Programare în Shell

Utilizarea Sistemelor de Operare

Paul Irofti

Universitatea din București
Facultatea de Matematică și Informatică
Department de Informatică
Email: paul.irofti@fmi.unibuc.ro

Motivație

- de multe ori vrem să avem o singură comandă pentru un șir de comenzi
- comenzile sunt preponderent comenzi shell sau din sistem, nu operații logice sau aritmetice
- avem nevoie să scriem un program scurt rapid
- vrem să funcționeze pe orice alt sistem pentru a repeta un set de operații de configurare sau administrare
- bazat pe CS2043 Unix and Scripting de la Cornell University (cursurile 9 și 10)

Variante Shell

- `sh(1)` – Bourne shell
  - printre primele shelluri
  - disponibilă oriunde în `/bin/sh`
  - funcționalitate redusă
  - portabilitate
- `csh(1)` – C shell, comenzi apropiate 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 instanța curentă a shell-ului
- convenție: variabilele locale se notează cu litere mici
- asignare: `x=1` (fără spații!)
- `x = 1`: se interpretează drept comanda `x` cu argumentele `=` și `1`
- folosirea valorii unei variabile se folosește prin prefixarea numelui cu `$`
- conținutul oricărei variabile poate fi afișat cu comanda `echo(1)`

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

Variabile de mediu

- folosite de sistem pentru a defini modul de funcționare a programelor
- shell-ul trimite variabilele de mediu proceselor sale copil
- `env(1)` – afișează 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 fișiere implicit
  - `$LANG` – limba implicită (ex. `ro_RO.UTF-8`)
- `export(1)` – se folosește pentru a seta o variabilă de mediu
- `printenv(1)` – se folosește pentru a afișa o variabilă de mediu

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

Exemplu: Variabilă locală

```
$ LANG=ro_RO.UTF-8
$ printenv LANG
$ echo $LANG
ro_RO.UTF-8
$ bash
bash-4.4$ printenv LANG
bash-4.4$ t
bash: t: command not found
bash-4.4$ echo $LANG

bash-4.4$ exit
```

Exemplu: Variabilă de mediu

```
$ export LANG=ro_RO.UTF-8
$ printenv LANG
ro_RO.UTF-8
$ bash
bash-4.4$ printenv LANG
ro_RO.UTF-8
bash-4.4$ echo $LANG
ro_RO.UTF-8
bash-4.4$ t
bash: t: comandă negăsită
```

Variable expansion

Variabilele pot fi mai mult decât simplii scalari

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

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

- `$((expr))` – evaluează întâi expresia aritmetică, iar rezultatul devine valoarea variabilei

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

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

Quoting

Șirurile de caractere sunt interpretate diferit în funcție de citare:

- Single quotes ''
  - toate caracterele își păstrează valoarea
  - caracterul ' nu poate apărea în șir, nici precedat de "\"
  - exemplu:

```
$ echo 'Am variabila $x'
Am variabila $x
```

- Double quotes ""
  - caractere speciale $ ` \ (opțional !)
  - restul caracterelor își păstrează valoarea
  - exemplu:

```
$ echo "$USER has home in $HOME"
paul has home in /home/paul
```

- Back quotes ` – funcționează ca $()

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

Înlănțuirea comenzilor

- `cmd1; cmd2` – înlănțuire secvențială, `cmd2` imediat după `cmd1`
- `cmd1 | filtru | cmd2` – ieșirea comenzii din stânga este intrarea celei din dreapta operatorului `|`
- `cmd1 && cmd2` – execută a doua comandă doar dacă prima s-a executat cu succes
- `cmd1 || cmd2` – execută a doua comandă doar dacă prima a eșuat
- exemplu:

```
$ mkdir acte && mv *.docx acte/
$ ls -lR | tee files.lst | wc -l
$ ssh example.org || echo "Connection failed!"
```

Scripting

Definition

Script-urile sunt programe scrise pentru un mediu run-time specific care automatizează execuția comenzilor ce ar putea fi executate alternativ manual de către un operator uman.

- nu necesită compilare
- execuția se face direct din codul sursă
- de acea programele ce execută scriptul se numesc interpretatoare înloc de compilatoare
- comenzile executate se mai numesc și byte-code
- astăzi granița dintre compilatoare și interpretatoare nu mai este atât de clară (JIT, tiered compilation)
- exemple: perl, ruby, python, sed, awk, ksh, csh, bash

Shebang (#!)

