Chapitre 8 : bash dans les détails

On va revenir sur de nombreux points
Comprendre le chapitre 7 avant celui-ci

Bash ?

- C'est un langage qui a grossi au dessus du tout premier shell d'Unix qui s'appelle `sh`
- Il y a de très nombreuses bizarreries
  - Syntaxe incohérente (ex : esac mais done)
  - Sensibilité extrême aux espaces

8.1 – Structure et exécution d'un script

Retour sur la composition d'un script et ce qui se passe quand on le lance

Composition d'un script

- C'est un fichier texte contenant

```
#!/bin/bash
# commentaire

Commande Unix complète...
...
```

- Taper `chmod u+x son nom`
- Taper `son nom` (ou `./son nom`)

La première ligne

- # ! nom complet du shell : `#!/bin/bash`
- Il faut comprendre qu'on redirige le script lui-même vers cette commande
  - Au lieu de bash, on peut mettre (presque) n'importe quelle commande qui utilise stdin

```
#!/bin/cat -n
Salut tout le monde,
Moi ça va, et vous ?
Tant mieux.
```

```
#!/usr/bin/wc -l
Salut tout le monde,
Moi ça va, et vous ?
Tant mieux.
```

L'exécution du script

- À chaque fois qu'on lance un script, ça crée un processus
  - Exemple :

```
#!/bin/bash
echo "je tourne, mon PID est $$"
ps -fu pierre
echo "j'ai fini"
```

  - Résultat :

```
pierre@port39$ essai.sh
je tourne, mon PID est 4584
UID        PID  PPID  C STIME TTY          TIME CMD
pierre    4388  4380  0 16:39 pts/0    00:00:00 bash
pierre    4584  4388  0 16:51 pts/0    00:00:00 /bin/bash ./essai.sh
pierre    4585  4584  0 16:51 pts/0    00:00:00 ps -fu pierre
j'ai fini
pierre@port39$
```

Exécution sans création de processus

- Pour ne pas créer de processus bash, il faut lancer le script par `.` ou `source` suivi d'un espace puis de son nom

```
pierre@port39$ source essai.sh
je tourne, mon PID est 4388
UID        PID  PPID  C STIME TTY          TIME CMD
pierre    4388  4380  0 16:39 pts/0    00:00:00 bash
pierre    4585  4388  0 16:52 pts/0    00:00:00 ps -fu pierre
j'ai fini
pierre@port39$
```

  - Dans ce cas, le shell de connexion lit lui-même directement les instructions du script sans lancer un nouveau processus bash

Chaque shell est indépendant

- Un shell (exécutant un script) est indépendant :
  - Exemple :

```
#!/bin/bash
echo "je tourne dans $PWD"
cd ..
echo "je vais dans $PWD"
```

  - Ça ne change rien pour celui qui le lance :

```
pierre@port39$ pwd
/home/pierre/tmp
pierre@port39$ essai.sh
je tourne dans /home/pierre/tmp
je vais dans /home/pierre
pierre@port39$ pwd
/home/pierre/tmp
pierre@port39$
```

Indépendance (suite)

- Les variables sont personnelles :
  - Exemple :

```
#!/bin/bash
MESSAGE='Bonjour à tous'
echo "MESSAGE=$MESSAGE"
```

  - Ça ne change rien pour celui qui le lance :

```
pierre@port39$ MESSAGE=salut
pierre@port39$ echo $MESSAGE
salut
pierre@port39$ essai.sh
MESSAGE=Bonjour à tous
pierre@port39$ echo $MESSAGE
salut
pierre@port39$
```

Un script peut en lancer un autre

- Les scripts sont des commandes comme les autres :
  - essai1 :

```
#!/bin/bash
echo "essai1: je lance essai2"
essai2
echo "essai1: essai2 a fini"
```

  - essai2 :

```
#!/bin/bash
echo "essai2: je suis essai2"
sleep 5      # petite pause
echo "essai2: j'ai fini"
```

Un script peut lancer un job

- Idem avec les jobs, on peut obtenir le PID du dernier lancé
  - essai1 :
  - essai2 :

```
#!/bin/bash
echo "essai1: je lance essai2"
essai2 &
echo "essai1: son PID est $!"
wait
echo "essai1: j'ai fini"
```

```
#!/bin/bash
echo "essai2: je suis essai2, mon PID=$$"
sleep 5      # petite pause
echo "essai2: j'ai fini"
```

