Laboratorul 4

Programarea shell
Shell scripting

Shell-ul poate fi folosit şi pentru a interpreta conţinutul unor fişiere care conţin comenzi şi a le executa. Aceste fişiere de comenzi poartă numele de scripturi. Ele pot fi văzute ca programe executabile în format text. Pentru acest lucru, este necesar ca fişierul script, `myscript` în exemplul de mai jos, să aibă permisiuni de execuţie setate cel puţin pentru proprietarul fişierului:

```
$ chmod u+x myscript
```

Ulterior adăugării permisiunilor de execuţie, fişierul script se poate executa ca orice fişier executabil compilat, de ex., considerând fişierul `myscript` de mai sus:

```
$ ./myscript
```

În general, pentru a preciza foarte exact pentru ce tip de shell (sau de interpretor, în sensul cel mai general) este potrivit scriptul, în mod uzual prima linie din fişierul script are o sintaxă specială prin care specifică programul care va interpreta scriptul. Această primă linie instruieşte shell-ul ce interpretor de comenzi să lanseze în execuţie pentru a interpreta şi executa comenzile din script. De exemplu, un shell `bash` poate lansa în execuţie un fişier script căruia i s-au dat drepturi de execuţie ca mai sus şi care conţine următoarele comenzi:

```
#!/bin/bash

echo "Hello scripting world!"
exit 0
```

Prima linie este un comentariu (începe cu caracterul #) special (imediat după # urmează !) şi care specifică interpretorul de comenzi (`/bin/bash`) care este folosit pentru a interpreta şi executa comenzile care urmează. Ultima linie care apelează comanda `exit` cu parametrul 0 evidenţiază o bună practică în programarea Unix în general (şi shell scripting-ul nu face excepţie) prin care se comunică procesului părinte (shell-ul care a lansat în execuţie scriptul de mai sus în cazul nostru) codul de terminare al programului (al scriptului în cazul de mai sus). Convenţia Unix spune că orice program care se termină fără eroare întoarce un cod 0. Orice altă valoare de retur reprezintă *a priori* o terminare eronată a programului.

În cazul cel mai general, prima linie din script poate desemna orice tip de interpretor care, desigur, va fi folosit pentru a interpreta comenzile care urmează în script. Iată un script `awk`, care poate fi lansat şi el în execuţie de către orice alt interpretor de comenzi, inclusiv

bash, şi care tipăreşte argumentele primite de script în linia de comandă:

```
#!/usr/bin/awk -f

BEGIN {
  for (i = 0; i < ARGC; i++)
    printf "ARGV[%d] = %s\n", i, ARGV[i]
  exit
}
```

După cum se observă, awk este practic un limbaj de programare cu structură apropiată de limbajele de programare compilate.

De fapt, toate interpretoarele de comenzi au şi propriul limbaj de programare, cu propria sintaxă şi propriile comenzi. În cele ce urmează, vor fi prezentate principalele "instrucţiuni" pe care le înţelege Bourne Again Shell: loop-uri (while/for), instrucţiuni condiţionale (if/case) şi funcţii.

1 Test

Operaţia de test uzual folosită în instrucţiuni condiţionale sau iterative este implementată de bash cu ajutorul comenzii interne test sau [. Pentru a verifica că [este de fapt o comandă puteţi folosi comanda internă type:

```
$ type test
test is a shell builtin
$ type [
[ is a shell builtin
$ type type
type is a shell builtin
$
```

Fiind un program ca oricare altul, [are nevoie de spaţii în jurul său. De pildă, expresia [$myvar = "somestring"] va genera o eroare de sintaxă fiind interpretată drept test$myvar = "somestring"]. Versiunea corectă a comenzii anterioare este [$myvar = "somestring"].

