Séance No 5

- Cours :
  - caractères de protection
  - regroupements de commandes
  - expressions arithmétiques
  - analyse et interprétation de la ligne de commande
  - comparaison de $* et de $@
  - mise au point des scripts shells
  - structures de contrôles
    - while/do/done
    - case/in/esac
    - application au décodage des options d'un script
  -
- TD: utilisation dans des scripts complexes

caractères de protection

- \: protège le caractère suivant (sauf si c'est un saut de ligne)
- \ en fin de ligne:
- 'chaine': protège tous les caractères de la chaîne
- "chaine": protège tous les caractères sauf $, ` et \ dans le cas où \ est devant $, ", \ et en fin de ligne.
- caractères à protéger :

  | & ; < > ( ) $ ` \ " ' <espace> <tab>

  <saut de ligne> * ? [# ~ = %

•caractères de protection: exemples

```
$ echo 'un test $#\c\$'
un test $#\c\$
$ echo "un test $#\c\$"
un test 0\c$
$ echo un test \$
un test $
$ echo un test \$ \\
un test $ \
```

regroupement de commandes

- () et {}
  - pour rediriger les sorties des commandes vers une même fichier
  - pour exécuter des commandes dans un même contexte
- (): exécution dans un shell enfant
  - cd /;(cd /tmp;ls);pwd
  - (pwd;w)> /tmp/test1.txt
- {}: exécution par le shell courant
  - il faut un espace avant et après chaque accolade et un ; après la dernier commande
  - cd /; { cd /tmp;ls; } ;pwd
  - { pwd; w } > /tmp/test2.txt

expressions arithmétiques

- mécanisme historique: via la commande expr
- mécanisme moderne (SUS V3, POSIX): (( )) et $(( ))

expr:

- l'expression arithmétique doit être passée sous la forme d'arguments séparés par des espaces (représentés ici par _)
  - 1_+_2 : 3
  - 1+2: chaîne 1+2
- les caractères qui peuvent être interprétés par le shell doivent être protégés:
  - * (et pas * seul)
  - \> (et pas > seul)
  - ...

expr: exemples

```
$ expr 2 + 5
7
$ expr 2 * 3
syntax error
$ expr 2 \* 3
6
$ x=1
$ y=$( expr $x + 3 )
$ echo $y
4
```

expr: quelques opérateurs

opérateur	sens
n1 + n2	
n1 - n2	
n1 * n2	
n1 / n2	
n1 % n2	modulo
\>, \<, \>=, \<=, =, !=	opérateurs de comparaison
chaine : expression_rationnelle	comparaison
\(expression \)	regroupement
s1 \& s2	vrai si s1 et s2 sont non nulles (valeur 0 et chaîne nulle)
s1 | s2	vrai si s1 ou s2 est non nulle

commande (( ))

- syntaxe: ((expression)) ou let expression
- avantages par rapport à expr:
  - plus grand choix d'opérateurs (ceux du C)
  - les caractères spéciaux du shell peuvent ne pas être protégés
  - les noms de variables peuvent ne pas être préfixé par $
  - les affectations se font directement dans la commande

(( )): quelques opérateurs

opérateur	sens
n1 + n2	
n1 - n2	
n1 * n2	
n1 / n2	
n1 % n2	modulo
>, <, >=, <=, ==, !=	opérateurs de comparaison
(expression)	regroupement
s1 && s2	ET logique
! e1	négation de e1
s1 || s2	OU logique
~, >>, <<, &, |, ^	travaillant sur les bits
+=, -=, *=, ...	n+=e équivaut à n=n+e

(( )) : exemples

```
$ x=1
$ ((y=x+1))
$ echo $y
2
$ z=7
$ z*=y
$ echo $z
14
$ if (( (x==1) && (y>10) ))
  then
```

substitution d'expressions arithmétiques

- syntaxe: $((expression arithmétique))
- Exemple:

```
$ echo 1 + 2 vaut $(( 1+2))
1 + 2 vaut 3
$ x=1
$ echo x vaut maintenant $(( x=x+1))
x vaut maintenant 2
$  echo essai ((1+2))
bash: syntax error near unexpected
  token `(('
$ echo essai $( ((1+2)) )
essai
```

