Introduction à `ksh`, le Korn Shell

Thierry Boudet

20 mars 2008

1 Introduction

Vous allez découvrir l'aspect «langage de programmation» d'un interpréteur de commande tournant dans un système Unix. Vous allez apprendre à développer vos propres fichiers de commandes, appelés "scripts", qui vous permettront d'automatiser une grande partie de vos travaux répétitifs.

Vous arriverez donc bientôt à créer vos propres utilitaires, vos propres outils, qui pourront être réutilisables sur n'importe quel système de la famille des Unices/POSIX[1].

1.1 Avertissement

N'ayant pas de système AIX sous le coude, tout ce que vous allez voir a été mis au point et testé sur plusieurs variantes d'Unices libres ou propriétaires.

J'ai, entre autres, sous la main, OpenBSD 3.8 dont le ksh semble très proche du standard, une vieille Debian testing, une toute neuve Slackware 11 et enfin un Solaris 10.

D'autre part, **ksh** est peut-être le shell POSIX standard dans le monde des Unices propriétaires, mais il n'est pas le seul shell utilisable. Le projet GNU[2] propose le Bash (Bourne again shell) que l'on trouve dans les Linux, et qui peut s'installer dans tous les Unices. Il existe bien d'autres shells : `zsh`, `ash`, `pdksh`, chacun ayant des avantages, des inconvénients, une philosophie personnelle. Mais les principes généraux que nous allons voir seront toujours applicables, et certains outils décrits sont quasiment universels.

Et si vous aimez les défis techniques, il est possible d'écrire un shell minimaliste en C (cf page 60) en quelques centaines de lignes de code...

[1] AIX, HP-UX, Linux, Irix, Solaris, OpenBSD, Mac OSX, Minix, Unicos, os400...

[2] `http://www.gnu.org/`

Table des matières

2 Les concepts du shell dans le monde Unix

En fait, il faut bien voir qu'un shell (ou interpréteur de commande) tel que le Ksh n'a rien de différent d'un autre programme utilisateur. Ce fut d'ailleurs une des idées novatrices d'Unix à ses débuts. Il utilise les services fournis par le noyau du système d'exploitation pour accéder aux périphériques, aux fichiers stockés, à la mémoire et pour créer[3] de nouveaux processus. Le shell est donc l'interface privilégié entre vous, l'utilisateur, et le coeur du système.

2.1 Les droits, les permissions

Dans le monde Unix, un utilisateur se voit attribuer un nom et un numéro d'identification. Il fait également aussi partie d'au moins un groupe, ce groupe ayant lui-même un numéro. Son numéro d'identification est connu sous le nom d'UID. Le numéro du groupe principal est son GID. Ces deux numéros sont enregistrés dans le fichier `/etc/passwd` que nous verrons plus en détail bientôt. Si vous désirez connaitre vos UID et GID, utilisez la commande `id` :

```
$ id
uid=4231(tth) gid=4231(tth) groups=4231(tth), 0(wheel)
```

Un bloc élémentaire de droits est constitué de trois bits. Chacun de ces trois bits correspond respectivement à un droit. Dans l'ordre, nous avons lecture, écriture et exécution. Ce groupe de trois bits peut être exprimé de plusieurs façons : par un chiffre octal, ou par une chaine de trois caractères : `rwx`.

Il y a trois de ces blocs, plus d'autres bits, par entité du système de fichier. Le premier bloc est associé à l'utilisateur à qui appartient l'entité. Le second est attribué au groupe propriétaire de l'entité. Quand au troisième bloc, il concerne le reste du monde.

Tout ces bits de permissions, de droits, de modifications, ces numéros d'UID, de UID sont là pour assurer une cohabitation pacifique entres tous les utilisateurs de la machine.

2.2 Les répertoires

Contrairement à d'autres systèmes, Unix se base sur une arborescence unique de répertoires pour stocker vos fichiers. La base de cet arbre est appelé la **racine** du système de fichiers, et on la désigne par le caractère `/`.

Pour arriver dans un répertoire quelconque, on utilise un chemin (path en anglais) tel que `/usr/local/share/libtth/` qui est un chemin **absolu**, car il commence par un `/`, et donc démarre à la racine, la base de notre arborescence.

Par opposition, un chemin **relatif** est, comme son nom l'indique, relatif au répertoire courant. Donc, si celui-ci est `/usr/local/`, utiliser le chemin relatif `share/libtth/` vous emmenera au même endroit que dans l'exemple précédent.

[3] Par la magie du tandem fork/exec.

Chaque répertoire contient aussi deux entrées spéciales : la premières est `.` qui le désigne lui-même. Nous verrons plus tard son importance dans la recherche des commandes[4]. L'autre entrée spéciale est `..` qui représente le répertoire parent. Si vous êtes dans `/opt/foo/bar/`, le chemin `..` représentera donc `/opt/foo/`.

2.3 Les systèmes de fichiers

Pour obtenir cette arborescence unique, et pouvoir utiliser plusieurs disques et/ou partitions, il faut utiliser la notion de point de montage (*mount point*). Un point de montage est tout simplement un répertoire classique, qui doit en principe être vide, mais ça n'est pas indispensable.

Ensuite, on peut «raccorder» une arborescence présente sur un nouveau disque dans ce répertoire point de montage. Cette opération s'appelle le **montage**, parfois prononcé à l'anglaise : mount.

Au passage, on peut utiliser différentes options pour un montage particulier : par exemple un montage en lecture seule, ou une limitation des accès en écriture[5] très utile sur des périphériques lents ou fragiles (disquettes, clef USB). Nous verrons un de ces jours (p. 21) que ces options peuvent avoir une influence sur les tests que proposent un shell.

2.4 Hiérarchie

La hiérarchie des répertoires est relativement standardisée dans les systèmes Unix. Relativement, parce que chaque fournisseur d'Unix dévie un peu par rapport à la structure «de base». Voici une liste, forcément incomplète :

/ Répertoire racine.

/bin/ Outils utilisateur fondamentaux.

/dev/ Accès aux périphériques blocs et caractères.

/etc/ Fichiers de configuration du système.

/home/ Enmplacement par défaut des répertoires des utilisateurs.

/root/ Répertoire personnel du super-user.

/sbin/ Programmes systèmes et outils d'administration fondamentaux.

/usr/ La majorité des applications utilisateur.

- **/usr/bin/** Utilitaires, outils de programmation...
- **/usr/lib/** Bibliothèques de fonctions système.
- **/usr/sbin/** Démons systèmes et leurs outils associés.

/var/ Emplacement des données variables.

- **/var/log/** Journaux du système et des démons.
- **/var/spool/** Répertoire tampons pour l'impression, le mail...

[4] Le point dans le PATH, c'est très mal, et il faudrait en parler...

[5] Option `noatime` qui enlève l'écriture de l'heure du dernier accès à un fichier.

/tmp/ Répertoire pour les fichiers temporaires.

Pourquoi deux répertoires pour les binaires, `/bin` et `/usr/bin` ? Le premier des deux contient les commandes indispensables au démarrage et à la maintenance (en mode single user) du système. Il se trouvera donc placé sur le disque primaire, forcément présent. Le second n'est utile qu'en fonctionnement normal, et peut donc se trouver sur un autre disque qui sera mounté dans `/usr` au cours du lancement du système.

Attention, le répertoire `/tmp` est très souvent vidé au démarrage de la machine. Si vous souhaitez stocker des fichiers temporaires qui survivent à un reboot sauvage, vous pouvez parfois les placer dans `/var/tmp/`, mais est-ce bien raisonnable de garder ce genre de fichiers ?

2.5 Les caractères spéciaux

Certains caractères ont une signification particulière pour le shell. Tout d'abord, les blancs (whitespace en anglais) qui servent par défaut de séparateurs de mots : l'espace, la tabulation et la fin de ligne (newline). Ensuite, les méta-caractères `<`, `>`, `|`, `;`, `(`, `)` et `&`. Ces méta-caractères permettent de former des tokens tels que les redirections (`<<` ou `>&`) ou les expressions conditionnelles (`||` et `&&`).

Enfin, certains caractères sont traités d'une façon spéciale : `\`, `"`, `'`, `#`, `$`, `` ` ``, ` `, `{`, `}`, `*`, `?` et `[`. Si on désire les représenter pour eux-même, il faut les quoter, les protéger. On peut les protéger individuellement en les faisant précéder par un \, ou en groupe en les encadrant pas une paire de simple ou double quotes.

2.6 Syntaxe des commandes

Le shell commence à explorer son entrée (eg : la ligne de commande que vous venez de taper) en la découpant en **mots**. Ces mots sont des séquences de caratères séparées par des blancs ou des méta-caractères (voir aussi la variable IFS). En découpant ces mots et ces tokens, le shell construit les commandes, qui peuvent être de deux types : les commandes simples, typiquement des programmes qui sont exécutés, et les commandes composées, comme les boucles ou les branchements, les groupements et les définitions de fonctions.

Une commande simple est constituée d'une combinaison d'assignation de paramètres, de redirections d'entrées/sorties, et d'instructions. Il y a une seule contrainte : les assignations doivent venir avant les instructions. L'instruction, qui est en fait la commande elle-même peut être une commande interne (*builtin*, cf page 34, une fonction, ou une commande externe. Chaque commande simple a un code de retour.

Une commande composée est crée en utilisant un des mots réservés suivant. Ils ne seront reconnus que s'ils ne sont pas précédés par une assignation de paramètre ou une redirection.

```
case     else     function     then     !
do       esac     if           time     [[
done     fi       in           until    {
elif     for      select       while    }
```

Beaucoup de ces commandes composées correspondent à des structures de contrôle que nous verrons ultérieurement (page 26). La commande `time` sert à chronométrer le temps d'exécution d'une liste de commandes.

3 Les commandes de base

Nous allons commencer par l'indispensable. Dès votre connexion au système, vous allez être positionnés dans votre répertoire personnel, que l'on appelle conventionellement votre HOME. L'interpréteur de commande va alors attendre vos instructions : pour signaler qu'il est prêt, il va afficher un court message que l'on appelle le prompt. Par défaut c'est le cacartère `$` suivi d'un espace. Nous verrons ultérieurement comment personnaliser ce prompt[6].

La commande `echo` est un moyen simple d'afficher des messages. Elle se contente d'écrire sur sa sortie standard les paramètres qu'elle trouve sur sa ligne de commande.

```
$ echo affichons un texte
affichons un texte
$ echo ceci \# est un diese
ceci # est un diese
```

Vous aurez bien entendu noté l'utilisation de l'anti-slash pour protéger le # d'une interprétation erronée (début de commentaire) par le shell.

Signalons au passage que cette commande a un comportement différent selon les variantes d'Unix, en particulier sur le traitement du passage à la ligne suivante. Pour obtenir des affichages plus sophistiqués, il existe une commande, `printf`, nettement plus évoluée.

Quand à une autre commande vitale en milieu professionnel, c'est celle qui permet de connaitre l'heure. Facile, la voici à l'oeuvre :

```
$ heure
ksh: heure: not found
$ alias heure='date +%H:%M:%S'
$ heure ; date
23:06:54
mercredi 18 avril 2007, 23:06:55 (UTC+0200)
```

Et au passage, nous avons découvert la notion d'alias, que nous allons bientôt développer. Je suppose que vous avez plus ou moins compris le principe...

3.1 Génération des noms de fichier

Cette génération des noms de fichier (en anglais : file globbing) s'effectue par rapport au répertoire courant ou, si il a été désigné explicitement, par rapport à celui-ci. Elle s'effectue à partir de caractères spéciaux : `* ? [ ]`.

[6] Recette rapide : `export PS1='\w $ '` (ou `export PS1='$PWD $ '` est d'un grand secours en affichant le répertoire courant dans le prompt.

L'étoite `*` capture un ou plusieurs caractères. Le point d'interrogation `*` un seul caractère. La construction `[...]` capture un quelconque des caractères contenus entre les crochets. Un intervalle peut être défini ainsi : `[A-Z]` accrochera toutes les lettres majuscules. Quand au point d'exclamation, il exprimera la négation : `[!0-9]` désignes les caractères non numériques.

Les fichiers ou répertoires dont le nom commence par un point `.` sont, par défaut, invisibles. En fait, l'`*` n'attrapera pas ce `.` si il est le premier caractère du nom. Si on veut lister ces fichiers invisibles, on devra utiliser `.*`.

```
$ cd ; echo .*
.cshrc .hexdiffrc .kshrc .login .mailrc .profile .ssh
```

3.1.1 Extensions Kornshell

Le Kornshell a rajouté de nouvelles constructions de globbing, basées sur la notion de listes de motifs. Ces extensions ont étés reprises par d'autres shells modernes. Cette liste de motifs est composée d'expressions régulières éventuellement séparées par un `|` (pipe).

`$(`*`liste motifs`*`)` Reconnait optionnellement l'un des motifs.

`*(`*`liste motifs`*`)` Reconnait zéro, une ou plusieurs occurences des motifs.

`+(`*`liste motifs`*`)` Recconnait une ou plusieurs occurences des motifs de la liste.

`@(`*`liste motifs`*`)` Reconnait exactement l'un des motifs de la liste.

`!(`*`liste motifs`*`)` Reconnait tout, sauf l'un des motifs.

XXX ici, il manque vraiment quelques exemples...

3.2 Les répertoires

La commande `pwd` (*print working directory*) affiche le répertoire courant du shell. Il sera aussi le répertoire pas défaut des commandes que vous lancerez à partir de ce shell-là. Si vous disposez de deux shells en même temps le répertoire courant de l'un est totalement indépendant de celui de l'autre shell.

La commande `cd` (*change directory*) est obligatoirement une commande interne (un *builtin*) du shell. Elle vous déplacera d'un répertoire à l'autre, en suivant les règles que nous avons déja vues page 8. La commande `cd` sans arguments vous ramenera dans votre répertoire personnel, votre maison, votre HOME.

Les commandes `mkdir` et `rmdir` permettent de créer et de supprimer un répertoire. Il n'est pas possible de supprimer un répertoire qui n'est pas vide, à moins d'utiliser la commande `rm` avec l'option `-F`

La commande `du` (*disk usage*) calcule l'occupation par sous-répertoire à partir d'un point donné de l'arborescence. L'unité affichée est en général le bloc de 512 bytes et l'option `-k` permet d'obtenir cet affichage en kilo-octets. Quand à `-s`, elle demande d'avoir uniquement le grand total, sans les détails.

