Utilisation Avancée du shell (Zsh)

Fil conducteur

- Présentation générale
- Utilisation interactive avancée
- Eléments pour l'écriture de scripts

Qu'est-ce qu'un Interpréteur de Langage de Commande (ILC) ?

- Un programme comme un autre
  - Ecrit en C
  - S'exécute dans un processus
  - Capable de lire (commandes) et écrire (résultats) sur un terminal
    - Console texte
    - Graphique
  - Sait lancer des processus
    - Programmes externes
  - Souvent, le premier processus d'un utilisateur
    - Dans la cas de connexions distantes
    - Exception : session graphique
  - Fournit un **vrai langage** de programmation interpété

Algorithme d'un ILC (shell)

```
Init : interprétation des fichiers de configuration
TANT QUE (entrée != {eof, exit, logout})
FAIRE
        Saisir commande (edition « à la emacs », complétion, …)
        Analyse lexicale et syntaxique
        Expansion des paramètres
        CAS de $0 dans
                fonction, cmde interne) : l'exécuter
                fichier binaire dans PATH) :
                        créer process ayant $O comme prog.
                fichier texte) : lancer l'interpréteur
                        (#!…) ou sinon /bin/sh + texte
        FIN CAS
        SI ('&' n'est pas en fin de chaîne)
        ALORS
                attendre fin d'exécution du processus lancé
        FIN SI
FIN TANT QUE
```

Bourne shell et dérivés

- Bourne Shell (/bin/sh)
  - Le plus « classique »
  - Utilisé pour les scripts système
- Korn Shell (/bin/ksh)
  - 1ère évolution
  - nombreuses améliorations, mais encore des défauts
- Bourne Again Shell (/bin/bash)
  - Le shell GNU/FSF (shell « officiel de linux »)
  - Récupère nombreuses améliorations d'autres shell (ksh, tcsh), ajoute ses propres améliorations
- Zsh
  - Le plus complet (idem bash/ksh, plus améliorations propres)
  - Ambitieux : tout ce dont les utilisateurs pourraient avoir besoin doit être disponible dans zsh …

Fichiers (scripts) de configuration personnels

- Fichiers de configuration dans répertoire HD ($HOME)
  - Fichiers cachés (commencent par .)
    - `ls -al $HOME` (ou `ls -ald $HOME/.?*`)
  - A la longue, très nombreux (pas seulement conf. ILC !)

- Bourne Shell (sh)
  - .profile
- ksh
  - .profile + .kshrc
- bash
  - .profile + .bash_profile + .bashrc + .bash_login
- zsh
  - .zprofile + .zshrc + .zlogin + .zlogout + .zshenv

Scripts personnels de configuration : le cas Zsh

- Les scripts sont recherchés dans $ZDOTDIR
  - $HOME par défaut (ZDOTDIR non définie)
- Plusieurs contextes sont considérés
  - shell non interactif
    - .zhsenv
  - shell interactif
    - .zshenv, .zshrc
  - shell interactif de login
    - .zshenv, .zprofile, .zshrc, .zlogin
  - Lors de la terminaison :
    - .zlogout
- Attention : cas particulier des sessions graphiques
  - Connexion, mais pas login (.zshenv + .zshrc)

Scripts de configuration système

- Un équivalent système pour chaque script personnel
  - /etc/zshenv, /etc/zprofile, /etc/zshrc, /etc/zlogin
  - Maintenus par root
  - Appelé juste avant son équivalent « utilisateur »
    - sauf $HOME/.zlogout avant /etc/zlogout

- Parfois /etc/zprofile exécute en cascade /etc/profile
  - Selon variante système Unix
  - profile : commun à tous les « Bourne Shell »
    - /etc/profile peut lui même exécuter d'autres scripts...
      - par exemple /etc/profile.d/*.sh sous certains linux …

Script de configuration : Illustration

- Instruction « echo 'lecture *fichier*' » au début
- Cas shell système (sh) non interactif :

```
[odalle@petiote]~% sh -c date
dim sep  4 17:24:10 CEST 2005
```

- Cas zsh non intercatif

```
[odalle@petiote]~% zsh -c date
lecture /etc/zshenv
lecture /user/odalle/home/.zshenv
dim sep  4 17:24:22 CEST 2005
```

