



LES SYSTÈMES D'EXPLOITATION

LA PROGRAMMATION AVEC LE SHELL/UNIX

🎓 3A - Cursus Ingénieurs - Dominante Informatique et Numérique
🏛️ CentraleSupélec - Université Paris-Saclay - 2025/2026



Idir AIT SADOUNE
idir.aitsadoune@centralesupelec.fr

PLAN

- Les variables dans le shell
- L'interprétation d'une commande
- Les scripts Shell
- Les structures de contrôle

[Retour au plan](#) - [Retour à l'accueil](#)

PLAN

- Les variables dans le shell
- L'interprétation d'une commande
- Les scripts Shell
- Les structures de contrôle

[Retour au plan](#) - [Retour à l'accueil](#)

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
 - ➡ non déclarées → définies à la première initialisation
 - ➡ non typées a priori → tout est **chaîne de caractères**
 - ➡ pas d'héritage par les processus fils
- **Syntaxe** de l'affectation → **variable=valeur** (sans espaces autour du **=**)
- **Référence** à la valeur de la variable → **\$variable** ou **\${variable}**
- **Exemples** → déclaration, initialisation et affichage du contenu des variables

```
$ alpha=toto
$ b=35
$ c2=3b
$ echo alpha, b, c2 contiennent ${alpha}, ${b}, ${c2}
alpha, b, c2 contiennent toto, 35, 3b

$ set | grep alpha # set affiche les valeurs de toutes les variables
alpha=toto
```

EVALUATION DES EXPRESSIONS

- Evaluation d'une expression ? **non typées** → tout est **chaîne de caractères**

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

- Il existe **deux méthodes** pour faire de l'arithmétique avec le **Shell**
 - utiliser la commande **expr** ou l'opérateur **\$()**
 - les arguments de **expr** doivent être séparés par des espaces

```
$ b=35
$ expr $b + $b
70
```

```
$ b=35
$ bb=$((b+b))
$ echo ${bb}
70
```

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- **export** → étend la portée d'une variable aux processus du **Shell** courant
 ⇒ rendre la **variable globale** dans le **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**

- **-p** → liste des variables exportées dans le shell courant
- **-n** → supprime la variable de la liste exportée
- **-f** → exporte une fonction

- **Exemple**

```
$ export JAVA_HOME="/usr/local/jdk"
$ export -p
...
HOME=/Users/idir.ait-sadoune
SHELL=/bin/zsh
...
JAVA_HOME=/usr/local/jdk
```

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont des variables dynamiques héritées et **utilisées** par les différents processus d'un système d'exploitation
- Quelques **variables d'environnement** :
 - **SHELL** → interpréteur de commandes utilisé (**bash**, **zsh** ...)
 - **TERM** → type de terminal utilisé (**vt100**, **xterm**, ...)
 - **HOME** → répertoire d'accueil
 - **USER** → identifiant (nom) de l'utilisateur
 - **PATH** → liste des chemins de recherche des commandes
- Quelques **commandes** :
 - **env** → lister les valeurs des **variables d'environnement**.
 - **set** → lister les valeurs de toutes les **variables** définies dans la session.

LES VARIABLES D'ENVIRONNEMENT

LA VARIABLE PATH

- PATH → liste des chemins de recherche des commandes

```
$ echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

- Deux méthodes d'exécution de commandes/exécutables :
 1. par le nom du script/le nom de la commande :
la recherche se fait dans les répertoires listés dans la variable PATH
 2. par un chemin explicite vers le script/la commande :
la recherche se fait uniquement dans le répertoire indiqué dans le chemin

```
$ mon_script.sh
```

```
$ ./mes_bins/mon_script.sh
```

INITIALISATION DU SHELL

- A l'ouverture d'une session (ex. via un terminal ou une connexion SSH), l'OS démarre un programme de type **shell** (comme **bash**, **zsh**, ou **sh**).
- Selon le type de session, le **shell** lit différents **fichiers d'initialisation** pour configurer l'environnement.
 - variables (**PATH**, **HOME**, etc.), alias (**ll**, **la**, etc.), fonctions personnalisées, etc.
- **Session de login (connexion complète)**
 - **/etc/profile** → paramètres globaux pour tous les utilisateurs
 - **~/.bash_profile**, **~/.profile**, ... → paramètres personnels de l'utilisateur
- **Shell interactif non-login** (ex. : ouvrir un nouveau terminal graphique)
 - **~/.bashrc**
- **Exemple** de modifications à mettre dans **~/.profile**