3.3 Copier, renommer et effacer

Copier avec `cp`. Il existe deux variantes de cette commande : `cp srcF dstF`, qui fait une copie d'un fichier sous un nouveau nom, et `cp srcFa srcFb ... dstD`, ouù chacun des fichers sources sera copié sous le même nom dans le répertoire de destination. Par défaut, `cp` ne recopie pas les méta-données du fichier (uid, gid, dates...) mais l'option `-p` lui demande de le faire.

Renommer (ou déplacer) avec `mv`. On retrouve la dualité de la commande `cp` sur l'organisation des paramètres fichier sources, et destination.

Effacer avec `rm`. Si le fichier concerné n'est pas lisible, la commande `rm` demandera la permission de faire des dégats.

```
tth:155$ chmod 000 toto
tth:156$ rm toto
override ---------  tth/tth for toto?
```

Ces trois commandes partagent une option `-i` qui a la même signification : un fonctionnement interactif qui demande une confirmation pour chaque opération potentiellement dangeureuse (écrasement ou effacement d'un fichier).

```
$ rm -i ekk*
remove ekko? y
remove ekko.c? n
$ ls ekk*
ekko.c
```

3.4 Lister et regarder vos fichiers

La commande `ls` vous donnera la liste, classée par ordre alphabétique, des fichiers visibles dans le répertoire courant. Par convention, les fichiers ou répertoires dont le nom commence par un point sont invisibles pour certaines commandes, dont `ls` ; l'option `-a` de `ls` autorise leur affichage. L'option `-1` (le chiffre 1) force l'affichage sur une seule colonne.

La méthode la plus primitive pour regarder le contenu d'un fichier est l'utilisation de la commande `cat`, qui se contente de recopier tout le contenu du fichier dans votre terminal. Bien entnedu, si le fichier a plus de lignes que votre affichage, vous perdez le début... D'une autre coté, `cat` propose certaines options dignes d'intérèt.

En utilisant `more`, l'affichage se fera page par page. Un appui sur la barre d'espace affichera la page suivante, et la touche *Enter* avancera d'une ligne. Le `more` traditionnel est assez limité, il ne peut, par exemple, pas remonter vers le début du fichier. Sur certains systèmes, on trouvera donc son successeur, `less`, beaucoup plus puissant.

Pour examiner le contenu d'un fichier octet par octet, `od` propose divers affichages : octal, hexadécimal, flottants. Voici rapidement quelques options utiles : `-a` et `-c` affichent caractère par caractère avec une présentation différente des chars non-imprimables, `-t x1` affichera les octets en hexadécimal. Il semble que le successeur d'`od`, `hexdump`, soit plus cohérent au niveau des options.

3.5 Modifier un fichier : l'éditeur Vi

Nous allons très brièvement voir l'utilisation de l'éditeur de texte **Vi**, outil dont la connaissance est indispensable. En effet, on le retrouve dans toutes les variantes, tous les dérivés de la famille Unix.

Vi semble très austère au premier abord, et même vraiment rébarbatif. Il a sa propre logique, qui semble au premier abord echapper à la compréhension humaine. **Vi** connait plusieurs modes de fonctionnement. Par défaut, il est dans le mode "commande". Et pour revenir dans ce mode, il faut utiliser la touche ESC.

En cas de panique, et si vous n'avez pas de serviette, la séquence magique pour quitter Vi est : deux, trois tapotements sur la touche ESC, puis la séquence `:q!` vous permettra de sortir de là, mais sans enregistrer vos modifications, Il vaut mieux parfois ne pas sauvegarder ses erreurs.

Il existe bien d'autres éditeurs de texte, autant en mode texte qu'en mode graphique. Nous en verrons peut-être quelques uns plus tard. Mais n'oubliez pas, Vi est universel, et Vim, un de ses clônes du monde libre est vraiment surpuissant.

4 Redirections d'entrées/sorties

Chaque processus dispose de trois canaux pour communiquer avec le monde extérieur : un pour l'entrée, et deux pour la sortie. En entrée, il est connu sous le nom de stdin et il a le numéro **0**. En sortie, ils sont deux : le premier (stdout, **1**) sort les données, les résultats, de l'application, et le second (stderr, **2**) affichera les éventuels messages d'erreur de cette application.

Le shell permet de rediriger ces divers canaux vers d'autres horizons.

4.1 Redirections basiques

L'usage le plus immédiat est l'enregistrement dans un fichier de ce que la commande concerné pensait afficher sur votre écran. Voici une petite série d'exemples qui va vous montrer, à partir des bases que nous avons, ce principe de redirection :

```
$ echo guinness > boisson
$ cat boisson
guinness
$ tr g G < boisson > breuvage
$ cat breuvage
Guinness
```

Quand le shell interprète la première ligne de cet exemple, il découvre plusieurs choses : la commande est `echo`, son paramètre est `guinness`, et le "token" > lui demande de transférer la sortie de la commande vers le fichier «boisson», qui sera crée, ou effacé si il existe à ce moment.

Quand le shell interprète la troisième ligne, il va s'arranger pour que l'entrée de la commande `tr` vienne en fait du fichier «boisson». Quand à la sortie, c'est déja vu.

Une autre forme de redirection permet d'ajouter des données à la fin d'un fichier : >>. Si le fichier cible n'existe pas, il sera créé à la volée.

```
$ echo "quelle heure est-il ?" > heure
$ date >> heure
$ cat heure
quelle heure est-il ?
Thu Apr 12 10:03:22 CEST 2007
```

4.2 Redirections avancées

C'est un bien grand mot... Pour le moment, nous avons travaillé avec l'entrée et la sortie standard. Maintenant, nous voulons enregistrer les éventuels messages d'erreur de notre programme. Exemple :

```
$ cat /foo/bar
cat: /foo/bar: No such file or directory
$ cat /foo/bar 2> erreur
$ cat erreur
cat: /foo/bar: No such file or directory
```

Si vous désirez emettre un message d'erreur à partir d'un script, vous pouvez utiliser la redirection `>&2` qui transfère la sortie standard vers la sortie d'erreur.

Certains shells (dont le Ksh) disposent d'une option, *noclobber*, qui peut permettre d'interdire à une redirection de type > de remplacer un fichier existant. Dans ce cas, vous pouvez utiliser `>|` qui forcera l'écrasement du fichier existant.

```
$ date > mon_heure
ksh: cannot create mon_heure: File exists
$ date >| mon_heure
$ cat mon_heure
Sat Apr 14 06:30:46 CEST 2007
```

Maintenant, petite question : pourquoi `cmd < foo > foo` ne donnera probablement pas le résultat espéré ? Ne vous en faites pas, tout le monde s'est déja fait piéger, et tout le monde se referra piéger un de ces jours, de préférence en `root`, sur un fichier vital et le vendredi soir.

4.3 Les documents sur-place

Aussi connus sous le nom de here-documents, ils permettent d'utiliser des textes multilignes dans le corps d'un script shell. Voici un exemple depuis la ligne de commande :

```
$ cat << __EOT__
> ceci est un texte
> multiligne.
> __EOT__
ceci est un texte
multiligne.
```

Les trois caractères > suivis d'une espace que vous voyez en début de ligne correspondent au deuxième prompt (`PS2`) et sont émis par le shell pour signaler qu'il attend plus d'entrées pour compléter la commande.

Dans les scripts que nous allons bientôt écrire, il est parfois nécessaire d'indenter le code afin d'en améliorer la lisibilité. En utilisant le redirecteur `<<-`, vous demandez la suppression des tabulations en début des lignes de votre texte.

Le délimiteur de fin (ici `__EOT__`) peut être choisi librement. A la fin du texte, il doit impérativement placé en fin de ligne. Si il est encadré par des quotes, l'interpolation des variables n'aura pas lieu :

```
$ cat demo_in_place.ksh
#!/bin/ksh
heure=$(date +%H:%M:%S)
echo heure actuelle $heure
cat << __EOT__
l'heure est $heure maintenant
__EOT__
```

```
cat << "__EOT__"
l'heure est $heure maintenant
__EOT__
$ ./demo_in_place.ksh
heure actuelle 18:03:11
l'heure est 18:03:11 maintenant
l'heure est $heure maintenant
```

5 Les pipelines

Un jour, très tôt dans la genèse d'Unix, un des créateurs[7] a eu l'idée de brancher la sortie d'une commande sur l'entrée d'une autre commande. En gros brancher des tuyaux sur d'autres tuyaux. Si nous reprenons un des exemples précédents, il est facile de le simplifier avec un pipe.

```
$ echo guiNNess | tr gN Gn > gourmandise
$ cat gourmandise
Guinness
```

La sortie de la commande `echo` va donc transiter directement vers l'entrée de la commande `tr`. Brillante idée, n'est-ce pas ? Le concept de «boite à outils logicielle» venait de naitre.

Il est possible d'enchainer ainsi plusieurs commandes, chacune d'entre elles recevant les données de la précedente et envoyant ses résultats à la suivante. Le shell met en place les mécanismes de communication et c'est le noyau qui se charge de la synchronisation entre les divers processus. Pour les programmes mis en oeuvre tout au long de la chaine, c'est une opération quasiment transparente.

Il est même possible de «prélever» le flux qui transite entre deux commandes en intercalant un dérivation, un `T`. C'est la commande `tee` qui se charge de ça.

```
$ date +%H:%M:%S | tee heure | tr 0123456789 abcdefghij
bb:aj:db
$ cat heure
11:09:31
```

Une des composantes de la «philosophie» Unix est : *chaque composant fait une tache, mais la fait bien, et on peut les faire travailler ensemble*. Surtout quand on les regroupe dans un script, afin de ne plus avoir à retaper toutes ces commandes ésotériques.

Nous verrons plus tard (p. 67) un ensemble d'outils, `netpbm`, destiné au traitement des images qui exploite à fond la notion de pipelines.

[7] Désolé, je ne sais plus trop qui c'est, peut être Dennis Ritchie.

6 Utiliser les variables

Une variable est une entitée nommée qui peut contenir une valeur, qui sera toujours une chaine de caractères. Elle pourra être vide, auquel cas on parlera de chaine nulle (null). Elle pourra être interprétable comme un nombre entier. Le type et les attributs d'une variable, en Ksh, peuvent être changés par la commande `typeset`.

La définition d'une variable classique est *presque* simple. Il n'y a pas de déclaration préalable à faire, une simple affectation suffit à l'emmener à l'existence :

```
$ foo=42
$ echo $foo
42
$ foo="Nicolas et Leon"
Nicolas et Leon
$ foo= "Nicolas et Leon"
ksh: Nicolas et Leon: not found
```

J'ai dit presque car il y a un piège à éviter : il ne doit **pas** y avoir d'espace de part et d'autre du signe "egale"[8].

La concaténation de plusieurs variables XXX

Ces variables peuvent se voir assignées une valeur de plusieurs façons différentes. Tout d'abord, le shell en positionne implicitement un certain nombre comme `#`, `!` ou `$`. C'est la seule façon de positionner les paramètres nommés par un seul caractère spécial. En second, ils peuvent être importés de l'environnement du shell lors de son démarrage.

Troisièmement, on peut faire des assignations sur la ligne de commande : `VAR=valeur`. On peut en faire plusieurs sur la même ligne, et les faire suivre d'une commande simple. La quatrième façon de positionner un paramètre est avec les commandes `export`, `readonly` et `typeset`, que nous verrons bientôt. Ensuite, les boucles `for` et `select` positionnent leur paramètre, tout comme les commandes `getopts`, `read` et `set -A`. Et finalement, il peut y avoir des assignations à l'interieur des expressions arithmétiques ou en utilisant la forme `${nom=valeur}` des substitutions.

6.1 Paramètres implicites

Les paramètres spéciaux qui suivent sont implicitements positionnés au moment opportun par le shell, et ne peuvent être positionnés directement par une assignation classique :

! PID du dernier processus démarré en tache de fond. Ce numéro pourra être utilisé avec `wait`.

Nombre de paramètres positionnels ($1, $2, $3, etc.).

$ PID du shell, ou PID du shell d'origine si on est dans un sous-shell.

? Le code de retour de la dernière commande non asynchrone exécutée. Si cette dernière commande a été tuée, `$` est positionnée à 128 + plus le numéro de signal (voir p. 24).

[8] Comme vous pouvez le constater, au passage, j'ai perdu mes gosses et le message d'erreur du kornshell n'est pas très explicite, bien que très logique.

0 Nom du shell, ou du script courant, ou de la fonction en cours.

1 .. 9 Les neuf premiers paramètres positionnels qui ont été fournis au shell, à la fonction ou au script. Les paramètres suivants sont accessibles par la notation ${numéro}/

***** Tous les paramètres positionnels, sauf le 0.

@ Tous les paramètres positionnels, sauf le 0.

6.2 Environnement

Certaines variables servent à paramétrer, à modifier le fonctionnement de diverses applications, dont bien sûr le shell que vous êtes en train d'utiliser...

On les nomme *variables d'environnement* car elles sont "exportées" vers tous les processus descendants du shell courant. Certaine de ces variables sont positionnées automatiquement et ne doivent **pas** être modifiées : `HOME`, `LOGNAME`, `USER`. D'autres sont aussi positionnées automatiquement et peuvent être modifiées : `SHELL`, `TERM`, `PATH`, `PAGER`... Finalement, d'autres peuvent être renseignées par l'utilisateur à partir de son `.profile` ou interactivement par les commandes suivantes :

```
$ FOO="premiere variable" ; export FOO
$ export BAR="seconde variable"
$ env | egrep FOO\|BAR
FOO=premiere variable
BAR=seconde variable
```

La commande `printenv` permet de les visualiser.

```
SSH_CONNECTION=192.168.0.42 32771 192.168.0.50 22
TTY=ttyp1
PS1=\w $
PATH=/home/tth/bin:/bin:/sbin:/usr/bin:/usr/local/bin:
SHELL=/bin/ksh
HOSTNAME=puce.none.invalid
USER=tth
HOME=/tth/korn
OS=OpenBSD
TERM=xterm
LANG=C
```

Pour ne pas encombrer le document, j'ai enlevé pas mal de lignes, mais vous pourrez constater par vous-même qu'il y a un certain nombre d'informations.