Code de retour

- Toute commande qui se termine renvoie un code : en C, c'est la valeur du return de `main()`
  - Code de retour = 0 => aucune erreur, tout ok
  - Code de retour > 0 => erreur
- Exemple : egrep (voir sa doc en ligne)
  - Code = 0 si des lignes correspondent
  - Code = 1 si aucune ligne de correspond
  - Code = 2 s'il y a une erreur dans le motif ou le fichier à parcourir

Émettre un code de retour

- Il suffit de faire exit avec ce code

  `Ex: exit 2`

  - Retourne le code 2 (non nul => signale une erreur)

Obtention du code de retour

- Un script qui lance une commande récupère son code de retour dans la variable spéciale $?

```
#!/bin/bash
echo "je lance egrep"
egrep 'ABSENT' meteoUS
echo "Le code renvoyé est $?"
```

- NB : $? change à chaque commande lancée
- Dans le cas d'un tube, c'est le code de la dernière commande

Utilité du code de retour

- Le code de retour peut être testé directement par la conditionnelle if

  if commande

  then elle a marché

  else elle a échoué

  fi

```
#!/bin/bash
if egrep 'SUNNY' meteoUS
then
   echo 'chouette soleil'
else
   echo 'pas de soleil'
fi
```

```
#!/bin/bash
if gcc -o prog prog.c
then
   ...
```

Redirection d'une commande

- Un script peut récupérer les affichages d'une commande dans une variable :

  variable=$(commande complète)

- Exemple :

```
NbProcs=$(ps -edf | wc -l)
```

Redirection d'entrée interne

- Un script peut fournir des données à une de ses commandes par une partie du script lui-même

```
commande <<data
Texte...
Texte...
data
```

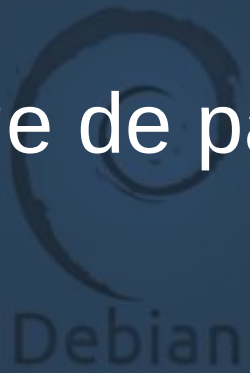
- Le texte situé entre les deux mots data est envoyé dans l'entrée standard de la commande
  - On peut mettre le mot qu'on veut à la place de data

Application de la redirection interne

```
for nom in pierre paul jacques
do
  mail ${nom}@univ-rennes1.fr <<message
Mr ${nom}, vous avez trop de fichiers.
Merci de les supprimer,
L'administrateur.
message
done
```

8.2 – Paramètres d'un script

Retour sur le passage de paramètres à un script



Paramètres d'appel (rappels)

- Une ligne de commande complète :
  - Nom du script paramètres...
  - Ex : essai3.sh bonjour tout le monde
- Le nom du script va dans $0 (pas comme awk)
- Le premier paramètre va dans $1
- Le deuxième paramètre va dans $2
- Etc. jusqu'à $9, après c'est ${10}, ${11}...

Comment sont délimités les paramètres ?

- Les paramètres sont séparés par des espaces
- S'il y a un espace à mettre dans un paramètre, alors, soit :
  - On met \ devant cet espace

```
mkdir Dossier\ de\ travail
```

  - On encadre par '...'

```
mkdir 'Dossier de travail'
```

  - On encadre par "..."

```
mkdir "Dossier de travail"
```

Exemple

- Si on tape :

```
essai.sh 'Bonjour le monde' voici\ un "texte court"
```

- On va avoir :
  - $0 = « essai.sh » (nom du script)
  - $1 = « Bonjour le monde » (premier param.)
  - $2 = « voici un » (deuxième param.)
  - $3 = « texte court » (troisième param.)

  NB : les « » ne font pas partie des valeurs

Attention ensuite

- Si un paramètre ou une variable est susceptible de contenir un espace, alors il faut l'encadrer par des "…"

  `mkdir "$1"` et non pas `mkdir $1`

  Sans ces ", les espaces vont à nouveau provoquer la séparation des mots de $1 avant de lancer mkdir

L'ensemble des paramètres

- Le nombre de paramètres fournis est dans `$#`
- La liste des paramètres est dans `$*` et `$@`
- Il y a une différence quand on écrit `"$*"` et `"$@"`
  - `"$*"` est une seule chaîne regroupant tous les paramètres passés au script
  - `"$@"` est remplacé par autant de chaînes que de paramètres, chacun étant comme encadré par des "...". C'est ça qu'il faut utiliser si on fait une boucle.