```
PATH="${HOME}/Scripts:${PATH}" # ajouter au PATH le répertoire ~/Scripts
PATH=".:${PATH}" # ajouter aussi le répertoire courant
export PATH
```

PLAN

- Les variables dans le shell
- L'interprétation d'une commande
- Les scripts Shell
- Les structures de contrôle

[Retour au plan](#) - [Retour à l'accueil](#)

L'INTERPRÉTATION D'UNE COMMANDE

- L'interprétation d'une commande → analyse et exécution une commande.
- Le processus passe par quatre étapes :

1. Analyse lexicale et syntaxique

- Le **shell** découpe la ligne de commande en mots et symboles (**tokens**) selon les espaces, les guillemets, et les opérateurs spéciaux (`|`, `>`, `&&`, etc.).

2. Substitutions → avant l'exécution, le **shell** remplace :

- les variables (`$HOME`, `$USER`, etc.)
- les commandes imbriquées (entre `` ... `` ou `$(...)`)
- les caractères génériques (globbing, comme `*.txt`)
- les expressions arithmétiques (`$((2+3))`)

3. Exécution → une fois la commande complètement interprétée, le **shell** :

- cherche le programme correspondant dans les répertoires du `$PATH`
- crée un processus fils pour l'exécuter

4. Redirections et pipelines

- Le **shell** gère les entrées/sorties (`|`, `>`, `<`, etc.)

LES MÉTACARACTÈRES

- Pour éviter d'exposer **les métacaractères** à l'interprétation par le **Shell**, on peut utiliser les **protections** suivantes :
 - `\` : protection individuelle du caractère suivant (**backslash**)
 - `'...'` : protection forte (aucune interprétation)
 - `"..."` : protection faible (substitution de paramètres/commandes)
- **Exemples**
 - affectation d'une chaîne comportant des blancs à une variable

```
$ v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3
```

- lecture du contenu d'une variable ou d'une commande

```
$ echo ${TERM} "${TERM}" '${TERM}'  
xterm-256color xterm-256color ${TERM}  
  
$ echo Je suis $(whoami) - "Je suis $(whoami)" - 'Je suis $(whoami)'  
Je suis idiraitsadoune - Je suis idiraitsadoune - Je suis $(whoami)
```

SUBSTITUTION DE COMMANDE

- `$(commande)` → récupération de **la sortie standard d'une commande**
 - ➡ chaîne de caractères stocké dans **une variable**
 - ➡ repris comme **argument** d'une autre commande
- **Utilisations** → sauvegardes, dans des **variables**, les résultats de :
 - ➡ calculs arithmétiques avec la commande `expr`
 - ➡ commandes dans le `shell-scripts`
- **Exemples**

```
$ qui=$(whoami)
$ echo "je suis ${qui}. On est $(date)"
je suis idiraitsadoune. On est Mar 17 oct 2023 05:45:25 CEST
```

```
$ s1=$(expr 12 + 2)
$ s2=$((($((12+2))+1))
$ echo "12 + 2 = ${s1}, (12 + 2) + 1 = ${s2}"
12 + 2 = 14, (12 + 2) + 1 = 15
```

PLAN

- Les variables dans le shell
- L'interprétation d'une commande
- Les scripts Shell
- Les structures de contrôle

[Retour au plan](#) - [Retour à l'accueil](#)

FICHIERS DE COMMANDES

- **Shell-scripts** → fichier (**.sh**) contenant une suite de **de commandes Shell**
- Il existe **trois méthodes** permettant d'exécuter un **Shell-scripts** :
 1. utiliser **la commande bash**

```
$ bash script.sh
```

2. **rendre le script exécutable** et utiliser **le chemin vers le script**

```
$ chmod +x script.sh
$ cp script.sh mes_bins/Scripts
$ mes_bins/Scripts/script.sh
```

3. **rendre le script exécutable** et le mettre dans un **dossier du PATH**

