

# Chapitre 6: Scripts shell

INF1070  
Utilisation et administration des systèmes informatiques

Jean Privat & Alexandre Blondin Massé

Université du Québec à Montréal

Hiver 2019

Notes

---

---

---

---

---

---

---

---

---

---

## Plan

- 1 Programmes et scripts
- 2 Script shell
- 3 Groupes et séquences de commandes
- 4 Plus de redirections
- 5 Développement et substitutions
- 6 Paramètres et variables
- 7 Code de retour
- 8 Structures de contrôle

Notes

---

---

---

---

---

---

---

---

---

---

## Programmes et scripts

Notes

---

---

---

---

---

---

---

---

---

---

## Programme

Ensemble d'instructions destinées à être exécutées par un ordinateur

### Programmes binaires

- Programme en **langage machine**
- Spécifique à une **architecture** (famille de processeurs)
- Nécessite souvent un environnement d'exécution spécifique (bibliothèques, système d'exploitation)

### Programmes scripts

- Programme en **langage de script**
- Sous forme **textuelle**
- Exécutés par un **interpréteur** (un autre programme)

**Script shell** → Programme destiné à être interprété par le shell

Notes

---

---

---

---

---

---

---

---

---

---

## Langages de script

### Spectre large

- Spécifique à un domaine (DSL)
- Généraliste
- Langage glue / langage d'extension

### Rapide à apprendre et à utiliser

- Concis (moins de verbosité)
- Typage dynamique (moins de structures et de contraintes)

### Environnement inclus

- Interprétés plutôt que compilés
- L'efficacité n'est pas un objectif

Notes

---

---

---

---

---

---

---

---

---

---

## Langages de script plus ou moins populaires



Python, PHP, JavaScript, R, Ruby, Perl, Visual Basic, Bash

- INF3190 Introduction à la programmation Web
- INF600A Langages de script et langages dynamiques
- INF5190 Programmation Web avancée
- INF1035 Informatique pour les sciences (pas pour les informaticiens)

Notes

---

---

---

---

---

---

---

---

---

---

## Programmes sous Unix

### Un programme = un fichier exécutable

```
$ file /bin/bash /usr/bin/vimtutor
/bin/bash: ELF 64-bit LSB pie executable
/usr/bin/vimtutor: POSIX shell script
```

### Droits nécessaires

- x pour les programmes binaires
- x et r pour les programmes scripts

### Question

- Peut-on exécuter un lien symbolique ?
- Peut-on exécuter un programme sans les droits x (mais avec r) ?

Notes

---

---

---

---

---

---

---

---

---

---

## Fichiers accessoires

La plupart des programmes ont besoin de fichiers supplémentaires

- Bibliothèques partagées binaires (.so) et/ou script
- Ressources (images, son, configuration, etc.)
- Autre programmes supplémentaires

Un programme échouera si ces fichiers manquent

Notes

---

---

---

---

---

---

---

---

---

---

## Gestionnaire de paquets

- L'approche recommandée pour installer un logiciel sous Unix
- Installe et gère les programmes et fichiers accessoires
- Gère les versions et les dépendances entre paquets
- Permet de désinstaller **proprement**
- Exemples: apt, dnf, pacman

De nombreux langages de programmation fournissent aussi des gestionnaires de paquets spécifiques à leurs écosystèmes

- Exemples: pip, npm, cabal

Notes

---

---

---

---

---

---

---

---

---

---

## Exécuter un programme



### Chemin absolu ou relatif

Avec au moins un slash « / »

- `/opt/INF1070/hello`
- `script/hello`
- `./hello` → exécute un programme du répertoire courant

### Nom simple

Le **nom** d'un fichier d'un des répertoires d'exécutables

Habituellement au moins `/bin`, `/usr/bin` et `/usr/local/bin`

- `type` affiche des informations sur le type de commande (POSIX)
- `which` localise une commande dans l'environnement courant
- `whereis` recherche une commande à d'autres endroits habituels
- `echo "$PATH"` pour voir les répertoires d'exécutables (les détails plus tard)

Notes

---

---

---

---

---

---

---

---

---

---

## Script ou binaire ?

- `/usr/bin/ts` (*timestamp*) horodate l'entrée standard (extra)
- `/usr/bin/chromium` navigateur web de Google (extra)
- `/usr/bin/ldd` lister les bibliothèques partagées nécessaires (extra)

Tester avec `file` ou ouvrir avec `vim`

### Shebang

Tout fichier texte lisible et exécutable (`r-x`) peut être exécuté

- Les premiers caractères sont « `#!` » (*shebang*)
- Ça indique le **chemin absolu** de l'interpréteur à utiliser
- Sans shebang, le défaut est souvent `/bin/sh`

```
$ cat hello
#!/bin/sh
echo Heeeeeelllllloooooo!!!!!!
$ ./hello
Heeeeeelllllloooooo!!!!!!
```

