Systèmes informatiques

Franck Guingne,
sur la base du cours d'Olivier Lecarme

Cours Licence 1; Semestre 2

2009–2010

Cinquième cours : le shell

Plan en cours

Retour sur les jokers

- mentionner un fichier dans une commande peut se faire :
  - explicitement, avec un chemin absolu
  - moins explicitement, avec un chemin local ou relatif
  - encore moins explicitement, grâce à l'utilisation de jokers
- trois formes de jokers acceptées par tous les shells :
  - le caractère `*` dénote une suite de caractères quelconques, éventuellement vide
  - le caractère `?` dénote un caractère quelconque
  - la construction `[ensemble]` dénote un caractère pris dans l'ensemble :
    - `[aeiouy]` dénote les voyelles minuscules non accentuées
    - `[A-Z]` dénote les lettres majuscules
    - `[0-9a-z]` dénote les chiffres et lettres minuscules

Encore les jokers

- la notation est interprétée par le shell lui-même
- elle est remplacée dans la commande par sa signification
- le programme appelé ne la voit donc pas
- si aucun fichier n'est décrit par la notation, c'est une erreur et la commande n'est pas appelée
- si plusieurs fichiers sont décrits, la commande reçoit une liste de noms
- deux particularités importantes :
  - le caractère `/` ne peut pas être décrit par un joker
  - le caractère `.` en début de mot (ou après un `/`) ne le peut pas non plus
- des notations plus puissantes sont offertes par Zsh

Accès aux fichiers exécutables

- le nom d'une commande (premier mot de la commande) est le plus souvent celui d'un fichier exécutable
- ce nom n'est en général ni absolu, ni relatif, ni local
- le fichier peut se trouver dans plusieurs répertoires, par exemple :
  - `/usr/bin`
  - `/usr/local/bin`
  - `/usr/bin/X11`
  - etc.
- pour éviter à l'utilisateur de se rappeler les emplacements des programmes exécutables, le shell :
  - utilise la *variable d'environnement* `PATH`
  - cherche dans les chemins mentionnés dans cette variable le premier répertoire contenant un fichier exécutable du nom indiqué

Les variables d'environnement

- tout processus peut utiliser plusieurs variables d'environnement
- il peut les affecter, les examiner, les communiquer aux processus qu'il lance
- on trouve par exemple :
  - `HOME`, répertoire personnel de l'utilisateur
  - `TERM`, type de terminal du processus
  - `DISPLAY`, nom du terminal graphique pour le serveur X
  - `PATH`, liste de chemins absolus séparés par le caractère `:`
- on affiche la valeur d'une variable par la notation
  `echo $PATH`
- on peut la modifier par la commande interactive `vared`
  `vared PATH`

Fichiers standard

- chaque programme de Unix accède à au moins trois *fichiers standard* :
  - entrée (numéro 0)
  - sortie (numéro 1)
  - sortie d'erreur (numéro 2)
- sauf indication contraire, ils sont associés au terminal du processus :
  - entrée depuis le clavier
  - sortie dans la fenêtre
  - sortie d'erreur dans la fenêtre

Re-direction

- tout fichier standard peut être *re-dirigé* vers un autre fichier
- c'est fait par le shell, indépendamment du programme qui s'exécute :
  - la notation `< fichier` re-dirige l'entrée standard, qui est prise sur le fichier indiqué plutôt qu'au clavier
  - la notation `> fichier` re-dirige la sortie standard, qui est envoyée dans le fichier indiqué plutôt que dans la fenêtre
  - la notation `2> fichier` re-dirige la sortie d'erreur
- quelques règles :
  - le fichier en entrée doit exister
  - le fichier en sortie ne doit pas exister (vrai avec zsh, pas nécessairement avec d'autres shell comme par exemple : Bash)
  - la notation `>> fichier` concatène la sortie standard à la fin d'un fichier existant

enchaîner des commandes

- on peut enchaîner plusieurs commandes :
  - sans rapport entre elles, grâce à l'opérateur `;`
    `cd SI/TP5 ; ls`
  - en re-dirigeant la sortie standard de la première sur l'entrée standard de la deuxième, grâce à l'opérateur `|` (*tube ou «pipe »en anglais*)
    `grep 'toto' fichier | less`
- les programmes qui transforment leur entrée standard en leur sortie standard sont des *filtres*

