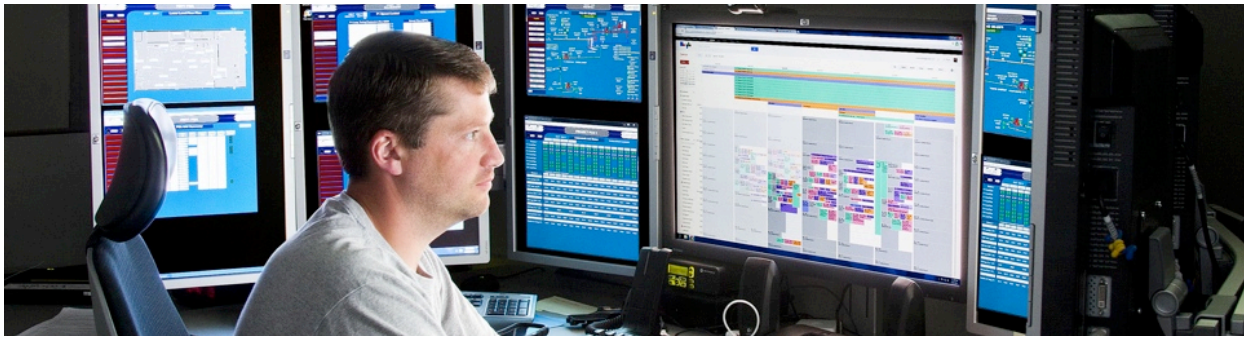
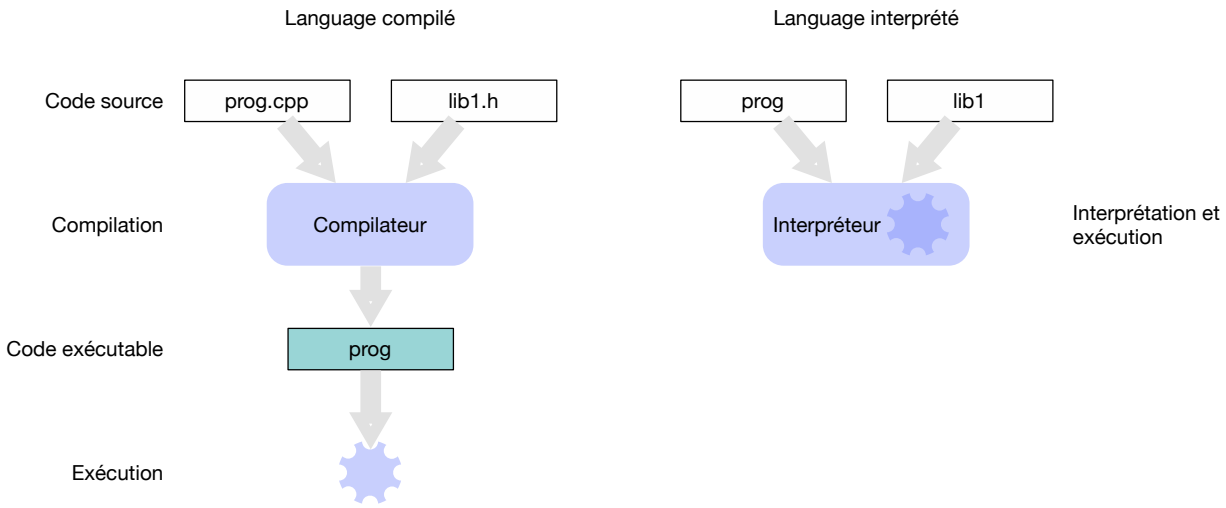
Administration Système — Scripts shell