Notes

---

---

---

---

---

---

---

---

---

---

## Autres exécution de scripts



### Script en ligne

Option `sh -c` exécute les instructions passées en argument

```
$ dash -c "echo hello | rev"
```

Utile pour profiter des fonctionnalités du shell quand une simple commande est attendue

### Sourçage

La commande interne « `.` » (point) exécute un script dans le processus courant

```
$ . ./niceprompt
```

Utile pour importer des bibliothèques et fichiers de configuration

Notes

---

---

---

---

---

---

---

---

---

---

## Script shell

Notes

---

---

---

---

---

---

---

---

## Objectif du cours

- S'initier à la programmation shell
- Voir plusieurs fonctionnalités avancées du shell

### Contenu

La suite décrit des fonctionnalités avancées et utiles du shell.

- Elles sont supportées par les shell POSIX (incluant `dash` et `bash`).
- Les extension spécifiques à `bash` (appelées **bashismes**) sont signalées.

### Bibliographie

De nombreuses fonctionnalités (et variations) ne sont pas décrites.

Pour une information plus complète, consultez

- `bash`
- Shell POSIX
- Advanced Bash-Scripting Guide (traduction en français)

Notes

---

---

---

---

---

---

---

---

## Pourquoi programmer en shell ?

- Universel: C'est toujours disponible et utilisé un peu partout
- Glue: Ça permet de facilement faire coopérer des programmes
- Efficace: Car la syntaxe est très concise et expressive
- Neutre: Ça fâche tout le monde pareil

### Mais

- Ça requiert une connaissance spécialisée
- C'est risqué: le naïf fait facilement des bogues (voir INF600C)
- C'est bizarrement portable (POSIX vs. `bash` vs. utilitaires)
- Les structures de données et de contrôle sont très pauvres
- C'est lent

Notes

---

---

---

---

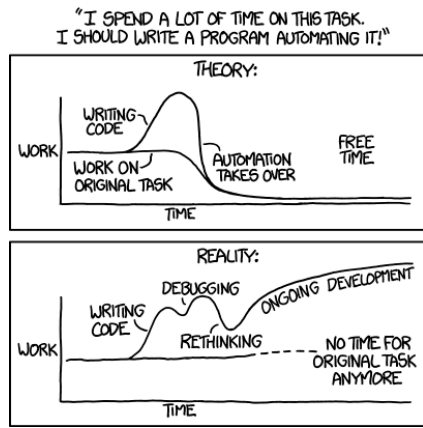
---

---

---

---

## Automatisation



Source: <https://xkcd.com/1819/> (2014)

Notes

---

---

---

---

---

---

---

---

---

---

## Commandements



- Tu feras seulement des petits scripts
- Tu respecteras les bonnes pratiques
- Tu protégeras tous tes développements de chaînes
- Tu chercheras la simplicité et éviteras les commandes superflues
- Tu maintiendras ton code clair et lisible
- Tu éviteras la programmation shell si ce n'est pas adapté
- Tu garderas tes *oneliners* fragiles pour la ligne de commande
- Tu ne regretteras pas de devenir bon en shell

Il y a beaucoup de pièges

- Newbie traps
- How to do things safely in bash
- Bash pitfalls

Notes

---

---

---

---

---

---

---

---

---

---

## Exécution de scripts

- Un script shell s'exécute ligne par ligne

```
#!/bin/sh
echo un
grep -x '[dm][ei][ur][xz]' /usr/share/dict/french
echo kagjl | rev | tr a-z i-z
```

- Les lignes vides sont ignorées
- Le carré « # » marque le début d'un commentaire

```
#!/bin/sh

# Dit « Bonjour »
echo Bonjour # Salut
```

Notes

---

---

---

---

---

---

---

---

---

---

## Attention à l'échec

Quand une commande échoue le script continue (par défaut)

```
#!/bin/sh
cd /tnp/reterpoire
rm -rf *
echo "Tout est effacé!"
```

On exécute...

```
$. /efface
bash: cd: /tnp/reterpoire: Aucun dossier de ce type
Tout est effacé!
$ ls -l
total 0
```

## Meilleure solution

```
#!/bin/sh
rm -rf /tnp/reterpoire/*
```

Notes

---

---

---

---

---

---

---

---

---

---

## Groupes et séquences de commandes

Notes

---

---

---

---

---

---

---

---

---

---

## Séquences de commandes

Le point virgule « ; » enchaîne les commandes

```
$ echo bonjour;echo le monde
bonjour
le monde
```

Le « ; » sépare les conduites

```
$ echo bonjour;echo le monde|rev
bonjour
ednom el
```