```
$ chmod +x script.sh
$ cp script.sh ~/Scripts # le dossier ~/Scripts doit être ajouté au PATH
$ script.sh
```

FICHIERS DE COMMANDES

LE PREMIER SCRIPT

Récupérer des informations sur l'utilisateur et sur la machine

```
1#!/bin/bash
2# file name : my_script.sh
3echo "nous sommes le $(date)"
4echo "mon login est $(whoami)"
5echo "mon calculateur est $(hostname)"
```

Exécution en passant par la commande **bash**

```
$ bash my_script.sh
nous sommes le Mar 17 oct 2023 06:40:29 CEST
mon login est idiraitsadoune
mon calculateur est MBP-de-Idir
```

LES PARAMÈTRES D'UN SCRIPT

Les **variables positionnées** lors du lancement d'une commande :

- **\$0** → nom du fichier de la commande (spécifié lors de l'appel)
- **\$1, \$2, ..., \$9, \${10}, ...** → les paramètres (arguments) de la commande
- **\$*** → chaîne formée par les paramètres d'appel ("\$1 \$2 \$3 ...")
- **\$@** → liste des paramètres d'appel ("[\$1, "\$2", "\$3", ...])
- **\$#** → nombre de paramètres lors de l'appel
- **\$\$** → numéro du processus lancé (**pid**)
- **\$?** → le code d'erreur de la dernière commande exécutée

LES PARAMÈTRES D'UN SCRIPT

EXEMPLES UTILISANT LES PARAMÈTRES

Récupérer des informations sur le script lui même et le processus associé

```
1#!/bin/bash
2# file name : prog.sh
3echo "la procedure $0 a ete appelee avec $# parametres"
4echo "le premier parametre est $1"
5echo "la liste des parametres est $*"
6echo "le numero du processus lance est $$"
```

Exécution en passant par la commande **bash**

```
$ bash prog.sh p1 p2 p3
la procedure ./prog.sh a ete appelee avec 3 parametres
le premier parametre est p1
la liste des parametres est p1 p2 p3
le numero du processus lance est 2960
```

UN EXEMPLE D'UN SCRIPT

Concatener deux fichiers (\$1 et \$2) dans le fichier \$3

```
1 #!/bin/bash
2 file_in1=$1
3 file_in2=$2
4 file_out=$3
5
6 echo '-----' > $file_out
7 echo \| $file_in1 \| >> $file_out
8 echo '-----' >> $file_out
9 cat $file_in1 >> $file_out
10
11 echo '-----' >> $file_out
12 echo \| $file_in2 \| >> $file_out
13 echo '-----' >> $file_out
14 cat $file_in2 >> $file_out
15
16 echo "le processus de concaténation est terminé"
17 exit 0
```

PLAN

- Les variables dans le shell
- L'interprétation d'une commande
- Les scripts Shell
- Les structures de contrôle

[Retour au plan](#) - [Retour à l'accueil](#)

LA COMMANDE `test`

- La commande `test` permet de vérifier les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code `0` ou `1` (vrai ou faux) qui est sauvegardé dans la variable `$?`.
- La commande `test` propose deux syntaxes équivalentes :

```
$ test expression
```

```
$ [expression]
```

- Consultez [ce lien](#) pour découvrir la commande `test`
- Consultez [ce lien](#) pour comparer `test` à `[[...]]`

LA STRUCTURE CONDITIONNELLE

if ... then ... fi

- L'**instruction if** permet d'exécuter des actions/des opérations **si** une condition est vérifiée.

```
if condition
  then instruction(s)
fi
```

- L'**instruction if** peut aussi inclure une **instruction else** pour exécuter des actions/des opérations **si la condition n'est pas vérifiée**.

