proširenje
            3
```
Bash koristi celobrojnu aritmetiku - komanda `echo $[3/4]` na ekranu prikazuje 0. Ukoliko je potrebno izvršiti neku operaciju sa realnim rezultatom ili više matematičkih operacija, može se koristiti program bc - rezultat komande echo 3/4|bc -l je 0.75, što je korektno.
Aritmetičko proširenje se može iskoristiti za određivanje istinitosti izraza. U tom slučaju, proširenje vra¤a status 0 ili 1 zavisno od vrednosti uslovnog izraza expression. Izraz se komponuje pomoću operatora <, <=, >, >=, == i ! =. Dodatno, izrazi mogu da se kombinuju koristeći sledeće operatore:

(expression)	vraća vrednost izraza expression.
! expression	tačno ukoliko je expression netačan (negacija)
exp1 && exp2	tačno samo pod uslovom ako su oba izraza (exp1 i exp2) tačni
exp1 || exp2	tačno ako je bar jedan od izraza (exp1 ili exp2 tačan).

Operatori && i || ne izračunavaju vrednost exp2 ako je vrednost izraza exp1 dovoljna da odredi povratnu vrednost celog uslovnog izraza.

```
$ echo $((1>3||2<4))
1
$ echo $((1>3&&2==2))
0
```

Kada se koriste operatori "==" i "!=" niz desnog operatora smatra se uzorkom, a provera identičnosti odgovara pravilima za pronalaženje uzorka (Pattern Matching). Vrednost 0 se vra¤a ako niz odgovara uzorku, a vrednost 1 ako ne odgovara. Razdvajanje reči i proširenje imena datoteka se ne izvršavaju unutar ovog proširenja; tilda proširenje, parametarsko proširenje, komandna zamena, procesna zamena i upotreba navodnika se izvršavaju.

```
$ ime=jsmith
$ echo $(($ime==jsmith))
1
$ echo $(($ime!=jsmith))
0
```

Proširenje imena datoteka (Filename Expansion)

Nakon zadavanja komande, Bash razdvaja reči koje predstavljaju parametre i u parametrima koji predstavljaju datoteke traži karaktere "*", "?", i "[". Ako se jedan od tih karaktera pojavi tada se reč smatra uzorkom i zamenjuje se alfabetski sortiranom listom imena datoteka koja odgovara uzorku. Ukoliko je Bash pokrenut sa parametrom -f ova zamena se ne izvršava.

Pronalaženje uzorka (Pattern Matching)
Prilikom pronalaženja uzorka specijalni karakteri imaju sledeće značenje:
* odgovara bilo kom nizu uključujući i niz nulte dužine. Takođe, na primer, komanda ls * prikazati sve datoteke, ls a* sve datoteke čije ime počinje sa a, a ls *.c sve datoteke koje imaju ekstenziju .c;
? odgovara bilo kom karakteru. Takođe, na primer, ls ? prikazati sve datoteke čije ime ima tačno jedan karakter, a ls fo? sve datoteke čije ime ima tačno tri karaktera, od kojih su prva dva fo;
[...] odgovara jednom od karaktera koji je naveden između zagrada. Ukoliko je prvi karakter iza otvorene zagrade "!" ili "^" tada odgovaraju svi karakteri koji nisu navedeni između zagrada. Na primer, ls [abc]* će prikazati sve datoteke čije ime počinje slovima a,b ili c, a ls [^abc]* sve datoteke čije ime ne počinje tim slovima;
[..-..] par karaktera razdvojen znakom "-" označava zonu, odnosno opseg. Ukoliko je prvi karakter iza otvorene zagrade "!" ili "^" tada odgovaraju svi karakteri koji ne pripadaju opsegu. Na primer, ls /bin/[a-f]* će prikazati sve datoteke direktorijuma /bin čije ime počinje slovima a,b,..f, a ls /bin/[^a-e]* sve datoteke direktorijuma /bin čije ime ne počinje tim slovima;

SHELL FUNCTIONS

To declare a function, simply use the following statement:

```
name () { commands; }
```

The name of your function is name, and that’s what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.
Primer.
Skript

```
# func
# A simple function
repeat() {
```

```
echo -n "I don't know $1 $2 "
}
repeat Your Name
```

The following should print to the screen:

```
I don't know Your Name
```

How It Works

The function repeat was declared with one `echo` command in the commands list. Once it has been declared, you can use the function from anywhere in the script. In this case, it was called it directly after the declaration, passing it two parameters: Your and `Name`:

```
repeat Your Name
```

A key point here, though, is that a **function must be declared before it is referenced within a script**.

Returning Values

To explicitly set the exit status of a function (by default it is the status of the last command), use the following statement:

```
return code
```

code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Scope

There are two types of scope—global and local. If a variable has global scope, it means that it is accessible from anywhere within a script. This is not true for variables with local scope, which are accessible only in the block in which they were declared.

To get a better feel for how scope works, tackle the short example in the following **Primer.**

```
#!/bin/bash
# scope
# dealing with local and global variables
scope ()
{
local lclVariable=1
gblVariable=2
echo "lclVariable in function = $lclVariable"
echo "gblVariable in function = $gblVariable"
}
scope
# We now test the two variables outside the function block to see what
happens
echo "lclVariable outside function = $lclVariable"
echo "gblVariable outside function = $gblVariable"
exit 0
```

You should receive the following output:

```
lclVariable in function = 1
gblVariable in function = 2
lclVariable outside function =
gblVariable outside function = 2
```

Signals and Traps

Certain process level events throw up signals, which can be useful for determining an action to take based on what has happened to that process. For example, a user might issue a Ctrl + C command to a process that is trying to write to a temporary file. This type of situation should be dealt with appropriately lest you lose important information when your temporary files are deleted on the next reboot.

Pressing Ctrl + C sends a signal to your shell process; how you react to this is up to you. You can use the `trap` command to take an action based on the type of signal you receive. To get a list of all the signals that can be sent to your processes, type the following:

```
$ trap –l
```

Some of the more common signals you might encounter and want to use in your programs:

Signal Name	Signal Number	Description
SIGHUP	1	Issued if a process is left hanging—the terminal is disconnected.
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted.
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations (such as closing files or removing temporary files).

Trapping these signals is quite easy, and the trap command has the following syntax:

```
trap command signal
```

command can be any valid Unix command, or even a user-defined function, and signal can be a list of any number of signals you want to trap. Generally speaking, there are three common uses for traps:

- removing temporary files,
- ignoring signals, and
- ignoring signals during special operations.

An example of how to use signals and traps is examined shortly. However, signals and traps can play a part in file handling, so you should take a look at that first.

File Handling

This section discusses how shell scripts can be used to perform file-related tasks efficiently. To begin with, you will look at how to determine whether a given file exists. This is obviously important for a healthy, robust file-handling script. Taking things a step further, you learn how to clean up files in the event something unforeseen happens to your scripts.

File Detection

Things can go horribly wrong if you try to overwrite files that already exist, or write to files that don't exist. You can also quite easily decide whether a file is readable or writable.

```
if [ -w writeFile ]
then
echo “writeFile exists and is writable!”
fi
```

The test in this case is given in generic form by:

```
[ option file ]
```

Where option and file are determined by what you need to test for and the file you want to test. The following table lists the most common options used.

Expression	**Meaning**
-d *file*	True if: file exists and is a directory.
-e *file*	True if: file exists.
-r *file*	True if: file exists and is readable.
-s *file*	True if: file exists and size is greater than zero.
-w *file*	True if: file exists and is writable.
-x *file*	True if: file exists and is executable.

You can combine these tests using && (logical and) or || (logical or) to fine-tune your test even more:

```
if [ -r writeFile && -x writeFile ]
then
echo “writeFile exists, is readable, and executable!”
```

