Programmation orientée données et systèmes Unix
coreutils, `sed`, `awk`, etc. [1]

Antoine Amarilli

7 novembre 2013

1. Merci à Pablo Rauzy pour sa relecture.

Idée générale

- Le shell est l'interface de choix pour utiliser les systèmes Unix.
- On veut s'en servir pour des tâches complexes
  - ⇒ Notamment : manipuler du texte.
- Les fonctionnalités built-in du shell sont insuffisantes.
- On utilise le shell pour raccorder des programmes entre eux.

⇒ Programmation orientée données.

Normes vs implémentations

- Le fonctionnement des systèmes Unix est standardisé :
  - **POSIX** Appels système, librairie standard C, signaux, shell, programmes...
  - **SUS** Essentiellement pareil que POSIX.
  - **FHS** `/bin`, `/usr/lib`, etc.
- Il y a différentes implémentations de ces normes (e.g., GNU).
- Beaucoup ajoutent des fonctionnalités non-POSIX.
- Certaines en dévient (`POSIXLY_CORRECT`).

$\Rightarrow$ Ici, on utilisera les versions GNU (répandues).

Jetabilité

- Faire des programmes propres, c'est fatigant.
- Faire des programmes lisibles, c'est fatigant.
- Quand on utilise le programme une seule fois, c'est inutile.
  - ⇒ Attention, pas de reproducibilité !
  - ⇒ Attention, debug pénible !
  - ⇒ Attention aux hacks qui prennent racine !

⇒ Jetable : un programme qu'on n'utilisera qu'une fois.

⇒ Write-only : un programme qui a été écrit d'une traite et qui n'est pas censé être relu ou modifié.

Write-only en images

```
cat reminders | grep '^1' | grep now | sed
's/days*/*3600*24/;s/hours*/ *3600/;s/weeks*
/*3600*24*7/;s/minutes*/*60/;s/months*/30*
24*3600/;s/now //;s/ /_/;s/ //g;s/_/ /' |
while read a; do A //=$(echo "$a" | cut -f1
-d' '); B=$(echo "$a" | cut -f2 -d' '); echo
-n "$A "; echo "$B" | bc; done | sort -k2,2n;
while read a; do echo "$a" | cut -d' ' -f1 |
tr '\n' ' '; D=$(echo "$a" |cut -d' ' -f2);
echo " AA$D"; done
```

À votre avis, que fait ce code ?

Write-only en images

```
cat reminders | grep '^1' | grep now | sed
's/days*/*3600*24/;s/hours*/ *3600/;s/weeks*
/*3600*24*7/;s/minutes*/*60/;s/months*/30*
24*3600/;s/now //;s/ /_/;s/ //g;s/_/ /' |
while read a; do A //=$(echo "$a" | cut -f1
-d' '); B=$(echo "$a" | cut -f2 -d' '); echo
-n "$A "; echo "$B" | bc; done | sort -k2,2n;
while read a; do echo "$a" | cut -d' ' -f1 |
tr '\n' ' '; D=$(echo "$a" |cut -d' ' -f2);
echo " AA$D"; done
```

À votre avis, que fait ce code ?

... ouais, moi non plus je ne veux pas savoir.

Robustesse

- La solution évidente échoue souvent dans certains cas :
  - ⇒ `head *` échouera si un nom de fichier commence par '-'.
- Souvent on peut supposer que l'entrée n'est pas malicieuse (mais pas toujours) :
  - ⇒ Si un fichier contient une liste de couples de mots pour une recherche, doit-on échapper ou non ?
- Souvent c'est pénible de gérer les erreurs.
- Pour des programmes jetables, il suffit souvent de supposer qu'on est dans le cas typique.
- Un échec sale est OK quand l'utilisateur est capable de mettre les mains dans le cambouis. (Typiquement, quand vous êtes l'utilisateur.)

Performance

- Instancier plein de process est plus long qu'instancier un unique process qui fait tout.
- Les langages interprétés (shell, sed, awk...) sont généralement plus lents que des langages compilés.

⇒ Mauvais en vitesse d'exécution.

⇒ Souvent, importe moins que la vitesse d'écriture.