### Note

- L'esperluette « & » fonctionne comme « ; »
- Mais passe la commande en arrière-plan

Notes

---

---

---

---

---

---

---

---

---

---

## Groupes de commandes

Les accolades « {} » et parenthèses « () » groupent des commandes  
Cela permet de combiner les redirections et les conduites

```
$ { echo ednom el | rev; echo bonjour; } | tac
bonjour
le monde
```

### Attention

- () s'exécutent dans un sous-shell
- {} nécessitent
  - un espace après le {
  - un ; (ou équivalent) avant le }

Notes

---

---

---

---

---

---

---

---

---

---

## Plus de redirections

Notes

---

---

---

---

---

---

---

---

---

---

## Redirections de base (rappel)

- « > fichier » redirige la sortie standard en tronquant
- « >> fichier » redirige la sortie standard en ajoutant
- « < fichier » utilise fichier comme entrée standard
- « 2> fichier » redirige la sortie d'erreur

### Le shell s'occupe

- D'ouvrir les fichiers
- De rapporter les problèmes d'ouverture
- En cas de problème, la commande n'est pas exécutée

```
$ cat fail
cat: fail: Aucun fichier ou dossier de ce type
$ cat < fail
bash: fail: Aucun fichier ou dossier de ce type
```

Notes

---

---

---

---

---

---

---

---

---

---





La position des redirections n'est pas importante

Habituellement, on les met à la fin

```
$ head -n 1 < /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ < /etc/passwd head -n 1
root:x:0:0:root:/root:/bin/bash
```

### Question

Qu'affiche « head -n < /etc/passwd 1 » ?

```
« root:x:0:0:root:/root:/bin/bash »
```

Notes

---

---

---

---

---

---

---

---

---

---

## Duplication de redirection

« 2>&1 » assigne la sortie standard à la sortie d'erreur

```
$ ls /usr/bin/perl fail >liste 2>&1
$ cat liste
ls: impossible d'accéder à 'fail': Aucun fichier
/usr/bin/perl
```

### Attention

L'ordre des redirections est important

```
$ ls /usr/bin/perl fail 2>&1 >liste
ls: impossible d'accéder à 'fail': Aucun fichier
$ cat liste
/usr/bin/perl
```

Extension bash: « &>fichier » est un raccourci de « >fichier 2>&1 »

Notes

---

---

---

---

---

---

---

---

---

---

## Descripteurs de fichiers



« [n]opérateur mot » est la forme générale des redirections où

- n est un nombre optionnel
- opérateur est un opérateur de redirection comme <, >, >>, etc.
- mot est un nom de fichier ou &m pour une duplication

n et m sont des numéros appelés **descripteurs de fichier**

Les descripteurs 0, 1 et 2 ont une signification spéciale

- 0 = entrée standard
- 1 = sortie standard
- 2 = sortie standard d'erreur

Notes

---

---

---

---

---

---

---

---

---

---

## Tubes et conduites (rappel)

- Le **tube** « | » (*pipe*) connecte des commandes
- Une **conduite** (*pipeline*) est une séquence de commandes connectées avec des tubes
- Les commandes d'une conduite s'exécutent en **parallèle**

```
$ ls | cowsay | lolcat
$ time sleep 2 | sleep 3 | sleep 1
```

Notes

---

---

---

---

---

---

---

---

## Tubes et redirections

Les tubes sont mis en place **avant** les autres redirections

```
$ echo a > b | wc -l
0
$ cat b
a
```

Ce qui permet aux redirections de profiter de l'existence du tube

```
$ cat fail 2>&1 | rev
epyt ec ed reissod uo reihcif nucuA :liaf :tac
```

Extension bash: « |& » est un raccourci de « 2>&1 | »

Notes

---

---

---

---

---

---

---

---

## Questions de redirection

Qu'affichent les commandes suivantes

- `echo hello | rev`  
→ olleh
- `echo hello >&2 | rev`  
→ hello
- `((echo hello>&2;echo bonjour)|rev)2>&1|tr -d o`  
→ hell et rujnb
- `((echo hello>&8;echo bonjour)|rev)8>&1`  
→ hello et ruojnob

Notes

---

---

---

---

---

---

---

---

## Duplication d'entrée standard

`tee` copie l'entrée standard vers des fichiers et vers la sortie standard

- `-a, --append` ajouter au fichier sans l'écraser

```
$ grep -x 'dr.*rd' french | tee liste | wc -c
11
$ cat liste
dreyfusard
```

Très utile pour

- Afficher et sauvegarder en même temps
- Extraire des résultats temporaires
- Déboguer des conduites

