Elemente de programare în Shell-ul Unix

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

noiembrie 2000

Subiect: programare în Bourne Shell

Cunoştinţe necesare: cunoştinţe elementare de programare; familiaritate cu Unix

Cuvinte cheie: shell, proces, variabilă, comandă, script

Cuprins

Aveţi adesea de manipulat multe fişiere? Procesaţi frecvent texte? Sunteţi un administrator de sistem? Dacă intraţi într-una din aceste categorii, atunci scula software pe care o voi descrie în continuare se poate dovedi exact obiectul de care aveţi nevoie.

În acest articol voi discuta despre "shell", într-una din înfăţişările pe care le are în sistemul de operare Unix. Am mai scris articole despre shell în PC Report; le recomand cititorilor cărora articolul de faţă le stîrneşte interesul să se uite şi peste cele mai vechi; un articol în PC Report din iunie 1997 discută despre cum este implementat un shell. Articolul meu din decembrie 1996 discută despre sistemele de operare şi menţionează în treacăt şi rolul shell-ului. Dacă între timp aţi făcut curat în bibliotecă şi nu mai aveţi numerele vechi, puteţi găsi on-line articolele scrise de mine în pagina mea de web.

Chiar dacă 95% din umanitate foloseşte sistemul de operare Windows, de data aceasta nu mai pot fi acuzat că ignor marea audienţă cu bună intenţie: sistemele create de Microsoft oferă doar facilităţi rudimentare în această privinţă. Există într-adevăr mai multe pachete software pentru Windows oferite de terţi care implementează funcţionalităţi de shell, dar shell-ul standard rămîne cel DOS.

1 Shell-ul: un limbaj de programare

Dacă nu vreţi să vă osteniţi prea tare să căutaţi articolele mele mai vechi, voi rezuma aici cîteva din informaţiile mai importante de care avem nevoie.

În primul rînd, "shell" înseamnă cochilie sau carapace. Numele acestui program vine din faptul că "înveleşte" nucleul sistemului de operare, precum cochilia miezul moale al melcului: utilizatorul nu are de-a face cu nucleul ci interacţionează prin intermediul shell-ului. Pentru că nu îmi vine în minte nici o traducere potrivită, voi continua să folosesc cuvîntul englezesc.

Shell-ul este un program care permite utilizatorilor să tasteze şi execute comenzi. Una din funcţiile foarte importante ale shell-ului este de a permite utilizatorilor să pornească în execuţie *alte* programe. Numim un program în curs de execuţie *proces*. Toate procesele utilizatorilor pe un sistem Unix descind dintr-un shell.

În Unix shell-ul oferă mai mult decît abilitatea de a lansa procese în execuţie: oferă o sumedenie de comenzi şi facilităţi suplimentare, care-l fac un mediu ideal pentru a activităţi de administrare a sistemului. Toate aceste comenzi formează un *limbaj de programare* deosebit de puternic; shell-ul este deci un *interpretor*, care citeşte şi execută comenzi.

2 Shell-uri interactive şi neinteractive

Putem deci distinge două utilizări separate ale shell-ului: prima utilizare constă în a executa comenzi simple, în general pentru a lansa în execuţie alte programe. În acest caz shell-ul prezintă utilizatorului un *prompt*, o invitaţie de a primi o comandă. După ce utilizatorul tastează comanda, shell-ul o execută imediat, iar cînd comanda îşi termină execuţia (sau chiar mai înainte, dacă este instruit în acest sens), shell-ul oferă din nou un prompt. Acest mod de utilizare, în care fiecare comandă este citită de la utilizator şi executată se numeşte *interactiv*. În acest text vom folosi semnul procent `%` pentru a indica prompt-ul.

A doua utilizare a shell-ului este pentru execuţia unor programe mai complicate, scrise dinainte şi depozitate în fişiere. Un fişier cu comenzi pentru shell se numeşte în engleză *shell script* (adică un “scenariu pentru cochilie”), sau pe scurt “script”. Executarea unui script se mai numeşte “procesare în vrac”, *batch processing*, pentru că shell-ul nu se mai opreşte după fiecare comandă afişînd un prompt.