- Cas zsh interactif :

```
[odalle@petiote]~% zsh
lecture /etc/zshenv
lecture /user/odalle/home/.zshenv
lecture /etc/zshrc
lecture /user/odalle/home/.zshrc
```

Script de configuration : Illustration (2)

- Cas zsh interactif de login :

```
[odalle@petiote]~% zsh -l
lecture /etc/zshenv
lecture /user/odalle/home/.zshenv
lecture /etc/zprofile
lecture /user/odalle/home/.zprofile
lecture /etc/zshrc
lecture /user/odalle/home/.zshrc
lecture /etc/zlogin
lecture /user/odalle/home/.zlogin
[odalle@petiote]~% exit
lecture .zlogout
lecture /etc/zlogout
```

Utilisation interactive de zsh

- Mécanismes de redirection
  - Descripteurs
  - Redirections élémentaires
  - Redirections avancées

- Mécanismes de substitution
  - Historique
  - Commandes
  - Processus
  - Noms de fichiers (*globbing*)

Descripteurs

- Chaque processus possède des **descripteurs de flux**
  - numérotés
  - Chaque descripteur peut être lié
    - à un fichier
    - un (autre) processus (tube = *pipe*)
  - Les descripteurs sont orientés
    - lecture, écriture, duplex
- 3 descripteurs actifs par défaut :
  - 0 désigne l'entrée standard du processus
  - 1 désigne la sortie standard
  - 2 désigne la sortie standard d'erreur
- Cas des processus shell
  - Descripteurs par défaut liés à la console

Redirections élémentaires

- < FIC
  - Ouvre le fichier FIC et le lie au descripteur 0 (stdin)
- <> FIC
  - Ouvre le fichier FIC en lecture/ecriture et le place sur stdin
  - Rarement utilisé...
- > FIC
  - Ouvre le fichier FIC et le lie au descripteur 1 (stdout)
- >> FIC
  - Ouvre le fichier FIC en mode "APPEND" et lie à stdout
- <<MOT, <<-MOT
  - Redirection « Here Document »
    - lignes suivante jusqu'à apparition de MOT envoyé sur stdin
    - permet de simuler une frappe interactive (shell script)

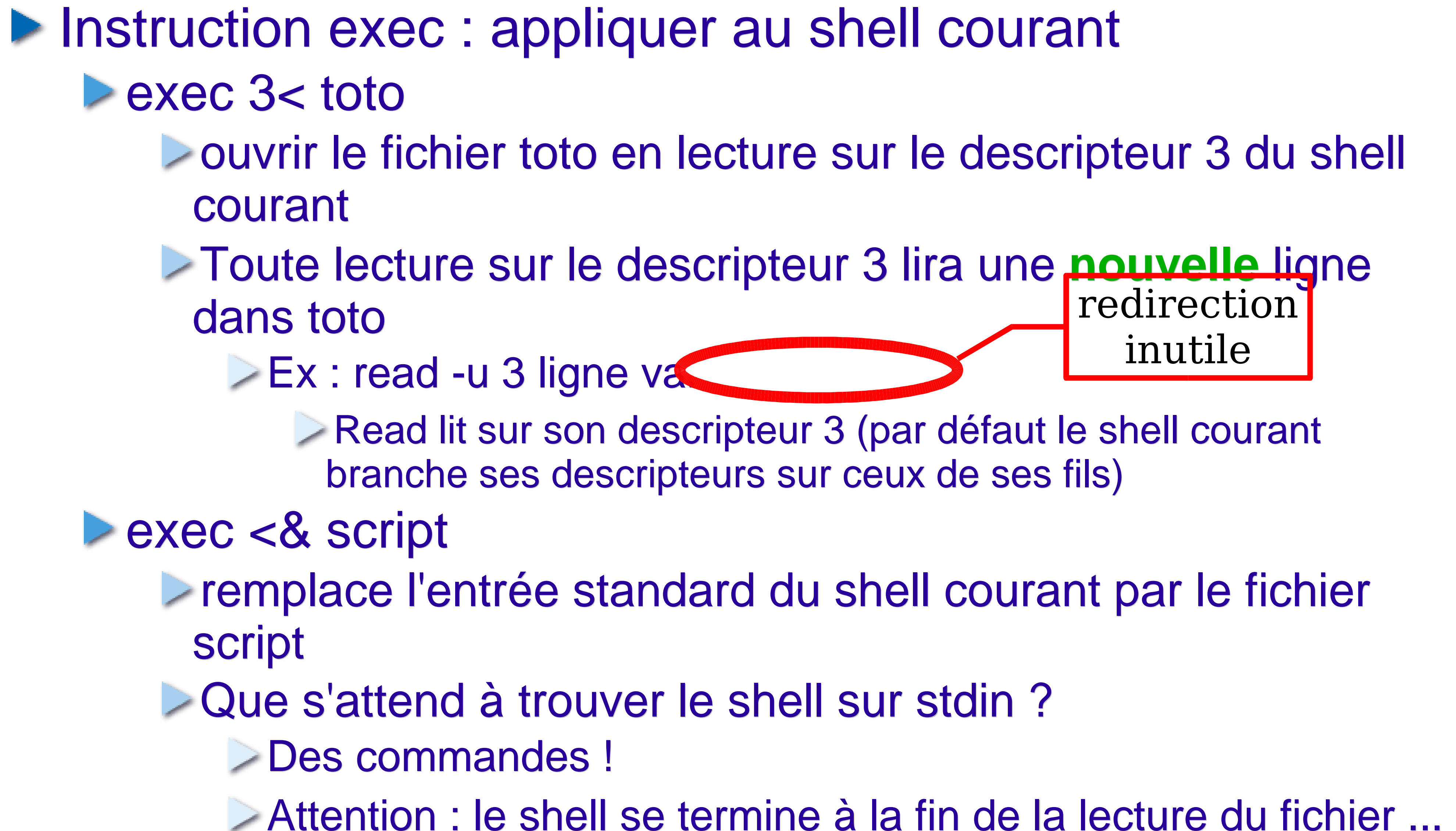
Redirections élémentaires de descripteurs quelconques