```
if condition
  then instruction(s)
else instruction(s)
fi
```

LA STRUCTURE CONDITIONNELLE

if ... then ... fi

EXEMPLE

Vérifier si un utilisateur est connecté

```
1#!/bin/bash
2if who | grep "^$1 "
3  then
4    echo "$1 est connecté"
5  else
6    echo "$1 n'est pas connecté"
7  fi
```

```
$ ./prog.sh idiraitсадoune
idiraitсадoune    console      18 oct 07:22
idiraitсадoune    ttys001     18 oct 07:40
idiraitсадoune est connecté
```

LA STRUCTURE CONDITIONNELLE

if ... then ... fi

UN AUTRE EXEMPLE

Vérifier le nombre de paramètres d'une commande

```
1#!/bin/bash
2if test $# -eq 0
3    then
4        echo "commande lancée sans paramètres"
5else
6    echo "commande lancée avec au moins un paramètre"
7fi
```

```
$ ./prog.sh p1 p2 p3
commande lancée avec au moins un paramètre
```

```
$ ./prog.sh
commande lancée sans paramètres
```

LA STRUCTURE CONDITIONNELLE

if ... then ... fi

UNE AUTRE SYNTAXE

- Il est possible d'imbriquer des **if** dans d'autres **if**
- Pour alléger le **if** imbriqué, le **shell ksh** fournit une autre syntaxe

```
if condition1
  then instruction(s)
else
  if condition2
    then instruction(s)
  else
    if condition3
      then instruction(s)
    ...
  fi
fi
fi
```

```
if condition1
  then instruction(s)
elif condition2
  then instruction(s)
elif condition3
  then instruction(s)
elif condition4
  then instruction(s)
elif condition5
  then instruction(s)
...
fi
```

LA STRUCTURE CONDITIONNELLE

if ... then ... fi

UN EXEMPLE DU **if** IMBRIQUÉ

Vérifier si une commande a des paramètres avant de l'utiliser

```
1#!/bin/bash
2if test $# -eq 0
3  then
4    echo "Relancer la cmd en ajoutant un parametre"
5else
6  if who | grep "^$1 " > /dev/null
7    then
8      echo "$1 est connecté"
9    else
10      echo "$1 n'est pas connecté"
11  fi
12fi
```

LA STRUCTURE CONDITIONNELLE

case ... in ... esac

ÉNUMÉRATION DE MOTIFS

case compare une valeur avec une liste de valeurs et exécute des instructions si une des valeurs de la liste correspond.

```
case valeur_testee in
  valeur1) instruction(s);;
  valeur2) instruction(s);;
  valeur3) instruction(s);;
  valeur4) instruction(s);;
  ...
  * ) instruction_else(s);;
esac
```

Exemple avec des expressions régulières

```
1 #!/bin/bash
2 echo "ecrirez OUI"
3 read reponse
4 case ${reponse} in
5   OUI)           echo "bravo"
6   [0o][Uu][Ii]) echo "trouvé";;
7   o*[0*)        echo "raté de peu";;
8   n*[N*)        echo "raté";;
9   *)            echo "complètement raté"
10
11
12 esac
```

LA STRUCTURE ITÉRATIVE

for . . . do . . . done

- La boucle **for** permet de parcourir une liste de valeurs, elle effectue un nombre d'itérations connu à l'avance.

```
for variable [in liste_valeurs]
    do instruction(s)
done
```

- La boucle **for** possède une deuxième syntaxe :

```
for ((exp1;exp2;exp3))
    do instruction(s)
done
```

- commence par exécuter **exp1**, puis tant que **exp2 $\neq 0$** le bloc d'instructions est exécuté et **exp3** également.

LA STRUCTURE ITÉRATIVE

for ... do ... done

EXEMPLE I

Parcourir une liste de valeurs

```
1 #! /bin/bash
2 for mot in 1 5 10 2 "la fin"
3 do
4     echo "mot vaut ${mot}"
5 done
```

```
$ ./my_prog.sh
mot vaut 1
mot vaut 5
mot vaut 10
mot vaut 2
mot vaut la fin
```

LA STRUCTURE ITÉRATIVE

for ... do ... done

EXEMPLE II

Parcourir les paramètres d'une commande

```
1 #! /bin/bash
2 for param in "$*"
3 do
4     echo "-${param}-"
5 done
```

```
$ ./my_prog.sh a b c d
-a-
-b-
-c-
-d-
```

LA STRUCTURE ITÉRATIVE

for ... do ... done

EXEMPLE III

Parcourir une liste de fichiers correspondant à un motif