Ca orice alt limbaj de programare, cunoaşterea doar a unei fracţiuni din elementele de bază se dovedeşte perfect satisfăcătoare pentru nevoile de zi cu zi. Voi ilustra aici numai comenzile cele mai puternice, şi pe acestea le voi descrie mai mult prin exemple decît riguros. Pentru cei doritori de aprofundarea subiectului, secţiunea finală despre surse de informaţii suplimentare se poate dovedi un punct bun de plecare.

3 Familii de shell-uri

Dacă Windows NT nu are un shell decent, în lumea Unix situaţia este chiar pe dos; un scurt istoric este necesar pentru a lămuri babilonia de opţiuni existente.

Primul shell tradiţional pentru Unix a fost scris în 1976 de Steve Bourne, care pe vremea aceea lucra la laboratoarele Bell ale companiei AT&T. În onoarea creatorului său, shell-ul acesta este numit “Bourne shell”. Programul cu pricina se numeşte simplu “sh”, şi se află de obicei în directorul `/bin` pe un sistem Unix (`/bin/sh`). Să nu uităm că sistemul de operare Unix însuşi fusese creat cu puţin timp în urmă în acelaşi loc, inspirat de sistemul de operare Multics (vedeţi şi articolul meu despre istoria Unix-ului). Shell-ul Bourne introducea o mulţime de concepte revoluţionare, care făceau viaţă utilizatorilor şi administratorilor mult mai simplă decît în sistemele de operare precedente. Shell-ul Bourne era mai curînd proiectat pentru utilizarea sub formă de interpretor, şi mai puţin pentru cea interactivă.

Cînd la începutul anilor ’80 la universitatea Berkeley din California a fost dezvoltată varianta locală de Unix, numită BSD, Bill Joy, un talentat programator, a scris un nou shell numit C Shell, sau `csh` (pronunţat de guru în engleză “sişi”). (Bill Joy este unul dintre fondatorii companiei Sun, la care actualmente este Chief Scientist.) Acest shell introduce facilităţi foarte utile pentru execuţia interactivă, dar din păcate nu este compatibil cu Bourne, şi suferă de o mulţime de neajunsuri.

În 1984 David Korn, tot de la Bell Labs ale lui AT&T, şi-a propus să modernizeze shell-ul Bourne adăugîndu-i facilităţi interactive à la `csh`. Astfel s-a născut shell-ul Korn `ksh`, care este excelent realizat şi compatibil cu Bourne.

Din păcate `ksh` iniţial nu era software “free”; aşa că unul din primele proiecte ale fundaţiei Free Software Foundation (vedeţi articolul meu despre Open Source din PC Report din iunie 1998) a fost să implementeze un nou shell, de data asta complet “liber”. Acest shell moşteneşte din ideile lui `ksh`, dar ia cîteva lucruri bune de la `csh`. Noul shell a fost implementat original de Brian Fox, care era plătit de FSF; numele shell-ului este “Bourne Again SHell” (“din nou Bourne”), sau `bash` (citit aproximativ “beş”). Acesta este shell-ul standard pe sistemele GNU/Linux.

O mişcare paralelă a transformat substanţial `csh` în ceea ce a devenit `tcsh`, sau “tisişel”. `tcsh` este compatibil cu `csh`, dar nu cu `sh`.

Prin 1981 Microsoft a lansat sistemul de operare MS-DOS. Acesta era echipat cu un shell foarte primitiv, numit `COMMAND.COM`. Deşi creat ulterior Unix-ului, şi inspirat de acesta, shell-ul MS-DOS este extrem de primitiv, inconsistent şi greu de folosit. Din păcate a rămas, cu minore îmbunătăţiri, shell-ul standard chiar şi sub Windows2000.

În Windows însă multe dintre funcţiunile unui shell sunt luate de un program grafic, numit Command Manager sub Windows 3.1. Pentru utilizatori novici un shell grafic oferă o interfaţă mult mai simplă şi intuitivă, dar pentru un ins experimentat sau pentru un administrator facilităţile acestuia (şi ale altor programe de administrare) sunt adesea frustrante.

De aici încolo avem de a face cu o explozie de noi shell-uri, care aduc tot felul de înflorituri şi varietăţi; voi cita astfel: `ash`, `zsh`, `scsh`, `rc`, `bsh`, `pdksh`, `es`.