- Redirection en lecture : commande N1< file
  - ouvrir fichier en lecture et l'associer au descripteur N1 du processus qui execute « commande »
  - Exemples :
    - read -u 3 var 3< toto
      - lire une ligne sur le descripteur 3 et placer le résultat dans la variable 'var'
    - while read -u 3 var ; do echo "ligne lue : $var" ; done 3< toto
- Redirection en écriture : commande N1> file
  - Rediriger ce qui "sort" par le descripteur N1 vers le fichier file
  - Exemples :
    - find ./ 2> find.log

Manipulations de descripteurs

- Equivalent shell des fonctions dup, dup2 et close en C
  - commande N1<&N2
    - L'entrée associée à N1 est remplacée par celle associée à N2
    - Si N1 absent, alors stdin est remplacée
  - commande N1>&N2
    - La sortie N1 est redirigée vers le descripteur N2
    - Si N1 absent, alors c'est stdout qui est redirigée
  - commande <&- : Fermeture de stdin
  - commande >&- : Fermeture de stdout
- A quoi ça sert ???
  - Surtout utile dans les scripts
  - Permet de créer ou supprimer des branchements

Manipulations des descripteurs du shell courant



Exemples de redirections

- find / 2>&1
  - Les messages d'erreur sont redirigés vers la sortie standard (en plus des messages normaux)
  - Très courant avant un *pipe*
    - find / 2>&1 | more
- exec 3> toto
  - Le descripteur 3 est redirigé en écriture sur le fichier toto : tout ce qui sera écrit sur le decripteur 3 ira dans le fichier toto
- ls –al >&3
  - Le résultat de 'ls –al' est envoyé vers le descripteur 3 (fichier toto)
- exec 3>&-
  - Le descripteur numéro 3 (fichier toto) est fermé

Mécanismes de substitution

- Le shell transforme systématiquement ce qui est saisi !
  - Dès que la ligne de commande est "envoyée" (entrée/CR)
  - Ce qui est réellement exécuté **n'est pas ce qui est saisi**
    - On peut prévisulaiser le résultat en invoquant la complétion
      - touche tabulation
- Plusieurs types de transformation, dans l’ordre suivant :
  - History expansion (shell interactif)
  - Alias expansion
  - Process substitution + Parameter expansion + Command substitution + Arithmetic expansion + Brace expansion
  - Suppression des \, ', et "
  - Filename expansion
  - Filename génération