Table des matières

Bases

```
commande -abc --option arg1 arg2
```

- Ce n'est pas le shell qui gère (même si tab-completion).
- Où trouver `commande` ? Variable d'environnement `PATH`.
- Les librairies dynamiques, suivant `LD_LIBRARY_PATH`.
- `--` pour indiquer la fin explicite des options.
- Souvent `-` pour l'entrée/sortie standard :
  - ⇒ Sinon, parfois `/dev/stdin` marche.
  - ⇒ Sinon, faire un wrapper.

Built-ins

- Certaines “commandes” (`echo`, ...) sont des built-ins
- Ils sont gérés par le shell et ne sont pas dans `PATH`...
- ... ou parfois si. `/bin/echo`
- Ne pas regarder `man echo` mais, e.g., `man zshbuiltins`.
- Pour savoir : utiliser `which` sous zsh (mais pas bash ! ;-P)

Structures de contrôle

- `if foo; then ...; else ...; fi`
- `if [ -f FICHIER ]; then ...; else ...; fi`
  - ⇒ `[` est en fait un alias pour `test` (gruiik).
  - ⇒ Autres tests disponibles.
  - ⇒ Tester la valeur de retour d'une commande.
- `for a in *; do ...; done`
- `for a in \`ls | grep '^a'\`; do ...; done`
  - ⇒ Attention aux espaces !
- `while read l; do ...; done`
  - ⇒ Attention aux retours chariot !

Expansion de variables

```
for a in *.bmp; do convert "$a" "${a%.bmp}.png"; done
```

Beaucoup sont disponibles :

- `${var#prefix}` pour les préfixes
- `${var/foo/bar}` rechercher-remplacer
- `${var//foo/bar}` rechercher-remplacer toutes occurrences
- `${#var}` longueur
- ...

Redirections et pipes

Écrire	`commande >fichier` (écrase !)
Lire	`commande <fichier`
Concaténer	`commande >>fichier`
Erreurs	`commande >fichier 2>&1`
Pipe	`commande1 | commande2`
Subshell	`diff <(commande1) <(commande2)`

Champs

- Attention aux champs :
  - `A="a b" ls $A`...
  - ⇒ Comparer à `ls "$A"`
- Sous-expressions :
  - `` `ls` `` (historique)
  - `$(ls)` (imbriquable)
  - Ne pas confondre avec `<(ls)` !
- Chaînes avec `"foo"` ou `'foo'`
  - Concaténation : `'foo''bar'`
  - Variables et sous-expressions : `"$foo"` et `'$foo'`
  - Échappement : `"\""` et `'\''`
  - Astuce sale : `'foo'"'"'bar'`

Trucs et astuces

- Utiliser la tab-completion.
- Utiliser un bon shell pour ça (ex : `zsh`).
- Configurer son shell.
- Utiliser les raccourcis d'édition de ligne (readline ou zle).
- Configurer son historique.
- La recherche incrémentale est votre meilleure amie.
- Conception pas à pas d'un pipeline.

Et aussi...

Signaux	lever, et attraper
Globbing	`**/a`, `*/*/b`
Job control	`fg`, `bg`, `jobs`...
Maths	`$((1 + 2))`
Aliases	parfois pratiques
Scripts	ou sous-fonctions
...	

Table des matières

Lire des fichiers : `cat`, `less`

- Afficher le contenu d'un fichier sur la sortie standard : `cat`
- Concaténer des fichiers.
- Attention aux UUOCs :
  - ⇒ `cat fichier | grep foo`
  - ⇒ `grep foo fichier` prend un processus de moins
  - ⇒ Ceci dit, est-ce vraiment si grave...
- `cat -n` numérote les lignes (aussi, non-blank ; voir aussi `nl`).
- `cat a - b` (entrée standard).
- `less` pour lire interactivement :
  - ⇒ Utile en fin de pipeline.

