

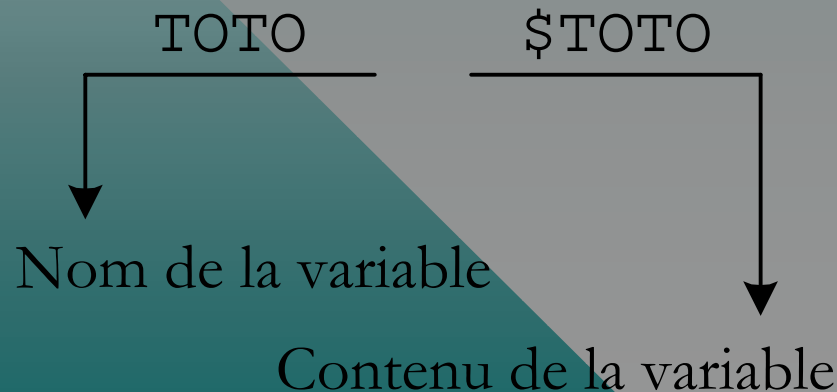
Systemes d'exploitation

Chapitre 7-4

Substitution: variables et commandes,
paramètres de Bourne shell,
éléments de programmation

Substitution: variables

- Une variable est identifiée par son nom.
- Le contenu de la variable est identifié par le symbole \$ placé devant le nom.
- Par exemple:



Substitution: variables

- L'interpréteur de commande Bourne shell réalisera la *substitution du contenu d'une variable* lorsqu'il rencontre le symbole \$ suivi d'un nom de variable.
- Deux comportements possibles:

| Résultat | Situation |
|-------------------------|---|
| substitution vide | La variable n'est pas définie ou la variable est définie mais son contenu est vide. |
| Substitution du contenu | La variable est définie et son contenu est non nul. |

Substitution: variables

➤ Exemples de substitution des variables:

```
$ MSG1="Jean est un "  
$ MSG2="chien fort réputé"  
$ echo "$MSG1 $METIER $MSG2"  
Jean est un   chien fort réputé
```

```
$ MSG1="Jean est un "  
$ MSG2="chien fort réputé"  
$ METIER=  
$ echo "$MSG1 $METIER $MSG2"  
Jean est un   chien fort réputé
```

Même résultat

```
$ MSG1="Jean est un "  
$ MSG2="chien fort réputé"  
$ METIER="dresseur de"  
$ echo "$MSG1 $METIER $MSG2"  
Jean est un   dresseur de chien fort réputé
```

Substitution: commandes

- L'interpréteur de commande Bourne shell est en mesure de substituer le résultat d'une ligne de commandes UNIX.
- Le symbole impliqué dans ce genre de substitution est l'accent grave (').
- Exemple: *pwd (print working directory)*

```
$ echo pwd
pwd
$ echo `pwd`
/export/home/exa/personnel/wong/DEMO
$ echo "Mon répertoire de travail est: `pwd`"
Mon répertoire de travail est: /export/home/exa/personnel/wong/DEMO
```

Substitution: commandes

- Il est possible d'assigner le résultat d'une ligne de commande UNIX à une variable.
- Voici comment:

Assignation du résultat des commandes UNIX à des variables

```
$ REPERTOIRE=`pwd`
$ JESUIS=`who am i`
$ MACHINE=`hostname`
$ echo "Utilisateur: $JESUIS\nRépertoire de travail: $REPERTOIRE\nMachine: \
> $MACHINE"
Utilisateur: wong          pts/5          fév 26 17: 22          (ppp40.etsi.fr)
Répertoire de travail: /export/home/extra/personnel/wong/DEMO
Machine: centi
```

Symbole anti-slash indiquant la continuation de la ligne de commande à la ligne suivante

Paramètres de Bourne shell

➤ Paramètres de position:

```
$ cmd par 1 par 2 par 3 par 4
```

Ce sont des paramètres de position.

Dans un programme Bourne shell, le contenu de ces paramètres de position est représenté par:

\$1, \$2, \$3 jusqu'à \$9.

Il s'agit du nom d'un fichier de commande Bourne shell. Le nom du fichier de commande est représenté par \$0.

