Instrumente si Tehnici de Baza in Informatica

Semestrul I 2025-2026

Vlad Olaru

Curs 5 - outline

- programare shell
- scripturi
- variabile
- teste
- instructiuni conditionale
- instructiuni iterative si repetitive
- functii

Motivatie

- de multe ori vrem să avem o singură comandă pentru un sir de comenzi
- comenzile sunt preponderent comenzi shell sau din sistem, nu operatii logice sau aritmetice
- dezvoltare rapida pentru programe scurte (in general)
  - in particular, folosite extensiv de sistemele de operare la pornire/oprire (eg, pt servicii) & administrare
- portabilitate: vrem să functioneze pe si pe alte sisteme pt. automatiza operatii de configurare sau administrare
- surse: tutoriale internet (eg, CS2043 Unix and Scripting de la Cornell University)

Variante Shell

- sh(1) – Bourne shell
  - printre primele shelluri
  - disponibilă oriunde în /bin/sh
  - functionalitate redusă
  - portabilitate
- csh(1) – C shell, comenzi apropiate de modul de lucru în C
- ksh(1) – Korn shell, bazat pe sh(1), succesor csh(1)
- bash(1) – Bourne Again shell
  - cea mai răspândită
  - există și în Windows 10
  - majoritatea scripturilor moderne scrise în bash(1)
  - acest rezultat a dus la o lipsă de portabilitate
  - proiect mare cu multe linii de cod și probleme de securitate

Variabile Shell

- două tipuri de variabile: locale și de mediu
- variabilele locale există doar în instanta curentă a shell-ului
- convention: variabilele locale se notează cu litere mici
- asignare: `x=1` (fără spatii!)
- `x = 1`: se interpretează drept comanda `x` cu argumentele `= si 1`
- folosirea valorii unei variabile se foloseste prin prefixarea numelui cu `$`
- continutul oricărei variabile poate fi afisat cu comanda `echo(1)`

```
$ x=1
$ echo $x
1
```

Variabile de mediu

- folosite de sistem pentru a defini modul de functionare a programelor
- shell-ul trimite variabilele de mediu proceselor sale copil
- env(1) – afisează toate variabilele de mediu setate
- variabile importante
  - $HOME – directorul în care se țin datele utilizatorului curent
  - $PATH – listă cu directoare în care se caută executabilele
  - $SHELL – programul de shell folosit implicit
  - $EDITOR – programul de editare fisiere implicit
  - $LANG – limba implicită (ex. ro_RO.UTF-8)
- export(1) – se foloseste pentru a seta o variabilă de mediu
- printenv(1) – se foloseste pentru a afisa o variabilă de mediu

```
$ export V=2
$ printenv V
2
$ echo $V
2
```

Variabile locale vs mediu

Variabila locala

```
$ V = 2

$ echo $V

2
$ b a s h
bash$ echo $V
bash$
```

Variabila de mediu

```
$ V = 2

$echo $V

2

$export V
$ b a s h
bash$ echo $V
2
```

Variable expansion

Variabilele pot fi mai mult decât simpli scalari

- $(cmd) – evaluează întâi comanda cmd, iar rezultatul devine valoarea variabilei

```
$ echo $(pwd)
/home/student
$ x=$(find . -name \*.c)
$ echo $x
./batleft.c ./pcie.c ./maxint.c ./eisaid.c
```

- $((expr)) – evaluează întâi expresia aritmetică + side effects

```
$ x=1
$ echo $((1+1))
2
$ echo $((x+1))
2
$ echo $((x<1))
0
```

```
$ echo $((x++))
1
$ echo $((x++))
2
$ echo $((++x))
4
```

Quoting

Sirurile de caractere sunt interpretate diferit în functie de citare:

- Single quotes ‘’
  - toate caracterele îsi păstrează valoarea
  - caracterul ’ nu poate apărea în sir, nici precedat de "\"
  - exemplu:
    - `$ echo ’Am o variabila $x ’`

      `Am o variabila $x`
- Double quotes ""
  - caractere speciale $ ‘ \ (optional !)
  - restul caracterelor îsi păstrează valoarea
  - exemplu:
    - `$ echo "$USER has home in $HOME"`

      `student has home in /home/student`

Quoting (cont.)