Année académique 2014/15

Les scripts shell

Introduction — Langages compilés et interprétés



Les scripts shell

Introduction — Langages compilés et interprétés

- Un script est un fichier contenant une série d'ordres que l'on va soumettre à un programme externe pour qu'il les exécute.
- Ce programme est appelé *interpréteur de commandes*.
- On appelle les langages qui suivent ce modèle les langages *interprétés*, par opposition aux langages *compilés* (C, C++, Fortran, Ada, Java, Scala, Go, ...)
- Parmi les langages interprétés on trouve les shells Unix, Windows PowerShell, Perl, Python, Tcl/Tk, Ruby, JavaScript, ...
- Principales différences des langages interprétés :
  - Après l'écriture du script on peut le soumettre directement à l'interpréteur de commandes. Il n'y a pas d'étape de compilation.
  - Un programme compilé est directement compris par le processeur du système, alors qu'un script doit être interprété dynamiquement, ce qui généralement ralentit l'exécution.
  - Le fichier exécutable issu d'une compilation n'est utilisable que sur un seul type de processeur et un seul système d'exploitation. Un fichier script peut être lancé partout où l'interpréteur de commandes est disponible.
  - Un fichier compilé est incompréhensible par un lecteur humain. Un script est directement lisible et modifiable.

Les scripts shell

Pourquoi écrire un script shell ?

- L'écriture de scripts shell peut répondre à plusieurs besoins différents.
  - L'administrateur système les utilise comme aide-mémoire pour lancer quelques commandes disposant d'options complexes.
  - Le système d'exploitation utilise des scripts pour son initialisation. Ces scripts sont livrés avec le système d'exploitation. Parfois l'administrateur doit les modifier pour personnaliser un serveur.
  - Certaines applications (logiciels métier) nécessitent un paramétrage complexe et des scripts shell sont employés pour assurer la gestion des configurations, la préparation des répertoires et des fichiers temporaires, etc.
  - La supervision d'un parc informatique réclame des tâches automatisées pour vérifier l'état des systèmes et assurer des tâches de maintenance périodique.
  - Pour assurer l'exploitation de serveurs réseau ou de machines industrielles, l'administrateur doit exploiter la sortie de certaines commandes de supervision, et les traces enregistrées dans des fichiers de journalisation *(log file)*.

Les scripts shell

Un exemple simple

- On peut tourner des commandes tapées sur la ligne de commande dans un script en les mettant dans un fichier texte, un par ligne.
- Voici un exemple simple d'un script shell :

```
$ cat simple_example
date
echo "Users currently logged in"
who
$ ./simple_example
Wed Mar 26 16:09:58 UTC 2014
Users currently logged in
ubuntu   tty1         2014-03-03 20:21
marcel.graf pts/0        2014-03-26 16:09 (10.192.20.8)
$
```


Les scripts shell

Invocation de l'interpréteur

- Pour lancer un script shell on peut
  - Lancer l'interpréteur de commandes et lui donner le nom du script à exécuter

```
$ cat myscript
echo "This is my first test."
$ bash myscript
This is my first test.
$
```

  - Appeler directement le script
    - Il faut le rendre exécutable
    - Il faut dire au shell où il se trouve

```
$ chmod +x myscript
$ ./myscript
This is my first test.
$
```

Les scripts shell

La variable d'environnement PATH

- Chaque commande que nous pouvons taper sur la ligne de commande (par exemple **`ls`**) est un fichier exécutable (binaire ou script) qui pourra être lancé par le shell.
  - À l'exception des commandes intégrées au shell comme **`cd`**, **`for`**, **`while`**, ...
- Quand nous tapons une commande sans donner son chemin de fichier c'est au shell de trouver le fichier.
- Au lieu de chercher parmi tous les fichiers le shell cherche dans un nombre restreint de répertoires. Ceux-ci sont listés dans la variable d'environnement **`PATH`**.
  - Pour consulter la valeur courante de la variable on peut taper **`echo $PATH`**

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$
```


Les scripts shell

La variable d'environnement PATH

- Pour ajouter des commandes personnelles au système la convention est de
  - Créer un répertoire nommé **`bin`** dans le répertoire personnel
  - Mettre les fichiers des commandes dans ce répertoire
  - Ajouter le répertoire à la variable **`PATH`**