Boucle avec "$@"

- Voici un exemple d'application :

```
#!/bin/bash
for param in "$@"
do
   echo "param = x${param}x"
done
```

```
pierre@port39$ essai3.sh bonjour le monde
param = xbonjourx
param = xlex
param = xmondex
pierre@port39$ essai3.sh bonjour 'tout le monde'
param = xbonjourx
param = xtout le mondex
pierre@port39$
```

NB : ça ne marche pas avec $*

Réaffecter les paramètres

- À l'intérieur d'un script, on peut réaffecter $1, $2... mais pas séparément :

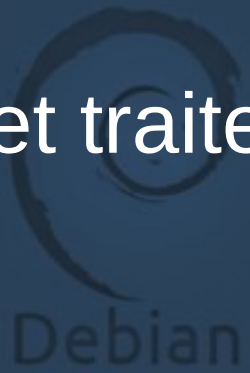
  `set` *nouveaux paramètres*

- Exemple :

```
set 'Salut la compagnie' tout va bien
echo $1
echo $2
```

8.3 – Variables

Pour stocker et traiter des chaînes



Affectation et valeur

- Affectation

  nom_de_la_variable=valeur

  NB : ne mettre aucun espace autour du =

  - Emballer la valeur entre '…' ou "..."
- Récupération de la valeur

  ${nom_de_la_variable}

  - Ou plus simple : $nom_de_la_variable

Rappels

- On peut affecter :
  - Un mot isolé
  - Une chaîne '…' constante, ou "…" variable
  - Les sorties d'une commande $(commande)
  - Le résultat d'un calcul $((calcul))
  - Une liste de mots (…) qui crée un *tableau*

Tableaux

- Bash a une notion de tableau : c'est simplement une liste de mots mis entre (…)
  - Ex : `utis=(pierre paul jacques)`
- Les mots sont désignés par un indice :
  - `${utis[0]}` donne le premier mot
  - `${utis[1]}` donne le deuxième mot, etc.
  - `${#utis[@]}` donne le nombre de mots
  - `${utis[@]}` donne tous les mots

  NB : $utis ou ${utis} ne donne que le premier mot

Affectation d'une case de tableau

- On peut affecter une case, même non existante du tableau :

```
utis[1]=jean
utis[6]=henri
echo ${utis[@]}
```

- La liste des indices employés s'obtient par :

```
${!utis[@]}
```

  - Ici : ça donne 0 1 2 6

Valeur par défaut

- On peut effectuer certains traitements lors de l'extraction des valeurs des variables :
  - `${nomvar:-valeur}` : si *nomvar* n'a pas été affectée, alors bash met *valeur* à la place mais sans changer la variable
  - `${nomvar:=valeur}` : si *nomvar* n'a pas été affectée, alors bash affecte la variable *nomvar* avec cette *valeur*

Exemples

- Pour proposer des valeurs par défaut :

```
gcc -o ${exec:-prog} ${source:=prog.c}
```

  - Si $exec a une valeur, alors c'est le nom du programme résultat, sinon c'est « prog »
  - Si $source a une valeur, alors c'est ok, sinon on affecte source=prog.c et c'est ça qu'on compile
- Autre exemple :

```
if test "${reponse:-non}" = oui
```

  - Si $reponse n'a pas de valeur, alors c'est non

Extraction de sous-chaînes

- On peut effectuer certains traitements lors de l'extraction des valeurs des variables :
  - `${#nomvar}` : donne le nombre de caractères de la valeur de cette variable (longueur de la chaîne)
  - `${nomvar:debut}` : extrait une sous-chaîne de *nomvar* commençant au caractère d'indice *début*
  - `${nomvar:debut:lng}` : extrait une sous-chaîne de *nomvar* commençant au caractère d'indice *début* et de longueur *lng*

Exemple

- Voici quelques extractions de sous-chaînes :

```
nomfich=P872887.JPG
```

  - On obtient la longueur du nom du fichier : 11

```
lng=${#nomfich}
```

  - On extrait son extension : .JPG

```
ext=${nomfich:lng-4}
```

  - On extrait son nom de base : P872887

```
base=${nomfich:0:lng-4}
```

Correspondance

- On peut extraire une partie d'une variable correspondant à un motif
  - `${nomvar#motif}` si le début de la valeur de la variable correspond au motif, alors ça donne le reste
    - Si on met ## au lieu de #, alors les jokers seront *gloutons*
  - `${nomvar%motif}` si la fin de la valeur de la variable correspond au motif, alors ça donne le début
    - Idem avec %% au lieu de %
  - `${nomvar/motif/remp}` affiche le remplacement de motif par remp dans la valeur de la variable

