Langages de scripts

Frédéric Goualard

Laboratoire des Sciences du Numérique de Nantes, UMR CNRS 6004
Office #112-11

Langages *shell* : l'exemple de BASH

Le *shell*

Interpréteur de commande

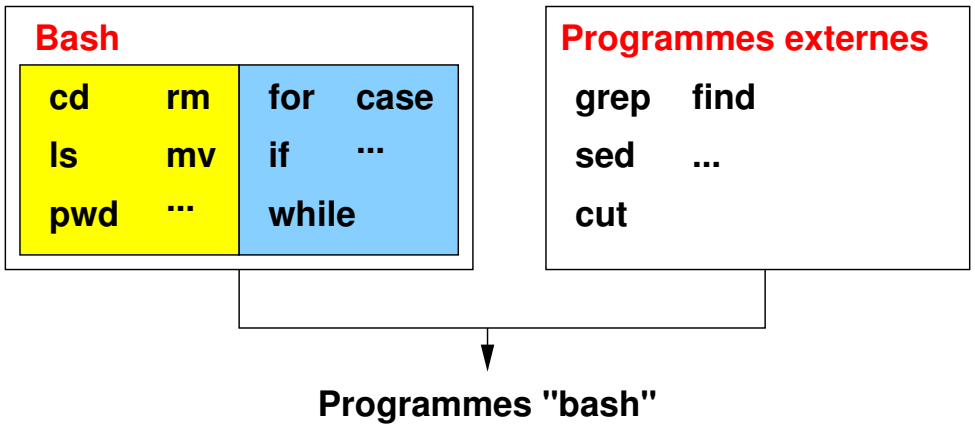
TQ vrai **FAIRE**
    lire commande au clavier
    exécuter commande
FTQ

invite ("*prompt*")

```
% pwd
/home/tutu
% cd toto
% pwd
/home/tutu/toto
```

- Quoi ?
  - Interpréteur de commandes
  - Langage de programmation
- Pour quoi ?
  - Manipulation des fichiers
  - Gestion des processus

Nombreuses variantes : **sh**, **csh**, **tcsh**, **ksh**, **bash**



Bases du *shell* BASH

- Variables :
  - Affectation : `a=5`, `toto="une jolie phrase"`
  - Utilisation : `echo $a`, `b=$toto`
- Commentaires : tout ce qui suit `#` jusqu'à la fin de la ligne

```
a=5 # ceci est un commentaire
```

- Pas d'évaluation explicite (tout est chaîne) :

```
a=5+6 # met "5+6" dans a
```

- Séquence de commandes : séparation par un retour-chariot ou un « ; » :

```
a=5; echo $a
```

Interprétation de commandes

- Appel *synchrone* : rend la main à la fin de l'exécution
  `% mozilla`
- Appel *asynchrone* : rend la main immédiatement (exécution en tâche de fond)
  `% mozilla &`

1. Lecture au clavier ou dans un fichier
2. Expansions
3. Exécution

% a = pomme **# affectation**
% echo $a

Interprétation de commandes

- Appel *synchrone* : rend la main à la fin de l'exécution
  `% mozilla`
- Appel *asynchrone* : rend la main immédiatement (exécution en tâche de fond)
  `% mozilla &`

1. Lecture au clavier ou dans un fichier
2. Expansions
3. Exécution

```
% a = pomme                                  # affectation
% echo $a          % echo pomme              # substitution de $a par sa valeur
```

Interprétation de commandes

- Appel *synchrone* : rend la main à la fin de l'exécution
  `% mozilla`
- Appel *asynchrone* : rend la main immédiatement (exécution en tâche de fond)
  `% mozilla &`

1. Lecture au clavier ou dans un fichier
2. Expansions
3. Exécution

```
% a = pomme                                  # affectation
% echo $a              % echo pomme          # substitution de $a par sa valeur
pomme                                        # exécution de la commande 'echo'
```

Les expansions

Règles :

- Pas d'expansion à l'intérieur d'apostrophes :
  a=5 ; echo '$a'** affiche "**$a"
- Expansion à l'intérieur de guillemets :
  a=5 ; echo "$a" affiche "**5**"