Notes

---

---

---

---

---

---

---

---

---

---

## Document en ligne (*Here document* ou *heredoc*)

« `<<délimiteur` » permet de déclarer un document en ligne

- Après le `<<` le shell n'exécute pas la commande
- Les lignes suivantes sont lues et utilisées comme document d'entrée
- Le document s'arrête quand une ligne `délimiteur` est lue

```
$ rev <<FIN
> ruojnob
> ednom el
> FIN
bonjour
le monde
```

Très utile pour

- Embarquer de la donnée dans un script

Notes

---

---

---

---

---

---

---

---

---

---

## Chaîne en ligne (bashisme)



« `<<<mot` » utilise `mot` comme document d'entrée

```
$ rev <<<"bonjour le monde"
ednom el ruojnob
```

C'est équivalent à un `echo`

```
$ echo "bonjour le monde" | rev
ednom el ruojnob
```

### Attention

Ceci est une extension `bash`

Notes

---

---

---

---

---

---

---

---

---

---

## Développement et substitutions

Notes

---

---

---

---

---

---

---

---

## Développement et substitutions

Notes

---

---

---

---

---

---

---

---

Il y a plusieurs étapes lors de l'analyse d'une commande

- Le développement des accolades (bashisme)
- Le développement du tilde « ~ »: déjà vu
- Le développement des paramètres et des variables
- La substitution de commandes
- Le développement arithmétique
- La substitution de processus (bashisme)
- Le découpage en mots, ou *split*
- Le développement des chemins, ou *glob*: déjà vu
- La suppression des protections: guillemets et backslash

## Développement des accolades (bashisme)



Notes

---

---

---

---

---

---

---

---

Bash accepte trois formes

- Liste: « x{a,b,c}y » → xay xby xcy
- Série: « a{1..5} » → a1 a2 a3 a4 a5
- Série et pas: « a{10..30..5} » → a10 a15 a20 a25 a30

Se combinent et s'imbriquent

```
$ echo f{1..3}.{txt,png}
f1.txt f1.png f2.txt f2.png f3.txt f3.png
$ mkdir -p \
velo/{roue/{jante,rayon,moyeu},selle,cadre/{tube,fourche}}
```

Note

- Il a lieu avant tous les autres développements
- Ce **n'est pas** du glob
- C'est une extension bash



« `$(commande)` » (et l'ancienne forme « ``commande`` »)

Remplace une commande par son résultat

- `commande` est exécutée
- la sortie de la commande (`stdout`) est capturée (au lieu d'être affichée)
- la capture remplace le « `$(commande)` » original

```
$ echo "Je m'appelle $(id -un)"
Je m'appelle privat
```

On peut les imbriquer

```
$ cd data
$ echo "Le répertoire courant est $(basename "$(pwd)")"
Le répertoire courant est data
```

## Guillemets simple et doubles (rappel)

- « ' » (guillemet simple, *simple quote*)
- Force l'interprétation littérale jusqu'au prochain « ' »
- « " » (guillemets doubles, *double quote*)
- Une version amoindrie de « ' »
- Permet des caractères 3 spéciaux internes: « \ », « \$ » et « ` »

### Question

Qu'affichent

- `cowsay "$(echo toto)"`
- `cowsay '$(echo toto)'`
- `cowsay $(echo toto)`
- `"cowsay $(echo toto)"`

## Split et glob



Dans l'ordre des développements, les 3 dernières étapes sont

- Le découpage en mots, ou *split*
- Le développement des chemins, ou *glob*
- La suppression des protections: guillemets et backslash

Pour un développement non protégé par des guillemets doubles « " »

- Le shell découpe le résultat en mots (*split*)
- Le shell développe les chemins pour chacun des mots (*glob*)
- Ce qui provoque souvent des problèmes

Par sécurité, lors des développements en shell:

Il faut **toujours** utiliser des guillemets doubles

## Split et glob: exercice

```
$ mkdir -p "data/ ta"
$ cd "data/ ta"
$ pwd
/tmp/data/ ta
```

Qu'affichent les commandes suivantes

- `echo "Le répertoire courant est «$(basename "$(pwd)")»"`  
→ Le répertoire courant est « ta»
- `echo Le répertoire courant est «$(basename "$(pwd)")»`  
→ Le répertoire courant est « ta»
- Les espaces n'ont pas été préservés
- `echo "Le répertoire courant est «$(basename $(pwd))»"`  
→ Le répertoire courant est «da»
- Sorcellerie !