Paramètres de Bourne shell

➤ Exemple de paramètres de position à l'aide d'un programme Bourne shell:

```
centi 24> cat param
#!/bin/sh
# Nom du fichier param
# param montrer l'utilisation des paramètres Bourne shell

# Paramètres de position
# Lancer de fichier de commande : param -A -B -C
echo "Numero PID de l'interpréteur de commande: $$"
echo "Nom du fichier de commande: $0"
echo "Nombre de paramètres: $#"
```

```
centi 25> param -A -b -C
Numero PID de l'interpréteur de commande: 17235
Nom du fichier de commande: param
```

```
Nombre de paramètres: 3
Parametre 1: -A
Parametre 2: -b
Parametre 3: -C
Parametre 4:
Toute la ligne de commande: -A -b -C
```

Instructions du programme param. Observez l'utilisation des paramètres de Bourne shell

Résultat obtenu

Paramètres de Bourne shell

➤ Les variables systèmes:

- ◆ Ces variables sont pré-définies par le Bourne shell et sont disponibles pour les programmes.

| Variable | Signification |
|----------|---|
| * | Contient les paramètres entrés dans la ligne de commandes lors de l'invocation du fichier de commandes. |
| 0 | Nom du fichier de commande. |
| n | Valeur du n ^e paramètre de la ligne de commande. |
| @ | Le contenu de la ligne de commande. |
| # | Le nombre de paramètres dans la ligne de commandes lors de l'invocation du fichier de commandes. |
| ? | L'état de terminaison (un numéro) de la dernière commande exécutée. |
| \$ | Le numéro du processus PID correspondant à l'interpréteur de commande. |
| ! | Le numéro du processus PID correspondant à la dernière tâche exécutant à l'arrière plan. |

Utiliser le symbole \$ pour obtenir leur contenu

Paramètres de Bourne shell

➤ Les variables systèmes (suite):

| Variable | Signification |
|----------|--|
| HOME | Répertoire de travail de l'utilisateur. |
| PATH | Chemins de fouille pour trouver une commande, un programme. Les chemins sont séparés par le caractère : . |
| MAIL | Chemin indiquant la boîte de courriers de l'utilisateur. |
| PS1 | Le caractère représentant le curseur. Par défaut, le caractère \$ joue le rôle de curseur. |
| PS2 | Lorsqu'une commande s'étend sur plus d'une ligne, l'interpréteur de commande affiche un second caractère pour indiquer qu'il faut continuer l'entrée. Par défaut c'est le caractère > qui est contenu dans cette variable. |
| IFS | Les caractères qui jouent le rôle de séparateur des commandes. En plus des caractères espace et TAB, l'interpréteur considère les caractères contenus dans IFS comme séparateurs supplémentaires. |
| CDPATH | Le contenu de cette variable modifie le comportement de la commande <code>cd(1)</code> . Voir le manuel en-ligne pour les détails de cette commande. |
| TZ | Cette variable indique la zone horaire (Time Zone) utilisée. |

Utiliser le symbole \$ pour obtenir leur contenu

Lecture et affichage

- La commande `read(1)` réalise la lecture à partir de l'entrée standard:

```
read var1 var2 var3
```

Lecture de l'entrée standard et placer les données dans les variables `var1`, `var2` et `var3`.

La séparation des données d'entrée en champs est réalisée par le Bourne shell à l'aide de la variable `IFS` (*Internal Field Separator*).

voici une-ligne de donnees

Il existe 4 champs.

Lecture et affichage

➤ Exemple d'utilisation de read(1):

```
cent i 6> cat lecture
#!/bin/sh
# nom du fichier: lecture
# lecture: montrer comment lire une donnée à partir de l'entrée standard

echo "Les répertoires de l'installation? \c"
read REPERTOIRE1 REPERTOIRE2 REPERTOIRE3
echo "Merci !"
echo "L'entrée lue: $REPERTOIRE1\n$REPERTOIRE2\n$REPERTOIRE3"
```

- La lecture est réalisée à partir de l'entrée standard.
- Les données lues sont placées dans trois variables (REPETOIRE1, REPETOIRE2 et REPETOIRE3).
- Le programme termine son exécution par l'affichage des données lues.