Sirurile de caractere sunt interpretate diferit în functie de citare:

- Back quotes ` – functionează ca $()

```
$ echo "Today is  `date`"
Today is Wed May  2 18:00:08 EEST 2018
```

Structura comenzilor bash

- pipeline-uri

  $ cmd_1 | cmd_2 | … |cmd_n # executie paralela a comenzilor

  $ cmd_1 |& cmd_2 |& … |& cmd_n # "|&" e totuna cu "2>&1 |"

- liste de comenzi

  $ cmd_1; cmd_2; …; cmd_n # executie secventiala a comenzilor

  $ cmd_1 && cmd_2 && … && cmd_n # executie conditionata a comenzilor

  $ cmd_1 || cmd_2 …. || cmd_n

- variabila bash "?" contine codul de terminare (*exit status*) al ultimei comenzi executate (valoarea zero inseamna succes)

  $ *echo $?*

Obs: *?* nu e variabila de mediu !

Exemple pipeline-uri si liste de comenzi

- pipeline-uri

```
$ ls -lR | tee files.lst| wc -l
```

- liste de comenzi

```
$ ls -lR `pwd` > files.lst ; wc -l files.lst
```

- comenzi conditionate

```
$ mkdir acte && mv *.docx acte/
$ chsh 2>/dev/null && grep `whoami` /etc/passwd || echo "chsh failed!"
```

Scripting

Script = program scris pentru un mediu run-time specific care automatizează executia comenzilor ce ar putea fi executate alternativ manual de către un operator uman.

- nu necesită compilare
- executia se face direct din codul sursă
- de acea programele ce execută scriptul se numesc interpretoare în loc de compilatoare
- exemple: perl, ruby, python, sed, awk, ksh, csh, bash

Indicii de interpretare

- semnalate cu ajutorul string-ului #! pe prima linie a scriptului
  - are forma `#!/path/to/interpreter`
- exemple: `#!/bin/sh`, `#!/usr/bin/python`
- pentru portabilitate folositi `env(1)` pentru a găsi unde este instalat interpretorul
- exemple: `#!/usr/bin/env ruby`, `#!/usr/bin/env perl`
- oriunde altundeva în script # este interpretat ca început de comentariu și restul linei este ignorată (echivalent // în C)
- introdusă de Denis Ritchie circa 1979

Exemplu: hello.sh

1. Scriem fisierul `hello.sh` cu comenzile dorite

```
#!/bin/sh

# Greet the user
echo "Hello, $USER!"
```

2. Permitem executia: `chmod +x hello.sh`
3. Executăm:

```
$ ./hello.sh
Hello, student!
```

Alternativ:

```
$ sh hello.sh  # nu necesita permisiunea de executie
```

Variabile speciale în scripturi

- $1, $2, ..., ${10}, ${11}, ... – argumentele în ordinea primită
- dacă numărul argumentului are mai mult de două cifre, trebuie pus între acolade
- $0 – numele scriptului (ex. hello.sh)
- $# – numărul de argumente primite
- $* – toate argumentele scrise ca "$1 $2 . . . $n"
- $@ – toate argumentele scrise ca "$1" "$2" . . . "$n"
- $? – valoarea iesirii ultimei comenzi executate
- $$ – ID-ul procesului curent
- $! – ID-ul ultimului proces suspendat din executie
- shiftarea parametrilor pozitionali la stanga: comanda *shift*

Exemple: argumente script

- add.sh – adună două numere

```
#!/bin/sh
echo $(($1 + $2 ))
```

apel:

```
$ sh add.sh 1 2
3
```

Varianta add2.sh:

-
```
#!/bin/sh
sum=$1
shift
echo $(($sum + $1 ))
```

Exemple: argumente script

- tolower.sh – imită functia din C tolower(3)

```
#!/bin/sh
tr '[A-Z]' '[a-z]' < $1 > $2
```

apel:

```
$ echo "WHERE ARE YOU?" > screaming.txt
$ ./tolower.sh screaming.txt decent.txt
$ cat decent.txt
where are you?
```

Blocuri de control

Pentru scripturi mai complexe avem nevoie, ca în orice limbaj, de blocuri de control

- conditionale – `if`, `test [ ]`, case
- iterative – `for`, while, `until`
- comparative – -ne, `-lt`, `-gt`
- functii – `function`
- iesiri – break, continue, `return`, `exit`