În articolul de faţă voi vorbi despre Bourne Shell; acest shell a evoluat şi el de-a lungul timpului, şi a fost şi standardizat de comitetul POSIX, care a standardizat Unix. De aceea, `sh` este de departe alternativa cea mai sigură: chiar dacă nu aveţi alte shell-uri la dispoziţie, `sh` este sigur disponibil (pe sisteme gen GNU/Linux se află `bash`, care are un mod de funcţionare în compatibilitate 100% cu `sh`).

4 Elemente fundamentale de programare în Shell

În restul acestui articol voi discuta doar despre programarea în shell. Elementele shell-ului legate de utilizarea interactivă vor fi trecute sub totală tăcere (cum ar fi editorul liniilor de comandă, mecanismul de re-execuţie a comenzilor (history), controlul job-urilor lansate în execuţie).

4.1 Comenzi externe

Shell-ul este un program ale cărui capacităţi pot fi înzecite de faptul că poate controla toate celelalte programe. Am spus deja că shell-ul poate porni în execuţie orice alt program; în plus, shell-ul poate influenţa în mai multe moduri mediul în care programul respectiv este executat. Vom vedea mai jos cîteva exemple.

Cel mai important de ştiut pentru moment este faptul numele oricărui fişier executabil (adică un fişier care conţine o aplicaţie) este o comandă shell. Astfel, fişierul "netscape" conţine imaginea executabilă a browser-ului Internet; cînd shell-ului îi este oferită spre execuţie comanda `netscape`, el va lansa în execuţie programul acesta. Un fişier executabil mai este de aceea numit "comandă externă".

Capacitatea de a controla alte programe dă shell-ului alura unui limbaj extensibil; creind noi programe de fapt creăm noi "comenzi" pentru shell. Shell-ul poate combina execuţia mai multor programe controlînd comunicaţia dintre ele, creînd astfel cu uşurinţă programe mai complicate din bucăţi simple.

Pentru ilustraţie vom folosi în acest text cu precădere cîteva comenzi externe simple; iată-le rezumate aici:

ls: Programul `ls` se citeşte "list", şi este echivalentul lui "dir" din DOS. Este urmat de o listă de nume de fişiere, iar efectul lui este de a afişa chiar numele acestor fişiere. Urmat de un director, afişează fişierele din acel director. (Fără argumente operează pe directorul "curent".) Iată un exemplu:

```
% ls
Mail  bin  data  lib  man  src  tmp
% ls src
Makefile  hello.c
%
```

Am ilustrat aici funcţionarea interactivă a shell-ului, pe care o vom folosi pînă avem destule elemente pentru a scrie programe mai mari. Caracterul `%` a fost scris de shell, fiind prompt-ul. utilizatorul a tastat apoi `ls`; shell-ul a identificat acest şir de caractere ca fiind numele unui fişier executabil, care a fost executat. Cînd se execută, `ls` afişează conţinutul directorului curent (`Mail`, `bin`, etc.). După ce `ls` s-a terminat, shell-ul afişează din nou prompt-ul, cu promptitudine.

wc: Programul Word Count numără liniile, cuvintele şi caracterele din fişierele care-i sunt date ca argumente. Iată un exemplu:

```
% wc src/hello.c
      6       9      74 src/hello.c
%
```

Vedem mai sus că fişierul `hello.c` din directorul `src` are 6 linii, 9 cuvinte şi 74 caractere.

cat: deşi înseamnă "pisică", programul `cat` de fapt tipăreşte conţinutul fişierelor care-i sunt oferite drept argument (echivalent cu `type` din DOS). Numele lui vine de la "CATalogue".

```
% cat src/hello.c
#include <stdio.h>

main() {
    printf("Hello world\n");
    return 0;
}
%
```

4.1.1 Succes şi eroare

În Unix, cînd un proces se termină returnează procesului său părinte[1] un cod de eroare, care permite părintelui să detecteze dacă odrasla-i s-a executat cu succes sau a întîmpinat nişte probleme.

În mod surprinzător, cel puţin pentru un programator C, 0 este codul pentru succes, şi orice valoare nenulă indică o eroare. Vom vedea mai încolo cum aceste valori pot fi folosite de shell pentru a crea programe complicate.