Interprétation de la ligne de commande

- séparation en mots
- expansion des accolades
- expansion du ~
- expansion des paramètres, des variables
- expansion des commandes
- évaluation des expressions arithmétiques
- découpage des mots
- développement des noms de fichiers
- suppression des " \ ' non protégés ne résultant pas d'un développement

découpage des mots

- découpage de tout ce qui n'est pas entre guillemets
- les caractères séparateurs sont dans la variable IFS (par défaut: espace, tab, retour chariot)
- les arguments explicitement nuls (" ou "") sont conservés. Ceux résultant du développement d'un paramètre sans valeur sont éliminés

Développement

- des accolades :
  - {mot1, mot2, mot3, ...}
  - file{2,34,IMP} donne file2 file34 fileIMP
- du tilde:
  - ~nomLogin
  - ~petit
- paramètres/variables:
  - ${parametre}ou $parametre
  - $!var: on remplace !var par la valeur de var avant de faire l'expansion (indirection)

Développement des commandes

- $(commande)
- ou `commande` (ancienne forme déconseillée)
- la sortie standard de la commande remplace l'expression
- ls -l $(grep -il toto *.c)

développement des expressions arithmétiques

- $((expression))
- l'expression est traitée comme si elle se trouvait entre guillemets (sans traitement spécifique pour le guillemet)
- l'expression subit le développement des paramètres/variables et des commandes
- x=1; echo nouvelle valeur $((x=x+3))

développement des noms de fichiers

- *: n'importe quelle séquence
- ?: un caractère quelconque
- [abcdef]: un caractère parmi tous les caractères entre crochet
- [a-f]: tous les caractères compris entre a et f
- [- : le caractère -
- [^ ou [!: tout sauf les caractères qui suivent ^ ou !

$* et $@

- contiennent la liste des arguments d'un script ou d'une fonction
- $* ou $@équivalentes si non entourées de guillements "
- "$@" : les espaces internes aux arguments sont protégés
- "$*": les espaces internes aux arguments ne sont pas protégés
- "$@" sert traditionnellement quand un script peut avoir des noms de fichiers contenant des espaces comme paramètres.

$*: exemple

```
cat exemple.sh
#! /bin/bash

for i in $*
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a
Program
Files
test
```

"$*": exemple

```
cat exemple.sh
#! /bin/bash

for i in "$*"
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a Program Files test
```

"$@": exemple

```
cat exemple.sh
#! /bin/bash

for i in "$@"
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a
Program Files
test
```

mise au point des scripts

- La mise au point est un gros point faible des shells
  - pas de débugger
  - portée dynamique des variables
  - pas de compilation: les erreurs de syntaxe sont détectées quand le flot d'exécution les atteint
- les outils utilisés pour la mise au point sont minimaux et archaïques
  - des options du shell permettant un affichage verbeux lors de l'exécution
  - afficher la valeur de variables à l'aide de commandes echo ou printf
  - mettre des points d'arrêt en lançant des sous-shell

option du shell: -e

- set -e: interrompt le shell à la première commande dont le code de retour est non nul
- Exemple: que se passe-t-il si le dossier n'existe pas ?

```
#! /bin/bash
cd /var/spool/aSupprimer
rm -rf *
```

- c'est une bonne pratique d'intégrer un 'set -e' à tous ses scripts.