Lecture et affichage

```
centi 34> lecture
Les repertoires de l'installation? /toto
Merci !
L'entree lue: /toto
centi 35> lecture
Les repertoires de l'installation? /toto /tata /minou
Merci !
L'entree lue: /toto
/tata
/minou
centi 36> lecture
Les repertoires de l'installation? /toto /tata /minou /coco /loulou
Merci !
L'entree lue: /toto
/tata
/minou /coco /loulou
centi 37> lecture
Les repertoires de l'installation? /toto /tata /minou /coco \
/loulou
Merci !
L'entree lue: /toto
/tata
/minou /coco /loulou
```

Les résultats obtenus. Observer bien la façon dont les données sont assignées aux variables.

Décision et bouclage

- Instruction if - elsif - fi
- Voici un exemple:

```

1  #!/bin/sh
2  # Nom du fichier affiche
3  # affiche: démontrer l'utilisation de l'instruction if
4
5  if [ $# = 0 ]      # aucun paramètre ...
6  then
7      echo "Usage: affiche [-v] NonFichier" 1>&2
8      echo "      -v utilise la commande more" 1>&2
9      exit 1
10 fi
11
12 if [ "$1" != "-v" ]
13 then
14     cat "$@"        # pas d'option -v alors c'est le nom du fichier
15 else
16     shift          # débarrasser l'option -v
17     more "$@"
18 fi

```

Instruction if - fi

**Instruction
if - else - fi**

L'instruction if teste le statut de retour des commandes.
La logique: vrai = 0, faux ≠ 0 (différent de zéro).

Décision et bouclage

- Dans le programme précédent, nous avons utilisé la commande `shift(1)`.
- Voici une explication de son rôle:

```

$@ → -v toto.txt lolo.txt
    $1 $2          $3
    |
    | shift
    |
$@ → toto.txt lolo.txt
    $1          $2
    |
    | shift
    |
$@ → lolo.txt
    $1
    |
    | shift
    |

```

La commande `shift(1)` agit sur les paramètres de position de Bourne shell.

Décision et bouclage

- Nous pouvons tester un ensemble de conditions à l'aide de if(1).
 - ◆ Voir les notes de cours à la page 124.
- Voici un exemple:

```

1 #!/bin/sh
2 # Nom du fichier : iftest
3 # iftest: démontrer l'utilisation des options et opérateur de if et de test
4
5 # voir si le fichier d'action terminal.dt existe ...
6 NOMFICH=$HOME/.dt/types/Terminal.dt
7 if [ -f $NOMFICH ]
8 then
9     # voir s'il possède les permissions lecture ou écriture mais pas d'exécution
10    if [ \( -r "$NOMFICH" -o -w "$NOMFICH" \) -a ! -x "$NOMFICH" ]
11    then -----
12        echo "$NOMFICH accessible en lecture ou écriture"
13    fi
14 else
15    echo "$NOMFICH n'existe pas"
16 fi
    
```


Décision et bouclage

- Instruction for - done
- Cette instruction sert à boucler:
 - ◆ sur les paramètres de position
 - OU
 - ◆ sur une liste de paramètres donnée directement dans le programme
- Attention! L'instruction for - done **ne ressemble pas** à celle des langages procéduraux (ex: Basic, Pascal, C, etc.).

Décision et bouclage

- Voici un exemple de son utilisation à l'aide d'une liste de paramètres donnés explicitement dans le programme:

```

1 #!/bin/sh
2 # Nom du fichier : forttest
3 # forttest: démontrer l'utilisation de l'instruction for
4
5 # voir si les fichiers ftp.dt, frp.mpm ftp.t.pm et ftp existent...
6 ACTION=$HOME/.dt/types/ftp.dt
7 ICONm=$HOME/.dt/icons/ftp.mpm
8 ICONt=$HOME/.dt/icons/ftp.t.pm
9 LIEN=$HOME/ftp
10 # Bouclage
11 for nomfich in "$ACTION" "$ICONm" "$ICONt" "$LIEN"
12 do
13     if [ -f $nomfich ]
14     then
15         echo "$nomfich existe"
16     else
17         echo "$nomfich n'existe pas"
18     fi
19 done
    
```