4.2 Variabile shell

Ca orice limbaj de programare, limbajul shell-ului conţine variabile. În mod tradiţional, variabilele shell sunt scrise numai cu majuscule, pentru că în Unix programele executabile au nume scrise cu minuscule; în felul acesta se evită confuziile. Valoarea unei variabile shell este un şir arbitrar de caractere. Numele variabilelor sunt aceleaşi ca în limbajul C: o literă urmată de litere, cifre şi semnul "subliniat" `_`.

[1] Acesta este procesul care l-a pornit în execuţie, în cazul nostru shell-ul însuşi.

Principala limitare a shell-ului, care îl face nepractic pentru scrierea de programe mari, constă în sărăcia tipurilor de date disponibile: există un singur tip de date, şirul de caractere. Nu există numere, structuri, matrici, liste, arbori, etc. Pentru a scrie programe mai complicate, aceste structuri de date trebuie implementate pornind de la şiruri, ceea ce e complicat şi adesea ineficient. Dar asta nu ne va îndepărta de la scopul de a vedea ce putem face totuşi folosind shell-ul.

4.2.1 Atribuiri

Cea mai simplă comandă a shell-ului este cea de atribuire; ea are forma:

```
variabila=valoare
```

Efectul acestei comenzi este de a atribui valoarea din dreapta variabilei din stînga. **Atenţie:** nu puteţi pune spaţii la stînga şi la dreapta semnului egal.

4.2.2 Valoarea unei variabile

`sh` este un limbaj straniu prin faptul că pentru a accesa valoarea unei variabile trebuie s-o prefixăm cu semnul dolar. Deci atribuirea şi citirea folosesc nume diferite! O eroare comună este de a folosi dolar la atribuire sau de a uita dolarul la citire. Atenţie, deci:

```
[1] % DIR=src
[2] % ls $DIR
[3] Makefile  hello.c
[4] % ls DIR
ls: DIR: No such file or directory
[5] % $DIR=bin
sh: src=bin: command not found
[6] % DIR=ls
[7] % $DIR src
Makefile  hello.c
[8] % DIR=$DIR$DIR
[9] % ls $DIR
ls: lsls: No such file or directory
[10] % DIR=src/
[11] % ls ${DIR}Makefile
Makefile
```

În linia [2] variabila `DIR` este substituita cu valoarea sa, `src`. În linia [4], `DIR` este chiar numele directorului căutat, care nu există, cauzînd `ls` să afişeze o eroare. Iar în linia [5], în loc să se facă o atribuire variabilei `DIR`, valoarea acesteia este substituită, generînd comanda `src=bin`, care *nu* este interpretată la rîndul ei ca o atribuire, ci este direct executată, şi care, fiind inexistentă, generează un mesaj de eroare. Dacă atribuim lui `DIR` valoarea `ls`, ca în linia [6], executînd linia [7] executăm de fapt `ls src`. În linia [8] observăm cum putem concatena valoarea a două variabile juxtapunîndu-le. În fine, linia [11] arată cum procedăm dacă vrem să concatenăm valoarea unei variabile `DIR` cu un alt şir de caractere: trebuie să folosim acolade pentru a delimita numele variabilei.

4.2.3 Ghilimele

Shell-ul permite construirea de şiruri de caractere folosind două tipuri de ghilimele: apostroful (') şi ghilimelele duble ("). Diferenţa între cele două tipuri este că între apostrofuri valorile variabilelor *nu* sunt substituite, pe cînd între ghilimele sunt. (La fel şi pentru expresiile regulate, despre care nu am vorbit încă). Ghilimelele sunt utile cînd vrem să includem spaţii în valorile variabilelor.

4.2.4 Variabile interne

Shell-ul însuşi foloseşte unele variabile pentru nevoile sale interne. Schimbînd valoarea acestor variabile putem afecta comportarea sa. De exemplu, variabila numită `PS1` este chiar prompt-ul. Variabila internă `PWD` este directorul curent.

```
% PS1="ordonati, stapine: "
ordonati, stapine: ls
Mail  bin  data  lib  man  src  tmp
ordonati, stapine:
```

Unele din variabilele interne nu pot fi atribuite, ci pot fi doar citite. Unele din acestea au nume stranii, formate din alte caractere decît litere şi cifre; de exemplu variabila `$?` conţine rezultatul cu care s-a terminat ultima comandă (0 pentru succes).