If

- cuvinte cheie: *if, then, elif, else, fi*
- ex:

  if test-cmd

  then

  cmds

  elif test-cmd

  then

  cmds

  else

  cmds

  fi
- rezultatul *test-cmd* apare in $? (0 –> true, !=0 ->false)

Exemplu: if

Caută în fisier date și le adaugă dacă nu le găseste

```
#!/bin/sh
if grep "$1" $2 > /dev/null
then
    echo "$1 found in file $2"
else
    echo "$1 not found in file $2, appending"
    echo "$1" >> $2

fi
```

apel:

```
$ ./text.sh  who decent.txt
who not found in file decent.txt, appending
$ cat decent.txt
where are you?
who
```

test sau [ ]

- nu dorim să testăm tot timpul rezultatul executiei unei comenzi
- există expresii pentru a compara sau verifica variabile
- `test expr` – evaluează valoarea de adevăr a lui `expr`
- `[ expr ]` – efectuează aceiasi operatie (**atentie la spatii!**)
  - [este comanda interna

    ```
    $ type [
    [ is a shell builtin
    ```
- expresiile pot fi legate logic prin
  - `[ expr1 -a expr2 ]` – conjunctie
  - `[ expr1 -o expr2 ]` – disjunctie
  - `[ ! expr ]` – negatie

Expresii test: numere

- [n1 -eq n2] – $n_1 = n_2$
- [n1 -ne n2] – $n_1 \neq n_2$
- [n1 -ge n2] – $n_1 \geq n_2$
- [n1 -gt n2] – $n_1 > n_2$
- [n1 -le n2] – $n_1 \leq n_2$
- [n1 -lt n2] – $n_1 < n_2$

Expresii test: siruri de caractere

- `[ str ]` – `str` are lungime diferită de 0
- `[ -n str ]` – `str` nu e gol
- `[ -z str ]` – `str` e gol ("")
- `[ str1 = str2 ]` – stringuri identice
- `[ str1 == str2 ]` – stringuri identice
- `[ str1 != str2 ]` – stringuri diferite

Expresii test: fisiere

- -e path – verifică dacă există calea path
- -f path – verifică dacă path este un fisier obisnuit
- -d path – verifică dacă path este un director
- -r path – verifică dacă aveti permisiunea de a citi path
- -w path – verifică dacă aveti permisiunea de a scrie path
- -x path – verifică dacă aveti permisiunea de a executa path

while

- execută blocul cât timp comanda *cmd* se execută cu succes

while cmd
do
    cmd_1
    cmd_2
done

- în loc de comanda *cmd* putem avea o expresie de `test`
- într-o singură linie: while cmd; do cmd_1; cmd_2; done

Exemplu: while

Afisează toate numerele de la 1 la 10:

```
i=1
while [$i -le 10 ]do
      echo "$i"
      i=$(($i+1))
done
```

Sau într-o singură linie:

```
i=1; while [ $i -le 10 ]; do echo "$i"; i=$(($i+1)); done
```

until

- execută blocul cât timp comanda *cmd* se execută **fără** succes
- *until* cmd
  do
    cmd_1
    cmd_2
  done
- în loc de comandă putem avea o expresie de `test`
- într-o singură linie: `until` cmd; do cmd_1 ; cmd_2; done

for

- execută blocul pentru fiecare valoare din listă

  for var in str_1 str_2 ... str_N
  do
  cmd_1
  cmd_2
  …
  done

- var ia pe rând valoarea str_1, pe urmă str_2 până la str_N
- de regulă comenzile din bloc (cmd_1, cmd_2) sunt legate de *var*
- comanda for are mai multe forme, aceasta este cea mai întâlnită
- într-o singură linie:
  for var in str_1 str_2 ... str_N; do cmd_1 ; cmd_2; done

Exemplu: for

Compilează toate fisierele C din directorul curent

```
for f in *.c
do
    echo "$f"
    cc $f -o $f.out
done
```

Sau într-o singură linie:

```
for f in *.c; do echo "$f"; cc $f -o $f.out; done
```

for traditional

- formă întâlnite doar în unele shell-uri, **nu este portabil**
- execută blocul urmând o sintaxă similară C

```
for (( i=1; i<=10; i++ ))
do
    echo $i
done
```

- i ia pe rând toate valorile între 1 și 10
- în exemplu, blocul afisează $i, dar pot apărea oricâte alte comenzi
- într-o singură linie:

for ((i=1; i<=10; i++)); do cmd_1; cmd_2; done

Case

- se comportă similar cu switch în C

```
case var in
        pattern1 )
                cmds
                ;;
        pattern2 )
                cmds
                ;;
        * )
                defaultcmd
                ;;
esac
```