Quel shell utilise-t-on

- le shell utilisé par défaut est le *programme de départ* :
  - il est déterminé par l'administrateur à la création du compte
  - l'utilisateur peut en changer par la commande `chsh`
  - il ne peut choisir que dans une liste établie par l'administrateur
  - pour tous vos comptes, c'est Zsh, qui a l'avantage d'inclure toutes les possibilités de tous les autres
  - le seul shell dont on peut garantir l'existence dans toute installation de Unix est Sh (`/bin/sh`)
- il est très facile de changer de shell de manière temporaire :
  - on appelle le shell voulu comme on le ferait de tout autre programme
  - le processus lancé est interactif et cache le shell sous-jacent
  - quand on termine ce shell (`exit`), on revient au shell interactif sous-jacent

Corriger la ligne de commande

- l'utilisateur communique avec le shell interactif en :
  - composant une ligne de commande (frappe)
  - la corrigeant (commandes similaires à celles d'Emacs)
  - la soumettant (touche `RET`)
- les commandes de correction sont les mêmes qu'en Emacs, en particulier :
  - `C-a` amène en début de ligne
  - `C-e` amène en fin de ligne
  - `M-DEL` efface le mot de gauche
  - `M-b` recule d'un mot
  - `M-f` avance d'un mot
  - `C-k` efface toute la fin de ligne
  - etc.
- il faut parfois taper `ESC b` plutôt que `M-b`

Utiliser l'historique

- le shell conserve un *historique* des commandes :
  - immédiat, accessible par des commandes spécifiques
  - plus permanent, conservé dans un fichier local, par exemple `~/.sh_history`
  - la longueur de ce fichier est paramétrable
- `C-p` ou ↑ affiche la commande précédente
- on peut la corriger, la soumettre
- on peut en chercher une autre dans l'historique
- `C-n` ou ↓ redescend
- `C-r` ou `C-s` effectue une recherche incrémentale dans l'historique :
  - `RET` soumet la commande sans la modifier
  - une touche de déplacement la laisse prête à être modifiée et soumise

Achèvement automatique

- la touche `TAB` demande au shell de compléter le mot en cours :
  - si c'est le premier mot de la commande, il cherche dans les fichiers exécutables (variable `PATH`)
  - si c'est un mot suivant, cela dépend de la commande
  - c'est en particulier utile pour les noms de fichiers avec des chemins compliqués
  - s'il y a plusieurs choix (début de nom ambigu), le shell les affiche et propose le premier ; on peut :
    - l'accepter et poursuivre la commande
    - demander le suivant de la liste (`TAB`)
    - le corriger pour lever l'ambiguïté
- le shell peut aussi proposer une correction des fautes de frappe :
  - nom de commande proche
  - nom de fichier proche
  - on peut refuser (`n`) ou accepter (`y`) la correction, renoncer à la commande (`a`), ou la corriger (`e`)

Autres commandes de préparation de la ligne

- `M-a` soumet la commande, puis la re-propose
- `C-o` soumet la commande puis propose la suivante de l'historique
- `C-l` efface toute la fenêtre puis réaffiche l'invite
- `M-h` appelle `man` pour la commande actuelle, puis la propose
- `M-?` appelle `whence` pour la commande actuelle (donne le chemin absolu de l'exécutable) puis la propose
- etc.

Suspendre ou terminer un processus de premier plan

- une des erreurs les plus fréquentes consiste à lancer en premier plan un processus depuis le shell
- le processus du shell devient donc inaccessible
- on peut intervenir quand même :
  - la commande `C-c` fait se terminer immédiatement le processus de premier plan
  - la commande `C-z` suspend l'exécution du processus de premier plan
- le processus peut résister à la première commande, pas à la deuxième
- on peut agir de trois manières sur un processus suspendu :
  - `fg` le fait reprendre au premier plan
  - `bg` le fait reprendre en arrière-plan ; il devient une *tâche*
  - `kill` lui envoie un signal (voir plus loin)

Plan en cours

Fichiers de script

- un *fichier de script* est un fichier de texte qui constitue un programme directement interprétable
- en particulier c'est un programme rédigé dans le langage du shell
- contrairement à un programme dans un langage de programmation ordinaire, le script n'est pas traduit
- fonctionnement quand on demande d'exécuter un script :
  - il faut la permission d'exécution
  - il faut atteindre le fichier par la variable `PATH` ou un chemin absolu
  - le shell demande au noyau d'exécuter le fichier
  - pour un script, le noyau appelle le shell pour interpréter le fichier
  - si le fichier commence par la ligne
    `#!nom absolu de shell`
    c'est ce shell qui interprète le fichier, sinon c'est Sh