option du shell: -x

- set -x: affiche les commandes telles qu'elles sont exécutées (après analyse de la ligne de commande et substitutions)
- Exemple:

```
$cat ex2.sh
#! /bin/bash
set -x
x=1
echo $x $(( 1+3)) t{a,b} *
$ ls
F3  ex2.sh  f1  f2
petit@ns:/tmp/Test$ ./ex2.sh
+ x=1
+ echo 1 4 ta tb F3 ex2.sh f1 f2
1 4 ta tb F3 ex2.sh f1 f2
```

option du shell: -v

- set -v : affichage des commandes avant substitution
- on reprend l'exemple précédente en remplaçant « set -x » par « set -v »:

```
$ ./ex2.sh
x=1
echo $x $(( 1+3)) t{a,b} *
1 4 ta tb F3 ex2.sh f1 f2
```

option du shell: -n

- set -n: lecture des commandes sans les exécuter (pour détecter des erreur de syntaxe)
- Exemple:

```
$ cat ./ex2.sh
#! /bin/bash
set -n
x=1
echo coucou
echo $x $(( 1+3) t{a,b} *
$ ./ex2.sh
./ex2.sh: line 6: unexpected EOF while looking
  for matching `)'
./ex2.sh: line 7: syntax error: unexpected end
  of file
```

structures de contrôle: case/in/esac

- syntaxe:

```
case mot in
   modele1) liste de commandes;;
   modele2|modele3|modele4) liste de
    commandes;;
   ...
esac
```

- les modèles sont des chaînes incluant éventuellement des caractères spéciaux * ? []^ (utilisés dans la substitution des noms de fichiers)

structures de contrôle: case/in/esac

- principe
  - la valeur du mot est comparée à chaque modèle
  - la liste de commandes du premier modèle correspondant est exécutée jusqu'au « ;; »
  - l'exécution se poursuit ensuite après le « esac ».

case/in/esac: exemples

```
$ cat codePostal.sh
#! /bin/bash

case "$1" in
  75[0-9][0-9][0-9])
    echo "code postal parisien"
    ;;
  7[78][0-9][0-9][0-9]|9[1-5][0-9][0-9][0-9])
    echo "code postal ile de France"
    ;;
  [0-9][0-9][0-9][0-9][0-9])
    echo "code postal de la france metropolitaine"
    ;;
  *)
   echo "code postal non reconnu"
   ;;
esac
```

structures de contrôles: while/do/done

- syntaxe

```
while commande1
do
  liste de commandes
done
```

- « commande1 » est exécutée,
  - si le code de retour vaut 0, la liste de commande est exécutée et on retourne exécuter « commande1 »
  - si le code de retour de « commande 1 » est différent de 0, la commande qui suit le « done » est exécutée.

while/do/done: exemples

```
$ cat facto.sh
#! /bin/bash
n=5
x=1
while (( n > 0))
do
  ((x*=n))
  ((n-=1))
done
echo "12!=$x"

$ ./facto.sh
12!=120
```

while/do/done: exemples

```
$ cat test.sh
#! /bin/bash
n=1
while (( $# > 0 ))
do
  echo "parametre No $n: $1"
  ((n=n+1))
  shift
done

$ ./test.sh a g df "ze rt"
parametre No 1: a
parametre No 2: g
parametre No 3: df
parametre No 4: ze rt
```

commande READ

- syntaxe: read var1 var2 ...
- lit son entrée standard et affecte le premier mot lu à var1, le deuxième à var2, ... le texte restant est affecté à la dernire variable
- Exemple:

```
echo hello world et salut les tepos
  read a b c
```

  - a contient hello
  - b contient world
  - c contient « et salut les tepos »

commande READ: utilisation

- 2 utilisations typiques:
  - pour mettre dans une variable la saisie d'un utilisateur
  - en liaison avec while :

```
while read a b
do
  traitement de a et b
done
```

  - la boucle s'arrête en fin de fichier (caractère Ctrl-D si sur une saisie utilisateur)

READ : exemple

- on suppose que le fichier nombres.txt contient deux entiers par ligne

```
$ cat test-read.sh
#! /bin/bash
while read a b
do
  echo $a + $b = $(( a+b ))
done
$ ./test-read.sh < nombres.txt
1 + 3 = 4
6 + 4 = 10
2 + -5 = -3
```