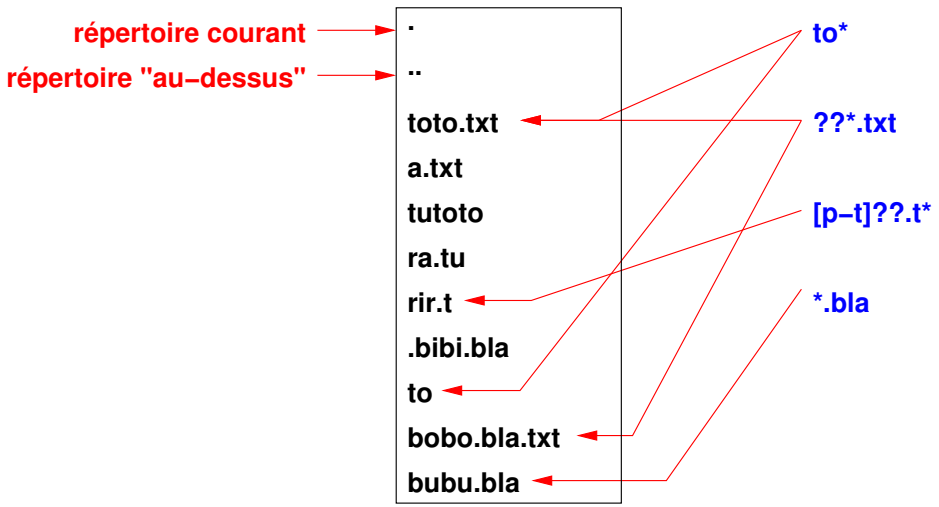
Types d'expansions :

- Variables : **$x** expansé en le contenu de **x**
- Commandes : **$(*cmd*)** expansé en le résultat de l'exécution de ***cmd*** :
  echo "Chemin courant : $(pwd)"
- Expressions arithmétiques : **$((*expr*))** :
  a=$(($a+1))
- Noms de fichiers : *globbing*

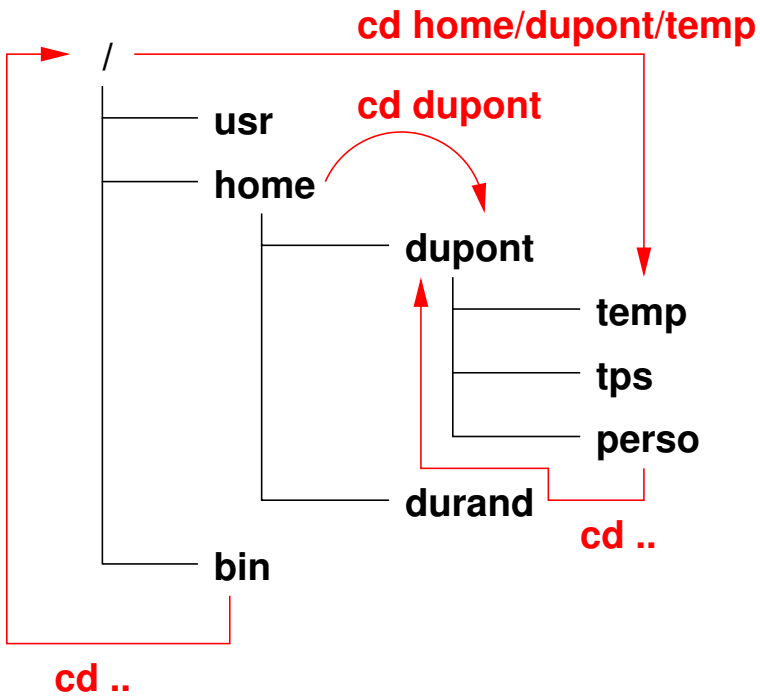
Le *globbing*

- Moyen de spécifier une liste de fichiers en intension avec un *patron*
- Trois jokers :
  - ***** : prend la place de 0 ou n caractères quelconques
  - **?** : prend la place d'un caractère quelconque
  - **[...]** : prend la place de n'importe quel caractère dans l'ensemble "**...**"
  - **{...}** : expansion de liste

Exemples :