Mots et délimiteurs

- le shell lit, analyse et exécute les commandes
- il décompose la commande en une suite de mots et opérateurs
- un *mot* est une suite de caractères délimitée par un séparateur à chaque extrémité :
  - espaces ou tabulations
  - fin de ligne
  - opérateurs
  - signes de citation

Opérateurs

- les *opérateurs* sont en particulier :
  - redirection
    `> >> >& >| < << <<- <& <> >! >>| >>! <<< ><& <&- >&-`
  - autres opérateurs
    `| & ; ( ) || && ; ; (( )) |&`
- la fin de ligne ne soumet la commande que si cette dernière est complète, sinon le shell analyse la ligne suivante

Citation, substitution, commentaire

- les mécanismes de *citation* permettent de :
  - changer la signification des caractères spéciaux
  - en particulier de mettre des blancs ou fins de ligne dans les mots
- les mécanismes de *substitution* permettent de :
  - remplacer une partie de la commande par une chaîne de caractères
  - cette chaîne est traitée comme faisant partie de la commande
  - le shell en recommence l'analyse, mais sans nouvelle substitution
- les *commentaires* sont utiles dans les fichiers de scripts

```
# ici un commentaire intéressant
```

Modèle généralisé

- les *modèles généralisés* de Zsh enrichissent l'idée de joker
- les notations déjà vues sont :
  - les *jokers* `*` et `?`
  - la notation `[...]` pour un ensemble de caractères
- Zsh reconnaît beaucoup d'autres notations :
  - `[^...]` prend le complément de l'ensemble
  - `<`n_1`-`n_2`>` dénote un nombre dans l'intervalle indiqué
  - chaque borne peut être omise
  - `^`*`modèle`* nie le modèle
  - `(`*`modèle`*$_1$`|`*`modèle`*$_2$`)` accepte les deux modèles
  - etc.

Commandes simples

- une *commande simple* est :
  - interne au shell (*commande intrinsèque*)
  - l'appel d'un programme
- elle est formée d'une suite de mots :
  - le premier mot est le nom de la commande
  - les mots suivants sont les arguments
  - la commande peut être précédée par des affectations locales à des variables d'environnement
    `TERM=vt100 emacs -nw`
  - l'affectation est elle-même une commande simple :
    `PRINTER=mips3o`
  - attention, pas de blancs dans l'affectation !

Commandes intrinsèques

- les *commandes intrinsèques* sont internes au shell, et ne correspondent pas au lancement d'un processus
  - `cd`, `pwd`, `fg`, `bg` sont déjà vues
  - `exit` fait se terminer le shell
  - `kill` envoie un signal à un processus ou à une tâche :
    - le signal est plus ou moins «fort », `-9 pour la destruction inconditionnelle du processus`, `-19 pour une suspension de processus`
    - le processus est indiqué par son numéro (donné par la commande `ps`)
    - la tâche est indiquée par son numéro de tâche (notation *%numéro*
  - `echo` envoie tout ce qui le suit sur la sortie standard (après éventuelles substitutions)

Autres commandes intrinsèques

- `exec` *`commande`* remplace le processus du shell en cours par celui de la commande
- *`./fichier`* exécute le script du fichier dans le shell en cours (ce qui permet de modifier les valeurs de variables)
- `break` termine l'itération en cours
- `continue` passe à l'élément suivant de l'itération en cours
- `shift` change la numérotation des paramètres positionnels (voir plus loin) en éliminant le premier
- `eval` *`commande`* évalue (expansions, substitutions) tous les mots de la commande, puis l'exécute (ce qui revient à l'analyser deux fois)
- `whence` *`noms`* indique l'interprétation par le shell de chaque nom

Lancement d'une commande

- le shell applique les règles suivantes pour déterminer quelle commande exécuter :
  - si le nom de la commande est un chemin absolu, il le soumet directement au noyau
  - sinon le shell cherche les commandes applicables dans l'ordre suivant :
    - fonction du shell
    - opération prédéfinie (commande intrinsèque)
    - fichier exécutable trouvé grâce à la variable `PATH`
- la notation sous forme de chemin absolu peut donc toujours servir en cas d'ambiguïté