```
$ mkdir /home/marcel.graf/bin
$ mv myscript /home/marcel.graf/bin
$ export PATH=$PATH:/home/marcel.graf/bin
$ myscript
This is my first test.
$
```

  - Pour rendre ce changement permanent il faut mettre la commande **`export`** dans un fichier d'initialisation du shell, le fichier `~/.profile`. Sur Ubuntu ce fichier contient déjà les lignes suivantes :

```
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

Exercice 03.01

- Exercice a)
  - Pour prendre conscience de l'utilité de la variable PATH, affichez-la, puis effacez-la ainsi :

```
$ PATH=
```

  - Puis essayez quelques commandes :
    - **`cd /etc`**
    - **`ls`**
    - **`echo "Hello"`**
    - **`cd /bin`**
    - **`/bin/ls`**
    - ...
  - Que se passe-t-il ? Pourquoi ? Comment y remédier ?
- Exercice b)
  - Écrivez un petit script et exécutez-le (1) en appelant bash, (2) directement en donnant son chemin de fichier, (3) directement en le mettant dans votre répertoire `bin` personnel que vous avez ajouté à la variable PATH.

Les scripts shell

La ligne shebang spécifie le shell

- Quand le shell a trouvé le fichier d'une commande, il demande au système d'exploitation d'exécuter le fichier.
- Quand il s'agit d'un script, comment le système d'exploitation sait-il dans quel langage est écrit le script et quel interpréteur de commandes utiliser ?
- Sur notre système l'interpréteur par défaut est bash.
- Généralement on utilise la première ligne du script pour spécifier l'interpréteur à utiliser :
  - `#!path_to_interpreter`
  - Cette ligne s'appelle la ligne *shebang* (contraction de *shell* et *bang*)
  - Pas d'espace permis avant ou entre les caractères `#!`
- Exemple : Si on a écrit le script **`myscript2`** qui contient les lignes suivantes

```
#!/bin/tcsh
echo "This is a second script."
```

et on lance le script comme ceci

```
$ ./myscript2
```

le système d'exploitation lance **`tcsh`** et passe le script en paramètre comme si on avait tapé

```
$ /bin/tcsh ./myscript2
```

Les scripts shell

La ligne shebang spécifie le shell

- Le mécanisme de la ligne shebang est disponible pour les shells et autres langages interprétés :
  - Bourne shell : `#!/bin/sh`
  - Bash : `#!/bin/bash`
  - C shell : `#!/bin/csh`
  - K shell : `#!/bin/ksh`
  - Perl : `#!/usr/bin/perl`
  - Python : `#!/usr/bin/python`
  - Ruby : `#!/usr/local/bin/ruby`
  - ...
- Ainsi la ligne shebang peut être aussi considérée une forme de documentation : elle indique le langage dans lequel le script est écrit.

Les scripts shell

Commentaires et en-tête des scripts

- On peut ajouter des commentaires dans un script en les introduisant avec le caractère **#**
  - Le shell considère que tous les caractères jusqu'à la fin de la ligne font partie du commentaire
  - Exemple
    - ```
      # Display who is currently logged in
      date # include current date
      echo "Users currently logged in"
      who
      ```
- Généralement on donne aux scripts un en-tête contenant le nom du script, son rôle et son utilisation, le nom de l'auteur, un historique des changements, etc.
  - Exemple
    - ```
      #!/bin/bash
      #
      # who_is_on - Displays who is currently logged in
      #
      # usage: who_is_on
      #
      # Albert Einstein 2014-03-26

      date # include current date
      echo "Users currently logged in"
      who
      ```

Les scripts shell
Les variables

- Par défaut les variables utilisées dans les scripts shell ne sont pas typées
  - Le contenu est considéré une chaîne de caractères
  - Sauf si on indique qu'elle doit être traitée comme une variable entière (calculs arithmétiques)
  - Le shell ne permet pas de manipuler des données en virgule flottante.
- À la différence des langages compilés, une variable n'a pas à être déclarée explicitement.
  - Dès qu'on lui affecte une valeur, elle commence à exister
- Syntaxe :
  - `variable=valeur`
- Exemple :

```
$ a=1
$
```