6.3 Substitutions

La forme `${nom}` de substitution de variable peut voir son comportement modifié par des expressions conditionnelles :

`${var :-valeur}` Si `var` existe et n'est pas nul, elle est substituée, sinon, c'est `valeur` qui est substituée.

${var :+valeur} Si `var` existe et n'est pas nul, c'est `valeur` qui est substituée, sinon rien n'est fait.

${var :=valeur} Si `var` existe et n'est pas nul, elle est substituée, dans le cas contraire, on lui assigne `valeur` et le résultat est substitué.

${var :?valeur} Si `var` existe et n'est pas nul, elle est subtituée, sinon `valeur`, précédé du texte `name :` est imprimé sur la sortie d'erreur et une erreur survient.

Note : la forme exacte est `${nom:-essai}` et l'espace que vous voyez vient d'un souci de LaTeXque je n'arrive pas à régler.

6.4 Les tableaux

Un tableau est aussi une variable utilisable dans le shell. Le nombre d'élements que peut contenir un tableau est parfois limité. Les indices ne peuvent être que des nombres entiers positifs.

On peut récupérer le nombre d'éléments d'un tableau avec ces deux formes de substitutions : `${#name[*]}` et `${#name[@]}`. Une expression de la forme `$((..))` peut être utilisée comme un indice de tableau.

Vous trouverez un autre exemple d'utilisation à la page 28.

7 Calculs arithmétiques et logiques

Le Korn Shell dispose de possibilités assez avancées pour faire des calculs simples.

```
$ let a=6\*7 ; echo $a
42
$ printf 'o %d o\n' $((6*7))
o 42 o
```

Les valeurs numériques manipulées par le shell sont codées en interne sur des nombres entiers et signés de 32 bits, ce qui nous donne comme valeur maximum 2147483648.

Il est également possible d'utiliser la commande externe `expr`, qu'on retrouve fréquemment dans les scripts écrits pour le Bourne shell :

```
$ echo `expr 6 \* 7`
42
```

7.1 Les calculateurs `bc` et `dc`

Pour des calculs plus avancés, il existe le tandem `dc`/`bc`. Bc est à la fois un langage et une calculatrice en précision arbitraire. Dc est une calculatrice structurée autour d'une pile, c'est à dire qu'il utilise la notation polonaise inverse. Bc peut servir de pré-processeur pour dc. A moins d'être adepte des calculettes Hewlett-Packard ou du RPL/2, il est peu probable que vous n'appréciez bc.

```
korn:39$ echo "scale=42; a(1)*4" | bc -l
3.141592653589793238462643383279502884197168
```

La version GNU de `bc` propose quelques extensions, telle que la fonction *read()* qui permet de saisir une valeur numérique au cours de l'exécution d'un programme bc.

8 Codes de retour, commandes de test

Chaque commande exécutée dans le système a la possibilité de dire à l'entité qui l'a invoquée si elle a réussi à faire ce qu'on lui avait demandé. Ce compte-rendu est représenté sour la forme d'un petit nombre entier appelé le 'code de retour'.

Dans un contexte «interpréteur de commande», par convention, la valeur **0** représente la réussite ou le **Vrai**, et une valeur différente de 0 indiquera l'échec ou le **Faux**. La valeur précise donne souvent le type de l'erreur, la cause du disfonctionnement.

Dans notre contexte, l'entité invocatrice est le `ksh`, celui-ci nous offre donc la possibilité de lire ce code de retour après l'appel d'une commande externe. Ce code est stocké dans la variable pré-définie `$ ?` mise à jour après chaque exécution.

```
$ true ; echo $? ; $ false ; echo $?
0
1
$ cat /foo/rien
cat: /foo/rien: No such file or directory
$ echo $?
1
```

En général, la documentation des divers outils vous précisera les valeurs et significations des différents codes de retour. On peut tester ce code et prendre des décisions adéquates pour la suite des opérations. Brièvement, voici la façon de faire :

```
fichier=/foo/rien
cat $fichier > /dev/morse
if [ $? -ne 0 ]; then
        echo je ne peux pas lire $fichier >&2
fi
```

Rappellons que la notation `>&2` redirige la sortie de la commande `echo`, notre message de désarroi, vers la sortie d'erreur.

Cet exemple est un peu artificiel, on aurait mieux fait de vérifier avant l'opération[9] que le fichier soit accessible en lecture. Les commandes de test sont là pour ce genre de chose.

[9] Bien que... l'état du fichier puisse changer entre le test et l'opération de lecture.

8.1 Tester

Nous allons commencer par voir les vérifieurs[10] unaires, qui s'appliquent pour la plupart à des fichiers :

`-d file` Vrai si le fichier est en fait un répertoire qui existe.

`-e file` Vrai si le fichier existe, quel que soit son type.

`-f file` Vrai si le fichier est régulier et existe.

`-h file` Vrai si le fichier est en fait un lien symbolique.

`-k file` Vrai si le fichier existe et a son bit *collant*. *Oui, mais c'est quoi ce bit ?*

`-n str` Vrai si la taille de la chaine est non nulle.

`-p file` Vrai si le fichier est un tube nommé (FIFO).

`-r file` Vrai si le fichier existe et est lisible.

`-s file` Vrai si le fichier a une taille supérieure à zéro.

`-w file` Vrai si le fichier existe et est writable. Le test se fait sur le flag d'écriture uniquement, et peut donc donner un résultat erronné si le fichier est sur un fs monté en read-only.

`-x file` Vrai si le fichier existe et est exécutable. Même remarque que pour `-w` : si le fs est monté en *noexec*.

`-z str` Vrai si la taille de la chaine est zéro.

Et maintenant, voici les comparateurs binaires :

`file1 -nt file2` Vrai si file1 existe et est plus récent que file2. Ce test peut être utilisé pour une compilation conditionnelle, mais l'utilisation de `Make` devrait alors être envisagée. Plus d'informations page 65.

`file1 -ot file2` Vrai si file1 existe et est plus ancien que file2.

`file1 -ef file2` Vrai si file1 et file2 existent, et se réfèrent au même fichier.

`str1 = str2` Vrai si les deux chaines sont identiques.

`str1 != str2` Vrai si les deux chaines sont différentes.

`str1 < str2`

`str1 > str2`

`str1` Vrai si la chaine n'est pas nulle.

`nbr1 -eq nbr2` Vrai si les deux entiers sont égaux.

`nbr1 -ne nbr2` Vrai si les deux entiers sont différents.

`nbr1 -gt nbr2` Vrai si nbr1 est supérieur à nbr2.

`nbr1 -ge nbr2` Vrai si nbr1 est supérieur ou égal à nbr2.

`nbr1 -lt nbr2` Vrai si nbr1 est inférieur à nbr2.

`nbr1 -le nbr2` Vrai si nbr1 est inférieur ou égal à nbr2.

[10] J'aime bien ce néologisme.

Toutes ces primitives unaires ou binaires peuvent être combinées entre elles par des opérateurs dont voici la liste :

! expr Vrai si l'expression est fausse.

expr1 -a expr2 Vrai si les deux expressions sont vraies.

expr1 -o expr2 Vrai si une des deux expressions est vraie.

(expr) Vrai si l'expression est vraie.

La valeur retournée par la commande `test` est une des suivantes :

0 L'expression a été évaluée vraie.

1 L'expression a été évaluée fausse, ou elle est manquante.

>1 Une erreur indéterminée est survenue.

La commande `test` a d'autres possibilitées, moins utiles dans un usage courant. La forme `test` *`expression`* peut aussi s'écrire `[` *`expression`* `]`.

9 Ecriture d'un script

Voilà, nous en avons assez vu pour passer à l'écriture de nos propres script. Un script shell n'est qu'un fichier texte contenant une suite de commandes, ressemblant à ce que vous auriez pu taper sur la ligne de commande :

```
$ cat > mon_premier_script
date
echo $RANDOM
^D
$ /bin/ksh mon_premier_script
Fri Apr 13 08:14:00 CEST 2007
10378
```

Rien de bien sorcier : que le shell lise ses instructions depuis votre terminal ou depuis un fichier, il fait la même chose. Il y a quand même quelques petites choses à préciser afin que votre script soit pleinement utilisable, comme le serait une commande classique.

9.1 Emplacement et permission

Votre shell, quand vous invoquez une commande, va la rechercher dans un certain nombre d'emplacement : est-ce un *builtin*, un alias, une fonction, une commande externe ? Et si c'est une commande externe, où est-elle dans l'arborescence ?

Pour rechercher un exécutable, le shell utilise la variable d'environnement PATH qui contient une liste de chemins, séparés par le caractère `:`.

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/home/tth/bin
```

Si le script que vous voulez lancer n'est pas dans le $PATH, il faut préciser le chemin (relatif ou absolu) vers celui-ci. Ensuite, il **faut** que le droit d'exécution soit positionné sur le fichier trouvé. Vous devrez donc utiliser la commande `chmod` pour le modifier.

```
$ ./mon_premier_script
ksh: ./mon_premier_script: cannot execute - Permission denied
$ chmod u+x mon_premier_script
$ ./mon_premier_script
Fri Apr 13 08:37:28 CEST 2007
10299
```

9.2 Le shebang

Une fois que le fichier a été localisé, le noyau se demande ce qu'il va en faire. Pour cela, il examine les premiers octets du fichier qui contient une signature spécifique à l'architecture. Si cette signature correspond à un format binaire, il l'exécute directement.

Par contre si les deux premiers caractères sont `#!`, que l'on nomme shebang, c'est un fichier de script qui doit être «joué» par l'interpréteur adéquat. Le chemin complet de celui-ci, et une éventuelle **unique** option se trouve alors sur la première ligne du fichier, après le shebang. Cet interpréteur est alors lancé, avec en premier paramètre sur sa ligne de commande le chemin du fichier script, et ensuite les options fournies l'utilisateur. Démonstration :

```
$ cat texte.numero
#!/bin/cat -n
hello world
$ ./texte.numero
     1  #!/bin/cat -n
     2  hello world
```

C'est donc la ligne `/bin/cat -n texte.numero` qui sera réellement executée lors de l'invocation de la commande `./test.numero` à partir du prompt.

Bien entendu, dans le cas de *mon_premier_script*, la première ligne sera `#!/bin/ksh`. Beaucoup d'outils Unix peuvent être utilisés par l'intermédiaire de ce mécanisme d'appel d'un interpréteur spécifique. C'est également pour cette raison que le caractère qui marque le début des commentaires est un # dans tous les langages de scripts relativement civilisés.

10 Contrôle des signaux

Un "signal" est un mécanisme que le noyau utilise pous indiquer à un processus en cours d'exécution un évènement exceptionnel tel qu'une division par zéro, un accès mémoire illégal, ou une interruption par l'utilisateur.

Voici une liste des signaux disponibles, au niveau OS, dans un OpenBSD :

```
~/Formation $ kill -l
 1    HUP Hangup                      17   STOP Suspended (signal)
 2    INT Interrupt                   18   TSTP Suspended
 3   QUIT Quit                        19   CONT Continued
 4    ILL Illegal instruction         20   CHLD Child exited
 5   TRAP Trace/BPT trap              21   TTIN Stopped (tty input)
 6   ABRT Abort trap                  22   TTOU Stopped (tty output)
 7    EMT EMT trap                    23     IO I/O possible
 8    FPE Floating point exception    24   XCPU Cputime limit exceeded
 9   KILL Killed                      25   XFSZ Filesize limit exceeded
10    BUS Bus error                   26 VTALRM Virtual timer expired
11   SEGV Segmentation fault          27   PROF Profiling timer expired
12    SYS Bad system call             28  WINCH Window size changes
13   PIPE Broken pipe                 29   INFO Information request
14   ALRM Alarm clock                 30   USR1 User defined signal 1
15   TERM Terminated                  31   USR2 User defined signal 2
16    URG Urgent I/O condition
```

Pour «envoyer» un signal à un processus en cours d'exécution, on utilise la commande `kill` qui prend en option le numéro ou le nom symbolique du signal, et en paramètre le PID du processus destinataire.

```
~ $ sleep 100 &
[1] 6754
~ $ kill 6754
[1] + Terminated           sleep 100
```

De part sa nature, l'arrivée d'un signal extérieur dans un processus en cours d'exécution est totalement asynchrone, et peut donc arriver à un moment inopportun.

10.1 Intercepter un signal

Dans un script shell, il est possible d'intercepter un certain nombre de signaux et de dérouter l'exécution vers une séquence d'instructions spécifique.

On utilisera pour ça la commande interne `trap` dont la syntaxe est `trap '`*`handler`*`' signal [signal]`. Le *handler* peut être une chaine nulle, indiquant que le signal doit être ignoré, un signe moins `-`, indiquant que l'action par défaut pour ce signal doit être prise, ou enfin une chaine contenant des commandes shell à évaluer et exécuter dès que possible (c'est à dire quand la commande

actuelle se termine, ou avant d'afficher le prochain prompt PS1) après réception d'un des signaux de la liste.

```
...
trap 'echo "Information Request >> log.appli" ; sleep 1' INFO
...
```

10.2 Signaux particuliers

Le signal `EXIT` (aussi connu sous le numéro 0) est traité quand le shell est sur le point de se terminer, et `ERR` qui est executé après qu'une erreur soit arrivée.

Le signal `EXIT` est souvent employé pour assurer un nettoyage final en fin de traitement : effacement des fichiers temporaires, en général. Si on a un certain nombre de ces fichiers, il est plus simple d'utiliser ce genre de séquence :

```
dirtemp="/tmp/coucou.$$" ;  mkdir $dirtemp
fictemp1=$dirtemp/1 fictemp2=$dirtemp/2 fictemp3=$dirtemp/3
trap 'rm -Rf $dirtemp' EXIT
```

Le signal `ERR` est généré par quelque chose qui causerait une sortie du shell si l'option `-e` ou `errexit` était positionnée. Voir la commande `set` pour plus de détails.

11 Structures de contrôle

Boucles, sélections, branchements, menus... Nous en avons parlé brièvement quand nous regardions le shell explorer sa ligne de commandes en page 9. Ce sont des commandes composées.

11.1 L'alternative

Vous avez procédé à une vérification quelconque, probablement avec la commande `test`, et vous avez obtenu une valeur logique : *Vrai* (**0**) ou *Faux* (**>0**). Cette valeur est donc une *condition* qui va vous servir à prendre une décision.