Comanda test este extrem de complexă şi poate testa stringuri, numere, fişiere. Pentru o imagine comprehensivă, consultaţi pagina de manual. În cele ce urmează vom evidenţia câteva dintre utilizările des întîlnite ale comenzii, apêlând la comenzile condiţionale puse la dispoziţie de către shell. În secţiunile următoare vom vedea cum poate fi folosită comanda test împreună cu instrucţiuni condiţionale şi iterative.

Pentru exemplele care urmează vom folosi o variabilă shell X căreia i se vor asigna valori diverse şi o vom folosi împreună cu comanda test. Pentru lizibilitate vom folosi caracterul \ care permite extensia unei singure linii de comandă pe linia următoare (atât interactiv, la promptul shell-ului, cât şi în scripturi). În mod interactiv, utilizarea \ implică automat apariţia promptului de continuare ">" stocat în variabila de mediu PS2.

```
$ X=4
$ [ "$X" -lt "0" ] && echo "X is less than zero" \
> || echo "X is greater than zero"
```

```
X is greater than zero

$ X=-1
$ [ "$X" -lt "0" ] && echo "X is less than zero" \
> || echo "X is greater than zero"
X is less than zero

$ X=0
$ [ "$X" = "0" ] && \
> echo "X is the string or number \"0\""
X is the string or number "0"

$ X=hello
$ [ "$X" = "hello" ] && \
> echo "X matches the string \"hello\""
X matches the string "hello"

$ X="not hello"
$ [ "$X" != "hello" ] && echo "X is not the string \"hello\""
X is not the string "hello"

$ echo $X
not hello
$ [ -n "$X" ] && echo "X is of nonzero length"
X is of nonzero length

$ X=somefile
$ [ -f "$X" ] && \
> echo "X is the path of a real file" || \
> echo "No such file: $X"
No such file: somefile

$ X=/etc/passwd
$ [ -f "$X" ] && \
> echo "X is the path of a real file" || \
> echo "No such file: $X"
X is the path of a real file

$ X=/bin/ls
$ [ -x "$X" ] && echo "X is the path of an executable file"
X is the path of an executable file

$ X=.
$ [ "$X" -nt "/etc/passwd" ] && \
> echo "X is a file which is newer than /etc/passwd"
X is a file which is newer than /etc/passwd
$
```

```
$ X=hello
$ [ -n "$X" ] && echo "$X is a non-zero string" \
> || echo "$X is the null string"
hello is a non-zero string

$ Y=bye
$ [ $X != $Y ] && echo "$X is not equal to $Y" \
> || echo "$X is equal to $Y"
hello is not equal to bye
$
```

Expresiile supuse comenzii `test` pot fi compuse cu ajutorul operatorilor logici: negaţie, conjuncţie şi disjuncţie.

```
$ echo $X
hello
$ [ ! "$X" = "" ] && echo "$X is not an empty string" \
> || echo "$X is the empty string"
hello is not an empty string

$ echo $X $Y
hello bye
$ [ "$X" = "hello" -a "$Y" = "bye" ] && \
> echo "Proper greetings have been made" \
> || echo "This is not quite polite"
Proper greetings have been made

$ Y=notbye
$ echo $X $Y
hello notbye
$ [ "$X" != "hello" -o "$Y" != "bye" ] && \
> echo "This is not quite polite" \
> || echo "Politeness has been satisfied"
This is not quite polite
$
```

2 Instrucţiunile de tip If şi Case

Instrucţiunile condiţionale de tip `if` din shell arată în felul următor:

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Observaţi poziţia cuvintelor cheie `then`, `else` şi respectiv `fi`. Ele trebuie să apară singure pe o linie chiar la începutul ei. Pentru a face economie de spaţiu, se poate folosi caracterul `;` care permite unirea a două linii consecutive pe aceeaşi linie:

```
if [ ... ]; then
  # if-code
else
  # else-code
fi
```

