

Shell et Outils Unix

Shell et Outils Unix - Eric Lecolinet - ENST Paris

E. Lecolinet

ENST

PLAN

Shell et Outils Unix

SHELL	3
FILTRES ET EXPRESSIONS REGULIERES	31
GREP	35
AWK	39
DEBOGUEURS	47

Shell et Outils Unix - Eric Lecolinet - ENST Paris

SHELL UNIX

Interpréteur de commandes du systèmes **Unix**

- utilisation conversationnelle :
 - ☞ interpréteur de commandes **interactif**
- utilisation non conversationnelle :
 - ☞ langage de programmation des **scripts**
 - ☞ création de nouvelles commandes avec :
notation concise, variables, paramétrage, structures de contrôle

Notation :

"quotation inverse" :

- ☞ pas de "marquage" des chaînes de caractères (sauf si nécessaire)
- ☞ marquage des valeurs des variables (prefixe \$)
- ☞ pas de virgule entre les arguments

Syntaxe

commande *[-options] [paramètres] [redirections]*

```
ls -al *.c
ps -aux
cd ~username
```

Redirections :

entrée standard:

```
sort < input-file
```

sortie standard:

```
man awk > output-file
```

sortie erreurs:

```
ls 2> kk           (sh, ksh, bash)
ls >& kk           (csh, tcsh)
```

Redirections

> fichier dirige la sortie standard vers **fichier**
>> fichier ajoute la sortie standard à **fichier**
< fichier prend l'entrée standard depuis **fichier**
com1 | com2 **tube:** connecte la sortie de **com1** à l'entrée de **com2**
<<s prend l'entrée jusqu'à **s** en début de ligne

Pour sh, ksh, bash :

n> fichier dirige la sortie du descripteur **n** vers **fichier**
n>&m fusionne la sortie du descripteur **n** à celle de **m**
n<&m fusionne l'entrée du descripteur **n** à celle de **m**

Descripteurs :

0 entrée standard
1 sortie standard
2 sortie des erreurs

Métacaractères du Shell

&	commande en "background" (arrière-plan)	<code>cc prog.c -o prog &</code>
;	succession de commandes	<code>cc prog.c -o prog; prog</code>
 	tube (pipe)	<code>ls -l wc</code>
*	chaîne générique	<code>ls *.c ; ls -a *</code>
?	caractère générique	<code>ls version?.c</code>
[ccc]	prend un caractère parmi ccc	<code>ls version[123].c</code>
[a-z]	prend un caractère entre a et z	<code>ls .[A-z]*</code>
\	neutralisation d'un métacaractère	<code>ls *toto\?</code>
'...'	neutralisation à l'intérieur de la sous-chaîne ...	<code>ls '*toto?'</code>
"..."	neutralisation partielle (interprétation de \$ et `)	<code>ls "\$HOME/*toto?"</code>

'...' exécute les commande dans ... ; le résultat remplace ...
`ls -l `which ps``

(...) exécute les commande dans ... par un sous-shell
`(make prog ; prog)&`

commentaire

`com1 && com2` exécute com1, s'il n'y a pas d'erreur exécute com2

`com1 || com2` exécute com1, s'il y a une erreur exécute com2

Caractères de contrôle

^D fin de fichier, fin de connexion

^C interruption (termine l'exécution)

^ quit (crée un "core dump" pour analyse post mortem)

^Z suspend l'exécution (reprise par **fg** ou **bg** ; pas supporté par sh)

Commandes usuelles

echo affichage des arguments sur sortie standard
`echo -n date = ; date`

who, rwho liste des utilisateurs connectés
`who am i ; whoami`

tty terminal utilisé

cd, pwd changer, montrer le répertoire courant

du espace disque

df partitions

ps liste des processus
`ps -aux` (Unix BSD)
`ps -ef` (Unix Système V)

stty	configuration des entrées stty -a stty intr ^y (taper Ctrl-v Ctrl-y)
passwd, yppasswd, nispasswd	changement du mot de passe
chmod	changer les attributs d'un fichier chmod a+x toto ; ls -l toto
telnet, rlogin	remote login: connexion sur machine distante
rsh	remote shell: exécution d'un programme sur machine distante rsh autre-machine who (dépend du fichier .rhosts)
which	retrouver une "commande" (= un fichier exécutable dans \$PATH) which ls
find	retrouver un fichier quelconque (en spécifiant des conditions) find ~ -name toto* -print

