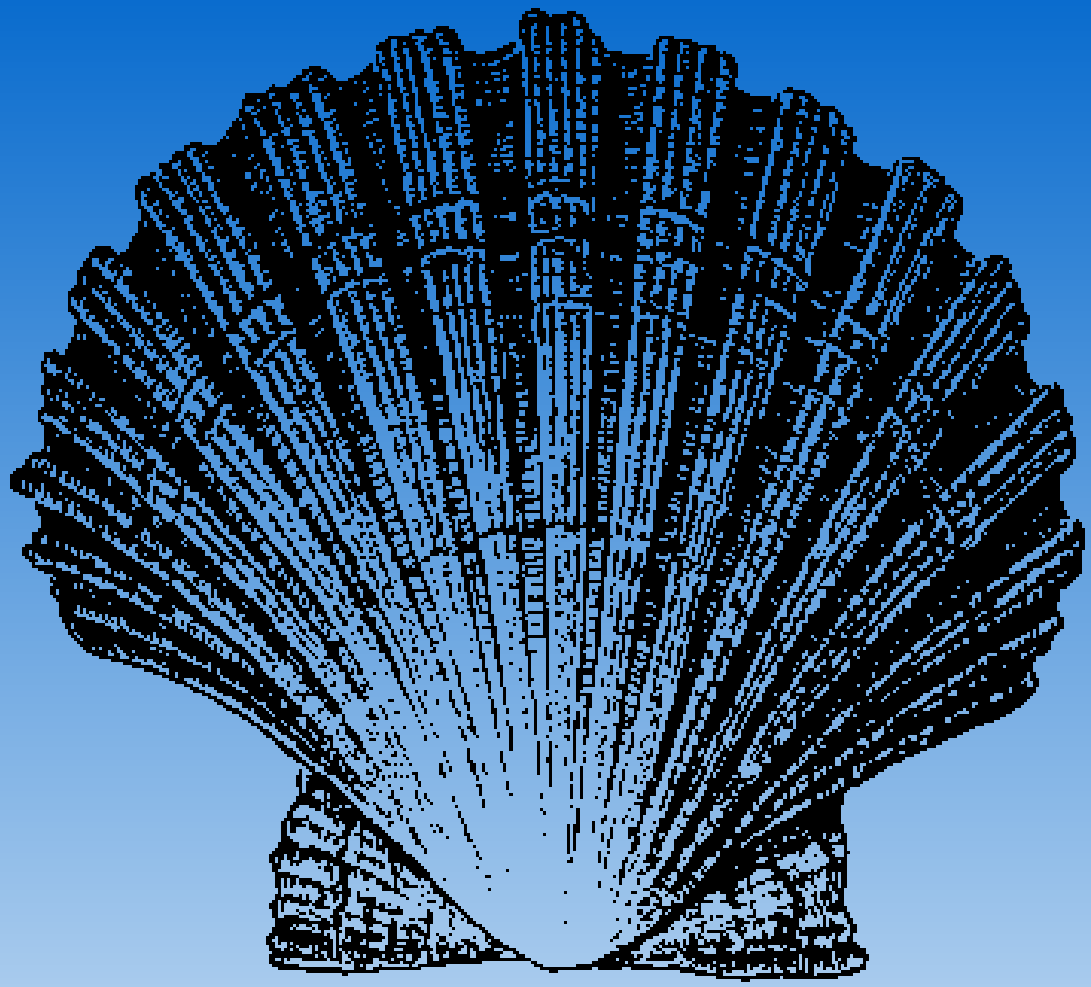

Mieux programmer en shell

Journées Perl 2008

Olivier Mengué
Paris.pm
dolmen@cpan.org
http://o.mengue.free.fr/

Voici ce que vous auriez pu voir en live si...

- Ubuntu, X11 et ATI géraient correctement et simplement l'affichage sur un écran externe avec une résolution différente
- Windows Vista n'était pas si leeeennnnnnt sur mon PC (presque 10 minutes pour arriver à OOo)
- Windows ne décidait pas tout seul de rebooter pour installer des mises à jour au bout de 5 minutes de présentation
- Vive la craie et le tableau noir !

Avertissement

- Tous les conseils présentés après s'appliquent à :
  - ksh88, pdksh : sûr
  - bash, ksh93 : probablement
  - N'importe quel autre shell POSIX (ex : dash) : en partie
- Lisez le manuel de votre shell !

Olivier Mengué

- Batch DOS/Windows depuis 20 ans
- Shell Unix depuis 15 ans
- Perl depuis 11 ans
- WSH / Javascript depuis 10 ans
- 4 ans de dév shell professionnel
  - chaînes d'exploitation
  - installation d'applications
  - scripts portables sur AIX et Linux

Pourquoi développer en shell...

Pourquoi développer en shell...

... alors que Perl existe ?

Pourquoi développer en shell...