4.3 Comenzi interne importante

Unele comenzi ar putea fi implementate atît sub formă de comenzi interne cît şi externe. Dar altele trebuie neapărat să fie implementate de către shell. De exemplu comanda `cd`, care schimbă directorul curent: aceasta nu poate fi o comandă externă; dacă `cd` ar fi implementată de un program separat, atunci cînd shell-ul ar porni programul numit `cd`, acesta s-ar executa, ar schimbă directorul său curent după care s-ar termina. Dar schimbările făcute de un proces fiu nu se propagă la părinţii lui (ci doar invers), deci shell-ul rămîne în directorul iniţial.

În această secţiune vom vedea alte cîteva comenzi interne importante.

cd: (change directory) este urmată de numele unui director şi schimbă directorul curent în acel director;

exit: este urmată de o valoare numerică; shell-ul îşi termină execuţia cu valoarea indicată. Deci `exit 0` înseamnă "succes". Adesea, în caz de eroare, programatorii utilizeaza acest cod pentru a transmite procesului-părinte informaţii despre eroarea petrecută;

echo: (ecou) îşi tipăreşte argumentele. Este o comandă deosebit de utilă:

```
[1] % DIR=src
[2] % echo DIR $DIR
DIR src
[3] % echo DIR         DIR
DIR DIR
[4] % echo "DIR         DIR"
```

```
DIR         DIR
[5] % echo "$DIR         $DIR"
src         src
[6] % echo '$DIR         $DIR'
$DIR         $DIR
```

Observaţi diferenţa între liniile [3] şi [4]; în linia [3] `echo` primeşte două argumente, iar în linia [4] primeşte unul singur, un şir care conţine şi spaţii.

eval: Aceasta este o comandă extrem de puternică. Argumentul ei este un şir de caractere, care este executat ca şi cum ar fi o comandă shell.

```
[1] % DIR=src
[2] % COMANDA=ls
[3] % echo "$COMANDA $DIR"
ls src
[4] % eval "$COMANDA $DIR"
Makefile  hello.c
```

test: e foarte utilă pentru a evalua expresii boolene (care adică generează un rezultat "adevărat" sau "fals"). Ca şi codurile de eroare ale proceselor, "adevărat" este 0, iar fals este orice valoare diferită de 0.

`test` poate face patru feluri de teste diferite:

- Teste asupra fişierelor; de exemplu:
  - `test -f fisier` returnează "adevărat", adică 0 (zero), dacă fişierul indicat există şi este un fişier ordinar (adică nu un director).
  - `test -d director` testează existenţa unui director
  - `test -r fisier` vede dacă fişierul poate fi citit (permisiunile de citire dau dreptul acestui utilizator)
  - `fisier1 -nt fisier2` e adevărat dacă primul fişier este mai nou decît al doilea (newer than).
- Teste pentru şiruri de caractere:
  - `test -z sir` testează dacă şirul conţine vreun caracter (e nenul)
  - `test sir1 = sir2` testează dacă cele două şiruri sunt egale
  - `test sir1 != sir2` testează dacă şirurile sunt diferite
- Teste pentru valori numerice: se pot folosi operaţiile `-eq,-ne,-le,-lt,-gt,-ge` pentru egal (equal), inegal (not equal), mai mic sau egal (less or equal), mai mic (less than), mai mare (greater than) sau mai mare sau egal (greater or equal), respectiv.

  Nu folosiţi `=` sau `==` pentru a compara numere!

- Teste pentru valori boolene: puteţi folosi `-a` pentru "şi" (and), `-o` pentru "sau" (or) şi `!` pentru negaţie.

  Din păcate nu puteţi folosi paranteze; pentru expresii mai complicate vedeţi mai jos comanda `expr`.

```
% DIR=src; FILE=Makefile
% test -f $DIR/$FILE
% echo $?
0
%
```

În exemplul anterior ilustrăm mai multe concepte noi: în prima linie punem două comenzi pe aceeaşi linie, separîndu-le cu punct-şi-virgulă. În linia a doua testăm existenţa unui fişier. În linia a treia tipărim rezultatul testului anterior (vă amintiţi că variabila `$?` conţine rezultatul ultimei comenzi executate, şi că 0 reprezintă succes).