Liste de paramètres
donnés directement à
l'instruction for - done

Décision et bouclage

➤ Un exemple utilisant les paramètres de position:

```

1 #!/bin/sh
2 # Nom du fichier : forttest2
3 # forttest2: montrer l'utilisation de l'instruction for itérant sur les
4 # paramètres de position d'une ligne de commande
5
6 # voir si les fichiers donnés à la ligne de commande existent...
7 if [ $# != 0 ]
8 then
9     # bouclage sur les paramètres de position
10    for nomfich
11    do
12        if [ -f $nomfich ]
13        then
14            echo "$nomfich existe"
15        else
16            echo "$nomfich n'existe pas"
17        fi
18    done
19 else
20    echo "Pas de paramètres de position"
21 fi

```

Pour obtenir le même résultat que l'exemple précédent, vous devez lancer le programme de la manière suivante:

```

forttest2 \ $HOME/.dt/types/ftp.dt\
$HOME/.dt/types/ftp.m.pm \
$HOME/.dt/types/ftp.t.pm\
$HOME/ftp

```

Décision et bouclage

- Instruction case - esac
- Il s'agit d'un ensemble de if - elsif - fi mais organisé d'une manière structurée.
- La syntaxe de cette instruction ressemble à ceci:

```
case var in
  desc r i pt eur 1 [| desc r i pt eur 2] ) commandes ;;
esac
```

- var est le contenu d'une variable.
- descripteur est une expression générique simple.

Décision et bouclage

- Voici la liste des descripteurs de case - esac:

| descripteur | Signification |
|-------------|---|
| * | une chaîne de caractères quelconque incluant la chaîne vide. |
| ? | un caractère quelconque excluant le caractère vide. |
| [.] | un caractère parmi ceux placés entre crochets (OU-logique implicite). |
| [a- z] | un caractère parmi la plage des caractères spécifiée entre crochets (OU-logique implicite). |
| [! .] | tous les caractères excluant ceux placés entre crochets. |
| x | un ou plusieurs caractère donnés explicitement. |

Décision et bouclage

➤ Voici un exemple qui peut nous aider à mieux comprendre cette instruction:

```

1  #!/bin/sh
2  # Nom du fichier : casesac
3  # casesac: montrer l'utilisation de l'instruction case - esac
4
5  # traiter les options d'une commande
6  if [ $# = 0 ]
7  then
8      echo "Usage : casesac -t -q -l NonFich"
9      exit 1
10 fi
11
12 # utiliser case - esac pour traiter les options
13 for option
14 do
15     case "$option" in
16         -t) echo "option -t reçu" ;;
17         -q) echo "option -q reçu" ;;
18         -l) echo "option -l reçu" ;;
19         [!-]*) if [ -f $option ]
20             then
21                 echo "fichier $option trouve"
22             else
23                 echo "fichier $option introuvable"
24             fi
25         *) echo "option inconnue $option recontree"
26     esac
27 done

```

Voici une façon (simpliste) qui permet le décodage des options reçues d'un programme. Quel est le rôle du descripteur [!-]* et celui de *

Décision et bouclage

- Voici un exemple de descripteur capable de détecter la présence d'un code postal dans une variable:
 - ◆ Code postal:

lettre-chiffre-lettre-chiffre-lettre-chiffre

Ex: H1L4C9
 - ◆ Descripteur:

[A-Z][0-9][A-Z][0-9][A-Z][0-9]

Décision et bouclage

- Instruction while - do
- La syntaxe de cette instruction est:

```
while [ commandes-test ]
do
    commandes
done
```
- Il y aura bouclage tant et aussi longtemps que le statut de retour de commandes-test est **vrai** (égal à 0).

Décision et bouclage

➤ Voici un exemple d'utilisation:

```

1 #!/bin/sh
2 Nom du fichier: whiletest
3 whiletest: montrer l'utilisation de la boucle while
4
5 #Boucler et demander à l'utilisateur un répertoire QUI
6 #doit exister.
7
8 REPERTOIRE=""
9 while [ ! -d "$REPERTOIRE" ]
10 do
11     echo "Entrer un nom de répertoire existant: \c"
12     read REPERTOIRE
13 done
14 echo "Merci !"
    
```

On doit comprendre que le programme bouclera tant et aussi longtemps que le nom de répertoire donné n'est pas un répertoire valide.

Décision et bouclage

- Instruction until - do
- La syntaxe de cette instruction est:

```
until [ commandes-test ]
do
    commandes
done
```
- Il y aura bouclage tant et aussi longtemps que le statut de retour de commandes-test est **faux** (non nul).