Bouts de fichier : `head`, `tail`

`head`	afficher le début des fichiers
`tail`	idem pour la fin
`tail -f`	afficher puis suivre (logs...)
`head *`	affiche le nom des fichiers (-q)
`head -42`	alias `head -n 42` : 42 premières lignes
`head -n +42`	tout sauf les 42 dernières
`head -c 1M`	premier méga

Découper : `split`

- Découper un fichier en morceaux (nommage contrôlable)

  `split -l 42` 42 lignes par fichier

  `split -b 42M` 42 mégas par fichier

  `split -n 10` séparer en 10 morceaux

⇒ Plein d'options : round-robin, etc.

Trier : `sort`

`sort file`	trier un fichier
`-k 1,3rn`	spécifier une clé
Attention	`-s` pour rendre stable (sinon, dernier recours)
Attention	`-k 1` et `-k 1,1`
`-t`	séparateur de champ
Big data	Plein d'options !

- `-S100M` pour utiliser plus de RAM.
- `-T` pour le répertoire temporaire.
- `--compress-program`.

Attention à la locale ! `LC_ALL=C`

Opérer sur les lignes : `uniq`, `shuf`

`uniq` supprimer les doublons

⇒ Trier avant !

⇒ Pour conserver l'ordre : `awk '!a[$0]++'`

`uniq -c` compte les répétitions

`uniq -d` n'affiche que les lignes répétées

`uniq -i` pour ignorer la casse

Champs utiliser `sort -u` dans ce cas

`shuf` mélanger un fichier

Combiner des fichiers : `join`, `paste`

`join foo bar` joindre `foo` et `bar` au sens relationnel

- Trier avant sur le champ de join !
- Champ de join puis autres champs foo puis autres champs bar
- Séparateur configurable
- Champ de join configurable dans chaque fichier

`paste foo bar` juxtaposer les lignes de `foo` et `bar`

- `paste - - - - -`

Comparer : comm, diff

`comm a b` afficher les lignes de $a \backslash b$, $b \backslash a$, $a \cap b$

- Trier avant !
- -1, -2, -3 pour retirer chaque output

`diff` afficher les différences

- `-r` recursive
- `-y` en colonnes (pratique)
- `-i` casse
- `-a` même sur le binaire
- `-q` seulement tester si fichiers différents (plus rapide)

`wdiff` faire un diff par mots

`colordiff` Couleur

- `colordiff a b`
- `wdiff a b | colordiff`

Lister : `ls`, `stat`, `du`

`ls` lister (duh)

- `-A` inclut les fichiers cachés mais pas `.` et `..`
- `-h` à utiliser, voire aliaser
- l'output de `ls` change si ce n'est pas un tty
- options de tri : ne pas trier `-l` à la main

`stat` voir les infos détaillées

- Format configurable

`du` sommer l'usage disque

Rechercher : `find`, `locate`

`find` rechercher des fichiers sur des critères et agir

- `-name '*blah*'`
- `-iname '*blah*'`
- `-print0`
- `-exec`, `-ok`
- ...

`locate` plus rapide (cache) mais plus limité

Exécuter sur des arguments : `xargs`, `parallel`

`find | xargs ls -l` exécuter sur chaque résultat

- Attention, `ls -l` lancé même si aucun résultat
- Par défaut `ls -l fichier1 fichier2 ...`
- `-n` pour le nombre d'arguments (e.g., `-n 1`)
- `-0` avec `-print0` pour séparer par NULL
- `-P` pour l'exécution concurrente

`parallel` `xargs` en plus parallèle

- Peut distribuer sur des machines distantes !
- `man parallel` et regarder `EXAMPLES`

grep

- `grep motif fichier`
- `grep motif fichier1 fichier2 ...`
- Options :
  - `-R` récursif
  - `-i` ignorer la casse
  - `-v` inVerser
  - `-C 10` 10 lignes de contexte (fusion si chevauchement)
  - `-A` et `-B` pour After et Before
  - `--line-buffered` désactiver le buffering
- Alternative : `ack`

grep (langage)

- Par défaut les caractères sont interprétés littéralement
- Utiliser `-E` pour changer ça (regexps non-triviales)
- `a?`, `a+`, `a*`, `[^a-z]`, `(foo|bar)`, `{2,}`, `^`, `$`, ...
- Regexps Perl avec `-P` (expérimental)
- Back-references : `<h([1-6])>[^<]*</h\1>`

Infos sur le fichier : `wc`

`wc` compter les lignes, mots, octets

- `-l` juste les lignes
- `-w` juste les mots
- `-c` juste les octets

`-m` pour le nombre de caractères

`-L` ligne la plus longue

Sur les champs : cut

- `cut -d: -f1` pour prendre le premier champ
- `-f1,4-5,8-` pour spécifier quels champs
- À partir de la fin ? `rev | cut -f1 | rev`
- Options :
  - `-b` pour les octets
  - `-c` pour les caractères
  - `-s` pour masquer les lignes sans délimiteurs
- Pas de fusion des délimiteurs adjacents.