```
if condition
then
        liste de commandes
else
        liste de commandes
fi
```

11.2 Conditions multiples

On a parfois une variable qui peut prendre plusieurs valeurs, par exemple un paramètre du script qui va choisir l'action à exécuter. Il nous faut donc un branchement multiple, qui permette également de gérer la condition «*mauvais choix*».

```
case identifiant in
        motif 1)
                liste de commandes
                ;;
        motif 2)
                liste de commandes
                ;;
esac
```

Et pour gérer le cas du «*mauvais choix*», il suffit de mettre en dernier `case` le motif `*` qui attrapera n'importe quoi.

11.3 Boucle conditionnelle

Il existe deux formes de cette boucle : soit on continue tant que tout va bien, soit on attend qu'enfin les choses soient bonnes. Il faudra donc qu'à un moment dans la boucle quelque chose soit modifié.

```
while condition
do
        liste de commandes
done
```

Nous allons boucler tant que la condition est vraie. Si on remplace le `while` par un `until`, nous tournerons jusqu'à ce que la condition devienne vraie. La boucle `while` est souvent utilisée pour lire un fichier ligne par ligne :

```
cat /etc/hosts | while read ligne
do
       echo $ligne
done
```

Une autre construction classique dans les scripts shell est la boucle infinie faite avec le `while`. La sortie éventuelle de cette boucle se faisant au milieu de la *liste de commandes* par un test spécifique et l'instruction `break` qui ordonne de quitter la boucle en sautant juste après le `done`. La condition à utiliser est `:` (le caractère "deux points") qui retournera toujours la valeur Vrai.

11.4 Boucle itérative

Quand vous devez refaire le même traitement sur plusieurs choses, par exemple plusieurs fichiers sélectionnables, vous procederez à une itération.

```
for identifiant in liste de valeurs
do
        liste de commandes
done
```

Il existe un certain nombre de manières pour construire la liste de valeurs. Si `in liste de valeurs` est omis, les paramètres positionnels seront utilisés à la place. Le code de retour de la boucle `for` est celui de la dernière commande de la liste, ou 0 si rien n'a été exécuté.

```
datas=${1:-"alpha+bravo+charlie+delta+echo"}
IFS='+'
for foo in $datas
do
        up=$(echo $foo | tr a-z A-Z)
        printf "%-15s %-15s\n"  $foo  $up
done
```

Si cet exemple a été sauvé dans un fichier nommé `boucle_for.ksh`, nous obtiendrons ce résultat :

```
$ ./boucle_for.ksh alice+bob+charly
alice           ALICE
bob             BOB
charly          CHARLY
```

Deuxième exemple de boucle `for`, qui reprend aussi l'utilisation des variables de type tableau et de la commande `typeset`.

```
#!/bin/ksh
typeset -i  i=0                          # type entier
typeset -u  cartes                       # Force en majuscules
for couleur in pique coeur carreau trefle
do  for valeur in as 2 3 4 5 6 7 8 9 10 valet roi dame
    do  cartes[i]="$valeur de $couleur"
        i=i+1          # i est une variable entiere, le "let" est inutile
    done
done
print -- ${cartes[RANDOM%52]}
```

A titre d'exercice, vous pouvez essayer de faire un 421.

11.5 Faire un menu

Nous allons découvrir une fonctionnalité très interessante à maitriser : la création facile de petits menus permettant un choix interactif entre diverses options.

```
select choix in liste de mots
do
         liste de commandes
done
```

Un petit exemple vous montrer l'essentiel de cette structure en action :

```
PS3="Votre Choix : "
select foo in Heure Jour Quitter
do
      echo $REPLY " -> " $foo
done
```

PS3, le troisième prompt, est utilisé pour demander la réponse de l'utilisateur et peut être modifié pour avoir une demande moins cryptique que le sobre `# ?` de la valeur par défaut.

Quand l'utilisateur exécute ce morceau de code, il obtient ceci :

```
1) Heure
2) Jour
3) Quitter
Votre Choix : _
```

A ce moment, il peut répondre par une des valeurs numériques proposées. Si la sélection est valide, la variable `foo` sera renseignée par la valeur correspondante de la liste de mots ou effacée

dans le cas contraire. La variable `REPLY` sera garnie avec ce qui a été lu, amputé des blancs de debut et de fin. Et finalement la liste d'instructions entre le `do` et le `done` sera exécutée.

Et pour ceux qui désire plus d'interactivité, il existe dialog qui permet d'avoir des menus, des boites de dialogue, des trucs et des machins.

12 Les fonctions

De la réutilisabilité des composants logiciels, de la difficultée de trouver des exemples...

Pour commencer, nous allons définir, depuis la ligne de commande, une fonction très simple, et que nous pourrons utiliser depuis cette même ligne de commande.

```
~/funk $ function plop
> {
>  echo PLOP $RANDOM
> }
~/funk $ plop
PLOP 15568
~/funk $ plop
PLOP 8380
~/funk $ type plop
plop is a function
```

Une syntaxe alternative pour déclarer une fonction est la forme `plop () { bla bla }`, que l'on retrouvera plus souvent dans les scripts anciens, écrits pour le shell Bourne.

Une liste des fonctions présentes peut être obtenue par la commande `typeset +f`. La liste de leurs définitions peut être obtenue avec `typeset -f`. On peut effacer une fonction existante avec `unset -f` *`nom-de-la-fonction`*. Il existe un mécanisme pour le chargement automatique de fichiers contenant le code d'une fonction.

12.1 Fonction de mise en boîte

Nous allons écrire un de ces fameux composants logiciels qui nous permettra d'afficher bien en evidence un message lors d'un traitement à rallonge. Nous allons refaire un clône minimaliste de la commande `boxes` dans une fonction. Voici un exemple d'utilisation :

```
/home/korn/Formation 66 : . ./texte_en_boite.ksh
/home/korn/Formation 67 : texte_en_boite "un message"
+----------------+
|   un message   |
+----------------+
/home/korn/Formation 68 : texte_en_boite "$(date)"
+----------------------------------+
|   Tue Apr 17 18:36:12 CEST 2007  |
+----------------------------------+
```

La commande 66 va *sourcer*, c'est à dire exécuter par le shell courant dans son propre contexte, le fichier contenant le code de notre fonction.

Et les deux commandes suivantes vont l'utiliser. Nous pouvons voir que notre fonction attend un argument : le texte à afficher. Que se passe-t-il si ce paramètre est absent ? Et si nous avons plus d'un argument ?

```
/home/korn/Formation 69 : texte_en_boite
+--------------------+
|   BOITE DE TEXTE   |
+--------------------+
```

Bravo, nous avons un comportement par défaut en cas de manque. Quand aux arguments surnuméraires, ils seront tout simplement ignorés.

Analyse, étude du code : nous avons deux fois l'affichage d'une ligne de tirets d'une longueur variable, nous avons la gestion d'une valeur par défaut, et nous avons en argument une chaine dont la longueur est variable. Le début de notre fonction est :

```
function texte_en_boite
{
##echo "texte_en_boite " $# $@          # verification des arguments
texte=${1:-"BOITE DE TEXTE"}            # message par defaut
taille=${#texte}                        # mesure de la taille
```

La ligne qui débute par `##` est une convention[11] personnelle : ces deux dièses m'indiquent qu'il y a la possibilité, à cet endroit, d'afficher un message de debug.

Ensuite, nous traitons successivement l'éventuelle absence de l'argument (qui se trouverait dans le paramètre `$1`) grâce à la construction `${nom:-default}` déja vue dans le passé, et la mesure de la taille de notre texte avec `taille=${#texte}`. Suite et fin du code de la fonction :

```
texte_en_boite_trait $taille
print  "|  " $texte "  |"
texte_en_boite_trait $taille
}
```

Une nouvelle fonction pour tracer nos lignes de tirets, et le premier souci de portabilité des scripts écrits en langage Shell : comment afficher du texte à l'écran ? Nous avons le choix : `echo`, `print`, `printf`. J'ai ici fait un choix arbitraire, je vais utiliser le *builtin* `print` du Ksh de l'OpenBSD 3.8. A vous d'adapter ça à votre shell. Voici le traçage de trait :

```
function texte_en_boite_trait
{
##echo $0 " -> " $1                    # verification du parametre
largeur=$1                              # nom de variable plus clair
limite=$((largeur + 6))                 # marge des deux cotes
print -n -- "+"                         # non portable ?
compteur=0
while [ $compteur -lt $limite ] ; do
        print -n -- "-"                 # non portable ?
        compteur=$((compteur + 1))
done
print    -- "+"                         # non portable ?
}
```

[11] Il exite une foultitude de conventions de ce genre. XXX FIXME TODO.

Voilà finalement notre fonction opérationnelle. En la combinant avec l'utilisation des alias, nous arrivons à ceci, qui démontre bien la versabilité d'un shell :

```
$ alias heure='date +%H:%M:%S'
$ alias h='texte_en_boite $(heure)'
$ h
+--------------+
|   23:08:57   |
+--------------+
```

12.2 Rechercher des infos sur un yuser

Nous avons un `login` : flo, et nous voulons connaitre l'UID de cette jolie dame. Nous allons devoir explorer un fichier bien connu : `/etc/passwd`. Le prototype de notre fonction shell sera dans ce genre : `get_uid_by_login` *`username`*.

Il y a certainement plusieurs façons de faire ça avec le shell, et/ou avec les outils (certains de ces outils n'ont pas encore été vus[12]) que nous propose Unix.

12.2.1 Lecture par le shell

Première approche, la plus immédiate, entièrement en code shell :

```
function get_uid_by_name_0
{
##echo "get_uid_by_name_0 " $# $@       # visualisation des arguments
yuser=$1                                 # login en argument
IFS=':'                                  # sep de champs de /etc/passwd
while read login pw uid gid gecos home shell
do
        if [ $login = $yuser ] ; then    # hop, found !
                return $uid
        fi
done < /etc/passwd
return -- -1
}
```

La variable du shell IFS contient une liste de caractères qui sont utilisés pour découper une chaine en plusieurs mots. En positionnant ce séparateur à la valeur adéquate, le builtin `read` nous découpe notre `/etc/passwd` en 7 parties et renseigne nos 7 variables.

12.2.2 Utilisation de Awk

Un petit essai depuis la ligne de commande nous montre le principe général, mais aussi ce qui se passe quand nous ne trouvons pas notre amie dans `/etc/passwd`. La gestion de ce genre d'erreur

[12] C'est pour ce genre de raisons qu'il est très difficile de faire un cours **structuré** sur les shells d'Unix. Il y a tellement de choses imbriquées que la meilleure façon de faire est d'explorer les mille et une facettes de la chose

est la partie la plus contraignante de la programmation, mais elle est indispensable pour éviter d'être réveillé à 3h du mat' par un yuser en furie.

```
awk -F':' -v user=flo '$1==user { print $4 }' < /etc/passwd
```

Vous noterez le positionnement de la clef de recherche depuis les options de Awk.

```
function get_uid_by_name_1
{
yuser=$1                                    # login en argument
uid=$(awk -F':' -v user=$yuser '
   BEGIN           { val=-1 }
   $1==user        { val=$3 }
   END             { print val }' < /etc/passwd)
return -- $uid
}
```

12.2.3 Utilisation de `grep` et `cut`

```
grep '^flo:' /etc/passwd | cut -d: -f4
```

Nous avons de la chance, le champ à rechercher est en début de ligne, ce qui évite de fouiller dans les options obscures de `grep` pour trouver comment isoler un champ. Un exercice pour vous, chers amis...

12.2.4 Donc...

Donc, on peut faire ça de plusieurs façons différentes, et il n'existe pas vraiment de règles précises pour choisir l'une plus que l'autre. A part peut-être mesurer la rapidité d'exécution, ou l'occupation mémoire, ou la charge subie par les disques, ou la facilité de maintenance, ou...

13 Commandes internes

Pour diverses raisons, certaines commandes ne sont pas traitées par un programme indépendant, mais sont *construites dans* l'interpréteur de commande lui-même. Il est courant de les nommer par l'expression anglaise *built-in*. Nous avons déja vu le positionnement (`cd`) dans l'arborescence des répertoires dont la présence dans le shell est indispensable, pourquoi[13] ?

Ces commandes internes sont classée en deux catégories : les spéciales et les normales. Les commandes internes spéciales diffèrent des normales pour les raisons suivantes :

- Une erreur de syntaxe dans une commande interne spéciale peut causer l'interruption du shell qui l'exécute, ce qui n'arrive pas avec les commandes normales.
- Les assignations de variables faites par une commande interne normale restent en effet après la fin de la commande
- Les redirection d'entrée/sortie sont traitées après les assignations de praramètres.

Et surtout, les commandes internes spéciales sont prioritaires quand le shell cherche à identifier une commande.

Certaines de ces commandes internes ont déja été vues, commme `test` (p. 21) ou `cd`.

En voici quelques autres.

13.1 export

Ce builtin `export` permet de gérer la «diffusion» des variables d'environnement auprès des processus dépendants du shell en cours d'exécution.

13.2 return

Revient d'une fonction ou d'un script sourcé (appel par `.`) et peut renvoyer une valeur donnée. Si aucun valeur n'est donnée, c'est le code de retour de la dernière commande exécutée qui sera retourné.

13.3 read

13.4 set

La commande `set` est utilisée pour plusieurs choses : manipuler les options du shell, donner des valeurs aux paramètres positionnels, paramétrer les tableaux. . .... Ce qui nous donne une syntaxe parfois confuse : `set [+-abCefhkmnpsuvXx] [+-o option] [+-A name] [-] [arg ...]`.

[13] Et c'est loin d'être une petite question.

13.5 trap

Cette commande sert à intercepter les signaux. Son utilisation est détaillée en page 24.

Sans arguments, la commande `trap` affiche une liste de l'état courant des trappes qui ont été positionnées depuis que le shell concerné a été démarré. De part son mécanisme interne, la sortie ne peut être transmise à une autre commande.