L'arborescence des répertoires



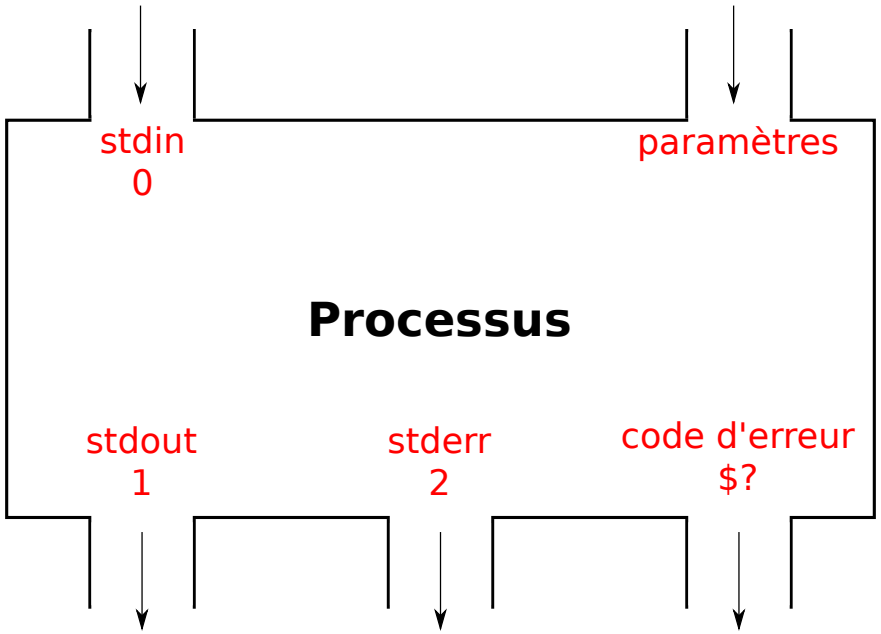
Commandes de base

déplacement :	**mv /home/durand/*.txt .**
copie :	**cp ../toto.txt ~dupont/temp**
effacement :	**rm ***
listage :	**ls /bin**
	ls
	ls ../*.txt
Position courante :	**pwd**

Les commandes

- Commandes internes : **cd**, **mv**, ...
- Commandes externes : **find**, **grep**, ...
- Scripts **bash** de l'utilisateur

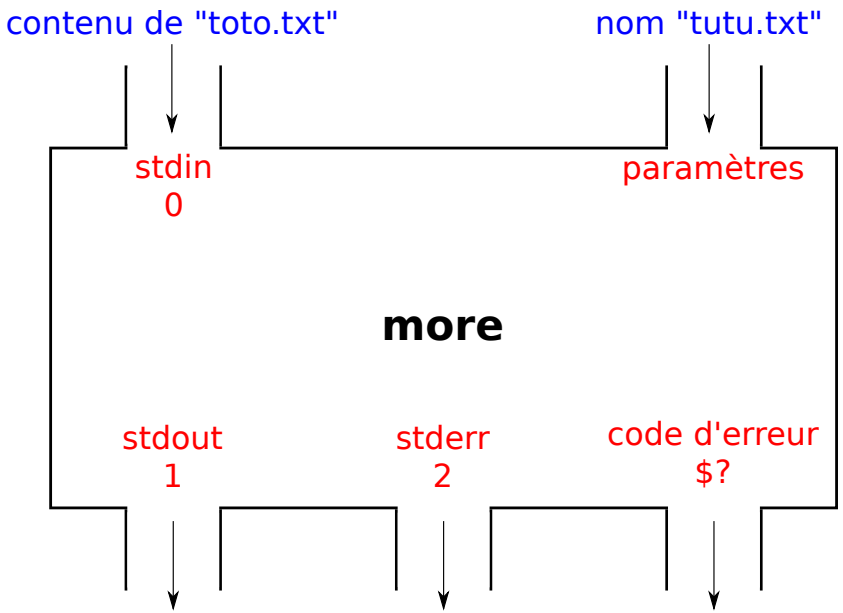
Entrées/sorties des commandes :



Les commandes

- Commandes internes : **cd**, **mv**, ...
- Commandes externes : **find**, **grep**, ...
- Scripts **bash** de l'utilisateur