Pentru convenienţa utilizatorului, există o comandă cu numele `[` (paranteză dreaptă deschisă), care este echivalentă cu `test`. Astfel în loc de: `test -f fisier` putem scrie: `[ -f fisier ]`. Observaţi că paranteza trebuie închisă, şi în plus *trebuie lăsate spaţii în jurul parantezelor.*

if: Putem face tot felul de teste, dar cum putem beneficia de rezultatul lor? Folosind instrucţiunea de execuţie condiţională, ca în următorul exemplu:

```
% DIR=src; FILE=Makefile
% if [ -f $DIR/$FILE ]; then echo "Exista"; else echo "Nu exista"; fi
Exista
%
```

Aici am combinat două comenzi: `if` şi `test`. Comanda internă `if` este urmată de o altă comandă. Dacă cea din urmă se termină cu succes (returnează 0), atunci comanda de după `then` este executată; altfel ramura de după `else`. Nu-l uitaţi pe `fi` la sfîrşit.

Cînd compunem un script putem separa părţile unui `if` pe mai multe linii. Iată un exemplu de script:

```
#!/bin/sh

# acest script tipareste fisierul argument, sau un mesaj de
# eroare daca acesta nu exista

if [ $# != 1 ]; then
        echo "Imi trebuie un argument"
        exit 1
elif [ -f $1 ]; then
        cat $1
```

```
else
        echo "$1 nu exista"
fi
```

Acesta este deja un script interesant, care introduce cîteva elemente noi:

- Comentariile în shell sunt introduse de semnul diez (`#`) şi se termină odată cu linia;
- Prima linie este o convenţie Unix: dacă un fişier începe cu semnele `#!/`, atunci urmează numele unui interpretor care trebuie să execute acest program;
- Variabila internă `$#` conţine numărul de argumente primite de shell în linia de comandă;
- Variabilele `$1`, `$2`, ..., `$9` conţin argumentele însele;
- Observaţi folosirea lui `elif` în loc de `else if`.

Dacă punem script-ul anterior în fişierul numit "arata.sh", putem să-l executăm cu secvenţa de comenzi:

```
% chmod a+x arata.sh
% ./arata.sh
Imi trebuie un argument
% ./arata.sh bibi
bibi nu exista
% ./arata.sh src/hello.c
#include <stdio.h>

main() {
    printf("Hello world\n");
    return 0;
}
%
```

Comanda `chmod a+x arata.sh` are drept efect de a face fişierul `arata.sh` executabil în aşa fel încît oricine să-l poată executa (CHange the MODe for All to add (+) eXecutable).

Observaţi că pornim script-ul în execuţie indicînd directorul unde se află (`./arata.sh`). Dacă atunci cînd pornim în execuţie un program, nu indicăm exact prin directorul unde se află, shell-ul foloseşte o variabilă numită `PATH` care conţine o serie de directoare (separate cu semnul două puncte) unde acest script trebuie căutat. Încercaţi şi `echo $PATH`.

read: citeşte cuvinte într-una sau mai multe variabile:

```
% read a b c
tastez multe cuvinte sa vedem cum merg
% echo ":$a:$b:$c:"
:tastez:multe:cuvinte sa vedem cum merg:
%
```

Read citeşte de la intrare o linie (linia care apare după comandă a fost introdusă de la tastatură, şi nu tipărită de shell); apoi `read` sparge linia în cuvinte separate de spaţii, şi fiecare cuvînt este atribuit unei variabile; ultima variabilă primeşte restul liniei pînă la sfîrşit.

for: permite să executăm un ciclu într-o listă de cuvinte, astfel:

```
for i in hello goodbye ok; do
    for j in .o .c~; do
        if [ -f $i$j ]; then
            echo "Sterg $i$j"
            rm $i$j
        fi
    done
done
```

Acest program caută fişierele `hello`, `goodbye` sau `ok` urmate de sufixul `.o` sau `.c~` (ceea ce în genere înseamnă fişier obiect, respectiv o versiune veche a unui fişier C) şi le şterge. Observaţi folosirea a două bucle imbricate.

Înainte de a prezenta alte cîteva comenzi interne foarte utile, vom discuta sumar despre alte funcţiuni foarte importante ale shell-ului.