Mécanisme d'historique

- Réutilisation de (parties) de commandes précédentes
  - Sauvegarde des commandes dans l'historique (taille = HISTSIZE)
- Substitution d'historique : commence par '!'
  - 3 catégories
    - Désignation d'événement (= commande complète)
    - Désignation de mots
    - Modificateurs
  - Syntaxe générale :

  !evt:word…:mod…

Mécanisme d'historique : Exemples

- [1] cp toto titi tata dest/
- [2] !!0- dest2/
  - tous les mots de la dernière commande sauf le dernier
  - => [2] cp toto titi tata dest2
- [3] ls !-2:$
  - Le dernier mot de l'avant dernière commande
  - => [3] ls dest
- [4] echo "!?cp?" > exemple.txt
  - La dernière commande qui contient cp
  - => [4] echo "cp toto titi tata dest2" > exemple.txt
- [5] mv !!$!!$:r.foo
  - le dernier mot de la dernière commande ; le dernier mot de la dernière commande dans lequel on a supprimé le .txt
  - => [5] mv exemple.txt exemple.foo

Mécanisme de substitution de commande

- De la forme $(CMD) ou `CMD`
  - La forme est substituée par le résultat de CMD
  - Exemples :
    - `echo "Répertoire courant : `pwd`"`
    - `echo "Votre groupe est : $(id -gn )"`
  - Les éventuels sauts de lignes en fin de résultat sont éliminés
  - Lorsque l'option GLOB_SUBST est positionnée, le résultat subit la substitution "filename génération" (cf plus loin)

Quelques effets du ~

- ~ seul : équivalent de $HOME
- ~+ : équivalent de $PWD
- ~- : répertoire avant le dernier cd
- ~login : désigne le homedir de l'utilisateur 'login'

Expansion/Génération de noms de fichiers (globbing)

- Désigner de façon simples les paramètres des commandes de type fichier
  - sans avoir besoin de les désigner tous explicitement
  - à l'aide d'expressions régulières
- méta-caratères
  - * : 0 ou plusieurs caractères quelconques
  - ? : un caractère quelconque
  - [xyzt] : un carctère parmis x, y z ou t
  - [^xyzt] : tout sauf un caractère parmis x,y,z ou t
  - <nb1-nb2> : un nombre compris entre nb1 et nb2
  - <-nb2>, <nb1-> : resp un nombre < nb2 ou un nb > nb1
  - <> : un nombre quelconque
  - (exp1|exp2) : exp1 ou exp2

Exemples de globbing élémentaire

- ls f* f?o
  - fichiers qui commencent par 'f'
  - fichiers de 3 lettres commençant par f et terminant par o
- ls f[aeiou]o
  - fichier de 3 lettres commençant formé de f + 1 voyelle + o
- ls (f?o|b?r)
  - fichier soit de la forme f?o soit de la forme b?r
- ls <0-9>*
  - fichier commençant par un nombre entre 0 et 9
- ls *<-15>
  - fichier terminant par un nombre < 15
- ls *<>*
  - fichier contenant un nombre quelconque

Globbing étendu

- L'option EXTENDED_GLOB doit être validée
  - setopt EXTENDED_GLOB
- Deux nouveaux meta-caractères :
  - ^ : tout sauf ce qui suit
  - exp# : 0 ou plusieurs occurrences de exp
  - exp## : 1 ou plusieurs occurrences de exp
- Exemples :
  - ls ^*.c
  - file *.[^(o|out)]
  - ls (fo)#* commence par 0 ou plusieurs fois 'fo'
  - ls fo##* commence par fo…ooo
  - ls (*TP*/)##* : contenu des (sous-)sous-répertoires contenant 'TP' dans leur nom

Globbing des fichiers cachés

- L'option GLOB_DOTS doit être validée
  - Le globbing s'applique alors aussi aux fichiers cachés (commençant par .)
- On peut éventuellement s'en passer
  - Fichier caché = commence par '.' et suivi au minimu d'un 2e caractère (car . est le nom du répertoire courant)
  - ls .?* : tous les fichiers cachés (et seulement les cachés)
  - ls (.?*|*) : tous les fichiers sans exception
    - Rem : résultat identique avec ls .?* *

