Plan

- Rappels sur la séance précédente
- les caractères de protection: ' " \
- les expressions arithmétiques entières avec (( ))
- les options et techniques de mise au point
- les structures de contrôle case et while
- la commande read

Exemple

- considérons le script suivant:

```
#! /bin/sh
echo "1ere commande"
maPremiereFonction(){
    echo "coucou depuis maPremiereFonction"
}
echo "2e commande"
maPremiereFonction
echo "après l'appel de maPremièreFonction"
```

- son exécution provoque l'affichage suivant:

```
1ere commande
2e commande
coucou depuis maPremièreFonction
après l'appel de maPremièreFonction
```

paramètres: exemple

- script: test.sh

```
#! /bin/bash
f1 () {
    echo "1er argument de f1: $1"
    echo "deuxieme argument de f1: $2"
    echo "nombre d'arguments de f1: $#"
    echo "tous les arguments de f1: $*"
    echo "nom du script: $0"
}
echo début du script
echo "1er argument du script: $1"
echo "deuxieme argument du script: $2"
echo "nombre d'arguments du script: $#"
echo "tous les arguments du script: $*"
echo "nom du script: $0"
f1 quand est-ce "qu'on" mange
```

paramètres: exemple

- execution du script: « ./test.sh salut les tepos »

```
début du script
1er argument du script: salut
deuxieme argument du script: les
nombre d'arguments du script: 3
tous les arguments du script: salut les tepos
nom du script: test
f1 quand est-ce "qu'on" mange#! /bin/bash
1er argument de f1: quand
deuxieme argument de f1: est-ce
nombre d'arguments de f1: 4
tous les arguments de f1: quand est-ce qu'on mange
nom du script: ./test.sh
```

code de retour d'une fonction

- comme toute commande, une fonction a un code de retour
- commande « return n»: termine la fonction et retourne le code d'erreur « n ».
- Erreur à ne pas faire: *return* ne retourne pas le résultat de la fonction mais son code d'erreur

structures de contrôle: for

- syntaxe:

  for *var* in *listeValeur*

  do

  commande1

  ...

  done

- sémantique:
  - la variable *var* prend chaque valeur de la liste de valeur
  - les commandes situées entre do et done sont exécutées pour chaque valeur de la variable *var*

Exemple 1: liste de valeurs citée

```
$for i in 1 2 3 4
 do
    echo $i
done
1
2
3
4
```

Exemple 2: liste des valeurs contenues dans une variable:

```
$ varTest="a b c d"
$for i in $varTest
do
   echo $i
done
a
b
c
d
```

Exemple 3: liste des valeurs via substitution de commande

```
$for i in $(ls)
do
  echo $i
done
fichier1
fichier2
... (résultat de la commande ls, un fichier par ligne)
```

Exemple 4: liste des valeurs via génération de noms de fichiers

```
$for i in f*
do
  echo $i
done
fichier1
fichier2
... (les noms des fichiers commençant par f dans le dossier courant, un fichier par ligne)
```

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: -u

- set -u: l'utilisation de la valeur d'une variable non définie est considérée comme une erreur.
- L'utilisation de ce paramètre supprime de nombreuses erreurs dues à des fautes de frappe
- Exemple:

```
$ echo $aaa
-bash: a: unbound variable
$ aaa=12
$ echo $aaa
12
```

•expressions rationnelles: éléments communs

- les caractères suivants sont communs aux expressions rationnelles de base et étendues:
  - ^: début de ligne
  - $: fin de ligne
  - .: un caractère quelconque
  - [liste]: un caractère de la liste (idem [] du shell, ^: négation)
  - *: 0 à n fois le caractère/expression précédente
  - \<, \>: chaîne vide de début et de fin de mot
  - \c: protège le caractère c de tout interprétation. ex.: \$: caractère $

•Exemples:

- petit: la chaîne petit
- ^petit: chaîne démarrant par petit
- petit$: chaîne finissant par petit
- DSCN[0-9]*\.JPG: DSCN suivi de 0 à N chiffres suivi de .JPG (noter la protection du caractère .)
- ^$: chaîne vide (utile pour détecter les lignes vides)
- ^[^:]*:[^:]*:[^:]*:1000 une chaîne contenant des champs séparés par le caractère : et dont le 4e champ vaut 1000. Appliqué sur un fichier /etc/passwd, sélectionne les lignes dont le groupe est 1000.