- Faire attention à ne pas mettre des espaces autour du =

```
$ a = 1
bash: a: command not found
$
```


Les scripts shell
Les variables

- Pour accéder au contenu d'une variable il suffit de préfixer son nom avec le caractère **$**

```
$ echo $a
1
$
```

- Le nom d'une variable peut être composé de
  - lettres
  - chiffres
  - le caractère souligné _
  - Le nom ne doit pas commencer avec un chiffre.
- Le shell distingue les majuscules et les minuscules, a et A sont deux variables différentes

Les scripts shell

Les variables

- Si la valeur comprend des espaces, il faut l'envelopper dans des guillemets " " ou des apostrophes ' '

```
$ variable="abc def"
$ echo $variable
abc def
$
```

- Une variable qui n'a jamais été affectée est considérée comme une chaîne vide

```
$ echo $inexistante

$
```

- On peut configurer le shell à déclencher une erreur si on essaie de lire une variable inexistante

```
$ set -u
$ echo $inexistante
bash: inexistante: unbound variable
$
```


Les scripts shell

Les variables — Variables arithmétiques

- Pour faire des calculs avec des valeurs entières on peut utiliser des variables arithmétiques
  - Elles peuvent contenir des valeurs entières
  - Lors de l'affectation d'une valeur le shell fera une évaluation arithmétique de la valeur
- On déclare une variable arithmétique avec
  - `declare -i variable` (forme propre à Bash)
  - ou
  - `typeset -i variable` (forme portable)
- Exemple montrant la différence entre une variable arithmétique et une variable normale :

```
$ declare -i A
$ A=1+1
$ echo $A
2
$ B=1+1
$ echo $B
1+1
$
```

Les scripts shell

Les variables — Variables arithmétiques

- Si on veut insérer des espaces dans la formule, il faut l'envelopper avec des guillemets " "

```
$ A=4*7 + 67
sh: +: command not found
$ A="4*7 + 67"
$ echo $A
95
$
```

- Les opérateurs disponibles sont les mêmes que dans la plupart des langages de programmation
  - Opérations arithmétiques : `+, -, *, /`
  - Modulo : `%`
  - Opérateurs de manipulation binaire : `&, |, ^, ~, <<, >>`

Exercice 03.02

- Exercice a)
  - Affectez la chaîne de caractères `1+2` à une variable normale avec le nom `B`.
  - Affectez le contenu de la variable B à une variable arithmétique avec le nom `A`.
  - Qu'observez-vous dans la valeur de la variable arithmétique `A` ?
- Exercice b)
  - Répétez l'exercice 1 mais avec l'affectation `A=$B*2`.
  - Puis répétez encore une fois mais omettez le **$** dans l'affectation : `A=B*2`. Qu'observez-vous ?
- Exercice c)
  - Affectez une chaîne de caractères non-arithmétique comme `ABC` à une variable arithmétique.
  - Quelle valeur contient la variable ?
- Exercice d)
  - En utilisant la commande `declare -i` seule affichez la liste des variables arithmétiques.

Les scripts shell

Les variables — Variables arithmétiques

- Les constantes numériques sont considérées par défaut en base 10
  - Si elles commencent avec un `0`, elles sont considérées comme étant octales
  - Si elles commencent avec un `0x`, elles sont considérées comme étant hexadécimales
- On peut préciser explicitement la base employée (jusqu'à 36) en utilisant la forme
  - `base#nombre`
- Exemple

```
$ declare -i masque=2#000110
$ declare -i capteur=2#001010
$ declare -i resultat="masque & capteur"
$ echo $resultat
2
$
```