Globbing : qualificatifs sur le type

- Une expression en fin de motif recherché permet de "spécialiser" le globbing :
  - exp(exp2)
  - qualificatifs de exp2 :
    - @ = lien symbolique
    - / = répertoire
    - r : lisible ; R : lisible other ; A : lisible groupe
    - w, W, I : ecriture pour user, other, groupe
    - x, X, E : exécution pour user, other, groupe
    - u : propriétaire, u[lui] : 'lui' est propriétaire
    - s: setuid, S: setgid
  - Combinaisons ET (), OU (,), NON (^)

Exemples d'utilisation des qualificatifs sur le type

- ls *(@)
  - lister les liens symboliques seulement
- ls *[aeiou](x)
  - lister les fichiers qui se terminent par une voyelle qui ont la permission d'exécution
- ls *to(/)
  - lister les répertoires dont le nom se termine par 'to'
- ls *(UX^@)
  - lister les fichiers qui m'appartiennent, exécutables pour tous, mais qui ne sont pas des liens
- ls *(x,X)
  - lister les fichiers qui sont exécutables pour moi ou pour les autres

Autres mécanismes de globbing

- ** : un chemin quelconque
  - ls **/*.c : rechercher et lister tous les fichiers qui se terminent par .c à partir du répertoire courant
- =bin : nom complet de l'exécutable 'bin'
  - ls =emacs
    - /usr/local/bin/emacs
  - Equivalent à : ls `which emacs`
  - Desactivé par l'option "noequals"
    - Utile quand on utilise régulièrement = dans les noms de fichiers

Variables, expressions

- Par défaut : variable = chaîne de caractères
  - Déclaration inutile
  - Initialement vide
- Variables entières
  - typeset –i var1 [var2 ...] : var1 et var2 de type entier
    - typeset –i e1 e2
    - echo $e1 $e2
    - => 0 0
  - Expressions arith : délimitées par des doubles parentheses
    - ((i1 = $i2 +1))
    - i1=$(expr $i2 + 1)

Variables de type chaîne

- Chaîne : Type par défaut des variables
- Affectation : gare aux espaces !
  - toto=bonjour
    - Ok
  - toto = bonjour
    - erreur
- Protection
  - guillemets (")
    - echo "$toto $USER"
    - => bonjour odalle
  - quote (')
    - echo '$toto $USER'
    - => $toto $USER
  - backslash (\)
    - echo $toto \$USER
    - => bonjour $USER
- Combinaison ?
  - echo "'$toto \$USER'"
  - => 'bonjour $USER'

Variables de type tableau

- Déclaration implicite lors de l'initialisation
  - tab=(e1 e2)
  - Indices : de 1 à N, mais 0 fonctionne
    - echo $tab[0] => e1
    - echo $tab[1] => e1 aussi !
    - echo $tab[2] => e2
  - Indices négatifs
    - echo $tab[-1] => e2
    - echo $tab[-2] => e1
  - Autres manipulations
    - echo $#tab => 2 (nb éléments)
    - $tab[*] => totalité du tableau (1 seul tenant)
    - $tab[@] => totalité du tableau (éléments séparés)

Mécanismes de substitution de variables

- **`${var:-word}`**
  - remplacé par le contenu de la variable si elle existe, ou par word sinon
- **`${var:=word}`**
  - idem mais variable est affectée avec word
- **`${var#exp}`**
  - retranche à var le plus petite chaîne en partant de la gauche correspondant à exp
- **`${var##exp}`**
  - idem mais retranche la plus grande chaine
- **`${var%exp}`, `${var%%exp}`**
  - idem mais en partant de la droite

Exemples de substitutions de variables

- Lecture d'une réponse avec valeur par défaut:

```
def_rep="yes"
echo -n "Votre choix [$def_rep] ?"
read rep
choix=${rep:-$def_rep}
echo $choix
```

- Suppression du chemin
  - `chemin=`pwd``
  - `echo ${chemin##/*/}`

Eléments pour l'écriture de shell-scripts

- Shell = langage de commandes
- Shell Script = programme écrit en langage de commandes
- Rôles du shell
  - Combiner/relier des commandes existantes avec redirections, |, ||, && et instructions de contrôle
    - Permettre la construction de nouvelles commandes
  - Regrouper dans des fichiers des suites de commandes
    - Exécution de tâches répétitives