- pattern – string sau orice expresie de shell (similar cu wildcards-urile folosite pentru grep(1))

Exemplu: case

```
echo “Greetings !”
read input_string
case $input_string in
        hi)
                echo “A good day to you too !”
                ;;
        salut)
                echo “Buna ziua !”
                echo “Ou, peut-etre, bonjour a vous aussi ? “
                ;;
        moin)
                echo “Guten morgen !”
                ;;
        *)
                echo “Sorry, I don’t understand !”
                ;;
esac
```

Functii

- intorc valori:
  - schimband valoarea variabilelor
  - folosind comanda *exit* pentru a termina scriptul
  - folosind comanda *return* pentru a termina functia
  - tiparind rezultate la *stdout*
    - recuperate de apelant in maniera tipica shell-urilor

    e.g., var=`expr …`
- nu pot modifica parametrii de apel, dar pot modifica parametrii globali

Exemplu functie

```
#!/bin/sh

add_host()
{
      IP_ADDR=$1
      HOSTNAME=$2
      shift; shift;
      ALIASES=$@
      echo “$IP_ADDR \t $HOSTNAME \t $ALIASES” >> $hostfile
}

# aici incepe scriptul
hostfile=$1
echo “Start script”
add_host 80.96.21.88 fmi.unibuc.ro fmi
add_host 80.96.21.209 www.unibuc.ro www
```

Obs: *add_host* parsata de shell si verificata sintaxa, dar executata doar la apelare

Scope-ul variabilelor

- cu exceptia parametrilor, nu exista scope pt variabile

e.g. *scope.sh*

```
#!/bin/sh
somefn()
{
        echo "Function call parameters are $@"
        a=10
}
echo "script arguments are $@"
a=11
echo "a is $a"
somefn first string next
echo "a is $a"
```

Apel:

```
$ ./scope.sh 1 2 3
script arguments are 1 2 3
a is 11
Function call parameters are first string next
a is 10
```

Recursivitate

- factorial.sh:

```
#!/bin/sh
factorial()
{
        if [ “$1” –gt “1” ]; then
            prev=`expr $1 – 1`
            rec=`factorial $prev`
            val=`expr $1 \* $rec`
            echo $val
        else
            echo 1
        fi
}
echo –n “Input some number: ”
read n
factorial $n
```

Biblioteci de functii

- colectie de functii grupate intr-un fisier CARE NU INCEPE cu linia speciala introdusa de “#!” !
- se pot defini variabile globale
- “apelul” propriu-zis se face folosind comanda *source*

e.g., *renamelib.sh:*

```
MSG=“Renaming files …”
rename()
{
      mv $1 $2
}
```

Apelul functiei de biblioteca in scriptul *libcall.sh:*

```
#!/bin/sh
. ./renamelib.sh          # echivalent cu “source ./renamelib.sh”
echo $MSG
rename $1 $2
```

Coduri de retur

```
#!/bin/sh
add_host()
{
	if [ “$#”  -eq  “0” ]; then
		return 1
	fi
	IP_ADDR=$1
	HOSTNAME=$2
	shift; shift;
	ALIASES=$@
	echo “$IP_ADDR \t $HOSTNAME \t $ALIASES” >> $hostfile
}
hostfile=$1
echo “Start script”
add_host
RET_CODE=$?
if [ “$RET_CODE” –eq “1” ]; then
	echo “No arguments to the add_host call !”
fi
```

read

- citeste una sau mai multe variabile din `stdin`
- sintaxă: `read` var_1 var_2 `...` var_N

```
$ read x y
1 2
$ echo $x $y
1 2
```

- fara nici o variabilă, pune tot rezultatul în `$REPLY`

```
$ read

hello
$ echo $REPLY
hello
```

- citire line cu linie dintr-un fisier:

```
cat foo.txt | while read LINE; do echo $LINE; done
```

Depanare

Pentru a depana un script apelati-l cu shell-ul folosit si optiunea -x.
Comenzile executate apar pe ecran prefixate cu + .

```
$ sh -x tolower.sh screaming.txt decent.txt
+ tr [A-Z] [a-z]
+ < screaming.txt
+ > decent.txt
```