```
1 #!/bin/bash
2 for fichier in *.f90
3 do
4     echo "fichier ${fichier}"
5 done
```

```
1 #!/bin/bash
2 motif=$1
3 for fic in $(grep -l ${motif} *)
4 do
5     echo "le fichier $fic contient le motif $motif"
6 done
```

LA STRUCTURE ITÉRATIVE

while ... do ... done

- La boucle **while** exécute un bloc d'instructions **tant qu'une certaine condition** est satisfaite

```
while condition
    do instruction(s)
done
```

- Exemple → un script qui boucle jusqu'à ce qu'un utilisateur se déconnecte

```
1#!/bin/bash
2utilisateur=$1
3while who | grep "^${utilisateur} " > /dev/null
4do
5    echo "${utilisateur} est connecté"
6    sleep 2
7done
8echo "${utilisateur} n'est pas connecté"
```

LA STRUCTURE ITÉRATIVE

until ... do ... done

- La boucle **until** est exécutée tant qu'une certaine condition est fausse.

```
until condition
  do instruction(s)
done
```

- Exemple → un script qui boucle jusqu'à ce qu'un utilisateur se connecte

```
1#!/bin/bash
2utilisateur=$1
3until who | grep "^\${utilisateur} " > /dev/null
4do
5  echo "\${utilisateur} n'est pas connecté"
6  sleep 2
7done
8echo "\${utilisateur} est connecté"
```

LA STRUCTURE ITÉRATIVE

LA COMMANDE **continue**

- La commande **continue** saute les commandes suivantes dans la boucle et **reprend l'exécution en début de boucle**

```
continue
```

- La commande **continue n** sort des **n-1** boucles les plus intérieures et **reprend au début de la n^{ième} boucle.**

```
continue n
```

- A Utiliser dans un bloc conditionnel pour **court-circuiter les instruction de la fin de boucle.**

LA STRUCTURE ITÉRATIVE

LA COMMANDE **continue** - EXEMPLE

Afficher les 4 premières lignes d'un fichier s'il est lisible

```
1 #!/bin/sh
2 for fic in *.sh
3 do
4     echo ****
5     echo "< fichier ${fic} >"
6     if [ ! -r "${fic}" ] # tester si le fichier existe et est lisible
7         then
8             echo "fichier ${fic} non lisible"
9             continue # sauter la commande head
10    fi
11    head -n 4 ${fic}
12 done
13
14 exit 0
```

LA STRUCTURE ITÉRATIVE

LA COMMANDE **break**

- La commande **break** permet de **sortir d'une boucle** avant sa fin.

```
break
```

- La commande **break n** permet de **sortir des n boucles** les plus intérieures.

```
break n
```

- Nécessaire dans **les boucles infinies**.
 - ➡ **while true** ou **until false**
 - ➡ insérée dans **un bloc conditionnel** pour arrêter la boucle.

LA STRUCTURE ITÉRATIVE

LA COMMANDE **break** - EXEMPLE

Répéter une boucle jusqu'à ce qu'une valeur soit lue

```
1 #!/bin/bash
2 while true # boucle infinie
3 do
4     echo "entrer un chiffre (0 pour finir)"
5     read i
6     if [ "$i" -eq 0 ]
7     then
8         echo '** sortie de boucle par break'
9         break # sortie de boucle
10    fi
11    echo "vous avez saisi $i"
12 done
13 echo "fin du script"
14
15 exit 0
```

LA COMMANDE `exit`

- `exit` arrête l'exécution du script et rend un `statut` (0 par défaut) au processus appelant (accessible via la variable `$?`)

```
exit [statut]
```

- Utilisé également pour arrêter le traitement en cas d'erreur
➡ rendre alors un `statut` $\neq 0$

```
1#!/bin/bash
2if [ $# -lt 1 ] # test sur le nb d'arguments
3then
4    echo "il manque les arguments" >&2 # sur la sortie d'erreur
5    exit 1 # sortie avec code d'erreur
6fi
```

MERCI

[Version PDF des slides](#)

[Retour à l'accueil](#) - [Retour au plan](#)