Shells Unix

Deux familles :

-- System V :

Bourne Shell: **sh**
 Korn Shell: **ksh**,
 Gnu Shell: **bash**
 ... **zsh** ...

-- BSD

C-Shell: **csh**
 Toronto C-Shell: **tcsh**

Plusieurs shells "cohabitent" souvent sur une même machine; *fréquemment* :

csh ... --> utilisation interactive
 = *interpréteur de commandes*

sh ... --> utilisation non conversationnelle :
 = *langage de programmation des **scripts***

Fichiers d'initialisation

Shells:

sh, ksh : .profile
 zsh : .profile .zshrc
 csh, tcsh : .login .cshrc .tcshrc .logout

Outils:

Rlogin: .rhosts
 Emacs: .emacs
 Débogueur: .dbxinit
 X Window: .xinitrc .Xdefaults .mwmrc .xmodmaprc

A mettre dans le "home directory"

Utilisation interactive

Editeur de ligne (ksh, bash, zsh, tcsh)

suivant les cas:

Flèches, mode Emacs (^B ^F ^P ^N), mode VI

Complémentation

ESC TAB ^D

Historique

liste: **history**
 rappel: **!!** **!string** **!no-commande**

Aliases

sh : pas d'aliases
 csh, tcsh : **alias name value** (à mettre dans .cshrc)
 ksh, zsh : **alias name="value"**

Exemples (csh):

```
alias ls ls -F
alias ll ls -la
alias E "emacs \!* &"
```

Job control

(pas de "job control" pour sh)

^Z : suspend l'exécution (met le processus en cours dans l'état dormant)

jobs : liste les jobs lancés depuis ce shell

bg *%no-job* ou **bg** *no-process* : met un processus en "background"

fg *%no-job* ou **fg** *no-process* : met un processus en "foreground"

Variables

- type chaîne de caractères
- nom commençant par une lettre
- déclaration implicite
- affectation :

sh: *name=value* (pas de blancs autour du signe =)

csh: **set** *name = value*

- valeur de la variable:

\$name

Exemples (sh)

```
compteur=0
```

```
echo $compteur
```

```
s=/usr/paul/src
```

```
mv *.c $s
```

```
l=/usr/lib/lib
```

```
ls ${l}c.a ( ≠ ls $lc.a !!!)
```

Variables d'environnement

- variables transmises aux **sous-shells**
- servent à configurer l'utilisation du Shell et des outils Unix

sh: *name=value ; export name*
csch: *setenv name value*

Variables usuelles

PATH
 LD_LIBRARY_PATH
 MANPATH
 MAIL, MAILPATH
 HOME, PWD
 PS1, PS2
 SHELL
 DISPLAY
 LINES, COLUMNS
 VISUAL, EDITOR, XEDITOR
 PRINTER, LPDEST, TAPE
 NAME, SIGNATURE
CSH et TCSH : prompt ignoreeof nobeep history filec ignore

Scripts

- fichiers exécutables contenant des commandes Unix
 - pour créer de "nouvelles commandes" de manière simple
- => Shell = vrai langage de programmation**

NB: par défaut: **Bourne Shell (sh)**

Créer et utiliser un nouveau script:

1. éditer le fichier de commandes (par exple: "myscript")
2. le rendre exécutable:


```
chmod a+x myscript
```
3. lancement comme une commande normale:


```
myscript
```
4. (éventuellement) mettre le script dans un des répertoires de **\$PATH**
 faire:


```
rehash
```