Mai multe `if`-uri se pot îmbrica folosind instrucţiunea `elif`, ca în scriptul următor, pe care îl puteţi salva în fişierul `if.sh` (atât pentru acest script cât şi pentru toate cele care urmează, nu uitaţi să setaţi permisiunea de execuţie înainte de a rula scriptul):

```
#!/bin/bash

read filename
if [ ! -e "$filename" ]; then
  echo "$filename does not exist"
elif [ -f "$filename" ]; then
  echo "$filename is a regular file"
else
  echo "$filename exists, but we don't know what kind of file it is"
fi
```

Rularea scriptului evidenţiază funcţionalitatea instrucţiunii `if`:

```
$ ./if.sh
somefile
somefile does not exist
$ ./if.sh
/etc/passwd
/etc/passwd is a regular file
$ ./if.sh
/etc
/etc exists, but we don't know what kind of file it is
$
```

Pentru a face scriptul de mai sus ceva mai complet se poate folosi instrucţiunea `case`, ca în scriptul de mai jos pe care îl vom denumi `case.sh`:

```
#!/bin/bash

echo -n "Please input a filename: "
read filename
longformat=`ls -ld $filename 2>/dev/null`
case "${longformat:0:1}" in
  -)
    echo "$filename is a regular file"
    ;;
  d)
    echo "$filename is a directory"
    ;;
  b)
    echo "$filename is a block file"
```

```
    ;;
  c)
    echo "$filename is a character file"
    ;;
  *)
    echo "Sorry, I don't know anything about $filename"
    ;;
esac
```

Rememoraţi din laboratoarele anterioare că orice comandă care apare între *backquotes* este înlocuită cu un string care reprezintă rezultatul execuţiei comenzii dintre *backquotes*. Redirecţionarea `stderr` reprezentată prin descriptorul de fişier 2 este necesară pentru a trata situaţia în care numele fişierului introdus de la tastatură nu corespunde unui fişier existent. `/dev/null` este un fişier de tip caracter cu un comportament special în sistem: orice scriere la `/dev/null` se pierde, iar de la `/dev/null` nu se poate citi nimic. Pe cale de consecinţă, redirecţionarea `stderr` la `/dev/null` are ca efect eliminarea mesajului de eroare generat de comanda `ls -l` atunci cînd numele de fişier citit de la tastatură nu corespunde unui fişier existent în sistem.

Aşa cum am menţionat la curs, listarea în format lung a atributelor unui fişier are ca efect, printre altele, tipărirea pe ecran ca prim caracter a tipului fişierului. Pentru a izola acest caracter în cadrul stringului generat de comanda `ls -l` se foloseşte sintaxa `${nume_variabila:offset:length}` care permite obţinerea unui substring al variabilei `nume_variabila` care începe la poziţia `offset` şi are lungimea `length`.

Rezultatele rularii scriptului de mai sus arată de manieră următoare:

```
$ ./case.sh
Please input a filename: /etc
/etc is a directory
$ ./case.sh
Please input a filename: /etc/passwd
/etc/passwd is a regular file
$ ./case.sh
Please input a filename: /dev/tty1
/dev/tty1 is a character file
$ ./case.sh
Please input a filename: /dev/sda1
/dev/sda1 is a block file
$ ./case.sh
Please input a filename: somefile
Sorry, I don't know anything about somefile
$
```

3 Instrucţiuni iterative

Limbajele interpretate oferă şi funcţionalitate de tipul instrucţiunilor iterative, a instrucţiunilor executate repetitiv într-o buclă. Principalele construcţii sintactice pe care le oferă `bash` pentru implementarea buclelor sunt instrucţiunile `for` şi `while`.