  NB : ce sont des motifs style bash (voir Partie2)

Jokers gloutons ou pas

- Soit un nom de fichier : toto.orig.jpg.bak
  - Le motif *. correspond à toto. si le * n'est pas glouton
  - Le motif *. correspond à toto.orig.jpg. si * est glouton
- Glouton (*greedy*) : essaie de correspondre au plus grand nombre de caractères à la suite

Exemples

- Soit un nom de fichier dans une variable :

```
nomfich='P7812.JPG.bak'
```

- On peut extraire le nom de base P7812 par :

```
base=${nomfich%%.*}
```

- On peut récupérer l'extension .JPG.bak par :

```
ext=${nomfich#*.}
```

- On peut récupérer l'extension .bak par :

```
ext=${nomfich##*.}
```

Variables : toutes locales

- Un script ne partage pas ses variables (cf 8.1)
  - essai2.sh ne reçoit pas NOMBRE
  - essai2.sh ne peut pas modifier NOMBRE pour essai1.sh

```
#!/bin/bash
NOMBRE=1
echo "essai1: $NOMBRE"
essai2.sh
echo "essai1: $NOMBRE"
```

```
#!/bin/bash
echo "essai2: $NOMBRE"
NOMBRE=2
echo "essai2: $NOMBRE"
```

Variables globales

- Il y a une variante d'affectation permettant d'envoyer une variable
  - essai2.sh reçoit la valeur de NOMBRE
  - Mais pas de retour en cas de changement

```
#!/bin/bash
export NOMBRE=1
echo "essai1: $NOMBRE"
essai2.sh
echo "essai1: $NOMBRE"
```

```
#!/bin/bash
echo "essai2: $NOMBRE"
NOMBRE=2
echo "essai2: $NOMBRE"
```

Liste des variables

- On peut afficher la liste des variables :

  ```
  set
  ```

- On peut avoir la liste des variables exportées :

  ```
  env
  ```

- Il y a un très grand nombre de variables, elles sont expliquées dans la doc en ligne : man bash

Variables prédéfinies

- Il faut connaître quelques variables :
  - PWD : chemin absolu courant
  - HOME : chemin absolu du compte
  - LOGNAME, USER : nom du compte connecté
- Ensuite, un peu moins utile :
  - RANDOM : un entier différent à chaque emploi
  - LANG, LC_ALL... : paramètres linguistiques
  - PATH : chemins contenant les commandes
  - PS1, PS2 : prompts

8.4 – Structure d'une commande

Les commandes peuvent être complexes

Commande simple

- Une ligne de commande bash contient le nom d'un programme à exécuter (venant de /bin, /usr/bin ou autre) et les paramètres

  Ex: `gcc tp3.c -o tp3`

- On peut y rajouter des redirections et le signe de mise en arrière-plan
- Si la ligne est trop longue, on peut placer un \ puis un retour à la ligne et continuer ensuite

Commande => processus

- Chaque commande lancée dans un script crée un nouveau processus
  - Recherche du fichier exécutable correspondant au nom de la commande dans les dossiers indiqués par $PATH, puis exécution de ce programme
  - Ex : ls => lancement de /bin/ls
- Sauf les commandes internes : ce sont des procédures de bash lui-même
  - Ex : la commande cd est interne

Affectation de variable à l'appel

- On peut affecter une variable juste pour lancer le script :

```
#!/bin/bash
echo "Je reçois $info"
```

- Lancement :
  - On écrit variable=valeur commande paramètres...

```
pierre@port39$ info=Bonjour essai4.sh
Je reçois Bonjour
pierre@port39$
```

Commandes complexes

- Parmi les commandes complexes, il y a :
  - Les **tubes** commande1 | commande2 | …
    - Toutes ces commandes sont exécutées simultanément et leurs entrées et sorties sont connectées (voir cours partie 3)
  - Les **listes de commandes** : c'est une séquence de commande connectées par des opérateurs logiques
    - On exécute la première commande puis selon qu'elle a réussi ou échoué et le type d'opérateur, on continue ou on arrête

Listes de commandes

- `commande1 ; commande2`

  Exécute commande1 puis commande2 (quelque soit le résultat de commande1)

- `commande1 && commande2`

  Exécute commande2 seulement si commande1 a réussi

- `commande1 || commande2`

  Exécute commande2 seulement si commande1 a échoué

Groupement de commandes

- On peut grouper des commandes

  (commande1 ; commande2 ; ...)