Entrées/sorties des commandes :

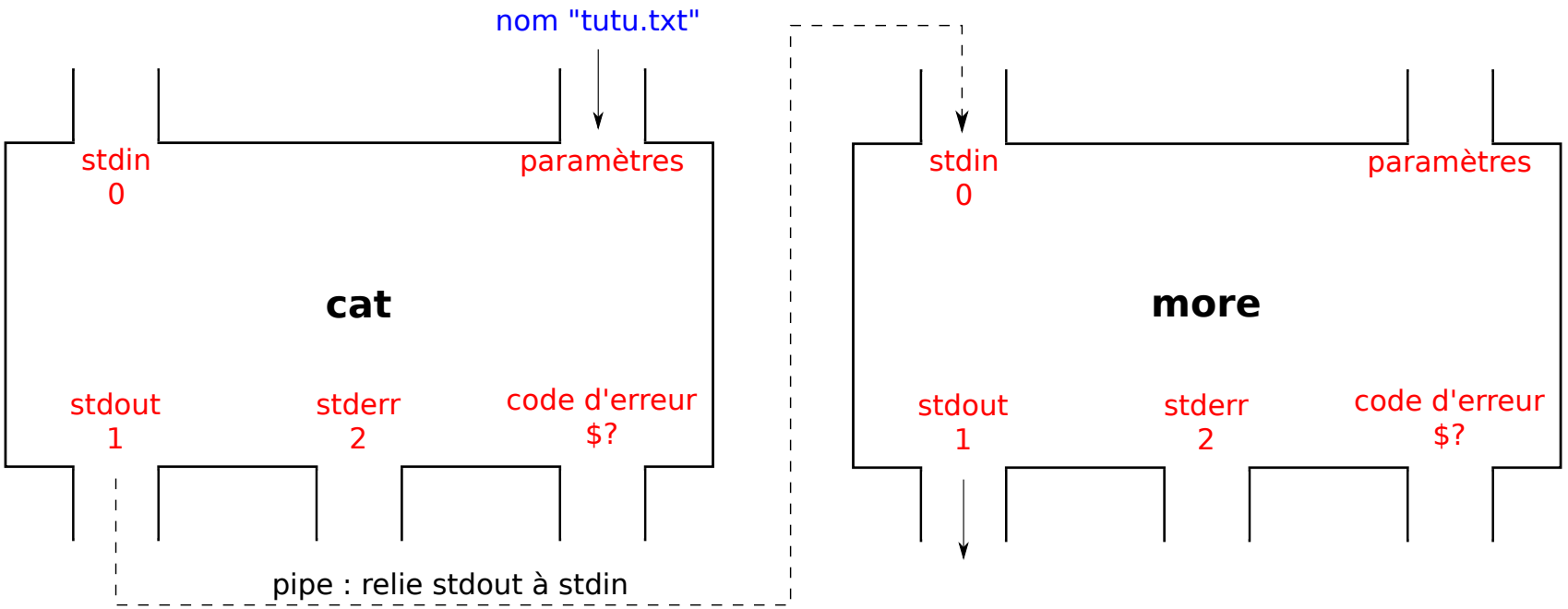



```
% more < toto.txt
% more tutu.txt
```

Les commandes

- Commandes internes : **cd**, **mv**, ...
- Commandes externes : **find**, **grep**, ...
- Scripts **bash** de l'utilisateur

Entrées/sorties des commandes :




```
% cat toto.txt | more
```

Les redirections

- Redirection de l'entrée
  (contenu d'un fichier envoyé sur **stdin**) :
  less $<$ **toto.txt**
- Redirection de la sortie
  (envoi de ce qui sort par **stdout** dans un fichier) :
  ls $>$ **liste.txt**
- Redirection de la sortie d'erreur
  (envoi de ce qui sort par **stderr** dans un fichier) :
  gcc toto.c 2$>$ **erreurs.log**
- Connection par un *pipe*
  (envoi de ce qui sort de **stdout** de **cmd1** vers **stdin** de **cmd2**) :
  cat toto.txt | **more**