3.1 For loops

Instrucţiunile de tip `for` iterează printr-o listă de valori. Executaţi următorul script pe care-îl salvaţi într-un fişier `myfor-loop.sh`:

```
$ cat > myfor-loop.sh
#!/bin/bash
for i in 1 2 3 4 5
do
  echo "Iteratia cu numarul $i"
done
$ chmod u+x myfor-loop.sh
$ ./myfor-loop.sh
Iteratia cu numarul 1
Iteratia cu numarul 2
Iteratia cu numarul 3
Iteratia cu numarul 4
Iteratia cu numarul 5
$
```

Observaţi că `do` şi `done`, cuvinte cheie ale instrucţiunii `for`, trebuie să apară pe o linie separată chiar la începutul ei. Dacă modificaţi scriptul şi mutaţi cuvîntul cheie `do` pe aceeaşi linie cu `for`, la execuţia scriptului veţi obţine o eroare de sintaxă. Puteţi repara această eroare folosind caracterul special `;` care uneşte două linii:

```
#!/bin/bash
for i in 1 2 3 4 5; do
  echo "Iteratia cu numarul $i"
done
```

Valorile din listă pot fi practic orice. De pildă, încercaţi să executaţi următorul script, pe care-îl puteţi salva în fişierul `myfor-loop2.sh`:

```
#!/bin/bash
for i in * 1.5 \* 2
do
  echo "Valoarea lui i este $i"
done
```

Ca să înţelegeţi comportamentul instrucţiunii `for` în acest caz este important să vă reamintiţi de interpretarea *wildcard*-urilor în `bash` prezentată la curs. Încercaţi să rulaţi scriptul de mai sus cu şi fără `*`. Dar dacă folosiţi *escape* character `\*`? Şi dacă folosiţi `\*`, ce se întâmplă dacă în corpul instrucţiunii `for` afişaţi valoarea variabilei `i` fără să o includeţi în ghilimele (ca mai jos)?

```
echo Valoarea lui i este $i
```

3.2 While loops

Instrucţiunea `while` este cel mai adesea folosită împreună cu comanda `test`, ca în exemplul de mai jos pe care îl puteţi salva în fişierul `while.sh`:

```
#!/bin/bash
```

```
while [ "${INPUT_STRING:-hello}" != "bye" ]
do
  echo "Introduceti date (bye pentru a iesi din bucla)"
  read INPUT_STRING
  echo "Ati introdus: $INPUT_STRING"
done
```

Scriptul de mai sus citeşte şiruri de caractere de la tastatură în buclă până când se tipăreşte `"bye"`. Ce se întâmplă dacă variabila `INPUT_STRING` nu are valoare iniţială?

Scriptul de mai sus se poate modifica în următoarea variantă (`while2.sh`) care nu foloseşte comanda `test` ci caracterul `:` (echivalent cu comanda `true`) care întoarce permanent valoarea de adevăr:

```
#!/bin/bash
while :
do
  echo "Introduceti date (Ctrl-C pentru a iesi din bucla)"
  read INPUT_STRING
  echo "Ati introdus: $INPUT_STRING"
done
```

De asemenea, instrucţiunea `while` se foloseşte des împreună cu `read`. Ca exemplu aveţi scriptul de mai jos (`while3.sh`), care modifică scriptul anterior `case.sh` pentru a permite introducerea de nume de fişiere în mod repetitiv:

```
#!/bin/bash

echo -n "Please input a filename: "
while read filename
do
  longformat=‘ls -ld $filename 2>/dev/null‘
  case "${longformat:0:1}" in
    -)
      echo "$filename is a regular file"
      ;;
    d)
      echo "$filename is a directory"
      ;;
    b)
      echo "$filename is a block file"
      ;;
    c)
      echo "$filename is a character file"
      ;;
    *)
      echo "Sorry, I don’t know anything about $filename"
      ;;
  esac
  echo -n "Please input a filename: "
done
```

Puteţi încheia introducerea datelor cu `Ctrl-d` sau să terminaţi programul cu `Ctrl-c`.

4 Mai multe despre variabile

Pe lângă variabilele shell-ului (variabile interne şi variabile de mediu) există un set de variabile speciale care, de cele mai multe ori, nu pot fi modificate.