Concaténation de commandes

- l'opérateur `;` dénote l'exécution séquentielle
- la fin de ligne d'une commande complète est équivalente à un `;`
- l'opérateur `|` dénote l'exécution en tube
- plusieurs opérateurs utilisent le *signal de sortie* de la première commande :
  - toute commande rend un nombre entier
  - s'il est nul il signifie « vrai » (succès)
  - sinon il signifie « faux » (échec), et sa valeur est significative et affichée
  - l'opérateur `||` exécute la deuxième commande si la première échoue
  - l'opérateur `&&` exécute la deuxième commande si la première réussit
- l'opérateur `&` lance la première commande en arrière-plan, puis passe à la suivante

Re-directions simples

- `< fichier` utilise le fichier comme entrée standard
- `<> fichier` utilise le fichier à la fois comme entrée et sortie standard
- `> fichier` utilise le fichier comme sortie standard (il ne doit pas exister)
- `>! fichier` remplace le fichier s'il existe
- `>> fichier` concatène la sortie standard au fichier (il doit exister)
- `>>! fichier` accepte un fichier inexistant

Autres re-directions

- `<< mot` prend l'entrée standard sur le terminal (ou l'entrée du shell non interactif) jusqu'à une ligne ne contenant que le mot (*entrée sur place*)
- `>&n` envoie la sortie standard sur le fichier standard de numéro n
- `n> ...` le fichier concerné est celui de numéro n
- `>&2 fichier` sorties standard et d'erreur sont envoyées sur le fichier

Commandes composées

- les *commandes composées* sont l'équivalent des énoncés structurés des langages de programmation
- la syntaxe est assez contraignante, puisque le premier mot de la commande détermine quelle est la commande composée
- dans ce qui suit :
  - une liste est une suite de commandes séparées par des points-virgules ou des fins de lignes
  - le signal de sortie d'une liste est celui que rend le dernier élément
  - les crochets dénotent des parties facultatives
  - les « mots-clés » tels que `then` ou `else` doivent apparaître en début de commande, donc après un point-virgule ou une fin de ligne

Constructions principales

- énoncé conditionnel

  `if` *`liste`* `then` *`liste`*
  `[ elif` *`liste`* `then` *`liste`* `] ...`
  `[ else` *`liste`* `]`
  `fi`

- énoncés itératifs
  - `while` *`liste`* `do` *`liste`* `done`
  - `until` *`liste`* `do` *`liste`* `done`
  - `repeat` *`mot`* `do` *`liste`* `done`
    ici le *mot* doit fournir la valeur numérique du nombre de répétitions (constante, variable, expression)

Constructions principales (suite)

- énoncé répétitif
  - `for` *nom* `[ in` *mot* `...]`
    `do` *liste*
    `done`
  - le *nom* prend comme valeurs successives chacun des *mots*
  - la *liste* est exécutée pour chaque valeur
  - si la liste de *mots* n'est pas donnée, on prend les paramètres de la commande
- énoncé sélectif
  - `select` *nom* `[ in` *mot* `... ]`
    `do` *liste*
    `done`
  - le shell affiche la liste de mots en les numérotant
  - il lit une ligne qui ne doit contenir qu'un nombre
  - le mot correspondant est affecté au *nom* et la *liste* est exécutée

Regroupement de commandes

- regroupement simple :
  - la notation `{ liste }` regroupe les commandes pour en faire une seule
  - c'est rarement utile à cause de la forme générale des commandes composées
- sous-shell :
  - la notation `( liste )` exécute la *liste* dans un shell fils
  - c'est un moyen de faire des changements temporaires : `tar cf - . | (cd /ailleurs/ici/la ; tar xf -)`
- expression conditionnelle :
  - la notation `[[ prédicat ]]` évalue le *prédicat* et fournit son résultat
  - `[[` et `]]` sont des commandes, donc précédées et suivies de blancs
  - on utilise cette notation après `if`, `elif`, `while` ou `until`