Sur les caractères : `tr`

- `tr a b` pour remplacer `a` par `b`
- `tr A-Z a-z` pour une séquence de caractères
- `tr -d '\n'` pour supprimer
- `tr -dc '0-9\n'`pour supprimer le complémentaire
- `tr -s ' '` pour fusionner les répétitions
- Attention, octets et non caractères (Unicode) :
  $\Rightarrow$ Ne pas faire : `tr à a`
- Attention, impossible de spécifier un fichier.

Séquences : `seq`, `yes`

- Trois syntaxes :
  - `seq LAST` (commence à 1)
  - `seq FIRST LAST` (`LAST` inclus)
  - `seq FIRST INCREMENT LAST`
- Padding avec des zéros possible.
- Attention : floating point en interne !
- `yes foo | head -10`

Plomberie : `tee`, `pv`

`tee fichier` copier `stdin` vers `stdout` et `fichier`

- `tee -a` pour concaténer

⇒ `foo | bar | tee intermediaire | baz`

`pv` Pipe Viewer

`pv fichier` `cat` mais avec barre de progression

`pv -L` limiter la vitesse

`pv -l` fonctionner par lignes

`sponge` (`moreutils`, incompatible avec `parallel`...)

- Ne pas faire : `cat fichier | ... >fichier`

⇒ `cat fichier | ... | sponge fichier`

Et aussi

`tac`	renverser l'ordre des lignes
`tsort`	trier topologiquement
`od`	dumper en hexa
`dd`	copier des octets (`SIGUSR1` !)
`base64`	et autres
`sha1sum`	et autres
`column`	afficher en colonnes
`csplit`	découper en morceaux suivant un motif
`fmt`	reformater du texte (aussi `pr`, `fold`)
`factor`	factoriser des entiers
`strings`	extraire des chaînes
fichiers	`cp`, `mv`, `rm`, `touch`, `chmod`...

⇒ Pour les coreutils, aide détaillée dans `info` !

Table des matières

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
```

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
  cut -d ' ' -f1 |
```

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
  cut -d ' ' -f1 |
  grep -vE '=|^$|^[~.]' |
```

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
  cut -d ' ' -f1 |
  grep -vE '=|^$|^[~.]' |
  sort |
```

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
  cut -d ' ' -f1 |
  grep -vE '=|^$|^[~.]' |
  sort |
  uniq -c |
```

Commandes les plus fréquentes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences.

```
cut -d';' -f2- ~/.history |
  cut -d ' ' -f1 |
  grep -vE '=|^$|^[~.]' |
  sort |
  uniq -c |
  sort -n
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
  cut -d' ' -f1 |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
  cut -d' ' -f1 |
  grep '^[a-z]\+$' |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
  cut -d' ' -f1 |
  grep '^[a-z]\+$' |
  sort |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
  cut -d' ' -f1 |
  grep '^[a-z]\+$' |
  sort |
  uniq -c |
```

Commandes les plus fréquentes après pipe

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre d'occurrences après un pipe.

```
grep '^:' ~/.history |
  sed 's/ | /\n/g' |
  grep -v '^:' |
  cut -d' ' -f1 |
  grep '^[a-z]\+$' |
  sort |
  uniq -c |
  sort -n
```

Commandes les plus longues

Afficher les commandes de l'historique zsh par ordre croissant de leur longueur.

Commandes les plus longues

Afficher les commandes de l'historique zsh par ordre croissant de leur longueur.

```
cut -d';' -f2- ~/.history |
```

Commandes les plus longues

Afficher les commandes de l'historique zsh par ordre croissant de leur longueur.