- Exemple (1 seule ligne) :

```
( egrep sunny < meteo | egrep -v windy ;
egrep windy  < meteo | egrep -v sunny ) >
reponse
```

Conditionnelles cachées

- Les programmeurs abusent des connecteurs && et ||
  - `commande1 && commande2` est plus court qu'écrire

    ```
    if commande1 ; then commande2 ; fi
    ```
  - `commande1 || commande2` est plus court qu'écrire

    ```
    if ! commande1 ; then commande2 ; fi
    ```
  - C'est plus court mais parfois moins lisible

8.5 – Conditionnelles

Quelques détails de plus...

Conditionnelle simple

- La syntaxe pour tester des conditions :

```
if test condition1
then
    commandes1...
elif test condition2
then
    commandes2...
else
    commandes3...
fi
```

Conditions

- Comparaisons de chaînes
  - Égalité : `"chaîne1" = "chaîne2"`
  - Différence : `"chaîne1" != "chaîne2"`
  - Chaîne non vide : `-n "chaine"`
  - Chaîne vide : `-z "chaine"`
- Connecteurs logiques
  - Et : `condition1 -a condition2`
  - Ou : `condition1 -o condition2`
  - Négation : `! condition`

Conditions, suite

- Comparaison de nombres :
  - Égalité : `nb1 -eq nb2`
  - Différence : `nb1 -ne nb2`
  - N1 < N2 : `nb1 -lt nb2`
  - N1 ≤ N2 : `nb1 -le nb2`
  - N1 ≥ N2 : `nb1 -ge nb2`
  - N1 > N2 : `nb1 -gt nb2`

Conditions sur les fichiers

- On peut vérifier beaucoup de choses, dont :
  - `test -f nom` vrai si nom est celui d'un fichier
  - `test -d nom` vrai si nom est celui d'un dossier
  - `test -x nom` vrai si nom existe : fichier ou dossier
  - `test -S nom` vrai si le fichier n'est pas vide
  - `test -N nom` vrai si le fichier a été modifié depuis sa dernière lecture (new)

Autres conditions

- On peut tester si une variable est initialisée

  `test -v` nom de la variable

  – Ex :

```
if test ! -v LOGNAME
then
    echo pas de nom de compte
    exit 1
fi
```

Variante

- Une variante utilise `[` à la place de `test` :

```
if [ condition ]
then
        Instructions...
else
        Instructions...
fi
```

- En fait, [et test sont la même commande
- Attention : mettre des espaces partout !

Autre variante intéressante

- La syntaxe `[ [ condition ] ]` est intégrée à bash
  - Mêmes opérateurs que test et [, d'autres en plus
  - Chercher CONDITIONS dans man bash
- Ex : pour comparer une chaîne à une expression régulière (type egrep) :

```
if [[ chaîne =~ 'expression régulière' ]]
then correspondance
else pas de correspondance
fi
```

Principe

- `Test`, `[` et `[[` retournent un code de retour qui vaut 0 si la condition est vraie et 1 si la condition est fausse
  - if se base sur ce code de retour pour choisir entre then et else
- Toute commande est utilisable en tant que test :

```
if egrep -q 'SUNNY' < meteoUS
then ...
```

Conditions multiples

- Pour comparer un mot à plusieurs possibilités :

```
case $variable in
valeur1) instructions1 ;;
valeur2) instructions2 ;;
...
*) instructions
esac
```

- Les valeurs peuvent contenir des jokers

Exemple de choix multiple

```
read -p "Votre choix (O/N/?)" reponse
case $reponse in
    O*|o*) echo "vous approuvez."
    ;;
    [Nn]*) echo "vous désapprouvez."
    ;;
    *) echo "vous hésitez."
esac
```

8.6 – Boucles

Voici quelques détails sur les boucles

Boucles while

- Boucle sur une condition booléenne :

```
while test condition
do
        Instructions...
done
```