Notes

---

---

---

---

---

---

---

---

---

---

## Explication

`basename` élimine le chemin d'accès et le suffixe d'un nom de fichier

```
$ pwd
/tmp/data/ ta
$ basename "$(pwd)"
ta
$ basename "/tmp/data/ ta"
ta
$ basename $(pwd)
da
$ basename tmp/data/ ta
da
```

Notes

---

---

---

---

---

---

---

---

---

---

## Question substitution de commandes

Que font les commandes suivantes

- `echo *`
- `echo "*"`
- `echo '*'`
- `echo $(echo '*')`
- `echo "$(echo '*')"`
- `echo '$(echo '*')`

Notes

---

---

---

---

---

---

---

---

---

---

## Substitution de processus (bashisme)



« `<(cmd)` » ou « `>(cmd)` »

- Exécute la commande `cmd`
- Redirige sa sortie (ou son entrée) vers un tube
- Substitue par le **fichier** de l'autre bout du tube
- C'est un peu magique...

Utile pour nourrir les commandes:

- Qui prennent seulement des fichiers et pas l'entrée standard
- Ou ont des entrées en plus de l'entrée standard

### Notes

- C'est une extension de bash
- N'est disponible que sur certains systèmes
- Les détails de la magie en INF3172

Notes

---

---

---

---

---

---

---

---

---

---

## Substitution de processus

```
$ cat fruits1.txt
banane
orange
$ cat fruits2.txt
orange
kiwi
banane
$ diff fruits1.txt fruits2.txt
1d0
< banane
2a2,3
> kiwi
> banane
$ diff <(sort fruits1.txt) <(sort fruits2.txt)
1a2
> kiwi
```

Notes

---

---

---

---

---

---

---

---

---

---

## Paramètres et variables

Notes

---

---

---

---

---

---

---

---

---

---

## Paramètres et compagnie

**Paramètre** = terme générique qui englobe 3 cas:

### Variable

- Un paramètre avec un nom, exemples « \$toto » et « \$tata »
- Utilisation relativement libre

### Paramètre positionnel

- Un paramètre avec un numéro, exemples « \$1 » et « \$3 »
- Pour récupérer les arguments

### Paramètre spécial

- Un paramètre avec un caractère, exemples « \$@ » et « \$\* »
- Pour des rôles spécifiques

Notes

---

---

---

---

---

---

---

---

---

---

## Variables du shell



- « nom="valeur" » définit une variable `nom` avec la valeur `valeur`
- « "\$nom" » se substitue par la valeur de la variable `nom`

```
$ toto="tata"
$ echo "hakuna ma$toto"
hakuna matata
```

### Attention

- Pas de « \$ » lors de l'affectation
- Pas d'espaces autour du « = »
- Mettre des « " » pour éviter les mauvaises surprises
- Il n'y a pas de types de données, ce sont des chaînes

Notes

---

---

---

---

---

---

---

---

---

---

## Split-glob sur les variables



Que fait « `rm $a` » sachant qu'on a défini `a` avec

- `a="toto"`
- `a="toto *"`
- `a="-rf ."`
- `a="toto; rm -rf ."`

Mêmes questions avec « `rm "$a"` » et « `rm -- "$a"` »

### Conclusion

Par sécurité, il faut

- Utiliser des **guillemets doubles** avec la substitution en shell
- Protéger les **options** des arguments, avec « `--` » par exemple

Notes

---

---

---

---

---

---

---

---

---

---



## Paramètres positionnels



- Permet de récupérer la valeur des arguments
- « \$1 » est le premier argument, « \$2 » le second, etc.

```
#!/bin/sh
echo "Le premier argument est $1"
echo "Le second argument est $2"
```

shift décale (renomme) les paramètres (commande interne)

```
#!/bin/sh
echo "Le premier argument est $1"
shift
echo "Le second argument est $1"
shift 2
echo "Le quatrième argument est $1"
```

Notes

---

---

---

---

---

---

---

---

## Quelques paramètres spéciaux



- « \$\* » et « \$@ » l'ensemble des paramètres positionnels
- « \$# » le nombre de paramètres positionnels
- « \$\$ » le PID du shell
- « \$! » le PID de la dernière commande en arrière-plan

### Ensemble des paramètres positionnels

« \$\* » et « \$@ » se comportent différemment entre guillemets doubles

- « "\$\*" » → « "\$1 \$2 \$3..." »
- « "\$@" » → « "\$1" "\$2" "\$3"... »

→ C'est souvent "\$@" que l'on veut

Notes

---

---

---

---

---

---

---

---

## Question des paramètres positionnels

Que fait le programme suivant?

```
#!/bin/sh
outfile=$1
shift
cat "$@" > "$outfile"
```

Notes

---

---

---

---

---

---

---

---

## La commande set



set affecte les options et paramètres positionnels (interne)

### Réaffecte les paramètres positionnels

```
$ set A C D
$ echo "$* $2"
A C D C
```

### Active des options comportementales du shell

- -e si une commande échoue, le script se termine
- -x affiche une trace de l'exécution
- -u lire un paramètre non affecté est une erreur
- -o affiche les options (ou en active une)

```
#!/bin/sh
set -ex
echo hello; ehco fail; echo "ya quelqu'un?"
```