Prédicats

- la notation `[[ prédicat ]]` est équivalente à la commande `test prédicat` mais elle est plus lisible
- parmi les prédicats :
  - `-a fichier` vrai si le fichier existe
  - `-d fichier` vrai si le fichier existe et est un répertoire
  - `-f fichier` même chose pour un fichier ordinaire
  - `-n chaîne` vrai si la chaîne est non vide
  - `-z chaîne` vrai si la chaîne est vide
- il y a également
  - des comparaisons portant sur des chaînes ou des nombres
  - des opérateurs logiques
  - des expressions arithmétiques de forme très spéciale

Plan en cours

Paramètres

- le terme de *paramètre* est générique et décrit :
  - les *paramètres positionnels* :
    - fournis à l'appel du script ou du shell
    - similaires à des paramètres de procédures ou fonctions
    - simplement numérotés de 0 (nom de la commande) à n
  - les *variables* :
    - même rôle que dans les langages de programmation ordinaires
    - normalement non déclarées et non typées
    - la variable existe dès la première fois où on la mentionne
    - elle n'est utilisable par un script ou un sous-shell que si elle est exportée :
      `export variable`

Tableaux

- une variable peut être un *tableau* :
  - on lui affecte une valeur en énumérant les valeurs entre parenthèses :
    `table=(a b c d)`
  - les éléments sont numérotés à partir de 1
  - la référence se fait de la manière habituelle :
    `echo $table[2]`
  - l'indice spécial `*` donne la concaténation de toutes les valeurs du tableau, séparées par des blancs (un seul mot)
  - l'indice spécial `@` donne la liste de toutes les valeurs (autant de mots que de valeurs)

Paramètres spéciaux

- `@` est la liste des paramètres positionnels (autant de mots que de paramètres)
- `*` est la concaténation des paramètres positionnels (un seul mot)
- `#` est le nombre de paramètres positionnels
- `$` est le numéro du processus du shell en cours (par définition, différent de tout autre)

Expansion

- l'opération d'*expansion* remplace un paramètre par sa valeur dans le texte de la commande
  - `$paramètre` ou `${paramètre}` fournit la valeur
  - si le paramètre est un tableau, c'est équivalent à la notation `$paramètre[@]`
  - `${paramètre:-mot}` fournit la valeur du paramètre s'il existe et n'est pas vide, sinon le mot
  - `${paramètre:+mot}` comme le précédent mais l'inverse (si pas de valeur, résultat vide)
  - `${paramètre:?mot}` comme la précédente, mais termine le shell avec le mot comme message d'erreur si le paramètre est non défini ou vide

Expansion (suite)

- encore des expansions :
  - `${paramètre:=mot}` affecte le mot au paramètre s'il est inexistant ou vide, puis fournit sa valeur
  - `${#paramètre}` fournit la longueur de la valeur du paramètre (chaîne, tableau, liste)
  - `${paramètre#modèle}` fournit la valeur du paramètre, tronquée de son plus court préfixe conforme au modèle (avec jokers)
  - `${paramètre##modèle}` même chose avec le plus long préfixe
  - `${paramètre%modèle}` même chose avec le plus court suffixe
  - `${paramètre%%modèle}` même chose avec le plus long suffixe

Citation

- la *citation* sert à éviter l'expansion :
  - le *caractère d'échappement* \ enlève sa signification au caractère qui suit :
    - un blanc ne termine plus un mot
    - un passage à la ligne ne termine plus la commande
    - un opérateur n'est plus qu'un caractère ordinaire

Citation (suite)

- les apostrophes délimitent un texte cité tel quel :
  - tous les caractères perdent leur signification sauf l'apostrophe
  - le shell retire les apostrophes mais transmet l'ensemble comme un mot sans l'analyser
- les caractères " délimitent un texte cité presque tel quel :
  - les quatre caractères " $ ‘ et \ gardent leur signification
  - le shell peut donc faire des substitutions dans une chaîne entre "
  - le shell retire les " mais transmet l'ensemble comme un mot

Substitution

- la *substitution de commande*
  - exécute une liste de commandes
  - utilise leur *résultat* comme texte à inclure à la place de la notation
  - le *résultat* d'une commande est ce qu'elle écrit sur sa sortie standard
- deux notations possibles :
  - `` `liste` `` (accents graves)
  - `$(liste)`
- la deuxième notation est préférable (emboîtable), mais non reconnue par Sh
- les changements de ligne dans la sortie standard sont transformés en séparateurs de mots :

```
rm -i $(cat liste)
```