Décision et bouclage

➤ Voici un exemple d'utilisation:

```
#!/bin/sh
# Nom du fichier: untiltest
# untiltest: montrer l'utilisation de la boucle until

# Boucler et demander à l'utilisateur un repertoire qui
# doit exister.

REPertoire=""
until [ -d "$REPertoire" ]
do
    echo "Entrer un nom de repertoire existant: \c"
    read REPertoire
done
echo "Merci !"
```

On voit très bien que l'instruction until - do est le complément de while - do.

Nous avons simplement éliminé la négation ! devant le test -d.

Fonctions Bourne shell

- Nous pouvons rendre la programmation plus structurée en utilisant des fonctions.
- La syntaxe est:

```
NomDeFonction ( )  
{  
    commandes  
}
```
- Une fonction Bourne shell joue le rôle d'une sous-routine.

Fonctions Bourne shell

➤ Un résumé des caractéristiques :

◆ Syntaxe:

NomDeFonction () { commandes }

- ◆ définition des fonctions Bourne shell: au début du fichier de commande;
- ◆ prend préséance sur les commandes systèmes de même nom;
- ◆ peut avoir une valeur de retour: exit n où n est une valeur numérique (=0 → OK, ≠0 → Erreur).

Fonctions Bourne shell

```

1  #!/bin/sh
2  # Nom du fichier: functest
3  # functest: montrer l'utilisation des fonctions Bourne shell
4
5  # Boucler et demander a l'utilisateur un repertoire qui
6  # doit exister.
7
8  # repertoire () : demande un nom de repertoire et verifie son existence
9  repertoire () {
10     echo "Entrer un nom de repertoire: \c"
11     read REPERTOIRE
12     if [ -d "$REPERTOIRE" ]
13     then
14         return 0 # repertoire existe
15     else
16         return 101 # repertoire inexistant
17     fi
18 }
19
20 # gestion_erreur () : Afficher un message d'erreur selon ERRNO
21 gestion_erreur () {
22     case $ERRNO in
23         0) ;; # pas d'erreur
24         101) echo "Répertoire inexistant" ;;
25         102) echo "Permission d'écriture obligatoire";;
26         *) echo "Code d'erreur inconnu"
27             exit 1
28             ;;
29     esac
30 }
31
32 # Programme principal
33 ERRNO=123
34 while [ $ERRNO -ne 0 ]
35 do
36     repertoire; ERRNO=$? # statut de sortie de repertoire () assigné à ERRNO
37     gestion_erreur # invoquer le gestionnaire d'erreur
38 done

```

Définition d'une fonction

Définition d'une fonction

Utilisation des fonctions

Fonctions Bourne shell

➤ Fonction et paramètres de position:

Programme prog

Fonct i onA

Fonct i onB

Reste du programme

prog -l -q / Par ano

Disponible aux fonctions par passage de paramètres seulement:

Funct i onA "\$@"

Funct i onB "\$@"

Disponible au programme directement sous forme de \$*, @\$1, \$2, etc.

Neutralisation des caractères

- Certains caractères ont des significations particulières pour l'interpréteur de commandes.
 - ◆ Par exemple: `&`, `(`, `)`, `*`, `!`, `{`, `}`, etc.
- ◆ Cependant, à cause du nombre limité de caractères, certaines commandes et programmes réutilisent ces mêmes caractères mais à d'autres fins.
 - ◆ Par exemple: `&&` (ET-logique), `*` (multiplication), `!` (négation), etc.

Neutralisation des caractères

- Sans un mécanisme d'échappement, ces caractères spéciaux seront interprétés par le Bourne shell.
- Les commandes et programmes qui utilisent ces caractères spéciaux ne pourront pas s'exécuter correctement.
- D'où la nécessité de neutraliser la signification particulière de ces caractères spéciaux pour le Bourne shell.