Notes

---

---

---

---

---

---

---

---

## Variables utilisées par le shell

Plusieurs variables sont créées et/ou utilisées par le shell

- Pour avoir de l'information sur le shell
- Pour contrôler son comportement

### POSIX en définit 16, dont

- HOME Le répertoire d'accueil de l'utilisateur (utilisé par ~ et cd)
- PATH La liste des répertoires où chercher les commandes
- PS1 L'invite de commande principale
- PWD Le répertoire de travail courant (mis à jour par cd)

### Bash en définit une centaine...

- RANDOM un entier aléatoire entre 0 et 32767
- OLDPWD le répertoire précédent (utilisé par cd)

Notes

---

---

---

---

---

---

---

---

## Sous-interpréteur (*subshell*)



Le shell peut créer des sous-environnements locaux

- Conduites « | »
- Commandes en arrière plan « & »
- Groupes avec « () »
- Substitution « \$( ) » et « <( ) »

### Caractéristiques

- Chaque sous-interpréteur est un processus indépendant
- Les variables du shell sont **héritées** par **copie**
- Les modifications ne sont **pas propagées** au parent

→ Source de **bogues** subtils

```
$ { a=1; }; echo "$a"
$ (a=2; echo "$a")
$ { a=3; echo "$a"; } &
$ a=4|a=5
$ echo $a
```

Notes

---

---

---

---

---

---

---

---



### Variables du shell

- Concept du **shell**
- **Limitées** au shell courant

### Variables d'environnement

- Concept du **système d'exploitation**
- S'appliquent à **tous les processus**
- **Propagées** aux processus enfants (par défaut)

### Le shell

- Permet de **marquer** certaines variable
- Elles seront **exportées** comme des variables d'environnement dans les sous-processus
- Permet d'exécuter dans un environnement modifié

## Exporter une variable

`env` permet de lister les variables d'environnement

Commande interne `export` exporte une variable

- `-p` liste les variables exportées

```
$ var1=toto
$ export var1 var2=tata
$ var3=tete
$ env | grep var
var1=toto
var2=tata
```

## Affectations facultatives de variables

Les affectations qui précèdent une commande

- Sont exportées à la commande comme variables d'environnement
- Ne changent pas l'environnement du shell courant

```
$ var1=titi var2=tutu env | grep var
var1=titi
var2=tutu
$ env | grep var
var1=toto
var2=tata
```



env exécute un programme avec un environnement

- -i, --ignore-environment débute avec un environnement vide

Tout seul il affiche la liste des variables d'environnement

```
$ env var1=titi var2=tutu env | grep var
var1=titi
var2=tutu
```

env est souvent utilisé dans le shebang

Exemple: « #!/usr/bin/env bash » OU « #!/usr/bin/env python3 »

- Son emplacement est relativement standard (plus de probabilité)
- Il invoque l'interpréteur en fonction du PATH de l'utilisateur

Notes

---

---

---

---

---

---

---

---

---

---



«  $$(expr)$  » évalue une expression mathématique

```
$ echo $((1+1))
2
$ x=10
$ echo $((x+1))
11
```

- Que des nombres entiers
- La plupart des opérateurs du C ( $\approx$  Java) sont reconnus

```
$ echo $((1+2*3%4))
3
```

Le mode mathématique est différent du shell

- Les mécanismes shell n'ont plus cours
- Les variables n'ont pas besoin de « \$ »
- Les variables n'ont pas besoin de « " » (pas de split-glob)

Notes

---

---

---

---

---

---

---

---

---

---

Code de retour

Notes

---

---

---

---

---

---

---

---

---

---

## Code de retour des commandes

Chaque commande exécutée a un **code de retour** (ou code de sortie)

### Convention

- 0 tout s'est bien passé: **succès**
- Une autre valeur signifie un problème: **échec**
- Voir le manuel des commandes pour les détails

Le paramètre spécial «  `$?` » contient le code de retour de la dernière commande

```
$ wc -l /etc/passwd
49 /etc/passwd
$ echo "$?"
0
$ wc -l fail
wc: fail: Aucun fichier ou dossier de ce type
$ echo "$?"
1
```

Notes

---

---

---

---

---

---

---

---

---

---

## Fin des scripts

### Terminaison du script

- La fin du fichier script est atteinte
- La commande `exit` est exécutée

### Code de retour

- Peut être indiqué lors du `exit`
- Sinon, c'est celui de la dernière commande exécutée

### Question

Quel est le code de retour du script suivant?

```
#!/bin/sh
cat epic.fail
echo "Au revoir"
```