Les scripts shell

Substitution de commande

- La *substitution de commande* permet d'exécuter une commande, capturer sa sortie comme chaîne de caractères et l'affecter à une variable.
- Elle a la forme
  - `variable=$(commande)`
- Exemple :

```
$ variable=$(date)
$ echo $variable
Wed Mar 26 20:24:04 UTC 2014
$
```

- Dans la commande on peut utiliser des variables et des pipelines. Exemple qui compte le nombre de lignes dans un fichier donné :

```
$ filename=report
$ typeset -i nb_of_lines
$ nb_of_lines=$(cat $filename | wc -l)
$ echo $nb_of_lines
5
$
```

Les scripts shell

Substitution de commande

- Souvent la sortie d'une commande est formatée et contient une série d' espaces. Aussi elle peut s'étendre sur plusieurs lignes et contenir des sauts de ligne.
  - Le shell conserve tous ces caractères dans la chaîne de caractères, sauf du whitespace à la fin de la chaîne qui sera éliminé.
  - Lorsqu'on désire examiner le contenu d'une telle variable on rencontre que le shell enlève des successions d'espaces multiples, les tabulations et les retours à la ligne, qu'il considère des séparateurs d'arguments, par des espaces uniques.

```
$ ls -l
total 16
-rw-rw-r-- 1 marcel.graf marcel.graf   10 Mar 13 07:32 chap03
drwxrwxr-x 2 marcel.graf marcel.graf 4096 Mar 13 07:31 play
drwxrwxr-x 2 marcel.graf marcel.graf 4096 Mar 13 07:31 work
$ variable=$(ls -l)
$ echo $variable
total 16 -rw-rw-r-- 1 marcel.graf marcel.graf 10 Mar 13 07:32 chap03 drwxrwxr-x
2 marcel.graf marcel.graf 4096 Mar 13 07:31 play drwxrwxr-x 2 marcel.graf
marcel.graf 4096 Mar 13 07:31 work
$
```


Les scripts shell

Substitution de commande

- Pour afficher correctement la variable, il faut protéger le contenu de l'interprétation du shell, ce qui s'obtient en l'encadrant par des guillemets `" "`.

```
$ echo "$variable"
total 16
-rw-rw-r-- 1 marcel.graf marcel.graf   10 Mar 13 07:32 chap03
drwxrwxr-x 2 marcel.graf marcel.graf 4096 Mar 13 07:31 play
drwxrwxr-x 2 marcel.graf marcel.graf 4096 Mar 13 07:31 work
$
```

Exercice 03.03

- Exercice a)
  - Dans un répertoire avec un grand nombre de fichiers (par exemple **`/usr/share`**) exécutez
    - la commande **`ls`** seule
    - la commande **`ls | cat`** (la sortie de **`ls`** pipé dans **`cat`**. **`cat`** sans arguments recopie stdin sur stdout)
    - Qu'observez-vous ?
    - Quelle forme est utilisée quand vous affectez la sortie de **`ls`** à une variable ?

- Exercice b)
  - Écrivez un script qui calcule pour un fichier texte donné le nombre moyen de caractères par ligne

Les scripts shell

Utilisation de variables — Concaténation

- Souvent on veut construire un paramètre d'une commande ou une variable comme concaténation de plusieurs chaînes de caractères.
- Le shell n'a pas d'opérateur pour la concaténation, on colle simplement les chaînes les unes aux autres, sans espaces.

```
$ prefix=proj1
$ file=motor.c
$ new_file=$prefix$file
$ echo $new_file
proj1motor.c
$
```

- Quand on mélange chaînes littérales et variables, il devient parfois nécessaire de délimiter précisément le nom de la variable. Si dans l'exemple précédent on veut ajouter un caractère souligné _ pour obtenir un nom de la forme `proj1_motor.c` on doit protéger le nom de la variable avec des accolades **`{ }`** :

```
$ prefix=proj1
$ file=motor.c
$ new_file=${prefix}_$file
```

- **Conseil :** En cas de doute, utiliser la forme `${variable}` au lieu de `$variable`.