```
$ trap
trap -- 'echo user signal one' USR1
$ trap | cat -n
$ trap > toto ; cat -n toto
     1  trap -- 'echo user signal one' USR1
```

13.6 typeset

13.7 wait

Nous avons vu quelque part qu'il est possible de lancer des processus en tache de fond, et de les manipuler par des commandes interactives.

13.8 whence

14 Chercher un fichier : `find` et `locate`

14.1 locate

`locate` permet de retrouver rapidement un fichier par son nom. Pour cela, elle s'appuie sur une base de donnée périodiquement reconstruite par l'intermédiaire du `cron`. Il est donc possible que le fichier que vous recherchez ait disparu depuis la dernière reconstruction. C'est la vie...

```
$ locate boisson
/home/korn/Formation/boisson
$ locate Boisson
$ locate -i Boisson
/home/korn/Formation/boisson
```

L'option `-i` permet d'ignorer les différences entre les majuscules et les minuscules. L'option `-l` *`number`* limite la sortie à *number* lignes. L'ordre dans lequel sont affichés les fichier trouvés n'est pas spécifié.

Une implémentation plus récente, `slocate` règle la plupart des problêmes de sécurité et de confidentialité de la version classique. Il stocke en plus dans sa bédédé, les UID et GID des fichiers, et ne montre aux gens que ce qu'ils sont censés avoir le droit de voir.

En principe, ce mécanisme est géré au niveau global du système, mais il est tout à fait possible de contruire et d'utiliser votre base de données personnelle, si vous avez par exemple une grosse arborescence de travail. Consultez la documentation de `updatedb` pour savoir comment procéder.

14.2 find

La commande `find` permet de parcourir récursivement toute ou partie de l'arborescence des systèmes de fichier, de faire des choix en fonction de ce que l'on voit et d'effectuer des actions sur ce que l'on trouve.

Sa forme générale est `find [options] chemin[s] expression[s]` . Dans un premier temps, nous laisserons les options de coté. Sachez tout de même que les options `-H`, `-L` et `-P` sont en rapport avec les liens symboliques. Certains des caractères utilises dans les expressions étant aussi spéciaux pour le shell, il faudra penser à les protéger.

Une expression est contituée de prédicats primaires, éventuellements regroupés, assemblés par des opérateurs. Nous allons d'abord voir quelques expressions primaires essentielles :

`-name` ***`motif`*** Vrai si le dernier composant du chemin en cours d'examen accroche le motif, construit avec les caractères utilisés par le shell.

`-ls` Toujours vrai. Affiche les méta-données du fichier comme la commande `ls -dgils`.

`-exec` ***`outil`*** **`{} ;`** Vrai si le programme *outil* retourne un code à 0. Des arguments optionnels peuvent être passés à cette commande. L'expression doit être terminée par un **`;`** (point-virgule).

`-nouser` Vrai si le fichier n'appartient à aucun utilisateur connu.

-size *n* Vrai si la taille du fichier correspond. Cette taille est exprimée en blocs, ou en octets si le nombre est suivi d'un `c`.

-type *t* Vrai si le fichier est du type spécifié.

- `b` Périphérique bloc
- `c` Périphérique caractère
- `d` Répertoire
- `f` fichier normal
- `l` Lien symbolique
- `p` Pipe nommé, FIFO
- `s` Socket

-user *nom* Vrai si le fichier appartient a l'utilisateur *nom*. Le nom peut aussi être un UID numérique.

Toutes les fonctions primaires qui utilisent un argument numériques comprennent les nombres précédés d'un signe +, qui signifie *plus que n* ou d'un signe −qui signifie *moins que n*. L'absence du signe veut donc dire *exactement*.

Il est possible de combiner ces diverses fonctionnalités primaires par des opérateurs que voici, listés dans l'ordre des priorités descendantes.

(expr) Vrai si l'expression est vraie.

! expr Vrai si l'expression est fausse. C'est un opérateur unaire.

expr -and expr C'est l'opérateur logique ET. Il est vrai si les deux expressions sont vraies. Il fonctionne en court-circuit, c'est à dire que la seconde expression n'est pas évaluée si la première est vraie.

expr expr Le `-and` peut être aussi implicite. Attention à la lisibilité...

expr -or expr Opérateur logique OU. Aussi en court-circuit.

Tous les opérateurs, les primaires et leurs argument doivent être des arguments séparés pour la commande `find`.

14.2.1 Exemples

Afficher la liste des fichiers dont le nom ne se termine pas par **.c** :

```
$ find / \! -name '*.c' -print
```

Rechercher dans une partie de l'arborescence les fichiers appartenant à Florence et qui sont plus récents que le fichier `/tmp/timestamp` :

```
find /d2a -newer /tmp/timestamp -and -user flo -print
```

Rechercher dans votre répertoire personnel les fichiers dont la taille est supérieure à 12 mégaoctets et essayer de savoir de quoi il s'agit :

```
find $HOME -size +25M -exec file {} \;
```

15 Manipulation des fichiers

En général, une grande majorité des fichiers que l'on est emmené à traiter à partir de la ligne de commande sont des fichiers texte. On doit donc, la plupart du temps, les voir comme des suites ordonnées de lignes, chaque ligne pouvant être découpée en plusieurs champs, séparés de diverses façons.

15.1 Découpages par lignes : `head, tail`

La commande `head` affiche les *n* premières lignes d'un (ou plusieurs) fichier(s). Elle peut s'utiliser de plusieurs façons : `head -42 fichier` ou `head -n 42 fichier`, la seconde syntaxe étant préférable de nos jours. La valeur par défaut pour le nombre de lignes est **10**.

La commande `tail` affiche les *n* dernières lignes d'un fichier. (`tail -42 fichier` ou `tail -n 42 fichier`). Elle permet également de «surveiller» un fichier qu'un autre processus est en train d'écrire, en utilisant l'option `-f`. C'est particulièrement utile pour garder un oeil sur un fichier de log. Nous en verrons un exemple page 56. La valeur par défaut pour le nombre de lignes est **10**.

En combinant ces deux commandes, on peut extraire une ligne particuliere d'un fichier : `head -500 foo | tail -1`.

15.2 Découpage par colonnes : `cut, paste`

Pour sélectionner une partie de chaque ligne, on utilisera `cut`, qui peut découper la ligne de deux manières : en des champs séparés par un caractère ou par des positions de colonnes.

```
$ cut -d: -f1,3,4 < /etc/passwd | tail -3
korn:1137:1137
leon:1002:1002
flo:1003:406
```

Quand à `paste`, il permet de coller les lignes de rang identiques de plusieurs fichiers sur une seule ligne en sortie.

```
$ cat f1
alpha   0
bravo   1
$ cat f2
Alice
Bob
$ paste -d : f1 f2
alpha   0:Alice
bravo   1:Bob
```

Nous verrons plus loin (page 52) un exemple plus complet de l'utilisation de `cut`'n'`paste` sur le traitement des fichiers à champs fixes.

Il existe aussi l'utilitaire `split` qui est destiné à découper un gros fichier en plusieurs fichiers afin d'en faciliter le transport ou le stockage. On pourra ensuite recoller ces fragments avec `cat`.

15.3 Translation de caractères : `tr`

Ce filtre recopie son entrée standard vers sa sortie standard en opérant, au passage, des substitutions ou des suppressions de caractères. Sa forme la plus simple est celle-ci : `tr eEA 334` qui remplacera chaque occurence de la lettre **e**, qu'elle soit en majuscule ou en minuscule, par le chiffre **3** et chaque **A** par un **4**, le w4rl0rdZ1t3ur du pauvre, en quelque sorte. On peut aussi utiliser des intervalles de caractères : `tr a-z A-Z` passera toutes les lettres en majuscules.

L'autre possibilité de `tr` est la suppression de caractères. Un des usages courant est la conversion d'un fichier d'origine MSDOS/VMS/CPM, où les fins de lignes sont codées par les deux caractères <CR><LF>, en fichier Unix qui n'utilise que <LF>, aussi connu sous le nom de newline.

```
$ od -c fichier.dos
00000  u   n       f   i   c   h   i   e   r       v   e   n   a   n
00020  t  \r  \n   d   e       l   '   a   u   t   r   e       m   o
00040  n   d   e  \r  \n
$ tr -d '\r' < fichier.dos | od -c
00000  u   n       f   i   c   h   i   e   r       v   e   n   a   n
00020  t  \n   d   e       l   '   a   u   t   r   e       m   o   n
00040  d   e  \n
```

L'opération inverse n'est pas possible avec `tr`. Il faut attendre d'être arrivé en page 41 pour découvrir une des façons possibles de rajouter ce fameux <CR> avec Sed.

16 Go Regular Expression, Print

Avec les outils de la famille des `grep`s, vous allez pouvoir retrouver des choses dans des machins. Ce nom, `grep`, vient des commandes d'une fonction d'un antique éditeur de texte.

16.1 `grep`

L'outil `grep` cherche dans les fichiers ou les données qu'on lui fournit et sélectionne ou capture les lignes qui correspondent à un «patron»[14] ou «motif» donné. Ce motif est une expression régulière. Rapidement, une expression régulière est une chaine dans laquelle certains caractères ont une signification spéciale : par exemple, un point `.` désigne pratiquement n'importe quel caractère.

16.2 `egrep`

Le premier cousin, `egrep`, peut utiliser les expressions régulières étendues (détaillées en page 40).

[14] Le terme anglais d'origine, *pattern*, vient de l'ancien français, au sens de "patron de couture".

16.3 `fgrep`

Le second cousin, `fgrep`, ne connait pas les expressions régulières, il ne connait que les *patterns* fixes. En contrepartie, il est prétendu plus rapide.

16.4 On a grépé...

Chacun de ces outils nous retournera une valeur que nous pourrons exploiter éventuellement dans un script. **0** signifie qu'au moins une ligne a été trouvée, **1** nous dit qu'aucune ligne n'a été trouvé, et une valeur supérieure affirme qu'il y a eu une erreur. On peut supprimer la sortie des lignes trouvées avec l'option `-q` si on n'est interessé que par le code de retour.

17 Les expressions "régulières"

Attention, nous allons aborder un domaine à la fois flou et complexe. Certaines des commandes que nous avons vues précedemment peuvent utiliser ces fameuses expressions régulières, les connaitre permet donc d'augmenter les capacités de ces commandes.

Il existe deux grandes catégories d'expression régulières : Les basiques `BRE` et les étendues `ERE`.

17.1 BRE

Basic Regular Expressions.

17.2 ERE

Extended Regular Expressions.

17.3 Remarques sur les regexp

Le langage Perl (cf page 64) a rajouté ses propres extensions/modifications aux regexps que nous venons de voir. Il est même possible que ces extensions arrivent un jour dans les outils Unix traditionnels.

Si vous voulez savoir jusqu'où peut aller une expression régulière, celle qui sert à valider une adresse e-mail, en tenant compte de toutes les possibilitées, fait environ 2000 caractères, et tient avec peine sur une seule page du livre [15] consacré aux regexp.

*Dans la suite de ce document, je me permettrais d'utiliser l'anglicisme **regexp** pour désigner une forme quelconque d'expression régulière.*

17.4 PCRE

[15] à chercher chez O'Reilly.

18 Modifier un fichier : `sed`

Sed est un "éditeur de flux", en anglais *stream editor*. C'est un outil qui n'est pas interactif. Il se comporte comme un filtre, Il se contente de transformer ce qu'il voit passer. Nous avons déja vu, avec `tr`, ce genre de manipulations. Sed permet de passer à la vitesse supérieure.

La forme générale d'une commande d'édition est *[adresse1,adresse2] fonction [paramètres]*. Une ou deux adresses peuvent être omises. La fonction doit être présente. Les paramètres peuvent être requis ou optionnels, selon la fonction.

18.1 les bases

Pour les quelques exemples qui arrivent, nous allons utiliser ce petit texte de cinq lignes en fichier d'entrée :

> Philip K. Dick a laissé une oeuvre considérable, qui a
> profondément marqué toute une génération d'auteurs et de
> lecteurs. Après Minority Report, Blade Runner, Total
> Recall, il livre avec Substance Mort son oeuvre la plus
> personnelle, la plus aboutie.

Une option importante à connaitre dès maintenant : `-n` empèchera Sed d'afficher une ligne après les divers traitement qui lui ont été appliqués. Les lignes de commandes `sed ' ' /etc/passwd` et `sed -n 'p' /etc/passwd` sont donc équivalentes.

18.2 Adresses : choisir les lignes à éditer

Tout d'abord, une ligne d'entrée peut être désignée par son numero. La première ligne traitée étant numéroté **1**. Ce compteur n'est pas remis à zéro si on traite plusieurs fichiers. On peut également utiliser un intervalle de lignes, deux numéros séparés par une virgule : `2,4` sélectionnera les lignes 2, 3, et 4. Le caractère `$` désigne la dernière ligne du dernier fichier d'entrée.