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
```

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
  - => pas au programme de l'examen
- mécanisme moderne (SUS V3, POSIX): (( )) et $(( ))
  - => au programme de l'examen

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
```

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
```

commande find

- find permet de chercher récursivement les fichiers vérifiant une ou plusieurs conditions
- outre les expression simples, l'expression que doit vérifier un fichier peut être de la forme (par priorité décroissante) :
  - (expression)
  - ! expression
  - expression1 -a expression2 : ET logique
  - expression1 expression2: ET logique
  - expression1 -o expression2: OU logique

commande find

- expressions élémentaires à argument numérique:
  - +n: toutes les valeurs supérieures ou égales à n
  - -n: toutes les valeurs inférieures ou égales à n
  - n: n exactement
  - par la suite, partout où on verra un argument numérique n, on pourra utiliser +n, n ou -n
  - exemples:
    - -size 1024k: les fichiers de taille égale à 1024 Ko
    - -size -1024k: les fichiers de taille inférieure égale à 1024 Ko
    - -size +1024k: les fichiers de taille supérieur ou égale à 1024 Ko

commande find: quelques expressions élémentaires

- quelques expressions élémentaires:
  - -name motifProtégé: les fichiers vérifiant le motif
  - -size n: les fichiers de taille n
  - -mtime n, -ctime n, -atime n
  - -perm p avec p ayant la forme numérique ou symbolique des arguments de chmod
  - -type c avec c=b,c,d (dossier),l (lien symbolique),p,f (fichier ordinaire),s
  - -user u
  - -group g
  - -link n : nombre de liens physique sur le fichier
  - -print: provoque l'affichage des noms des fichiers vérifiant l'expression (par défaut sur le Gnu find)
  - ...

commande find: -exec

- -exec commande;
  - pour chaque fichier trouvé, la commande est exécutée.
  - si {}apparaît parmi les arguments de la commande, il est remplacé par le nom du fichier trouvé
- -exec commande arguments {} +
  - syntaxe POSIX/SUSv3 (standard mais pas disponible sur toutes les plateformes)
  - les noms des fichiers trouvés sont accumulés dans une liste l
  - la commande est exécutée une seule fois, à la fin de la recherche et {}est remplacé par la liste des arguments

commande find : exemples

- fichiers ordinaire nommés core de plus de 1024Ko
  - find -size +1024k -type f -name core -print
- fichier ordinaires de l'utilisateur petit ou fichiers ordinaires de taille supérieure à 1024 Ko et de nom core
  - find -type f \(-user petit -o \(-size +1024k -name core \) \) -print
- fichier dont le nom commence par C
  - find -name c* -print

commande find: -exec et les espaces (&Co)

- rm est une commande qui ne lit pas sur son entrée standard. 3 méthodes pour effacer un ensemble de fichiers sélectionnés par find :
  - on lance un rm par fichier (lourd)
    - find -name *.bak -exec rm -f {} \;
  - syntaxe POSIX: un seul rm global est lancé :
    - find -name *.bak -exec rm -f {} \+
  - une solution avec les options spécifiques de Gnu find:
    - find -name *.bak -print0 |xargs -0 rm -f
  - ne marche pas avec les noms de fichiers contenant des caractères à problème (espace, saut de ligne, ', etc.):
    - find -name *.bak -print |xargs rm -f

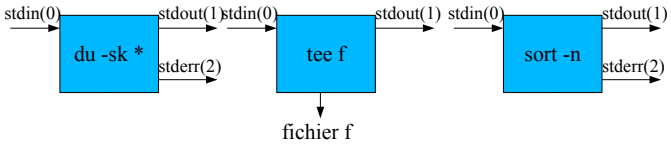
commande xargs

- xargs [options] commande
- xargs rassemble ce qu'elle reçoit sur son entrée standard dans une liste l et exécute « commande l »