- L'intégration au code existant
- Les compétences des pairs
- Les contraintes techniques
- La lisibilité du code, pour certaines tâches (ex : enchaînement de programmes)
- Souvent suffisant pour la tâche

Pourquoi ne pas développer en shell...

- Passez à Perl dès que la tâche est ardue :
  - Sécurité
    - Le shell est sensible à certains caractères ; chaque commande Unix a les siens également, différents.
    - Sous Unix, seul '/' et '\0' ne peuvent pas être utilisé dans un nom de fichier.
    - /usr/bin/xdg-email
  - Nécessité d'effectuer des opérations de bas niveau non disponibles dans les commandes Unix
  - Ex: vérifier le propriétaire et le groupe d'un fichier

```
perl -e '($u,$g,$f)=@ARGV;exit!((@s=stat $f)
[4]==getpwnam($u)&&$s[5]==getgrnam($g))' \
user group file
```

spéciale dédicace à L.Dami

  - Opportunité de grouper des opérations sur un fichier effectuée autrement par plusieurs commandes → perf

Commandes internes / externes

- Interne :
  - une commande directement interprétée par le shell
  - peut modifier l'état du shell
    - ex : `read`, `exit`, `umask`, `export`
- Externe :
  - fait appel à un programme externe (fichier binaire dans $PATH)
  - crée un nouveau processus (fork+exec)
- Reconnaître une commande interne : `type`

Commandes internes / externes

```
$ time /bin/echo Hello
Hello

real   0m0.003s
user   0m0.000s
sys 0m0.004s
$ time echo Hello
Hello

real   0m0.000s
user   0m0.000s
sys 0m0.000s
```

Privilégiez les commandes internes

- Groupez les appels à une commande externe
- Bannissez les « `cat | grep | grep -v| grep | awk` »
  - Utilisez la redirection de l'entrée plutôt que cat
    - `toto=$(cat toto.txt | head -n 1)` !!
    - `toto=$(head -n 1 < toto.txt)` ☺
    - `toto=$(head -n 1 toto.txt)` ☺
  - Utilisez un unique programme de filtre plus évolué (sed, perl), et donnez lui un script évolué

Entrée par la ligne de commandes

- Privilégiez les commandes qui prennent leurs données d'entrée sur la ligne de commande plutôt que sur un flux
- Comparez :
  - ligne="abc def"
  - toto=$(echo "$ligne" | cut -d' ' -f1)
  - toto=$(expr "x$ligne" : 'x\([^]*\)')
- Moins d'I/O pour expr

Entrée par la ligne de commandes

- Mais évitez pour les listes de fichiers (sauf si vous aimez les ennuis)
- Solution : `xargs`
- Ex :
  ```
  find /tmp -user dolmen | xargs chown toto
  ```
- La limite du nombre d'arguments du shell
  - Ex : "rm *" ne fonctionne pas si vous avez plus de 512 fichiers avec ksh88

Practical Extract...

- Pour extraire du texte (d'un fichier, d'un résultat de commande), utiliser le bon outil
- Pour un texte d'une seule ligne
  - Parameter expansion s'il s'agit juste de supprimer le début et/ou la fin : `${ligne#*,} ${ligne%,*}`
  - `expr` avec une capture : `expr "x$ligne" : 'x\([^,]*\)'`
- Pour plusieurs lignes
  - `cut`
  - `sed`
  - `perl`

Et les autres ?

- `grep`
- `tail`
- `head`

Et les autres ?

- `grep`
- `tail`
- `head`
- Je ne m'en sert pas !

Practical Extract...

- En général :
  - On isole les lignes intéressantes
  - Sur ces lignes on isole le texte intéressant

Comment se passer des pipes...

Il est plus efficace de faire les deux opérations avec un seul programme plutôt que deux (ou plus) avec des pipes

Comment se passer des pipes...

```
grep toto
```

Comment se passer des pipes...

```
grep toto

sed -n '/toto/p'
```

Comment se passer des pipes...

```
grep toto | cut -f2 -d,
```

```
sed -n '/toto/ s/^[^,]*\([^,]*\).*$/\1/
p'
```

Comment se passer des pipes...

```
head -n 2
```

```
sed '2q'
```

Comment se passer des pipes...

```
tail -n1

sed -n '$p'
```

Comment se passer des pipes...

```
tail -n2
```

```
sed -n '${x;p;x;p}
        h'
```

Rappel sur les types de données

- Les chaînes d'octets
  - Le seul type de variable du Bourne Shell
- Le code de retour d'une commande
  - Utilisable immédiatement
  - Transformable en texte avec $?
  - Représente un booléen (0 || !=0) pour les conditions des structures de contrôle (if/while/until/&&/||)