Contrôle de flot

```
for var in liste; do
    cmd1
    cmd2
    ...
    cmdn
done
```

```
while [ test ]; do
    cmd1
    cmd2
    ...
    cmdn
done
```

```
if [ test ]; then
    cmd1
else
    cmd2
fi
```

```
for x in *.txt; do
   echo $x
done
```

```
x=0
while [ $x -ne 4 ]; do
   x=$(($x+1))
done
```

```
x=banane
if [ "$x" == "banane" ]; then
   echo "banane trouvée"
else
   echo "rien trouvé"
fi
```

Tests

Nombres	(**-eq** $=$), (**-ne** $\neq$), (**-le** $\leq$), (**-lt** $<$), (**-ge** $\geq$), (**-gt** $>$)
Chaînes	(**==** $=$), (**!=** $\neq$)

Les scripts

- Fichier texte contenant une séquence de commandes
- Un script s'utilise comme une nouvelle commande
- Paramètres du script : accès par **$1**, **$2**, ...
- nombre de paramètres passés : dans **$#**

Exemple :

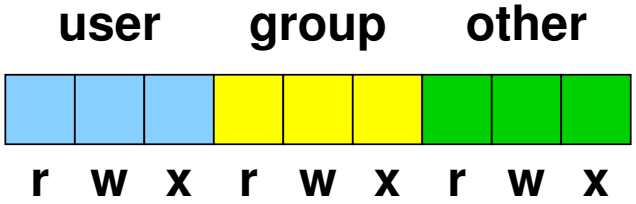
monrm.sh

```
#!/usr/bin/env bash
if [ $# -ne 1 ]; then
   echo "Utilisation : monrm.sh fichier"
   exit 1 # code d'erreur 1
fi
echo "Effacer $1 (o/n) ?"
read choix
while [ "$choix" != "o" -a "$choix" != "n" ]; do
   read choix
done
if [ "$choix" == "o" ]; then
   rm -f $1
   echo "$1 effacé"
fi
# exit 0 implicite
```

```
% monrm.sh tutu.txt
Effacer tutu.txt (o/n) ? o
tutu.txt effacé
%
```

La gestion des droits

- Droits associés à chaque fichier/répertoire
  - Lecture (**r**)
  - Écriture (**w**)
  - Exécution (**x**)



- Trois types de personnes
  - Propriétaire (**u**)
  - Groupe (**g**)
  - Autres (**o**)
- Modification des droits : **chmod**

Exemples :

```
# ajout du droit de lecture pour le propriétaire
% chmod u+r toto.txt
# retrait du droit d'écriture pour les autres
% chmod o-w tutu.bla
# Modification en octal (ajout de tous les droits à tous)
% chmod 777 pomme
```

Les expressions régulières

- *Globbing* : matching de noms de fichiers
- *Regexp* : matching de texte

Programmes utilisant les expressions régulières :

- **expr**
- **grep**
- **sed**
- **find**
- **AWK**
- ...

La puissance d'expression des regexp est supérieure au globbing.

Regexp vs. globbing

Regexp		Sens	Globbing
Basic	Extended		
`[^a-z05]`	`[^a-z05]`	N'importe quel caractère non dans l'ensemble	Idem
.	.	N'importe quel caractère	.
`^$`	`^$`	Ancres de début/fin	`^$`
`\?`	?	Répéteur (0 ou 1 fois)	1 caractère
*	*	Répéteur (0 ou n fois)	0 ou n caractères
`\+`	+	Répéteur (1 ou n fois)	+
`\(\)`	()	Groupement de patrons	—
$\backslash\{n\backslash\}$	$\{n\}$	Répéteur (exactement n fois)	Expansion d'accolades
$\backslash\{n,\backslash\}$	$\{n,\}$	Répéteur (n fois ou plus)	Expansion d'accolades
$\backslash\{n,m\backslash\}$	$\{n,m\}$	Répéteur (entre n et m fois)	Expansion d'accolades
`|`	|	Alternative (un patron ou l'autre)	—

- Tous autres caractères (+ caractères spéciaux précédés de « \ ») : sens habituel
- Les trois règles fondamentales du matching (dans l'ordre de considération) :
  1. Matching de la gauche vers la droite
  2. Matching le plus long possible ne remettant pas en cause le matching de la suite
  3. Pas de retour en arrière