```
cut -d';' -f2- ~/.history |
  awk '{print length, $0}' |
```

Commandes les plus longues

Afficher les commandes de l'historique zsh par ordre croissant de leur longueur.

```
cut -d';' -f2- ~/.history |
  awk '{print length, $0}' |
  sort -n |
```

Commandes les plus longues

Afficher les commandes de l'historique zsh par ordre croissant de leur longueur.

```
cut -d';' -f2- ~/.history |
  awk '{print length, $0}' |
  sort -n |
  uniq
```

Commandes avec le plus de pipes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre de pipes.

Commandes avec le plus de pipes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre de pipes.

```
tr -dc '|\n' <~/.history |
```

Commandes avec le plus de pipes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre de pipes.

```
tr -dc '|\n' <~/.history |
awk '{print length}') |
```

Commandes avec le plus de pipes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre de pipes.

```
tr -dc '|\n' <~/.history |
awk '{print length}') |
paste - ~/.history |
```

Commandes avec le plus de pipes

Afficher les commandes de l'historique zsh par ordre croissant de leur nombre de pipes.

```
tr -dc '|\n' <~/.history |
awk '{print length}') |
paste - ~/.history |
sort -n
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Autre méthode ?

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Autre méthode ?

```
join <(cut -f1,4 lexique | sort -k1,1) \
  <(cut -f1,4 lexique | sort -k1,1) |
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Autre méthode ?

```
join <(cut -f1,4 lexique | sort -k1,1) \
  <(cut -f1,4 lexique | sort -k1,1) |
  grep 'NOM VER' |
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Autre méthode ?

```
join <(cut -f1,4 lexique | sort -k1,1) \
  <(cut -f1,4 lexique | sort -k1,1) |
  grep 'NOM VER' |
  cut -d ' ' -f1 |
```

Mots ambigus

Afficher les mots du lexique qui peuvent être un nom ou un verbe.

```
comm -1 -2 \
  <(cut -f1,4 lexique | grep NOM | cut -f1 | sort)
  <(cut -f1,4 lexique | grep VER | cut -f1 | sort)
```

Autre méthode ?

```
join <(cut -f1,4 lexique | sort -k1,1) \
  <(cut -f1,4 lexique | sort -k1,1) |
  grep 'NOM VER' |
  cut -d ' ' -f1 |
  uniq
```

Table des matières

Histoire

- Initialement `qed`, 1965-1966.
- Éditeur interactif `ed`, par Ken Thompson, en 1971.
- `grep` (1973), également par Ken Thompson.
- Influence : `ex` (1976) et `vi` (1976), Bill Joy.
- `sed` (1974), Lee E. McMahon.
- Standard, dans POSIX.

Aspects généraux

- Spécifier un programme.
- Programme exécuté sur chaque ligne de l'entrée.
- Sélecteurs pour choisir les lignes.
- Seulement `goto` comme structure de contrôle.
- Seulement deux "variables" pour conserver de l'état.
- Turing-complet.
- Très concis (donc illisible).

Invocation

- `sed 'script'`
- `sed 'script' fichier`
- `sed -f script`
- Options :
  - `-n` désactiver l'affichage implicite des lignes (voir après)
  - `-z` pour des lignes séparées par NULL
  - `-i` pour éditer en place ; `-i.bak`

Pattern et hold space

- Pour chaque ligne, le '\n' de fin est retiré.
- La ligne est placée dans le pattern space
- Les commandes vont modifier le pattern space.
- Finalement le pattern space est affiché implicitement (sauf `-n`)
- Et on passe à la ligne suivante.
- Le hold space est conservé entre chaque itération.