```
$ cat -n petit_texte  | sed  '2,4d'
     1  Philip K. Dick a laissé une oeuvre considérable, qui a
     5  personnelle, la plus aboutie.
```

Ensuite une adresse peut être trouvée par un motif (expression régulière) encadrée par une paire de / slashes. On nomme parfois cela une *adresse contextuelle*. Les expressions régulières reconnues par Sed sont contruites comme ceci :

1. Un caractère ordinaire (qui n'est pas un de ceux que nous allons voir) accroche ce caractère.
2. Un circonflexe `^` au début d'une regexp accroche le pseudo-null au début de la ligne.
3. Un dollard `$` à la fin d'une regexp accroche le pseudo-null en fin de ligne.
4. Les caractères `\n` accroche un newline dans l'espace de motif, sauf celui de la fin.
5. Un point `.` accroche n'importe quel caractère, sauf le newline finale.

6. Une regexp suivie d'une asterisque `*` accroche zéro, une ou plusieurs occurences adjacentes de la regexp qu'elle suit.
7. Une chaine encadrée par les crochets carrés `[ ]` accroche un des caractères de la chaine, et aucun autre. Sauf si le premier caractère de cette chaine est la carotte, `^`, auquel cas, les choses sont inversées.
8. *La concaténation de regexps est une regexp qui accroche la concaténation des chaines accrochées par les composants de la regexp.*[16]
9. Une regexp entre les séquences `\(` et `\)` est équivalente à une regexp normale, mais a des effets de bord décrits à la commande `s`.
10. L'expression `\`*d* représente la même chaine que celle accrochée par l'expression entre `\(` et `\)` précedemment dans le même pattern. Ici, *d* est un simple chiffre ; la chaine spécifiée est celle qui commence par la *d*ième occurence de `\(`.
11. La regexp nulle toute seule `//` est équivalent à la dernière regexp compilée.

Pour utiliser un des caractères spéciaux d'une façon littérale, (c'est à dire pour accrocher une occurence de lui-même dans le flus d'entrée) il faut le faire préceder immédiatement par un `\` backslash.

Si une commande n'a pas d'adresses, elle s'applique à toutes les lignes.

18.3 Fonctions disponibles

Voici une liste exhaustive des fonctions à notre disposition. Avec un description un peu succinte, mais que nous allons développer ultérieurement par des exemples.

`d` Suppression de lignes.
Elle ne sont pas écrites vers la sortie, la nouvelle ligne est lue depuis l'entrée et on reprend la liste de commandes au début.

`n` Ligne suivante.
Lecture de la ligne suivante, qui va remplacer la ligne courante.

`a\ texte` Ajouter des lignes.
Cette fonction ajoute son argument texte dans la sortie **après** la ligne accrochée par l'adresse. Le texte lui-même doit avoir ses lignes terminées par un \, sauf la dernière. Les blancs en tête des lignes du texte sont supprimés, sauf si le premier blanc est précédé d'un \.

`i\ texte` Insérer des lignes.
Même chose que la commande `a` excepté que le texte est ajouté **avant** la ligne.

`c\ texte` Changer des lignes.
Les lignes sélectionnées sont effacées et remplacées le texte. Cette commande peut avoir deux adresses, et sélectionne donc un ensemble (textslrange) de lignes.

[16] Moi-même, j'avoue avoir du mal à comprendre, monsieur MacMahon est parfois dur à suivre.

s Substitutions.

Cette fonction, qui s'écrit sous la forme `s+motif+remplace+flag`, remplace une partie (sélectionnée par le motif) de la ligne, par la chaine 'remplace'. Le motif est identique à ceux utilisés pour les adresses, sauf que le `/` peut être remplacé par un autre caractère non-blanc, un + dans notre exemple. La chaine de remplacement commence juste après le second délimiteur du motif, et se termine par ce même délimiteur. Par défaut, seule la première occurence trouvée est remplacée.

La partie 'remplace' n'est pas un motif, et les caractères spéciaux des motifs n'y ont pas de signification particulière. A la place, deux autres caractères sont spécialisés : & est remplacé par la chaine accrochée par le motif, et `\`*d* est remplacé par la *d*ième sous-chaine accrochée par la partie du motif encadrée par `\(` et `\)`.

Quand aux flags, nous avons `g` qui remplace **toutes** les occurences trouvées, `p` qui imprime la ligne si une substitution a eu lieu, et `w` *`filename`* écrira la ligne dans le fichier *filename* si une substitution a eu lieu.

p Printation.

Ecrit la ligne accrochée sur la sortie, peut fonctionner en plein accord avec l'option `-n`.

w Ecriture dans un fichier.

r Lecture d'un fichier.

N Ajout ligne suivante.

La ligne d'entrée suivante est ajoutée à la ligne courante dans l'espace de travail. Ces deux lignes sont séparées par un newline intermédiaire.

D Effacement du début.

P Impression première partie.

h Attrape l'espace.

Recopie l'espace de travail (la ligne en cours de traitement) dans la zone de mémoire, dont le contenu précédent est perdu.

H Ajoute l'espace.

Ajoute le contenu de l'espace de travail à la fin de la zone de mémoire, en séparant les deux parties par un newline intermédiaire.

g Lit la mémoire.

Recopie la zone de mémoire dans l'espace de travail, dont le contenu disparait.

G Lit la mémoire.

Cette fonction ajoute la zone de mémoire à l'espace de travail, avec l'habituel newline de séparation.

x Echange l'espace de travail et la zone de mémoire.

! Ne fait pas !

{ Groupage.

: Etiquette.

b Branchement.

t Test substitution.

= Egalité.

Cette fonction écrit sur la sortie standard le numéro de la ligne accrochée par l'adresse contextuelle.

q Quitter. Cette instruction est exécuté immédiatement.

18.4 Quelques exemples

18.4.1 Encodage de caractères

En HTML, certains caractères ne sont pas affichable directement, il faut les représenter par des entitées spécifiques. Il faut convertir < en `<`, > en `>`, & en `&`, et ainsi de suite...

```
#!/usr/bin/sed -f
s/&/\&amp;/g
s/"/\&quot;/g
s/</\&lt;/g
s/>/\&gt;/g
```

Notez l'ordre dans lequel sont faites les substitutions et l'utilisation de l'option **g** afin de traiter plusieurs occurences d'un même caractère sur une ligne.

18.4.2 Reformatage de lignes

Un outil fournit par notre installation d'Unix peut nous donner des informations utiles sur nos fichiers, mais son format de sortie ne nous plait pas. Nous allons essayer de les remettre dans la bonne forme. Voyons ce que nous avons et ce que nous voudrions avoir :

```
$ md5 ekk*
MD5 (ekko) = 44cf99df0aef9740ce451d65266ff1df
MD5 (ekko.c) = 826c4de1e713df48c2fd26a41a399aa8
$ md5 ekk* | sed -f reformate.sed
44cf99df0aef9740ce451d65266ff1df ekko
826c4de1e713df48c2fd26a41a399aa8 ekko.c
```

Si nous décomposons le problème, nous pouvons arriver à quatre opération élémentaires, qui seront des substitutions ou équivalent[17] :

1. Suppression de la chaine MD5
2. Suppression des parenthèses
3. Suppression du signe =
4. Inversion des deux items restants

Dans une première approche, nous pouvons coller à l'analyse, et enchainer quatre instructions Sed qui remplissent les quatres tâches sorties de l'analyse :

[17] Après tout, supprimer, c'est remplacer par rien.

```
s/^MD5 *//
s/(\(.*\))/\1/
s/ *= */ /
s/\([^ ]*\) \([^ ]*\)/\2 \1/
```

Gardez cet exemple sous le coude, Sauvez le dans un fichier `reformat-md5.sed` nous allons nous en servir plus tard.

18.5 Un peu de magie ?

Sed, de part son langage cryptique et ses étranges possibilitées, est considéré par certains comme un outil quasiment artistique.

```
sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//'
sed -e :a -e '$q;N;11,$D;ba'
sed -n '/^.\{65\}/p'
ed 'N;$!P;$!D;$d'
sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d
```

Tout cela donne à réfléchir...

19 Exploiter un fichier : `Awk`

Awk est un langage de programmation à la fois puissant et méconnu. Sa fonction principale est d'itérer le même traitement sur chacune des lignes d'un fichier texte, avec une éventuelle sélection des lignes à prendre en considération.

Il découpe chaque ligne qu'il lit, en utilisant comme séparateur un caractère ou une expression régulière (voir page 40). Par défaut, c'est une suite de *blanks* qui est utilisée. Il positionne également quelques variables décrivant la ligne en question. Voyons tout de suite un petit exemple…

```
$ head -n 4 /etc/passwd
root:*:0:0:Charlie Root:/root:/bin/ksh
daemon:*:1:1:The devil himself:/root:/sbin/nologin
operator:*:2:5:System Operator:/operator:/sbin/nologin
bin:*:3:7:Binaries Commands and Source,,,:/:/sbin/nologin

$ awk -F: '{print length, NF, $1, "=", $5}' /etc/passwd
38 7 root = Charlie Root
50 7 daemon = The devil himself
54 7 operator = System Operator
57 7 bin = Binaries Commands and Source,,,
```

… Et les explications : l'option **-F :** demande l'utilisation du caractère **:** comme séparateur de champs. Ensuite, à l'interieur du bloc délimité par **{** et **}** (qui est exécuté pour chacune des lignes), nous affichons la taille de cette ligne, le nombre de champs à l'issue du découpage, le premier champ, le caractère = et finalement le cinquième élément. Le retour à la ligne est implicite dans l'instruction `print`. Je pense maintenant que vous saisissez bien l'intérêt de la chose.

19.1 Faire un script

```
#!/usr/bin/awk -f
BEGIN           {
                FS=":"
                nombre = 0
                }
$3 >= 1000      {
                printf "%-6d   %-10s = %s\n", $3, $1, $5
                nombre ++
                }
END             {
                print "on a vu passer", nombre, "yusers"
                }
```

Nous venons d'utiliser Awk directement depuis la ligne de commande. Il est également possible d'écrire un script pour pouvoir répéter *ad nauseum* ces quelques opérations inutiles. Au passage, nous allons modifier notre affichage et rajouter quelques traitements.

```
#!/usr/bin/awk -f
BEGIN           {
                FS=":"
                nombre = 0
                }
$3 >= 1000      {
                printf "%-6d   %-10s = %s\n", $3, $1, $5
                nombre ++
                }
END             {
                print "on a vu passer", nombre, "yusers"
                }
```

Nous découvrons ici plus précisément la structure d'une programme Awk de base. La première ligne est classiquement l'appel de l'interpréteur par le mécanisme du shebang. Il faudra donc penser à rendre ce script exécutable : `chmod u+x lstyusers.awk`.

Ensuite, nous découvrons le bloc BEGIN : il est exécuté **avant** la lecture de la première ligne du fichier. Nous l'utilisons ici pour initialiser le séparateur de champ (la variable prédéfinie **FS**) et un compteur [18].

Le deuxième bloc est conditionnel. Le troisième champ du fichier `/etc/passwd` contient l'UID'indexUID de l'utilisateur. Par convention, dans l'Unix[19] que j'utilise actuellement, les UID à partir de 1000 sont pour les utilisateurs lambdas.

Quand au troisième bloc, END, il est exécuté après la lecture de la dernière ligne du (ou des) fichier(s) en entrée. Nous en profitons pour afficher notre compteur. Et tout cela nous donne enfin ceci :

```
$ ./lstyusers.awk < /etc/passwd
 32767   nobody     = Unprivileged user
  1000   toto       = toto
  4231   tth        = Tonton Th
  1001   nicolas    = Tetard
  1137   korn       = David Korn
  1002   leon       = Petit Oiseau
  1003   flo        = Florence D.
on a vu passer 7 yusers
```

Nous avons également amélioré la présentation en formatant les lignes en sortie grâce à l'instruction `printf` de Awk. `%6d` définit un champ numérique entier de 6 caractères, et `%-10s` un champ texte de 10 caractères cadré à gauche.

19.2 Les variables de Awk

Il existe deux classes de variables dans Awk : les variables scalaires, qui contiennent une seule valeur, texte ou numérique flottant, et les tableaux associatifs que nous verrons un peu plus loin. Une variable n'a pas besoin d'être déclarée à l'avance.

[18] Bien que cela ne soit pas indispensable, à sa première utilisation, une variable est initialisée à une valeur nulle.

[19] OpenBSD 3.8

Awk propose un certain nombre de variables pré-définies qui sont soit initialisées à une valeur par défaut, ou mises à jour à la lecture de chaque enregistrement. Certaines de ces variables sont modifiable en cours d'exécution pour agir sur le comportement du script.

`FILENAME`	Nom du fichier d'entrée courant.
`FNR`	Numero d'ordre de l'enregistrement courant dans le fichier courant.
`FS`	Séparateur de champ en entrée, c'est une expression régulière.
`NF`	Nombre de champs dans l'enregistrement courant.
`NR`	Numero d'ordre de l'enregistrement courant.
`OFS`	Séparateur de champs en sortie.
`ORS`	Séparateur d'enregistrement en sortie.
`RS`	Séparateur d'enregistrement en entrée.

Par défaut le séparateur de champs en entrée, `FS`, contient une espace, qui est dans ce cas une valeur particulière puisqu'en fait il veut dire un ou plusieurs caractères blanc ou tabulation. Cette valeur peut, avec l'option `-F`, être modifiée au lancement de Awk.

19.3 Tableaux associatifs

Ce sont ces tableaux "associatifs" qui font partie des énormes avantages de Awk. Ils ont d'ailleurs été repris par beaucoup de langages de script modernes ; et parfois connus sous le nom de hash.

Pour illustrer rapidement cette puissance de ces tableaux, nous allons entamer la chasse aux clônes. Nous avons vu dans le chapitre sur Sed un outil pour obtenir une liste de couples signature/nom de fichier. Sachant que si deux fichiers ont la même signature, ils sont certainement[20] identiques.

19.4 Tableaux pré-définis

Awk propose également certains tableaux associatif pré-définis, analogues aux quelques variables telles que `FS` ou `ORS` vue précedemment :

```
$ awk 'BEGIN { print ENVIRON["HOSTNAME"] }'
puce.none.invalid
$ foo="grumble" awk 'BEGIN { print ENVIRON["foo"] }'
grumble
```

19.5 Passer des paramètres à un programme Awk

Il existe deux méthodes pour passer un (ou plusieurs) paramètre à un script Awk. La première est l'affectation de valeurs à des variables depuis la ligne de commande. La seconde est l'exploration des arguments de cette ligne depuis le script.

La première méthode :

```
$ awk -v rep=42 -v OFS=" -> " 'BEGIN { print "reponse", rep }'
reponse -> 42
```

[20] Enfin, presque, tout dépend de la méthode de calcul de cette signature, et des risques de collision.

Cette méthode est la seule utilisable avec certaines vieilles versions de Awk, qui ne savent pas explorer les arguments de la ligne de commande. Voyons comment se débrouille un Awk plus récent :