Paramètres et variables spéciales des scripts

\$1 \$2 ... \$9 argument 1, 2 ...9 de la ligne de commande
\$* tous les arguments
\$# nombre d'arguments
\$0 nom du script

ligne de commande : toto xxx yyy zzz
 -> variables: \$0 \$1 \$2 \$3

\$\$ numero du processus shell
\$_ numero du dernier processus en arrière plan
\$? code de retour (valeur renvoyée par **exit**)
 vaut **0 (= True)** si terminaison normale

Spécification du shell à employer pour interpréter le script:

- par défaut: **Bourne Shell (sh)**
- sinon indiquer: **#!shell-a-executer** sur la **première ligne** du script

Par exemple: #!/usr/local/bin/tcsh

Boucle for

Syntaxe: **for i in liste ; do commande ; done**

Exemples:

```
for i in *.c
do
    echo "fichier $i :"
    diff ../old/$i $i
done
```

Dans un script:

```
for i in $*
do
    echo "fichier $i :"
    cat $i
done | lpr
```

Choix dans une liste

Syntaxe:

```

case mot in
  exp1) commandes1 ;;
  exp2) commandes2 ;;
  ...
done

```

```

set `date`          # $1 $2 $3 ... = Tue May 21 18:51:04 MET DST 1996



case $1 in
Sun) jour=Dimanche;;
...
Sat) jour=Samedi;;
esac

case $2 in
Jan) mois=Janvier;;
...
Dec) mois=Decembre;;
esac

echo "$jour $3 $mois $7 $4"

```

Alternatives

- Les commandes **Unix** renvoient une valeur qui est un "**compte-rendu**" de l'exécution
- Cette valeur vaut :
 -  **0 ("vrai")** *si la commande s'est exécutée sans erreur*
 -  **une valeur > 0** *qui indique le type d'erreur dans le cas contraire*
- Cette valeur peut être utilisée pour faire des **alternatives** avec la commande **if**

Syntaxe:

```

if commande
then
  commandes-a-faire-si-vrai
else
  commandes-a-faire-si-faux
fi

```

Note: la partie "**else**" est optionnelle

Tests

Syntaxe:

test *expression*

ou

[*expression*]

(Attention: ne pas oublier les blancs !)

Cette **commande Unix** renvoie la **valeur de vérité** d'un test

Options:

-f <i>fichier</i>	<i>fichier</i> existe et n'est pas un répertoire
-d <i>fichier</i>	<i>fichier</i> existe et est un répertoire
-r <i>fichier</i>	<i>fichier</i> existe et est lisible
-s <i>fichier</i>	<i>fichier</i> existe et est de taille non nulle
<i>s1 = s2</i>	les chaînes <i>s1</i> et <i>s2</i> sont identiques
<i>s1 != s2</i>	les chaînes <i>s1</i> et <i>s2</i> sont différentes
<i>n1 -eq n2</i>	les nombres <i>n1</i> et <i>n2</i> sont égaux
<i>n1 -ne n2</i>	les nombres <i>n1</i> et <i>n2</i> sont différents
<i>etc ...</i>	

Exemples:

```
if test $# != 2
then
    echo "usage: $0 file1 file2"
    exit 2
fi
```

```
if cmp -s $1 $2
then
    echo "$1 et $2 sont identiques"
    exit 0
else
    echo "$1 et $2 sont differents"
    exit 1
fi
```

Différence entre:

```
if test $1 = hello; then commande; fi

if test "$1" = hello; then commande; fi

case "$1" in
hello) commande ;;
esac
```

Calcul

Syntaxe:

`expr expression`

Opérateurs:

`+ - * / % | & < > <= >= !=`

Exemple:

```
...
truc='expr $truc + 1'
echo $truc
```

Note:

ne pas oublier les **blancs** et les `'`

Evaluation

<code>\$var</code>	valeur de var ; rien si var indéfinie
<code>\${var}</code>	même chose
<code>\${var-val}</code>	valeur de var si définie, val dans le cas contraire \$var est inchangée
<code>\${var=val}</code>	même chose mais \$var devient val si indéfinie
<code>\${var?mess}</code>	valeur de var si définie, impression de mess sinon
<code>\${var?}</code>	valeur de var si définie, sinon impression de : var: parameter not set
<code>\${var+val}</code>	val si var est définie, rien sinon
eval :	double évaluation
	n=1 eval echo Argument: \$n : "\$n"

Boucles while et until

Syntaxe:

```
while commande
do
    commande-à-faire-tant-que-vrai
done
```

```
until commande
do
    commande-à-faire-tant-que-faux
done
```

Utilisent les "**comptes-rendu**" de l'exécution comme la commande **if**

Exemple:

```
while read line
do
    echo line
done < $1
```

Exemples

```
n=1
for i in $*
do
    echo arg $n : $i
    n=`expr $n + 1`
done
```

```
n=1
while [ $# -gt 0 ]
do
    echo arg $n : $1
    n=`expr $n + 1`
    shift
done
```

```
n=1
while [ $n -le $# ]
do
    eval echo arg: $n : "$n"
    n=`expr $n + 1`
done
```

Divers

break	sortie de for, while, until
continue	itération suivante de for, while, until
exit <i>n</i>	sortie du script en renvoyant la valeur <i>n</i>
read	lecture sur l'entrée standard
echo	affiche ses arguments sur la sortie standard
set	initialisation des variables
unset	effacer des variables

opérateurs || et &&

jouent le rôle de **OU** et de **ET** logiques (même principe qu'en langage C)

```
test -f fichier || echo Fichier inconnu
```

équivalent à:

```
if test ! -f fichier; then echo Fichier inconnu; fi
```

Interruptions

```
trap suite-de-commandes liste-d'interruptions
```

```
...
trap 'rm -f /tmp/fichtemp; exit 1' 1 2 15
...
```

commande vide = ignorer l'interruption
(trap '' 1; commande) &

Interruptions:

- 0 fin du shell
- 1 fin de connexion (^D)
- 2 interruption par ^C
- 3 fin par ^\ (produit un core dump)
- 9 destruction impérative
- 15 terminaison; interruption par défaut de kill

Fonctions

```
usage()
{
    echo Usage: $0 arg1 [arg2 ...]
    exit 1
}

main()
{
    if test $# -le 0; then usage $0; fi

    for i in $*
    do
        echo $i
    done
}

main $*
```

LES FILTRES

Programmes qui :

- ☞ lisent une entrée
- ☞ effectuent une transformation
- ☞ écrivent un résultat (sur la sortie standard)

- grep, egrep, fgrep** : recherche d'une expression dans des fichiers
- sed** : éditeur de flots
- awk** : outil programmable de transformation de texte
- diff, cmp, uniq, tr, dd ...** : outils de comparaison, conversion ... de fichiers

Ces outils utilisent les "expressions régulières"

EXPRESSIONS REGULIERES (Regexp)

- Moyen algébrique pour représenter un langage régulier
- Les expressions régulières
 - ☞ permettent de décrire une **famille de chaînes de caractères**
 - ☞ au moyen de **métacaractères**
- Principe
 - ☞ un "caractère simple" "**matche**" avec lui-même :
 - a *matche avec a*
 - 6 *matche avec 6*
 - ☞ un métacaractère génère ou précise un ensemble de possibilités
 - . *matche avec n'importe quel caractère*
 - ^ *indique un début de chaîne, etc ...*
 - ☞ les métacaractères sont neutralisés par le caractère \

Métacaractères des Regexp

.	⇒	n'importe quel caractère	(? du shell ≡ . des regexp)
*	⇒	répétition du caractère précédent	(* du shell ≡ .* des regexp)
*	⇒	caractère *	
^a	⇒	a en début de chaîne	
a\$	⇒	a en fin de chaîne	
^a\$	⇒	correspond exactement au mot: a	
^.\$	⇒	chaîne d' un seul caractère	
^...\$	⇒	chaîne d' exactement 3 caractères	
...	⇒	chaîne d' au moins 3 caractères	
\.\$	⇒	point en fin de chaîne	

Attention aux confusions avec les métacaractères du Shell :

!!! sens différents pour métacaractères: * . ?