Primul set de variabile de interes este cel al variabilelor `0`, `1`, ... `9`. Variabila `$0`, de pildă, reprezintă numele programului, numit în general *basename*, în vreme ce restul variabilelor până la 9, `$1`, `$2`, ..., `$9` reprezintă parametrii cu care a fost apelat scriptul. Variabila `@` reprezintă aceşti parametri, al căror număr exact este stocat în variabila `#`. Pentru înţelegerea mai bună a acestor aspecte, rulaţi următorul script `param.sh`:

```
#!/bin/sh

echo "I was called with $# parameters"
echo "My name is $0"
echo "My nicer name is ‘basename $0‘"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

Comanda externă `basename` este folosită pentru a elimina calea din numele scriptului. Iată aici rezultatele unei posibile rulări:

```
$ ./param.sh
I was called with 0 parameters
My name is ./param.sh
My nicer name is param.sh
My first parameter is
My second parameter is
All parameters are
$ ./param.sh first second third fourth
I was called with 4 parameters
My name is ./param.sh
My nicer name is param.sh
My first parameter is first
My second parameter is second
All parameters are first second third fourth
$
```

Cu ajutorul comenzii `shift` se pot folosi mai mult de 9 parametri în linia de comandă. Scriptul următor, `param2.sh`, evidenţiază funcţia comenzii `shift` care, atunci cînd e apelată, iterează printre parametrii de apel ai scriptului (*"shiftează"* parametrii la stânga). Pe măsură ce se shiftează parametrii, numărul lor, conţinut în variabila `$#`, scade.

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
```

```
done
```

Rezultatul execuţiei comenzii demonstrează felul în care se pot accesa toţi parametrii de apel, indiferent de numărul lor:

```
$ ./param2.sh 1 2 3 4
$1 (first parameter) is 1
$1 (first parameter) is 2
$1 (first parameter) is 3
$1 (first parameter) is 4
$
```

În final, aşa cum am discutat la curs şi în laboratoarele anterioare, variabila `?` conţine valoarea codului de retur cu care s-a încheiat comanda anterioară. Valoarea acestui cod de retur se poate folosi pentru a notifica utilizatorul în privinţa rezultatului execuţiei unei comenzi ca în scriptul de mai jos (`retur.sh`):

```
#!/bin/sh

read cmd
eval "$cmd" 2>/dev/null
if [ "$?" -ne "0" ]; then
  echo "$cmd has failed!"
fi
```

Comanda internă a shell-ului `eval` evaluează stringul furnizat ca argument drept comandă şi întoarce codul de return al acestei comenzi. Iată câteva exemple de rulare:

```
$ ./retur.sh
somecmd
somecmd has failed!
$ ./retur.sh
[ "0" -gt "1" ]
[ "0" -gt "1" ] has failed!
$
```

5 Sarcini de laborator

1. Executaţi toate comenzile prezentate în acest laborator.
2. Modificaţi unele dintre scripturile de la laborator pentru a primi datele necesare ca parametri în linie de comandă, în loc să fie citite de la tastatură (de ex., scripturile care identifică tipuri de fişiere).
3. Scrieţi propria versiune a comenzii `ls` fără nici un parametru (adică pentru a lista conţinutul directorului curent).
4. Scrieţi un script care foloseşte o comandă de tip *pipeline* pentru a afişa utilizatorii şi PID-urile proceselor lor aşa cum sunt afişate de comanda `ps auxw`. Scriptul trebuie să captureze într-un pipe outputul comenzii `ps auxw` şi să itereze prin fiecare linie tipărind primele două câmpuri ale outputului, cele care corespund utilizatorului şi PID-ului.

5. Scrieţi un script `myfind` care emulează comportamentul simplificat al comenzii `find` cu flagurile `-name`, `-type` şi `-exec`. Comanda primeşte ca prim parametru un director şi nu funcţionează recursiv (i.e., `maxdepth = 1`). Trebuie să fie capabilă să găsească un fişier după nume şi/sau tip şi, odată identificat, să poată executa o comandă asupra lui. Numele, tipul şi comanda de execuţie sunt furnizate ca parametri în linia de comandă, exact ca pentru comanda `find` (`man find`).