```
      fi
```
and
```
      if [ -r writeFile || -w writeFile ]
      then
      echo “writeFile exists and is readable or writable!”
      fi
```

Cleaning Up Files
It is not very good practice to allow scripts to create a whole bunch of temporary files and then leave them lying around. For example, you may decide that a script is taking too long to execute and it’s time to go home. What do you do?
Most often, people will hit `Ctrl + C` to terminate their script’s execution. If there’s no proper file handling in place, it could lead to some sticky problems. By capturing signals sent to your processes, you can call functions that will act appropriately.
Primer.
Captures an interrupt sent by a user and uses `trap` to clean up a temporary file created during the course of the script.
```
      #!/bin/bash
      # sigtrap
      # A small script to demonstrate signal trapping
      tmpFile=/tmp/sigtrap$$
      cat > $tmpFile
      function removeTemp() {
      if [ -f “$tmpFile” ]
      then
      echo “Sorting out the temp file... “
      rm -f “$tmpFile”
      fi
      }
      trap removeTemp 1 2
      exit 0
```
How It Works
The first line creates a temporary file using the shell’s process ID, suffixed by the word sigtrap:
```
      tmpFile=/tmp/sigtrap$$
```
The next line uses the cat command without any parameters. This causes the script to wait for input from the user. You need the program to wait for a little while to give you time to send the appropriate signal:
```
      cat > $tmpFile
```
Next, a function was created to remove any temporary files the script had created before exiting:
```
      function removeTemp() {
      if [ -f “$tmpFile” ]
            then
            echo “Sorting out the temp file... “
            rm -f “$tmpFile”
      fi
      }
```
This is the main use of trapping. You will often want to call a specific function based on the type of signal you receive and the type of task your script is completing.

Finally, the trap command was set to call the `removeTemp()` function in the event that the process received signals 1 or 2 (the signals for the process being hung, or the user sending an interrupt, Ctrl + C):
```
      trap removeTemp 1 2
```
You might find that you want to perform different operations depending on which signal is

received. In that instance, simply create several trap statements with their corresponding functions to handle execution.

Exercise 1

Using if statements, write a script named filescript.sh that will:

1. Take an argument from the command line. This argument should be a directory path. If no argument is given, the program will use the current directory as the default.
2. List all text files in that directory (files whose names contain the suffix .txt).
3. Along with listing the filenames, give the user the option to choose the file's size, permissions, owner, or group, or "all of the above" information to be displayed. Do this interactively.

Solution

```
#########################################################################
# filescript.sh
# # Dec. 9, 2004
## A program to find certain types of files, and report certain types
# of information, selectable by the user.
## CHANGELOG:
## 12/9/04 -- This is version 1. No changes at this point
###########################################################################
#!/bin/sh
# Get directory from command line. Otherwise, directory is current.
if [ $1 ] # NOTE: This form of the ‘test’ command returns
# true, as long as the value specified actually
# exists.
then
DIR=$1
else
DIR=`pwd`
fi
cd $DIR
echo “Working directory is $DIR”
# Prompt for an option from the keyboard.
echo “What information would you like to know (size, permission, owner,
group, all)?”
read OPTION
# If the option is “size”, find all “.txt” files in directory and print
# their names and file sizes.
#
# If the option is “permission”, print the permissions held by the file
#
# If the option is “owner”, print the owner of the file
#
# If the option is “group” print the group
#
# If the option is “all”, print all of the above info
if [ $OPTION = “size” ]
then
ls -la *.txt | awk ‘{print $9”: “$5}’
elif [ $OPTION = “permission” ]
then
ls -la *.txt | awk ‘{print $9”: “$1}’
elif [ $OPTION = “owner” ]
then
ls -la *.txt | awk ‘{print $9”: “$3}’
elif [ $OPTION = “group” ]
then
ls -la *.txt | awk ‘{print $9”: “$4}’
elif [ $OPTION = “all” ]
then
ls -la *.txt | awk ‘{print $9”: “$1”, “$3”, “$4”, “$5}’
```

```
else
echo “Must be size, permission, owner, group, or all.”
fi
```

Exercise 2

Write the same program using case statements instead of if statements.

Solution

```
###############################################################
# case $OPTION in
# “size”)
# ls -la *.txt | awk ‘{print $9”: “$5}’
# ;;
# “permission”)
# ls -la *.txt | awk ‘{print $9”: “$1}’
# ;;
# “owner”)
# ls -la *.txt | awk ‘{print $9”: “$3}’
# ;;
# “group”)
# ls -la *.txt | awk ‘{print $9”: “$4}’
# ;;
# “all”)
# ls -la *.txt | awk ‘{print $9”: “$1”, “$3”, “$4”, “$5}’
# ;;
# *)
# echo “Must be size, permission, owner, group, or all.”
# ;;
# esac
#######################################################################
```