[abc]	⇒	chaîne contenant a ou b ou c
[a-c]	⇒	idem
[a-zA-Z]	⇒	chaîne contenant un caractère alphabétique
[a-z] [0-9]	⇒	chaîne contenant une lettre minuscule suivie d'un chiffre
[^0-9]	⇒	chaîne ne contenant pas de chiffre
^[^0-9]	⇒	chaîne ne contenant pas de chiffre au début
[.]	⇒	chaîne contenant le caractère . (pas d'interprétation)
^[^^]	⇒	chaîne ne commençant pas par ^
/ /	⇒	délimite une chaîne ou une expression régulière
Exemple:		/^[A-Z]..\. /

GREP

grep *expression fichiers ...*

Recherche et imprime les lignes contenant l'expression spécifiée

```
grep From $MAIL
grep From $MAIL | grep -v Marie
who | grep Marie
ls | grep -v temp
```

Options:

- i** ignore distinction minuscules / majuscules
- v** *inverse* : imprime les lignes **ne** contenant **pas** l'expression
- n** affiche le numéro de ligne
- f** lit l'expresion dans un **f**ichier

Remarque: résultat sur sortie standard (utiliser les redirections)

Grep avec expressions régulières

```
grep '^From' $MAIL
ls -l | grep '^d'
```

Attention: *neutraliser l'interprétation des métacaractères par le Shell au moyen de ' '*

Shell+Grep

```
ls [a-Z]* ; ls [a-zA-Z]*
grep -n nom-var *[ch]
grep 'nom-var[1-9]' *[ch]
```

Métacaractères traités par le Shell ou par grep suivant neutralisation

Variantes

fgrep commande optimisée mais simplifiée pour traiter de gros fichiers

egrep grep étendu (équivalent à **grep -E** sur certains systèmes)

Caractères génériques de grep

Rappel:

- `.` ➔ caractère quelconque
- `*` ➔ répétition du caractère précédent :
 - `y*` ➔ autant de `y` que l'on veut (ou aucun)
 - `xy*` ➔ `x xy xyy xyyy`
 - `.*` ➔ n'importe quelle chaîne

Exemple:

trouver les utilisateurs sans password en utilisant :

- `cat /etc/passwd`
- ou : `ypcat passwd`

Sortie :

```
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1:/:
bin:x:2:2:/:usr/bin:
toto:x::3:/:
adm:x:4:4:Admin:/var/adm:
```

Commande : `grep ...`

AWK

awk 'programme-awk' [variable=valeur] fichiers ...
 ou
awk -f fichier-awk [variable=valeur] fichiers ...

- Outil programmable de transformation de texte
- Syntaxe proche du langage C
- Calculs de base, fonctions, tableur programmable
- Intègre la recherche des expressions régulières

Options

- f** fichier-awk : pratique si plusieurs lignes
- Fc** : initialise la variable **FS** (voir ci-après)

Note: AWK = Aho Weinberger Kernighan

Structure d'un programme awk

```
BEGIN      {action0}
regexp1    {action1}
regexp2    {action2}
.....
END        {actionF}
```

Principe

- Initialisation ➞ effectuer *action0*
- Corps ➞ pour **chaque ligne** du texte entré
 - ➞ si *regexp1* est vérifiée effectuer *action1*
 - ➞ si *regexp2* est vérifiée effectuer *action2*
 - etc ...
- Terminaison ➞ effectuer *actionF*

De plus:

- *regexp* omise ➞ *action* toujours effectuée
- *action* omise ➞ affichage de la ligne

Variables et Structure d'une ligne

☞ Chaque *ligne* ("Record") automatiquement séparée en *champs* ("Fields")