```
$ awk 'BEGIN {print ARGC, ARGV[2]}' foo bar quux
4 bar
```

Bien entendu, ces deux méthodes peuvent être combinées, chacune présentant des avantages et des inconvénients. La première permet d'initialiser des choses comme `OFS`, la seconde permet d'utiliser les expansions du shell.

19.6 Awk propose des fonctions

19.6.1 Fonction mathématiques

atan2, cos, exp, int, log, rand, sin, sqrt, srand. Ils ne sont pas tous présents, mais il y a le minimum vital.

19.6.2 Fonction sur les chaines

`gsub(r, t, s)` et `sub(r, t, t)` `sub` remplace la première occurence de la regexp `r` par `t` dans la chaine `s`. Si `s` n'est pas donnée, c'est l'enregistrement en entier, `$0` qui sera traité. Un `&` dans `t` est remplacé dans `s` par la regexp `r`. `gsub` fait la même chose, à part que toutes les occurences de la regexp sont remplacées. Ces deux fonctions retrournent le nombre de substitutions opérées.

`index(s, t)` Donne la position dans `s` où on trouve la chaine `t`, ou 0 si on ne la trouve pas.

`length(s)` Taille de la chaine `s`, ou celle de `$0` si il n'y a pas d'argument.

`match(s, r)` La position dans `s` où se trouve l'occurence de la regexp `r`, ou 0 si on ne la trouve pas. La variable `RSTART` est renseignée avec ce résultat. La variable `RLENGTH` est renseignée avec la taille de la chaine capturée.

`split(s, a, fs)` Découpage de la chaine `s` dans les éléments du tableau `a[n]` et retourne le nombre de champs trouvés. La séparation est faite avec la regexp `fs` ou le séparateur `FS` si l'argument est absent. Une chaine vide comme séparateur découpera caractère par caractère.

`sprintf(fmt, expr, ...)` Retourne la chaine obtenue en formatant les paramètres `expr, ...` par la chaine `fmt`.

`tolower(s)` et `toupper(s)` Conversion en minuscules/ majuscules.

19.6.3 Fonction d'entrée/sortie

`close(expr)` Fermeture du fichier ou du tuyau désigné par la chaine `expr`.

`cmd|getline`

`fflush(expr)` Pousse les données en attente pour le fichier ou le tuyau désigné par la chaine `expr`.

`getline` Rempli `$0` avec le prochain enregistrement tiré du fichier d'entrée courant, puis positionne les variables `NF`, `NR` et `FNR`.

`getline var` Rempli `var` avec le contenu de `$0`, puis positionne les variables `NF`, `NR`, `FNR`.

`getline var < file` Lecture d'une variable depuis un fichier. L'ouverture est automatique, le fichier désigné par `file` restera ouvert tant qu'on ne fera pas appel à `close(expr)` ;

`system(cmd)` Exécute la commande `cmd` et renvoie son code de retour.

19.7 Ecrire ses fonctions

On peut définir ses propres fonctions dans un script Awk, à n'importe quel endroit du source. La forme générale de cette définition est :
`function foo(a, b, c) { ...; return x }`
Les paramètres sont passés par valeur si ils sont scalaires, et les tableaux sont passés par référence.

Les paramètres sont privés à la fonction et toutes les autres variables sont globales, donc vues de l'ensemble du programme.

19.7.1 Obtenir des variables locales

Ceux d'entre vous qui connaissent un peu la programmation savent qu'il y a une différence entre une variable locale et une variable globale. Si vous avez besoin, dans une fonction, d'une variable pour stocker un résultat intermédiaire, vous n'avez pas vraiment envie qu'une autre fonction vienne modifier le contenu de cette variable dans votre dos.

Awk ne propose pas de façon de déclarer une variable qui doit être locale. A la place, il propose un ***hack*** que je trouve assez élégant. Il ne contrôle pas, lors de l'appel d'une fonction, le nombre de paramètres.

Si une fonction est déclarée avec trois paramètres, et que l'appelant ne passe que deux arguments, la fonction va se retrouver avec un paramètre inutilisé dont elle peut faire ce qu'elle veut, c'est à dire une variable locale.

19.8 Exemples pratiques

Le premier exemple que nous allons voir utilisera à la fois les tableaux associatifs, et un morceau de Sed (p. 44) qui reformate pour l'affichage les signatures md5 d'un fichier. Il permet de détecter deux fichiers identiques.

```
#!/usr/bin/awk -f
        {
        if ( $1 in md5s )
                print md5s[$1], " = ", $2, "?"
        else
                md5s[$1] = $2
        }
```

Nous avons utilisé un tableau associatif dont l'index est la somme MD5, et les valeurs le nom du fichier. Avec le test `$1 in md5s` nous pouvons vérifier si une signature a déja été vue, et connaire le nom du fichier auquel elle était associé. Voici l'exemple à l'oeuvre :

```
$ md5 t?t?
MD5 (titi) = d55bedd19c2e7eb24595e8e6fe074088
MD5 (toto) = fc67638d056c2960ef48dae131b115b1
MD5 (tutu) = fc67638d056c2960ef48dae131b115b1
$ md5 t?t? | sed -f ./reformat-md5.sed | ./duplicates.awk
toto  =  tutu ?
```

Deuxième exemple. Nous savons que dans notre fichier l'information que nous cherchons est dans les lignes qui suivent immédiatement les lignes qui commencent par `abc`. Le motif sera donc la regexp `^abc` et l'action «lit la ligne suivante et affiche-la».

```
$ awk '/^abc/ { getline; print }' < mes_donnees
```

20 Traiter des données en champs fixes

Par `champs fixes`, j'entends des fichiers dont les divers champs ne sont pas séparés par un caractère particulier, mais ont tous une taille fixée. Pour les besoins de la démonstration, nous allons voir rapidement comment en fabriquer un à partir du fichier passwd, avec un premier chanp de 10 caractères pour le username, et deux champs de 5 caractères pour les UID et GID :

```
awk -F: '$3>1000 { printf "%-10s%5d%5d\n", $1, $3, $4 }'\
< /etc/passwd > uidgid
$ cat uidgid
nobody    3276732767
tth       0423104231
nicolas   0100101001
korn      0113701137
leon      0100201002
flo       0100300406
```

De la même façon, nous pouvons obtenir un fichier avec les champs "user", "home" et "shell".

```
< /etc/passwd > homeshell       \
        awk -F: '$3>1000 { printf "%-10s%-20s%-20s\n", $1, $6, $7 }'
```

Cette étape préparatoire terminée, nous disposons maintenant de deux fichiers à champs fixe et ayant une clef commune : le login de l'utilisateur.

20.1 Découpage et recollage des champs

Nous avons brièvement parlé en page 38 de l'outil `cut` qui permet de découper les lignes d'un fichier. Nous pouvons le voir à l'oeuvre maintenant, sur des fichiers à champs de taille fixe :

```
$ cut -c1-10,16-20 < uidgid | head -n 1
nobody    32767
```

Le résultat que nous voulons en finale est un fichier contenant le nom de l'utilisateur, son GID et l'emplacement de son répertoire personnel. Ces champs doivent être séparés par un caractère donné.

Nous allons procéder par étapes. Tout d'abord, à partir du premier fichier (`uidgid`), nous allons extraire les deux colonnes "user" et "gid", puis la commande `paste` va les réunir dans un nouveau fichier, en utilisant notre séparateur. Au passage, nous en profitons pour trier sur la clef primaire, car nous aurons besoin d'enregistrements ordonnés dans l'étape suivante. Et nous refaisons une opération analogue sur les champs "user" et "home".

```
cut -c1-10      < uidgid        > $tmpuser
cut -c16-20     < uidgid        > $tmpgid
paste -d $separateur $tmpuser $tmpgid   | sort > u-gid
cut -c1-10      < homeshell     > $tmpuser
cut -c11-30     < homeshell     > $tmphome
paste -d $separateur $tmpuser $tmphome  | sort > u-home
```

20.2 Agrégation finale

Et finalement, grâce à l'opérateur relationnel `join`, nous pouvons tenter de réunir nos deux tables. La clef primaire étant le premier champ dans nos deux fichiers `u-gid` et `u-home`, et aussi le comportement par défaut de `join`, nous n'avons qu'à préciser le séparateur dans la commande.

```
join -t $separateur u-gid u-home > resultat
```

Bingo !

Nous avont obtenu le résultat demandé. Nos trois champs sont là, avec le séparateur correct.

```
$ cat resultat
flo       |00406|/home/flo
korn      |01137|/home/korn
leon      |01002|/home/leon
nicolas   |01001|/home/nicolas
nobody    |32767|/nonexistent
toto      |01000|/home/toto
tth       |04231|/home/tth
```

20.3 Conversion en CSV

Maintenant, pour transférer ces données vers une application bureautique quelconque, il est nécessaire de les convertir vers le format CSV[21] Puisque nous avons les données sous une forme très classique, un traitement avec Awk devrait faire l'affaire. Une première approche rapide est presque satisfaisante :

```
$ awk -F'|' '{printf "\"%s\",%d,\"%s\"\n",$1,$2,$3}' < resultat \
                                                  head -n 1
"flo       ", 406, "/home/flo           "
```

Presque correct, à par les espaces superflus qui restent en fin des chaines de caractères. La fonction `sub(r, t, s)` de Awk semble faite pour ça. Notre expression régulière sera `/ *$/`, qui capture les espaces en fin de chaine et le remplacement sera `""`, une chaine vide. Nous sommes donc arrivés au script Awk suivant :

```
$ cat to_csv.awk
#!/usr/bin/awk -f
BEGIN   {               # initialisations
        FS="|"          # separateur de champs
        }
        {               # pour chaque ligne...
        foo = sub(/ *$/, "", $1)
        foo = sub(/ *$/, "", $3)
        printf "\"%s\",%d,\"%s\"\n", $1, $2, $3
        }
```

[21] Comma separated values.

```
$ ./to_csv.awk < resultat
"flo",406,"/home/flo"
"korn",1137,"/home/korn"
"leon",1002,"/home/leon"
"nicolas",1001,"/home/nicolas"
"nobody",32767,"/nonexistent"
"toto",1000,"/home/toto"
"tth",4231,"/home/tth"
```

Voilà, c'est fini, l'usinagaz peut partir en production, et nous allons passer à la pause café. Je laisse aux lecteurs le soin de traiter la suppression de ces espaces superflus à l'aide de Sed.

20.4 L'intégrale du script v1

```
 1  #!/bin/ksh
 2
 3  # === declaration de constantes ===
 4  separateur='|'
 5
 6  # === declaration des fichiers temporaires ===
 7  tmpuser=/tmp/user.$$
 8  tmpgid=/tmp/gid.$$
 9  tmphome=/tmp/home.$$
10
11  # === preparation de la sortie du script ===
12  trap 'rm -f $tmpuser $tmpgid $tmphome'          EXIT
13
14  # === creation du fichier user/gid ===
15  # --- extraction des deux champs, chacun dans son fichier
16  cut -c1-10      < uidgid        > $tmpuser
17  cut -c16-20     < uidgid        > $tmpgid
18  # --- reunion de ces deux champs dans un fichier a separateur
19  paste -d $separateur $tmpuser $tmpgid   | sort > u-gid
20
21  # === creation du fichier user/home ===
22  cut -c1-10      < homeshell     > $tmpuser
23  cut -c11-30     < homeshell     > $tmphome
24  paste -d $separateur $tmpuser $tmphome  | sort > u-home
25
26  # === jointure entre nos deux fichiers ===
27  join -t $separateur u-gid u-home > resultat
```

Quand il sera à peu près figé, il faudra que je fasse quelques commentaires sur les construction utilisées...

Quand aux performances, les chiffres sont assez étonnants. Sur une machine très rudimentaire (PII à 133 MHz) et sur des fichiers d'entrée de 32000 lignes, nous obtenons ceci :

```
$ time ./execute_80.ksh ; time ./to_csv.awk < resultat > csv
```

```
0m11.40s real     0m5.15s user     0m1.22s system
0m27.50s real     0m26.85s user     0m0.21s system
```

20.5 Quelques idées

La fonction `split(s, a, "")` du langage Awk permet fort probablement de simuler un découpage par champs fixes d'une ligne. Je vous propose de voir cela comme un exercice à résoudre par vous-même.

20.6 Je n'y arrive pas...

Et dans les cas vraiment désespérés, il reste toujours la possibilité de passer à un langage plus évolué que l'équipe sed/awk/sh. Vous pouvez vous tourner vers un autre langage de script, comme Perl ou Python, ou un langage compilé, comme le C.

21 Astuces diverses

Nous sommes arrivés à un certain niveau. Nous allons maintenant essayer de voir comment tirer encore plus de ce que nous avons déja vu. Certaines commandes, en apparence anodines, sont en fait pleines de possibilitées. En général, il faut prendre le temps de lire leurs pages de man, pour découvrir ces choses là.

21.1 `cat` et assimilés

Pour numéroter les lignes d'un texte, il faut utiliser l'option `-n` de cat [22]. A titre d'exercice, vous pouvez tenter de reproduire ce comportement avec Awk.

L'option `-v` permet de rendre «visible» certains caractères nom-imprimables, et `-t` rend visible sous la forme `^I` les caractères de tabulation.

```
$ cat control_chars
une     tabulation, une cloche, u retour arriere.
$ cat -v control_chars
une     tabulation, une^G cloche, un^H retour arriere.
$ cat -t control_chars
une^Itabulation, une^G cloche, un^H retour arriere.
$ od -c control_chars
0000000    u   n   e  \t   t   a   b   u   l   a   t   i   o   n   ,
0000020    u   n   e 007       c   l   o   c   h   e   ,       u   n  \b
0000040        r   e   t   o   u   r       a   r   r   i   e   r   e   .
0000060   \n
```

Sur certains systèmes, deux commandes, `vis` et `unvis` propose une variante nettement plus avancée.

La commande `pr` permet un rangement en plusieurs colonnes de petites lignes de fichier, avec cette syntaxe : `pr -t -`*`n`* `fichier`, *n* étant le nombre de colonnes souhaité. Cette commande a bien d'autres possibilités de mise en forme, consultez la page de man pour en savoir plus.

Quand nous avons vu la commande `tail`, j'ai brièvement parlé de l'option `-f` de surveillance. Il est temps de fournir un exemple. Selon que vous êtes en mode console ou dans X11, ce sera un peu différent, mais le principe est le même. Nous commençons par créer un script qui va diffuser lentement une information.

```
$ cat slowprint.ksh
#!/bin/ksh
while :
do
        date ; sleep 5
done
```

Vous pouvez l'essayer, en général, il fonctionne et vous affiche l'heure toutes les cinq secondes. Maintenant lancez-le en tâche de fond si vous êtes en console, ou dans un `xterm` si vous avez X11.

[22] A propos de `cat`, n'oubliez jamais de faire attention aux **UUOC** qui feront toujours bien rire vos petits camarades...