La commande s

`s/regexp/remplacement/options` (ou autre caractère que `/`)

- Options :
  - `g` : remplacer tous les matches (seulement le premier par défaut)
  - `42` : seulement le 42e match
  - `i` : insensible à la casse
- Aussi (si la substitution réussit) :
  - `p` : afficher le pattern space
  - `e` : exécuter le pattern space comme une commande et récupérer le résultat
  - `w fichier` : écrire dans fichier (en dernier)

Regexps

- Syntaxe usuelle des regexp
- Attention aux échappements !
- `a*`, `a\?`, `a\+`, `^`, `$`, `[^a-z-]`, `a{12,}`, `a\|b`, etc.
- Back-références :
  - `\( ... \)` et `\1`, `\2`...
  - `&` pour tout le match (à échapper sinon)
  - `sed 's!<a href="\([^"]*\)">[^<]*</a>!\1!'`
  - `sed 's/./\n/g'`

Autres commandes

`p` afficher le pattern space

- si `p` implicite, afficher encore

`n` ligne suivante

- inclut le `p` implicite

`q` quitter

`q 42` pour retourner un code

`Q` pour désactiver le `p` implicite

`d` vide le pattern space et ligne suivante

- désactive le `p` implicite

Sélection d'une ligne

`10q` affiche les 10 premières lignes

`$d` supprime la dernière ligne

`/regexp/` seulement les lignes qui matchent regexp

- aussi `\_regexp_‘` pour un autre caractère
- peut ajouter 'I' pour ignorer la casse

`1~2p` (avec `-n`) affiche les lignes paires

`1,3d` supprime les lignes 1 à 3

`/CUT/,/CUT/d` supprime entre les CUT (≥ 2 lignes)

`/SKIP/,+1d` ignore SKIP et la ligne suivante

grouper `10{p;p}`

négation `/^yes /!d` supprime sauf ce qui commence par `yes`

- ici, plutôt `-n` et `/^yes /p`

Hold space

- `h` mettre le pattern space dans le hold space
- `g` récupérer le hold space dans le pattern space
- `x` échange hold et pattern
- `H`, `G` pour append au lieu de remplacer

Exemples :

- `sed '1x;1d;$G'`
- `sed '/ABOVE/{x;p;x}'`
- `sed '/BELOW/G'`

Goto

`:addresse` définit une adresse

`b addresse` saute à l'adresse

`t adresse` saute si le dernier `s` a réussi (en gros)

`T adresse` saute si le dernier `s` a échoué

Exemple :

- `sed ':a;s/^.\{1,79\}$/ &/;ta'`

Fichiers

`r fichier` lit le fichier dans le pattern space

`R fichier` lit la prochaine ligne du fichier dans le pattern space

`w fichier` écrit le pattern space dans `fichier`

- Non-exemple :

  ⇒ `sed 's/^include \(.*\)/cat "\1"/e'`

Autres commandes

`i`	insère du texte avant le pattern space
`a`	concatène après le pattern space
`c`	change le pattern space
`D, N, P`	pour la gestion multiligne du pattern space
`=`	affiche le numéro de ligne courante
	⇒ `sed = filename | sed 'N;s/\n/ /'`
`y/ab/AB/`	comme tr mais en moins bien
`autres`	Gestion des lignes multiples, changement de casse, commentaires...

Plus d'infos

- Des one-liners courants et leur explication :
  ⇒ `http://www.catonmat.net/blog/sed-one-liners-explained-part-one/`
- Un tutoriel touffu sur sed :
  ⇒ `http://www.grymoire.com/Unix/Sed.html`
- Le manuel de sed (avec exemples) :
  ⇒ `https://www.gnu.org/software/sed/manual/sed.html`

Table des matières

Histoire

- Aho Weinberger Kernighan
- Sorti en 1977
- Non pas hawk (la buse) mais auk (un genre de pingouin)[2]
- Également dans POSIX

2. Alcidae : mergules, guillemots, pingouins, stariques, macareux

Aspects généraux

- Spécifier un programme.
- Programme exécuté sur chaque ligne de l'entrée.
- Sélecteurs pour choisir les lignes.
- Entrée découpée en champs
- Véritables structures de contrôle !
- Variables !
- Expressions !
- Turing-complet.
- Moins concis, plus proche d'un vrai langage.