Neutralisation des caractères

- Voici un exemple:
- Nous désirons afficher la chaîne de caractère "TOTO & TATA". Voici le résultat:

```

1 $ echo TOTO & TATA
2 TOTO
3 6821
4 TATA: Introuvable
5 $ echo TOTO \& TATA
6 TOTO & TATA
7 $
    
```

Le méta-caractère & est neutralisé par le symbole \

Le caractère & est interprété par Bourne shell comme une demande d'exécution en arrière-plan

Neutralisation des caractères

- Donc, le symbole \ permet la neutralisation du caractère qui le suit.
- Nous pouvons neutraliser la signification spéciale du caractère Espace par les symboles " " et ' '.
- Le guillemet: élimine la signification spéciale du caractère Espace mais permet la substitution des variables et commandes.
- L'apostrophe: élimine la signification spéciale du caractère Espace et empêche la substitution des variables et commandes.

Neutralisation des caractères

- Voici un exemple qui aide à la compréhension de la neutralisation des caractères:

```

0 $ MACHINE=`host name`
1 $ echo La machine \' $MACHINE\' est en panne
2 La machine 'cent i' est en panne
3 $ echo La machine \"$MACHINE\" est en panne
4 La machine 'cent i' est en panne
5 $ echo La machine '$MACHINE' est en panne
6 La machine $MACHINE est en panne
7 $ echo "La machine '$MACHINE' est en panne"
8 La machine 'cent i' est en panne
9 $ echo 'La machine '$MACHINE' est en panne'
10 La machine cent i est en panne
11 $
    
```

Commandes `exec(1)` et `trap(1)`

- Dans un programme Bourne shell, la commande `exec(1)` permet l'exécution d'une commande sans la création d'un nouveau processus.
- Vous pouvez donc passer des paramètres du programme à la commande exécutée.
- Attention, ce n'est pas un appel de sous-routine car Le contrôle n'est pas retourné au programme !

Commandes exec(1) et trap(1)

➤ Voici un exemple:

```
1  #!/bin/sh
2  # Nom du fichier exect est
3  # exect est: montrer l'effet de la commande exec
4
5  pwd          # afficher le répertoire courant
6  exec date    # exécuter la commande date(1) et lui passer le contrôle
7  echo "Fin du programme" # jamais exécutée à cause de exec date
```

➤ Voici le résultat de son exécution:

```
cent i 6> exect est
/ export / home/ exa/ personnel / wong/ CHAPI TRE6
di manche, 27 février 2000, 18:04:54 EST
cent i 7>
```

- On voit que la ligne echo "Fin du programme" n'est jamais exécutée!
- La commande exec(1) sert aussi à rediriger les entrées et les sorties standards.

Commandes `exec(1)` et `trap(1)`

➤ La commande `trap(1)` est fort utile pour la programmation Bourne shell.

➤ Sa syntaxe est:

`trap 'commandes' signaux`

où `commandes` est un ensemble de commandes UNIX (les apostrophes sont nécessaires pour neutraliser les caractères spéciaux des commandes)

où `signaux` sont des numéros (entiers)

Commandes `exec(1)` et `trap(1)`

- À la réception d'un signal listé, la commande `trap(1)` exécutera les commandes entre apostrophes et termine le programme Bourne shell.
- Il s'agit donc d'une forme de communication asynchrone entre les programmes (ou entre un programme et son terminal).
- On peut considérer les signaux comme des interruptions logicielles (comparaison grossière).

Commandes `exec(1)` et `trap(1)`

➤ Voici quelques numéros de signal:

| Numéro du signal | Condition |
|------------------|---|
| 2 | Interruption du terminal par les touches <code>ctrl-c</code> .. |
| 3 | Quitter par les touches <code>ctrl- </code> ou <code>ctrl-\</code> . |
| 9 | Tuer. Ce signal ne peut être capturé. |
| 15 | Terminaison. Valeur par défaut de la commande <code>kill (1)</code> . |
| 18 | Changement d'état du processus. |

Commandes `exec(1)` et `trap(1)`

- Voici un exemple utilisant `trap(1)`:

```

1  #!/bin/sh
2  # Nom du fichier trap est
3  # trap est: montrer la capture des signaux
4
5  trap 'exit 1' 1 2 3 15 # capturer ces signaux
6  trap '/bin/rm/tmp/* 2> /dev/null' 0 # exécuter à la fin du programme
7  while :
8  do
9      echo trap est en execution
10 done
    
```

- Lancer le programme et appuyer sur les touche `ctrl-c`.
- Le programme se terminera par la commande `exit` et provoque le signal 0.
- La 2e commande `trap(1)` est alors exécutée.