4.4 Expansiunea expresiilor regulate

O trăsătură extrem de puternică a shell-ului este că poate expanda *expresii regulate* care descriu succint nume de fişiere. Pentru că am scris de curînd un articol despre acest subiect (în PC Report din aprilie 2000), voi fi foarte sumar aici, prezentînd doar cîteva exemple:

Expresie	**Semnificaţie**
`a*`	Toate fişierele al căror nume începe cu `a`
`*.c`	Toate fişierele al căror nume se termină cu `.c`
`.??*`	Toate fişierele al căror nume începe cu punct şi conţine cel puţin 3 caractere
`*/src/*.c`	Toate fişierele terminate cu `.c` aflate în subdirectorul `src` al oricărui director din directorul curent.
`*[0-3]*`	Toate fişierele care conţin una din cifrele 0,1,2 sau 3 în nume
`{ab,cd}.{c,h}`	fişierele existente în mulţimea `ab.c`, `ab.h`, `cd.c`, `cd.h`

Iată şi un program care foloseşte expresiile regulate:

```
#!/bin/sh

ACEST_SCRIPT=$0

for i in *.c; do
       if [ -f $i ]; then echo $i; fi
```

```
done

for i in *; do
        if [ -d $i ]; then
                cd $i; $ACEST_SCRIPT; cd ..
        fi
done
```

Dacă script-ul anterior este în undeva într-un director indicat de variabila `PATH`, atunci execuţia lui va cauza tipărirea tuturor fişierelor C dintr-o ierarhie de directoare. Observaţi cum script-ul foloseşte variabila `$0`, care conţine propriul lui nume, pentru a se auto-executa în toate subdirectoarele.

4.5 Comenzi paralele

Am văzut că putem folosi semnul punct-şi-virgulă pentru a lansa în execuţie mai multe comenzi succesiv. Dacă vrem să lansăm în execuţie două comenzi simultan, punem între ele semnul `&`:

```
% netscape & xterm &
%
```

Această comandă va lansa în execuţie programul netscape, şi fără a aştepta terminarea sa va lansa şi programul xterm. Semnul `&` de la sfîrşit îi spune shell-ului să nu aştepte nici terminarea lui xterm, ci să ofere un nou prompt.

Se pot de asemenea folosi semnele `&&` şi `||`. Dacă scriem `c1 && c2`, înseamnă că vrem ca `c2` să se execute dacă şi numai dacă `c1` se termină cu succes. Dimpotrivă, `c1 || c2` înseamnă că `c2` se execută dacă şi numai dacă `c1` a eşuat.

4.6 Redirectare şi conducte (pipes)

Cînd shell-ul lansează un proces are posibilitatea de a cupla intrarea şi ieşirea acestuia în diferite moduri. În Unix fiecare proces la pornire are 3 canale de comunicaţie deschise: un canal de la care primeşte date (intrarea), unul la care scrie date (ieşirea) şi unul la care scrie erori. Pentru shell în mod normal aceste canale sunt cuplate la tastatura şi respectiv ecran. Ele pot fi însă redirecţionate spre fişiere foarte simplu:

```
% ls >lista
% cat lista
Mail  bin  data  lib  man  src  tmp
%
```

Semnul `>` este urmat de un fişier; ieşirea comenzii executate este “depusă” atunci în acel fişier.

Mai spectaculos este că putem cupla ieşirea unui proces la intrarea altuia folosind un singur caracter, scris `|` şi citit “ţeavă” (pipe). Pentru a afla de exemplu cîte fişiere sunt în directorul curent, putem folosi:

```
% ls | wc -w
7
% ls >lista
% wc -w <lista
7
% rm lista
```

`wc -w` număra doar cuvintele (words). Pentru a număra fişierele putem fie trimite rezultatul lui `ls` într-un fişier `lista` folosind redirectare, după care putem număra cuvintele, sau, mult mai eficient şi rapid, putem cupla ieşirea lui `ls` la intrarea lui `wc` cu o ţeavă, ca în prima linie.

Un idiom ades folosit este de a redirecta erorile spre un dispozitiv fictiv numit `/dev/null`, care este găleata de gunoi pe un sistem Unix:

```
% wc -l *.c */*.c 2>/dev/null
```

Această comandă numără liniile din toate fişierele care se termină cu `.c` în directorul curent şi în toate subdirectoarele sale. Dacă unele din fişiere sunt însă ilizibile (de exemplu pentru că avem directoare al căror nume se termină cu `.c`), atunci `wc` va tipări nişte erori, care însă vor fi trimise spre "null" datorită redirectării canalului de eroare, designat de `2>`.