$\Rightarrow$ Présentation sommaire

Arguments

- `awk 'script'`
- `awk 'script' fichier`
- `awk -f script`
- `-F` pour définir le séparateur de champ (`FS`)

Exemples simples

`awk '{ print }'`	identité
`awk '{ print $1 }'`	affiche le premier champ
`awk '{ print $2, $1 }'`	inverse les deux premiers champs
`awk 'NR > 1'`	supprime la première ligne
`awk 1`	identité

Motifs

`/regexp/`	comme dans `sed`
`expression`	n'importe quelle expression
`expr ~ /regexp/`	l'expression matche la regexp
`motif1,motif2`	comme dans `sed`
`BEGIN`	début du fichier
`END`	fin du fichier

Exemple

```
awk '{ s += $1 } END { print s+0; }'
```

- Sans condition, on s'exécute sur chaque ligne.
- `s` est une variable.
- Pas besoin de la déclarer, vide par défaut.
- `+0` si l'entrée est vide

Structures de contrôle

- `if (...) { ... } else { ... }`
- `for (i=1; i<=NF; i++) { s += $i }`
- `while`

Variables built-in

`FS` le séparateur de champ pour l'entrée

- Définissable avec `-F`
- Par défaut, whitespace
- Par défaut, fusion des délimiteurs adjacents
- Possibilité de mettre un caractère ou une regexp
- À part pour le défaut, pas de fusion !

`RS` le séparateur d'enregistrement pour l'entrée

`NR` le nombre de lignes lues

`NF` le nombre de records de la ligne (`$NF`)

`OFS` le séparateur de champ pour `print`

`ORS` le séparateur d'enregistrement pour `print`

`OFMT` le format des nombres pour `print` (sinon, `printf`)

Fonctions built-in

`length`	longueur (e.g., de la ligne en cours)
`split`	découper avec un délimiteur
`tolower`	et `toupper`
`sub`	faire un rechercher-remplacer avec regexps
`system`	exécuter une commande externe
`maths`	`int`, `sqrt`, `exp`, `log`, `rand`, etc.

Et encore

- Des one-liners courants et leur explication :
  ⇒ `http://www.catonmat.net/blog/awk-one-liners-explained-part-one/`
- Un tutoriel touffu sur awk :
  ⇒ `http://www.grymoire.com/Unix/Awk.html`
- Le manuel de awk (plus long !) :
  ⇒ `https://www.gnu.org/software/gawk/manual/gawk.html`

Table des matières

Perl

- Perl, Larry Wall, 1987
- Perl 5, 1994
- Perl 6, langage distinct
- Support copieux pour des one-liners
- Plus puissant que les outils présentés
- Évidemment, plus complexe
- “There is more than one way to do it.”

Outils utiles

graphviz dessiner des graphes

- ⇒ `dot -Tps`
- ⇒ diverses commandes pour divers algos
- ⇒ copieux langage de formatage ad hoc
- ⇒ `awk 'BEGIN {print "digraph G {"} ; 1; END {print "}"}'`

gnuplot produire des tracés ; peu commode en one-liner

feedGnuplot produire des tracés en temps réel

curl ou `wget`, télécharger avec HTTP

- ⇒ mode récursif
- ⇒ mécanisation avec POST

xmlstarlet traiter du XML (aussi : XSLT)

jq traiter du JSON (aussi `jshon` et autres)

Outils utiles (suite)

`convert` convertir des images

⇒ `convert file.png file.jpg`

⇒ `convert *.jpg file.pdf`

`pdftk` traiter des pdf

⇒ `pdftk a.pdf b.pdf cat output ab.pdf`

⇒ `pdftk a.pdf cat 1E output arot.pdf`

⇒ `pdftk a.pdf cat 1-2 output a12.pdf`

`ffmpeg` convertir l'audio, la vidéo

⇒ `ffmepg -i input.mp4 output.ogg`

`xclip` gérer le presse-papiers

⇒ `| xclip -i`

`iconv` convertir du texte (encodages)

`formail` extraire des headers mail

Conclusion

- Pratique dans pas mal de situations.
- Être conscient des limites.
- ... ou les ignorer comme un goret. :-)
- Outils anciens
- Pas beaucoup de concurrence (wink).

Conclusion

- Pratique dans pas mal de situations.
- Être conscient des limites.
- ... ou les ignorer comme un goret. :-)
- Outils anciens
- Pas beaucoup de concurrence (wink).

⇒ Merci pour votre attention !