Notes

---

---

---

---

---

---

---

---

---

---

## Vrai et faux

- `true` ne fait rien mais réussit
- `false` ne fait rien mais échoue
- `:` fait comme `true`

```
$ true; echo $?
0
$ false; echo $?
1
$ ;; echo $?
0
```

Notes

---

---

---

---

---

---

---

---

---

---

## Liste conditionnelle de commandes

« && » et « || » chaînent les commandes en fonction de leur retour

### Et « && »

Exécute la seconde commande ssi la première réussit

```
$ cd toto && ls
```

```
bash: cd: toto: Aucun fichier ou dossier de ce type
```

### Ou « || »

Exécute la seconde commande ssi la première échoue

```
$ cd toto || echo "tant pis"
```

```
bash: cd: toto: Aucun fichier ou dossier de ce type
tant pis
```

### Attention

- Ne pas confondre « && », « & », « || » et « | »

Notes

---

---

---

---

---

---

---

---

---

---

## Structures de contrôle

Notes

---

---

---

---

---

---

---

---

---

---

## Structures de contrôle



Fonctionnalités utilisées dans le développement de scripts

- Vous les utiliserez et approfondirez en laboratoire et TP
- On ne demande pas de les connaître par cœur pour l'examen

### Par rapport à INF1120

Les structures de contrôle du shell sont

- **Pauvres**: il y en a peu et elles sont limitées
- **Fragiles**: il est facile de mal les utiliser
- **Bizarres**: syntaxe et sémantique → des accidents historiques

### Consignes

- Respectez les **bonnes pratiques**
- Évitez l'utilisation superflues de structures de contrôles

Notes

---

---

---

---

---

---

---

---

---

---

## Construction if

```
if liste; then liste; elif liste; then liste; else liste; fi
```

- La liste du `if` est exécutée
- Si le code de retour est 0
- Alors la liste du `then` associé est exécutée
- Sinon la liste du `elif` est exécutée
- Si le code de retour est 0
- Alors la liste du `then` associé est exécutée
- Sinon la liste du `else` est exécutée

### Notes

- `fi` est le mot clé de fin
- Il peut y avoir 0, 1, ou plusieurs `elif`
- Le `else` est optionnel

### Notes

---

---

---

---

---

---

---

---

---

---

## Exemple de if

Qu'affiche le programme suivant?

```
#!/bin/sh
mot=pâtisserie
if grep -q -e "$mot" -x /usr/share/dict/*; then
    echo "C'est du gâteau"
else
    echo "C'est pas d'la tarte"
fi
```

### Note de style

- On aligne `if`, `elif`, `else` et `fi`
- On met le `then` à la fin des lignes (après le `;`)
- On indente le corps des `then` et `else`

### Notes

---

---

---

---

---

---

---

---

---

---

## Des meilleurs tests

`test` teste des expressions

- L'utilitaire « `[` » est similaire à `test`
- `test` et « `[` » sont souvent des commandes internes
- « `[` » nécessite un argument « `]` » final (pour faire joli)

### Exemples

```
$ test -e /etc/passwd; echo $?
0
$ [ -e /etc/pas-ce-word ]; echo $?
1
$ test toto = tata; echo $?
1
```

### Attention

- Mettre des espaces autour des « `[` » et « `]` »
- Bien échapper les variables

### Notes

---

---

---

---

---

---

---

---

---

---

## Opérateurs de test des fichiers

- « `-e f` » le fichier `f` existe
  - « `-f f` » le fichier `f` existe et est régulier
  - « `-d f` » le fichier `f` existe et est un répertoire
  - « `-r f` » le fichier `f` existe et est lisible
  - « `-s f` » le fichier `f` existe et n'est pas vide
- Plusieurs autres en fonction des métadonnées des fichiers

### Exemple d'utilisation

- Tester l'existence et la validité des arguments

```
if [ -e "$output" ]; then
    echo "Erreur: $output existe déjà" >&2
    exit 1
fi
```

### Notes

---

---

---

---

---

---

---

---

---

---

## Opérateurs de test des chaînes

- « `-z s` » la chaîne `s` est de longueur nulle (chaîne vide)
- « `-n s` » la chaîne `s` est de longueur non nulle (chaîne non vide)
- « `s = s'` » les chaînes `s` et `s'` sont identiques
- « `s != s'` » les chaînes `s` et `s'` sont différentes

### Exemple d'utilisation

- Comparer des variables

```
if [ -z "$output" ]; then
    # Pas d'output demandé, on termine là
    exit 0
fi
```

### Attention

- L'opérateur d'égalité est « `=` » (et pas « `==` »)