- reprezintă prima linie din orice script Unix
- are forma `#!/path/to/interpreter`
- exemple: `#!/bin/sh`, `#!/usr/bin/python`
- pentru protabilitate folosiți `env(1)` pentru a găsi unde este instalat interpretatorul
- 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 fișierul `hello.sh` cu comenzile dorite

```
#!/bin/sh

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

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

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

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 ieșirii ultimei comenzi executate
- `$$` – ID-ul procesului curent
- `$!` – ID-ul ultimului proces suspendat din execuție

Exemple: argumente script

- `add.sh` – adună două numere

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

apel:

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

- `tolower.sh` – imită funcția 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

- condiționale – `if`, `test [ ]`, `case`
- iterative – `for`, `while`, `until`
- comparative – `-ne`, `-lt`, `-gt`
- funcții – `function`
- ieșiri – `break`, `continue`, `return`, `exit`

If

- cuvinte cheie: `if`, `then`, `elif`, `else`, `fi`
- exemplu ce folosește toate cuvintele cheie:

```
if    cmd1
then
      cmd2
      cmd3
elif  cmd4
then
      cmd5
      cmd6
else
      cmd7
fi
```

- o condiție este adevărată dacă comanda asociată este executată cu succes ($?=0)

Exemplu: `if`

Caută în fișier date și le adaugă dacă nu le găsește

```
#!/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 execuția unei comenzi
- există expresii pentru a compara sau verifica variabile
- `test expr` – evaluează valoarea de adevăr a lui `expr`
- `[ expr ]` – efectuează aceiași operație (**atenție la spații!**)
- expresiile pot fi legate logic prin
  - `[ expr1 -a expr2 ]` – și
  - `[ expr1 -o expr2 ]` – sau
  - `[ ! expr ]` – negație

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`: șiruri 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`: fișiere

- `-e path` – verifică dacă există calea `path`
- `-f path` – verifică dacă `path` este un fișier
- `-d path` – verifică dacă `path` este un director
- `-r path` – verifică dacă aveți permisiunea de a citi `path`
- `-w path` – verifică dacă aveți permisiunea de a scrie `path`
- `-x path` – verifică dacă aveți permisiunea de a executa `path`

while

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

```
while cmd
do
    cmd1
    cmd2
done
```

- în loc de comandă putem avea o expresie de `test`
- într-o singură linie: `while cmd; do cmd1 cmd2; done`

Exemplu: `while`

Afișează 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
    cmd1
    cmd2
done
```

- în loc de comandă putem avea o expresie de `test`
- într-o singură linie: `until cmd; do cmd1 cmd2; done`

for

- execută blocul pentru fiecare valoare din listă

```
for var in str1 str2 ... strN
do
    cmd1
    cmd2
done
```

- `var` ia pe rând valoarea `str1`, pe urmă `str2` până la `strN`
- de regulă comenzile din bloc (`cmd1`, `cmd2`) sunt legate de `var`
- comanda `for` are mai multe forme, aceasta este cea mai întâlnită
- într-o singură linie:
  `for var in str1 str2 ... strN; do cmd1 cmd2; done`

Exemplu: `for`

Compilează toate fișierele 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` tradițional

- 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 afișează `$i`, dar pot apărea oricâte alte comenzi
- într-o singură linie:
  `for (( i=1; i<=10; i++ )); do cmd1 cmd2; done`

Case

- se comportă similar cu `switch` în C

```
case var in
pattern1 )
    cmd1
    cmd2
    ;;
pattern2 )
    cmd3
    ;;
* )
    defaultcmd
    ;;
esac
```

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

Exemplu: case

Parsare subcomandă pentru un program SQL:

```
case "$1" in
'CREATE')
  do_creat
  ;;
'INSERT')
  do_insert
  ;;
'SELECT')
  do_select
  ;;
'USE')
  do_use
  ;;
'DROP')
  do_drop
  ;;
*)
  print_usage
esac
```

read

- citește una sau mai multe variabile din `stdin`
- sintaxă: `read var1 var2 ... varN`

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

- dacă nu este dată nici o variabilă, se așteaptă să fie una singură și pune rezultatul în `$REPLY`

```
$ read
hello
$ echo $REPLY
hello
```

- citire line cu linie dintr-un fișier:
  `cat foo.txt | while read LINE; do echo $LINE done`

Depanare

Pentru a depana un script apelați-l cu shell-ul folosit și opțiunea `-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
```