- xargs s'utilise avec des commandes qui n'acceptent pas de données sur leur entrée standard
- Exemple:
  - grep -l perso * | xargs chmod 700

xargs : options utiles

- -p: prompt mode. une confirmation est demandée à l'utilisateur pour chaque invocation de la commande
- si le nombre ou la taille des arguments transmis via l'entrée standard est important, il est possible d'indiquer à xargs d'exécuter plusieurs fois la commande avec une liste limitée:
  - -n nombre: la commande est invoquée plusieurs fois et chaque invocation a au plus nombre arguments
  - -s taille: la commande est invoquée plusieurs fois et chaque invocation fait au plus s octets

commande tee



- la commande tee envoie simultanément son entrée standard vers un fichier et vers sa sortie standard.
- options:
  - -a: ajoute au fichier
  - -i : ignore le signal

de la bonne utilisation de la commande test et des valeurs des variables

- "$var" vs $var
- les bonnes pratiques concernant l'utilisation de la commande test :
  - [-n "$a"] && ["$a" = "$b"] au lieu de [-n "$a" -a "$a" = "$b"]
  - toujours mettre les variables entre apostrophes ": "$var" et pas $var
  - utiliser les operateurs arithmetiques pour les nombres et literaux pour les chaines
  - utiliser ["x$A" = "aexpression"] pour comparer la valeur de variables (pour éviter des problèmes si $A est vide)
  - Les operateurs -e, -nt, -ot, -N, -L, -h, >, <, ne sont pas portables. -e, -h et -L sont POSIX toutefois. -h et -L sont raisonablement portables

analyse des options de ligne de commande

- 2 méthodes:
  - utiliser la commande interne getopts qui permet de gérer des options d'une lettre
  - à la main : boucle while/case/shift (avantage: on peut utiliser des noms longs pour les options)

getopt

- une option est composée d'une lettre précédée d'un + ou d'un -
- une option peut être suivie d'un argument
- exemple: getopt "abc:d" var
  - les options possibles sont -a, -b, -c, -d
  - le « : » après le « c » indique que l'option -c attend un paramètre
  - var est le nom de la variable dans laquelle sera mis le nom de l'option lue
  - l'argument éventuel sera mis dans la variable réservée OPTARG
  - getopt s'utilise dans une boucle while

getopt: gestion des erreurs

- lorsque getopt rencontre une option invalide:
  - la variable var contient le caractère ?
  - un message est affiché à l'écran
- si : est en première position dans la liste des options:
  - aucun message ne sera affiché à l'écran
  - OPTARG contiendra la valeur de l'option fautive
  - exemple: getopt `":abc:d" var`
- paramètre manquant:
  - var contient : et OPTARG contient le caractère de

getopt

- Exemple:

```
while getopt "ac:" option
do
  case "$option" in
    a) echo option a
       ;;
    c) echo option a avec l'argument
  "$OPTARG"
       ;;
    \?) echo option invalide
       ;;
  esac
done
```

gestion artisanale des arguments

```
while [ $# -gt 0 ]; do
  case "$1" in
    -Prefix) shift
             [ $# -gt 0 ] || { affErr; exit 1; }
             Prefix="$1"
             shift
             ;;
    -remoteComputer) shift
            [ $# -gt 0 ] || { affErr; exit 1; }
            remoteComputer="$1"
            shift
            ;;
    *) repASauver="$1"
       shift
       ;;
  esac
done
```

Bilan

- A la fin de cette cinquième séance, vous devez savoir utiliser:
  - les caractères de protection: ' " \
  - les expressions arithmétiques entières avec (( ))
  - les options et techniques de mise au point
  - les structures de contrôle case et while
  - la commande read