### Notes

---

---

---

---

---

---

---

---

---

---

## Opérateurs de test numériques

- « `n -eq n'` » les nombres entiers `n` et `n'` sont égaux (*equal*)
- « `n -ne n'` » `n` et `n'` sont différents (*not equal*)
- « `n -gt n'` » `n` est strictement plus grand que `n'` (*greater than*)
- « `n -ge n'` » `n` est plus grand ou égal `n'` (*greater or equal*)
- « `n -lt n'` » `n` est strictement plus petit que `n'` (*less than*)
- « `n -le n'` » `n` est plus petit ou égal `n'` (*less or equal*)

### Exemple d'utilisation

- Compter des trucs

```
if [ "$#" -lt 2 ] || [ "$#" -gt 3 ]; then
    echo "usage: outil IN OUT [EXTRA]" >&2
    exit 1
fi
```

### Notes

---

---

---

---

---

---

---

---

---

---



## De meilleurs tests encore (bashisme)

« `[[ ]]` » construction spéciale de `bash`

- Pas besoin de protéger les variable (pas de `split` et `glob`)
- « `&&` » et « `||` » connecteur logiques « et » et « ou »
- « `( )` » groupements
- « `<` » et « `>` » comparaison lexicographique de chaînes
- « `==` » la 2nd opérande est un motif style *glob*
- « `--` » la 2nd opérande est une ERE GNU

```
$ a=bonjour
$ [[ $a == ?o* && $a == (o[un]j?){2} ]]; echo $?
```

### Attention

- Les `[` et `test` internes `bash` acceptent certaines extensions
- En `bash` préférez « `[[ ]]` », en shell POSIX utilisez « `[ ]` »

Notes

---

---

---

---

---

---

---

---

---

---

## Boucle for

```
for nom in mot ...; do liste; done
```

- La liste des mots est développée en une liste d'éléments
- La variable `nom` prend tour à tour la valeur de chaque élément
- Les instructions de la liste sont exécutées pour chaque valeur de `nom`

### Notes

- `done` est le mot clé de fin
- « `in mot...` » est optionnel et vaut par défaut « `in "$@"` »
- Attention au *split-glob* des mots

Notes

---

---

---

---

---

---

---

---

---

---

## Exemple de for

Qu'affiche le programme suivant?

```
#!/bin/sh
for i in pommes poires; do
  for j in 1 2 3; do
    echo "$i$j"
  done
done
```

### Note de style

- On aligne `for` et `done`
- On met le `do` à la fin des lignes (après le `;`)
- On indente la liste d'instructions

Notes

---

---

---

---

---

---

---

---

---

---

## Attention au for

Quel est le problème de

```
for f in $(ls *.gz); do
  gunzip "$f"
done
```

Quel est le problème de

```
for f in "$(ls *.gz)"; do
  gunzip "$f"
done
```

Quelle est la solution ?

```
for f in *.gz; do
  gunzip -- "$f"
done
```

Notes

---

---

---

---

---

---

---

---

---

---

## Sous-programmes

```
nom() { liste; }
```

- Définit une fonction `nom`
- `nom` utilisé en commande appelle la fonction
- Les arguments de l'appel sont passés en paramètres positionnels (`$1`, `$2...`)
- Les autres variables ont une portée globale par défaut

### Note

- Le mot clé `return` termine une fonction avec un code de retour
- Le mot clé `local` permet de déclarer des variables locales
- Les fonctions s'utilisent comme des commandes: dans des conduites, en arrière plan, dans des substitutions de commandes, etc.
- On peut faire des fonctions récursives

Notes

---

---

---

---

---

---

---

---

---

---

## Exemple de fonction

Qu'affiche le programme suivant ?

```
#!/bin/sh
foo() {
  echo "F00 $@"
  local c=X
  d=Y
}
c=cé
d=dé
foo a b "$c" "$d"
foo a b "$c" "$d"
```

### Note de style

- Nommez les fonctions en minuscules (de « a » à « z » et « \_ »)
- On aligne le nom et l'accolade « } »
- On met l'accolade « { » à la fin de la ligne (après le « () »)
- On indente le corps de la fonction

Notes

---

---

---

---

---

---

---

---

---

---



Nous avons vu *rapidement* les fonctionnalités de base d'un shell Unix

- La spécification POSIX du langage du shell
- Quelques extensions `bash` (GNU)

Nous vous laissons le loisir d'en apprendre plus par vous-même

- `$IFS`: séparateur interne de champ (*internal field separator*)
- `while`: boucle while
- `case`: construction switch/case
- `read`: lecture de ligne
- `alias`: définition d'alias
- `${}`: développement avancé de paramètres
- `exec`: recouvrement (et contrôle des flots)
- `eval`: exécution de commande construite
- `trap`: capture de signaux
- et plein d'extensions `bash`

Notes

---

---

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

---

---