4.7 Accentul grav

O facilitate extrem de puternică a shell-ului este oferită de semnele de accent grav `. Orice este cuprins între semne de accent grav este executat ca o comandă; rezultatul acelei comenzi devine un şir de caractere, care înlocuieşte comanda între accente grave.

```
% wc -w src/hello.c
9 src/hello.c
% a=`wc -w src/hello.c`
% echo $a
9 src/hello.c
%
```

4.8 Alte comenzi

while: ne permite, ca şi `for`, să executăm instrucţiuni în mod repetat. În script-ul de mai jos, `while` se execută atîta timp cît comanda `read` funcţionează cu succes; astfel citim fiecare linie din fişierul `text-de-prelucrat`. Tipărim apoi numai liniile care au mai mult de 10 cuvinte.

```
#!/bin/sh

cat text-de-prelucrat | while read linie; do
    cuvinte=`echo $linie | wc -w`
    if [ $cuvinte -ge 10 ]; then
        echo $linie
    fi
done
```

expr: permite shell-ului să opereze şi cu numere. Am văzut că singurul tip de date al shell-ului sunt şirurile de caractere. Cum putem atunci face ceva aritmetică? Comanda `expr` este urmată de o expresie aritmetica a cărei valoare o tipăreşte la ieşire. E o metodă un pic cam complicată pentru a evalua simple expresii aritmetice, dar dacă faceţi puţine calcule, e foarte practică. Trebuie să fiţi atenţi să separaţi *toate* argumentele lui `expr` cu spaţii.

```
#!/bin/sh

contor=0
while [ $contor -le 100 ]; do
        echo $contor
        contor=`expr $contor + 1`
done
```

5 Încheiere

Aici pun capăt acestei incursiuni blitz în programarea în shell. Sper să vă fi pus la-ndemînă suficiente scule ca să puteţi să scrieţi programe utile. Shell-urile moderne sunt foarte sofisticate, oferă o sumedenie de facilităţi suplimentare, şi pot fi configurate şi adaptate în mii de moduri. Dar 90% din problemele întîlnite în practică pot fi rezolvate folosind cele 10% din limbaj pe care le-am prezentat.

Oricum, nu puteţi să vă perfecţionaţi decît folosind limbajul; m-aş bucura dacă v-am dat motive să încercaţi.

6 Alte surse de informaţie

- S-au publicat o mulţime de cărţi despre programarea în shell; din cauză ca eu însumi nu am citit nici una, mă voi feri să recomand ceva; un manual de referinţă tipărit cu exemple bune este însă foarte util;
- În general pe grupurile de discuţii Usenet se vehiculează multe gunoaie; de departe însă cel mai bun grup la care am fost vreodată abonat este `news:comp.unix.shell`. O sumedenie de experţi vor răspunde imediat la orice întrebare legată de shell aţi avea; în plus discuţiile sunt uneori foarte interesante;
- Foarte multe informaţii interesante sunt în fişiere de tip FAQ: Frequently Asked Questions, arhivate în multe locuri pe Internet, de pildă la `faqs.org`. Recomand următoarele fişiere:
  - Originea shell-ului şi a numelui `http://www.multicians.org/shell.html`.
  - `http://www.faqs.org/faqs/unix-faq/shell/bash`: întrebări frecvente despre Bash;
  - `http://www.faqs.org/faqs/unix-faq/faq/part5`: partea a cincea a acestei "Unix FAQ" discută în mod special despre shell-uri;
  - `http://www.faqs.org/faqs/unix-faq/shell/shell-differences` discută despre istoricul shell-ului şi despre diferenţele între shell-uri.

- O pagină interesantă aflată la `http://www.fi.uib.no/~btk/data/shell-100.BetaA.html` discută echivalenţele comenzilor din diferitele shell-uri;
- Dacă aveţi un sistem Linux, atunci paginile de manual `man bash` şi `man tcsh` sunt probabil instalate. Am apelat la ele adesea în cursul scrierii acestui articol;
- Pagina shell-ului `bash` la `http://www.gnu.org/software/bash/bash.html`;
- Tcsh este disponibil (cod şi manuale) la `ftp://ftp.gw.com/pub/unix/tcsh`.