☞ séparateur par défaut: blancs et/ou tabulations

NR, NF	numéro de ligne (Record), nombre de champs (Fields)
\$0	contenu de la ligne courante
\$1, \$2 ... \$NF	contenu du ième ... dernier champ
RS, FS	séparateur de lignes (défaut = \n), de champs (defaut = blanc et tab)
ORS, OFS	séparateurs en sortie (pour modifier l'impression)
OFMT	format d'impression des nombres (défaut = %.6g)

Remarque

La variable **FS** peut aussi être initialisée lors de l'appel de **awk** via l'option:

-Fc : le séparateur de champs prend la valeur du caractère **c**

Fichier de données *population*:

```
URSS 8649 275 Asie
Canada 3852 25 Amerique
Chine 3705 1032 Asie
France 211 55 Europe
```

Programme *totpop*:

```
BEGIN {printf("%10s %6s %5s %s\n", "Pays", "Superf", "Pop", "Cont")}
{
    printf ("%10s %6s %5s %s\n", $1, $2, $3, $4)
    superf = superf + $2
    pop = pop + $3
}
END {printf("\n %10s %6s %5s\n", "TOTAL", superf, pop)}
```

Résultat sur *sortie standard*:

```
    Pays Superf   Pop Cont
URSS   8649     275 Asie
Canada 3852      25 Amerique
Chine  3705    1032 Asie
France  211      55 Europe

TOTAL 16417   1387
```

Remarques

- variables et tableaux définis implicitement
- opérateurs de comparaison et de calcul du langage C
- fonctions d'impression et structures de contrôle similaires au C

Programme *maxpop*:

```
{
    if (maxpop < $3) {
        maxpop = $3
        pays = $1
    }
}
END { print "Pays: " pays "/ Max-Pop: " maxpop }
```

Résultat

Pays: Chine / Max-Pop: 1032

Tableaux et Mémoire associative

Programme *contpop*:

```
{ contpop[$4] += $3 }
END {
    for (c in contpop)
        print c, contpop[c]
}
```

Résultat

Europe 55
Amerique 25
Asie 1307

Expressions Régulières

```
$2~/[0-9]+/    { print $1 ": " $3 / 1000 " milliards" }

$1 != prev    { print; prev = $1 }
```

Fonctions et structures de contrôle

if (condition) instruction [**else** instruction]
while (condition) instruction
for (expression; condition; expression) instruction
for (identificateur in tableau) instruction
break, continue, next, exit

print [expr1 [expr2]]
printf (format, expr1, expr2 ...)
sprintf (format, expr1, expr2 ...)

length
length (chaîne)
index (chaîne, caractere)
substr (chaîne, caractere [, long])

sqrt (x)
log (x)
exp (x)
int (x) enlever les répétitions de lignes

AUTRES FILTRES

sort tri alphanumérique
-f ignore majuscules et minuscules
-n tri numérique
-r renverse l'ordre

uniq enlever les répétitions de lignes

diff, comm comparaison de fichiers

tr conversion de caractères dans des fichiers
tr a-z A-Z <in >out

dd conversion format entre systèmes d'exploitation différents
(sert à la lecture de fichiers sur bandes magnétiques ...)

cut extrait des champs dans les lignes d'un fichier

paste concatène les lignes de plusieurs fichiers

DEBOGUEURS

❑ à l'origine: niveau assembleur : **adb**

❑ maintenant: gestion du code source : **débogueurs symboliques** :

❑ variantes suivant systèmes :

⇒ SunOS : **dbx**, **dbxtool** (SunOS4), **debugger** (SunOS5)

⇒ multi-plateformes : **gdb** (GNU), **xxgdb** (X), **mxgdb** (Motif)

Attention:

⇒ *dépendance entre compilateur et débogueur :*

exemple SunOS5: **cc** <-> **dbx** **gcc** <-> **gdb**

⇒ *compiler avec :* **option -g**