Variété des instructions

- Toute une "palette" d'instructions :
  - Mémoriser (variables, types, arguments)
  - Compter (expressions arithmétiques et logiques)
  - Tester (test, [[ ]])
  - Bifurquer (if, case, ||, &&)
  - Boucler (while, for, select, break, continue)
  - Interagir par E/S (read, print, <, >)
  - Sous-programmes : function, unfunction, return, exec, source, export

if ... then ... else ...

```
if test1
then
    instruction
elif test2
then
    instruction
else
    instruction
fi
```

Tests conditionnels

- 1ere possibilité : **toutes** les commandes ont un résultat
  - **if** mkdir /toto ; **then** echo reussite ; **else** echo echec ; **fi**
- 2e possibilité : instruction test et dérivés (bourne shell)
  - if **test ...** ; then ...
  - eqv à : if **[...]** ; then ...
    - liste des tests possibles (man test ...)
      - Opérateurs logiques :
        - -a : AND, -o : OR, ! : NOT, (EXPR) ...
      - tests arithmétiques
        - -le, -lt, -gt, -ge, -ne, -eq
      - tests fichiers
        - -f, -x, -d, -l, -s, ...
      - test chaînes
        - STR1 = STR2, STR1 != STR2, -z STR (vide), -n STR (non vide)

Tests conditionels (syntaxe spécifique zsh)

- [[test]]
  - Plus proche de la syntaxe C :
    - STR == PATTERN
    - STR < PATTERN (ordre alphab)
    - STR > PATTERN
    - EXP1 && EXP2 : ET logique
    - EXP1 || EXP2 : OU logique
  - Des tests supplémentaires
    - -o OPTION : option zsh positionnée
  - Attn : Quelques différences avec test
    - -a FILE : fichier existant (au lieu du ET logique)
    - -o OPTION

Exemples de tests

- Test avec syntaxe zsh

```
if [[ -d /opt/bin && -a /opt/bin/java ]]
then
   /opt/bin/java toto.class
fi
```

- Idem avec syntaxe Bourne shell
  - if [-d /opt/bin -a -e /opt/bin/java] ; then ... ; fi
- Version simplifiée sans if !
  - [-d /opt/bin -a -e /opt/bin/java] && /opt/bin/java toto.class

```
if [ "$USER" = "odalle" -a -f $HOME/.odalle ]
then
    source $HOME/.odalle
fi
```