- On peut aussi utiliser une commande à la place du test :

```
while ps -edf | egrep -v egrep | egrep povray
do
```

While et read

- La commande read sépare une ligne reçue sur stdin en mots, on peut s'en servir comme condition :

```
while read mot1 mot2 reste
do
    Faire quelque chose avec $mot1 $mot2 $reste
done
```

- Lancer le script avec une redirection de stdin

While, read et redirection

- On peut ainsi traiter les affichages d'une commande :

```
#!/bin/bash
ls -l | while read prots liens user group \
 taille date1 date2 date3 nom
do
    echo $nom : $taille
done
```

NB : le \ en fin de ligne veut dire que ça continue sur la ligne suivante, mais c'est vu comme une seule ligne

Arrêt d'une boucle

- On peut sortir d'une boucle à l'aide de break

```
n=0
while true
do
    echo -n 'bip '
    sleep 1
    n=$((n+1))
    if test $n -gt 20 ; then break ; fi
done
```

Passage à l'itération suivante

- On peut forcer le passage immédiat à l'itération suivante dans une boucle à l'aide de continue

```
while read ligne
do
    # ignorer les lignes vides
    if test "$ligne" = ""
    then continue
    fi
    ... suite du traitement de la ligne...
done
```

Boucle pour

- Les boucles du langage C existent en bash, avec une syntaxe assez proche :

```
for ((var=valeur0; var<valeurN; var++))
do
    faire quelque chose avec $var
done
```

- break et continue sont utilisables

Énumération d'une liste

- Le type de boucle le plus utile en bash est celui qui permet de parcourir une liste de valeurs

```
for item in valeur1 valeur2 valeur3...
do
    faire quelque chose avec $item
done
```

- À chaque tour, $item prend pour valeur le mot suivant de la liste

Applications de la boucle for

- Cette boucle a des applications importantes :
  - Parcours des paramètres :

    for param in "$@" ; do … ; done
  - Parcours des fichiers :

    for fich in *.c ; do … ; done

Affichage d'un choix

- Voici une structure de boucle très originale :

```
select choix in valeur1 valeur2 valeur3...
do
    instructions utilisant $choix
done
```

- Elle sert à proposer des choix à l'utilisateur, la variable $choix vaut l'une des valeurs écrites
- Select tourne en boucle, il faut taper ^D pour terminer cette boucle

Exemple de select

```
PS3="Votre choix ? "
select $reponse in oui non euh
do
    if [[ $reponse = oui ]]
    then
        rm *.tmp
    fi
done
```

8.7 – Définition de fonctions

On peut définir et appeler des fonctions dans un script

Définition de fonction

- La syntaxe ressemble à celle du C en plus simple :

```
Nom() {
    Instructions...
}
```

- Ensuite, on peut appeler la fonction par :

```
Nom paramètres
```

comme si c'était une commande

Paramètres d'une fonction

- À l'intérieur d'une fonction, les variables $1... contiennent les différents paramètres passés lors de l'appel

```
#!/bin/bash

Salutations() {
    echo "Bonjour $2 $1"
}

Salutations Nerzic Pierre
```

8.8 – Configuration de bash

Certains scripts sont exécutés au lancement de bash

Shell interactif ou non

- Il faut d'abord noter que bash détecte s'il est le shell de connexion (interactif) ou un simple interpréteur de script
  - La variable `$-` contient la lettre i si ce shell est interactif

Fichiers de configuration

- Au lancement, s'il est interactif, bash exécute les scripts suivant (s'ils existent) :
  1) /etc/profile
  2) ~/.bash_profile
  3) ~/.bash_login
  4) ~/.profile
- A la terminaison, il exécute le script :
  1) ~/.bash_logout

Fichiers de config, suite

- Quand bash n'est pas interactif (exécution d'un script), il exécute seulement :
  1) /etc/bash.bashrc
  2) ~/.bashrc
- À noter que toutes ces exécutions se font dans le même processus (lancement par .)

  => les variables conservent leurs valeurs

Que met-on dedans ?

- L'affectation de certaines variables :
  - PATH : les dossiers contenant les commandes connues
  - PS1, PS2 : les prompts
- Les alias :
  - alias nom='commande avec ses paramètres'

TODO

- Alias
- Signaux : voir en 2e année
- Pushd, popd, dirs
- Eval ?
- Exec ?