Maîtrisez la portée des variables

- Dans les fonctions, utilisez typeset pour définir une variable "locale"
  - ```
    function fatal
    {
        typeset code="$1" msg="$2"
        echo "$msg" >&2 ; exit "$code"
    }
    ```
- export uniquement pour les variables utilisées par les sous processus
  - Commandes externes
  - Tâches en background
- `unset` : pour nettoyer
    ```

Les variables peuvent être typées

- Les shells "modernes" permettent de typer une variable comme étant numérique
  - pour éviter les conversions texte / entier
  - pour améliorer la lisibilité du code
- `typeset -i i=0`
  `i=i+1`
- Dans une affectation de variable numérique les mots sont pris comme des noms de variables, mais sans conversion en texte
  - `i=$i+1` est moins efficace

Les opérations mathématiques

- Le shell sait faire des maths, même sur les variables non typées !
- Oubliez `expr` !
- Let : commande interne
-
```
let x=3+2
let x=i+5
let x+=1
```
  - Attention aux méta-caractères !
```
$ touch 'x@@@@=2'
$ let x*=2
-bash: let: x@@@@=2: syntax error: invalid arithmetic
operator (error token is "@@@@=2")
$ let x\*=2
$ let 'x\*=2'
```

Les opérations mathématiques

- Utilisez plutôt la syntaxe :
  - ```
    (( x = 6 * 7 ))
    echo $x
    ```
- Oubliez `let` !
- Analysé directement par le shell sans évaluation de méta-caractères.
    ```

Les conditions

- Les condition des structures de contrôle (if/while/until) sont une commande
- `[` n'est qu'une commande parmi d'autres :

```
$ ls -l /usr/bin/[
-rwxr-xr-x 1 root root 40192 2008-04-04 08:44 /usr/bin/[
```

- Utilisez les résultats de commande directement
  Comparez :

```
1 grep "$user" /etc/passwd > /dev/null
  if [ $? = 0 ] ; then ...
2 if grep -q "$user" /etc/passwd ; then ...
3 if id -u "$user" > /dev/null ; then ...
```

Booléens

- `true` et `false` sont des commandes internes
  Comparez :
  - `ok=0`
    `if [ $ok = 0 ] ; then`
  - `ok=true`
    `if $ok ; then ...`
- `!` est l'opérateur interne de négation
  - `if ! $ok ; then ...`
  - `if ! grep -q $user_host ~/.ssh/authorized_keys`

Test interne

- Utilisez `(( ... ))` pour les comparaisons d'entiers :
  `if (( $# == 0 )) ; then ...`
- `[[` est la version interne plus élaborée de `[`
  - Mais analyse syntaxique avant l'évaluation des méta-caractères
  - Les méta-caractères à droite de = et != sont utilisés pour faire du matching :
    `[[ "x$f" = x*.pl ]] && echo "C'est du Perl !"`
    - Gardez les méta-caractères en-dehors des guillemets, sinon
    - Entourez les variables utilisées en partie droite avec des guillemets pour échapper les méta-caractères
  - &&, || intégrés. Comparez la lisibilité :
    - `[ -f toto.txt ] && [ -w toto.txt ]`
    - `[ -f toto.txt -a -w toto.txt ]`
    - `[[ -f toto.txt && -w toto.txt ]]`

Quelques pièges du shell

- Les chaînes vides et les espaces
  - `arg=`
    `if [ $arg = -h ] ; then`
  - `If [[ "x$arg" = -h ]] ; then`
- Les caractères spéciaux
  - Métacaractères : voir ex "let" plus tôt
  - Fichier dont le nom commence par '-'
    - Utilisez "--"
      `rm -f -- "$fichier"`
- La locale : LANG, LC_ALL...

echo

- Ne fait pas *que* ce que vous pensez
- ```
  \0NNN   the character whose ASCII code is NNN
  (octal)
  \\      backslash
  \a      alert (BEL)
  \b      backspace
  \c      suppress trailing newline
  \f      form feed
  \n      new line
  \r      carriage return
  \t      horizontal tab
  \v      vertical tab
  ```
- Utilisez print -r --
  ```

Portabilité

- Ksh sous Linux n'est pas ksh88 ou ksh93 mais pdksh : une réimplémentation
- (Au moins) une différence :

```
a=0
echo 1 | read a
echo $a
```

  - Ksh88/ksh93 : 1
  - Pdksh : 0

Ksh88 : coprocess

- http://publib16.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixuser/usrosdev/input_output_redir_korn.htm#a58c221de
- Un exemple concret d'utilisation : log dans un fichier en plus de la sortie standard

```
# Copie 1 vers 3
exec 3>&1
# Lance tee comme un coprocessus, en redirigeant la sortie
# vers 3 au lieu de l'entrée de ce shell
tee "$logfile" >&3 |&
# Finalement redirige stderr vers l'entrée de 'tee' input
exec >&p 2>&1
```

Références

- Ksh : guide de style OpenSolaris. La plupart de recommandations s'appliquent aussi à bash.
  http://www.opensolaris.org/os/project/shell/shellstyle/
- Mes bookmarks :
  http://del.icio.us/dolmen/shell
  http://del.icio.us/dolmen/ksh

Questions ?

Olivier Mengué

dolmen@cpan.org
http://o.mengue.free.fr/