Boucle for

- syntaxe

  for *var* **in** *list*

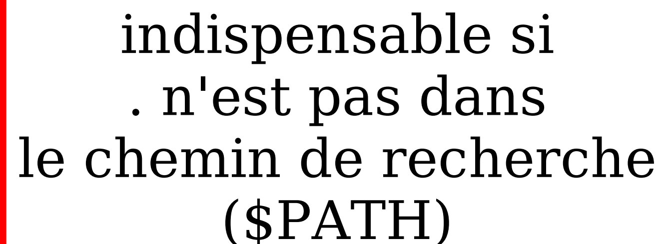
  do

  instruction

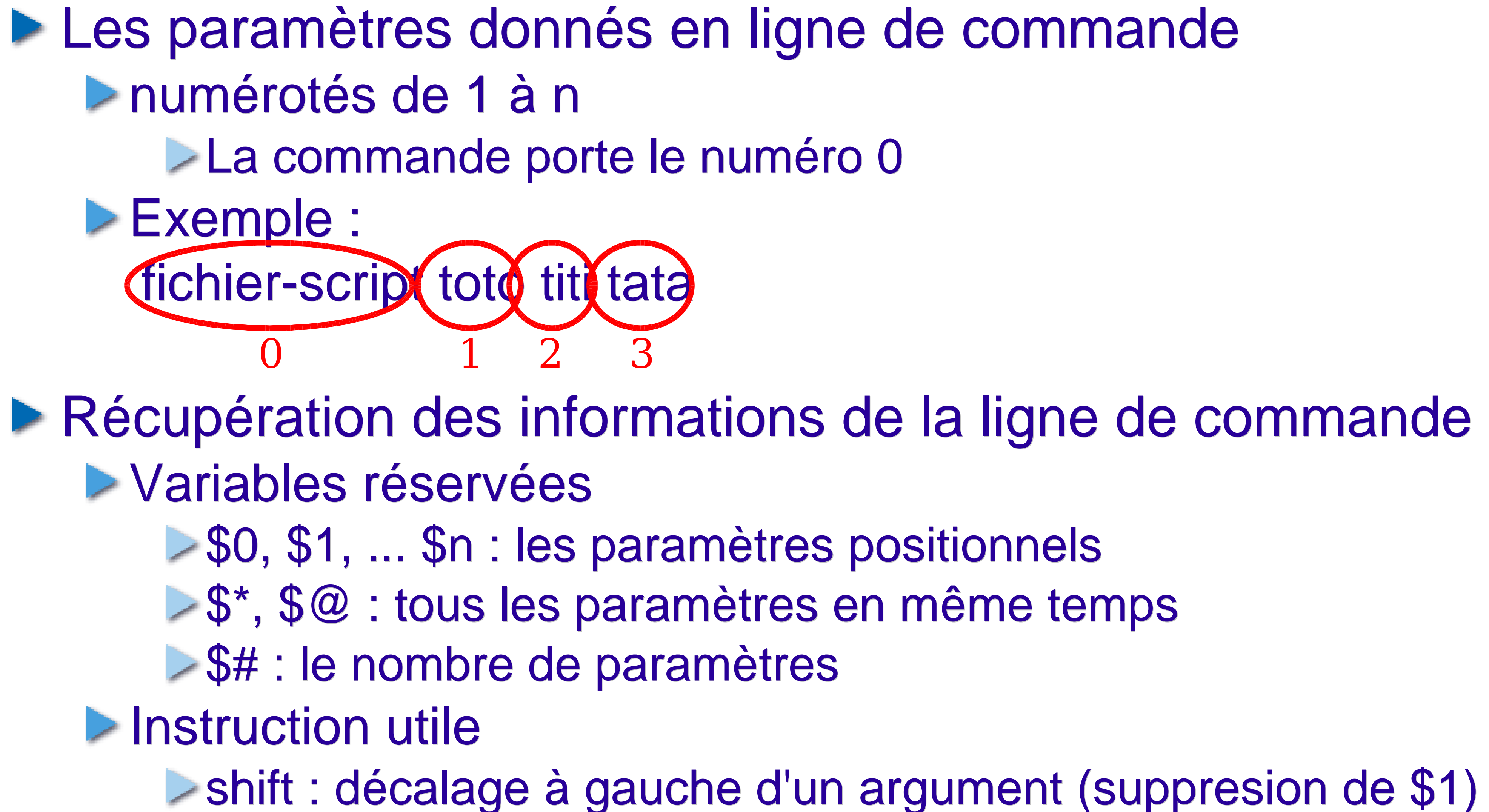
  done
- *var* : un nom de variable
- *list* : un liste de chaînes séparées par des espaces
  - une liste de fichiers (f*, /opt/bin/em*, **/*.c, ...)
  - le contenu d'un tableau ("$tab[@]")
  - une liste explicite (a b c d ...)
- Exemples :
  - for i in a b c d ; do touch fic_$i ; done
  - for i in *.txt ; do mv $i $i.old ; done

Création d'un shell-script

- Créer un fichier texte (avec son éditeur favori)
- Préciser l'interpréteur sur la **première** ligne
  - `#! /chemin/vers/interpreteur`
  - Ex : `#! /bin/zsh`
- Donner les permissions d'exécution
  - `chmod a+x fichier-script`
- Lancer exécution
  - `./fichier-script`



Paramètres positionnels



Exemple de script utilisant les paramètres

```
#! /bin/zsh
echo "Exécution de la commande $0"
echo "Nombre de paramètres : $#"
echo "Liste des paramètres : $*"

for i in "$*"; do echo "boucle1: $i" ; done
for i in "$@"; do echo "boucle2: $i" ; done

if [ $# -ge 1 ]
then
    echo "Premier paramètre : $1"
else
    echo "Pas de paramètres !"
fi
while [ $# -ge 1 ]
do
    echo "paramètre : $1"
    shift
done
```

Tester les cas d'exécution suivants

```
# test.sh
# test.sh toto
# test.sh toto titi tata
```