```
$ ./slowprint.ksh > regardez_moi &
[1] 22698
$ ls -rtl | tail -1
-rw-r--r--  1 korn  korn     60 Apr 12 12:14 regardez_moi
$ sleep 133
$ ls -rtl | tail -1
-rw-r--r--  1 korn  korn    870 Apr 12 12:17 regardez_moi
```

Quelques `ls -rtl` plus tard, vous remarquez que le fichier «regardez_moi» grossit régulièrement, il y a probablement des choses qui sont écrites dedans. Invoquez donc l'outil de surveillance par l'incantation : `tail -f regardez_moi` , attendez quelques secondes, et voilàhà.

Oui, mais non. Tout cela fonctionne, c'est bien beau, mais en fait, je n'ai pas vraiment besoin de cet énorme fichier qui ne contient que des dates et des heures. Comment s'en passer, de ce monstre ? En bref, est-il possible de transférer des informations d'un processus vers un autre sans utliser les `|` ?

21.2 Les FIFO

FIFO est un anglo-acronyme qui signifie First-In, First-Out, premier entré, premier sorti. Pratiquement, c'est une file d'attente. Sans que vous ne le sachiez, nous avons déja vu ce genre de chose. Rappelez vous la page 14, où nous parlions des redirections : C'est effectivement un mécanisme de FIFO qui sert de «tuyau», de «connecteur» entre deux processus de la chaine.

Et bien, ce mécanisme là est à notre disposition, sous le nom de *tube nommé*, et nous permet de refaire l'expérience "toc"[23] de la lecture de l'heure, en éliminant le fichier grossissant.

```
$ mkfifo mon_tuyau
$ ls -l mon_tuyau
prw-r--r--  1 korn  korn  0 Apr 12 13:39 mon_tuyau
```

Comme l'endroit où se trouvent nos données n'est plus un fichier régulier (notez le **p** au début de la ligne du ls, qui signifie pipe) la commande `tail` n'a pas la possibilité d'en détecter les changements. Par contre, un simple `cat` fera très bien l'affaire.

21.3 `ls, stat`

Pour savoir sur quoi vous avez travaillé récemment : `ls -rtl`. Dans cette option, le `t` demande un tri sur la date de dernière modification, le `r` demande un tri en ordre inverse, et le `l` devrait déja être connu. Bien entendu, si vous utilisez souvent cette commande, pensez à `alias lrtl='ls -rtl'` dans votre point-profile.

Avec `-F`, le nom du fichier sera éventuellement suivi d'un caractère décrivant son type : `*` pour un fichier exécutable, `/` pour un répertoire, `|` pour un tube nommé et `@` pour un lien symbolique.

[23] Trouble obsessionnel compulsif

Au passage, un petit avertissement : il est toujours tentant de *parser* la sortie de la commande ls avec divers outils pour en extraire certaines données. C'est, à mon sens, assez risqué à long terme, le format d'affichage étant POM dependant[24]. Il est courant de voir les dates exprimées sour la forme *Mmm jj hh :mm* pour les fichiers récents, et *Mmm jj yyyy* pour ceux qui ont plus de six mois.

Certains systèmes (les BSD, par exemple) proposent une commande `stat` permettant d'obtenir un affichage plus formaté, donc plus facilement exploitable par les outils déja vus, des informations sur un fichier ou un répertoire.

```
$ stat -f "%Sp %N" ekko*
-rwxr-xr-x ekko
-rw-r--r-- ekko.c
```

Hélas, là aussi, cette commande a un comportement différent selon son système d'origine.

21.4 Enregistreur

Il vous arrive souvent de taper une suite de commandes interactives et d'en contempler les résultats avec un peu de sceptiscisme. Parfois, vous souhaiteriez en garder une trace écrite. La commande `script` est faite pour ça. Exemple :

```
$ script enregistrement
Script started, output file is enregistrement
$ date
Fri Apr 13 07:54:34 CEST 2007
$ echo $RANDOM
24311
$ ^D
Script done, output file is enregistrement
```

Nous avons ici donné explicitement un nom de fichier de destination. Si nous l'avions omis, le nom par défaut est *typescript*.

D'autre part l'option `-a` demande l'ajout à la fin d'un fichier déja existant.

Le fichier que vous obtenez contient **tout** ce qui est passé par votre terminal, avec deux lignes supplémentaires au début et à la fin donnant la date et l'heure.

```
Script started on Thu Apr 19 12:54:57 2007
$ date^M
Thu Apr 19 12:55:01 CEST 2007
$ echo $RANDOM^M
24261
$ ^D^M

Script done on Thu Apr 19 12:55:06 2007
```

[24]POM : phase of the moon.

Donc vous y trouverez, entre autres, des <CR> que vous avez déja appris à éliminer, avec `tr` ou `sed`.

21.5 Gestion de l'écran

La gestion des écrans textes et le positionnement du curseur est possible, à un modeste niveau, avec l'utilitaire `tput` et une lecture attentive de la documentation termcap/terminfo.

Vous pourrez utiliser des commandes comme `tput clear cup 5 10` qui va effecter l'écran puis positionner le curseur ligne 5 et colonne 10. Il vous faut un exemple ?

22 Programmer en C

Le langage C et le système Unix sont intimement liés depuis les origines de l'un et de l'autre. La toute première version d'Unix, appelée alors Unics[25], a été écrite en assembleur sur un `pdp7` pour faire tourner un jeu : Spacewar.

La seconde version d'Unix a servi de test-bed pour le langage C, dont le but avoué été de permettre la portabilité d'Unix sur d'autres modèles de la gamme `pdp`.

Je n'ai pas la prétention de vous apprendre à programmer une application complète en C sous Unix, mais plutôt de vous montrer quelles sont les possibilitées et les interactions[26] entre le shell, le système et un programme basique écrit en C.

22.1 Hello, world

L'exemple canonique d'un programme en C dans pratiquement tous les contextes (à part peut-être les grille-pains ou les ascenseurs) est celui-ci, que nous devons à Messieurs *K&R* :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("Hello world.\n");
return 0;
}
```

Pour utiliser ce logiciel indispensable, que vous venez de sauver dans le fichier `hello.c`, il faut tout d'abord le transformer en un binaire exécutable, puis l'appeler depuis la ligne de commande par la suite d'opérations suivante :

```
$ cc hello.c -o hello ; ./hello ; ./hello | tr a-z A-Z
Hello world.
HELLO WORLD.
```

Voilà, vous en savez assez maintenant pour comprendre le reste, pause café. Dans 10 minutes, nous allons voir la suite.

22.2 Interfaces avec le shell

Un exécutable Unix, quel que soit le langage utilisé pour l'écrire a, en gros, deux façons de recevoir des paramètres du shell : il a accès aux arguments de la ligne de commande, et il a accès aux variables d'environnement exportées.

[25] Un jeu de mot sur MULTICS, système très ambitieux qui n'a jamais trop vu la lumière.

[26] Interférences ?

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int     foo;
for (foo=1; foo<argc; foo++)
        printf("%s ", argv[foo]);
if (argc > 1)
        printf("\n");
return 0;
}
```

Ce bout de code est une implémentation simpliste de la commande `echo`. La boucle commence à 1, parce que le premier élément du tableau des arguments (*arg[0]*) est le nom de la commande elle-même.

Voyons l'exploitation des variables d'environnement :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
char *cptr;
if ( NULL != (cptr=getenv("QUUX")) )
        printf("QUUX est '%s'\n", cptr);
else
        printf("Pas de QUUX ici\n");
return 0;
}
```

Nous utilisons une fonction de la bibliothèque standard : `getenv()`. Elle prend en paramètre une chaine de caractères contenant le nom de la variable à examiner, et retourne un pointeur NULL si cette variable n'existe pas ou un pointeur vers son contenu si elle existe. Démonstration :

```
~/demc 70 $ ./rdquux
Pas de QUUX ici
~/demc 71 $ QUUX=plop
~/demc 72 $ ./rdquux
Pas de QUUX ici
~/demc 73 $ export QUUX
~/demc 74 $ ./rdquux
QUUX est 'plop'
~/demc 75 $ QUUX=bazz ./rdquux
QUUX est 'bazz'
```

Et enfin, après l'exécution de notre programme, nous voudrions bien dire au shell si nous sommes arrivé a faire quelque chose. Et ça, on sait déja le faire. On l'a vu vers la page 20 quand nous avons parlé des codes de retour. Il ne reste qu'à écrire un bout de code :

```
#include  <stdio.h>
int main(int argc, char *argv[])
{
/* this is very dirty, sorry */
if (2 == argc)
      return atoi(argv[1]);
return 111;
}
```

C'est simple, et ça marche :

```
$ ./return_code ; echo $?
111
$ ./return_code 42 ; echo $?
42
```

Vous pouvez même envisager d'intégrer ce petit programme dans votre boite à outils de déverminer de scripts complexes, pour piloter de façon controlée un `case/esac` rebelle.

22.3 Faire un filtre

Un filtre est un programme souvent destiné à être intégré dans un pipeline de traitements. Il va lire ses données depuis son entrée standard, agir sur ces données, puis renvoyer les résultats obtenus sur sa sortie standard.

Prenons un exemple simple et peut-être artificiel, puisqu'il peut être probablement écrit avec les outils classiques. Nous voulons obtenir la taille des lignes d'un fichier texte, avec leur numéro.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int     numligne, taille, caractere;

numligne = 1; taille = 0;
while ( EOF != (caractere=getchar()) ) {
        if ( caractere == '\n' ) {
                printf("%9d  %9d\n", numligne, taille);
                numligne++;
                taille = 0;
                }
        else {
                taille++;
                }
        }
return 0;
}
```

Simple comme bonjour, quand on a quelques rudiments de C.

22.4 Pour en finir avec un mythe

Non, il n'est pas nécessaire de connaitre et pratiquer le langage C pour utiliser un système Unix, mais parfois c'est vraiment très utile... Nous verrons probablement d'autres exemples plus tard.

23 Autres outils

En dehors de ce que nous venons de voir, il existe une foultitude d'autres choses, dont certaines ne sont pas livrées par votre fournisseur, mais qui peuvent vraiment vous rendre la vie plus tranquille. Cette liste n'est pas limitative, bien entendu.

23.1 Perl

Le langage Perl (*Practical Extraction and Report Language*) [27] a été conçu par Larry Wall, un linguiste, pour remplacer à la fois Sed et Awk. Son universalité, sa puissance, et la quantité quasi indénombrable de modules additionnels en font un outil de plus en plus apprécié.

Avec Perl, vous disposez même d'un translateur (`a2p`) permettant de ré-utiliser vos scripts Awk. Le script que nous avons vu page 46 deviendra ceci :

```
$FS = ':';
$nombre = 0;
while (<>) {
    ($Fld1,$Fld2,$Fld3,$Fld4,$Fld5) = split(/[:\n]/, $_, 9999);
    if ($Fld3 >= 1000) {
        printf "%-6d   %-10s = %s\n", $Fld3, $Fld1, $Fld5;
        $nombre++;
    }
}
print 'on a vu passer', $nombre, 'yusers';
```

C'est certe un peu différent, mais la comparaison des deux codes source est assez instructive.

23.2 Les pages de man

Chaque outil, chaque fonction, chaque appel système, chaque fichier particulier est (théoriquement) documenté dans une *page de man*. Pour visualiser une de ces pages, on utilisera la commande du même nom, donc `man man` vous affichera l'aide de la commande man.

Ces pages de man sont classée par catégorie. Par exemple, sur l'Unix que j'utilise actuellement[28], ces catégories sont :

```
1    General commands (tools and utilities).
2    System calls and error numbers.
3    Libraries.
3p   perl(1) programmer's reference guide.
4    Device drivers.
5    File formats.
```

[27] aka *Pathological Eclectic Rubbish Lister* :)

[28] D'une variante d'Unix à l'autre, les catégories et la classification peuvent varier.

```
6    Games.
7    Miscellaneous.
8    System maintenance and operation commands.
9    Kernel internals.
```

Prenons un exemple : `printf` est à la fois une commande exécutable et une fonction dans la biblothèque standard du langage C. Pour lire la documentation de la commande, il faut utiliser `man 1 printf` et pour lire celle de la fonction, c'est `man 3 printf` qui fera l'affaire.

Régulièrement, le système relit toutes les pages de man en sa connaissance et récupère la ligne contenant le nom de la page et une briève description ; à partir de là, il construit une database. La commande `apropos un_mot` consulte cette database et affiche la liste des pages de man qui parlent de «un_mot». La commande `whatis un_mot`.....

Une page de man correspond à une structure précise et est écrite dans un langage particulier : `troff`. Nous allons voir brièvement ce qu'il en est. Une page doit contenir (obligatoirement ?) ces sections, et dans cet ordre :

NAME Nom de la commande, et en quelques mots, ce qu'elle fait.

SYNOPSIS Lignes de commande utilisables.

DESCRIPTION

FILES Fichiers impliqués dans l'utilisation de cette commande : commandes d'initialisation, données, stockage des highscores.

SEE ALSO Références vers des outils apparentés.

DIAGNOSTICS Messages d'erreur, codes de retours.

BUGS Problèmes connus, limitations, features non documentés, incompatibilités d'humeur avec d'autres logiciels.

```
.TH NAME SECTION local
.SH NAME
.SH SYNOPSIS
.SH DESCRIPTION
.SH FILES
.SH SEE ALSO
.SH DIAGNOSTICS
.SH BUGS
```

Dernière astuce : si vous enregistrez la sortie de la commande `man` dans un fichier, le résultat contiendra des codes de contrôle (backspaces) servant à gérer le soulignement ou la surbrillance. Il est possible de les éliminer avec la chaine de commandes `man foo | col -b > foo.doctxt` qui voud donnera un document plus clair.

23.3 Make

Make est l'auxiliaire idéal. Vous lui expliquez une seule fois ce que vous avez l'habitude de faire, et il est capable de répéter à l'infini vos actions, en en faisant à chaque fois le moins possible.

Pour cela, il se base sur deux choses : la date de dernière modification des fichiers, et un graphe de dépendances que vous aurez décrit dans un fichier dont le nom par défaut est `Makefile`.

Reprenons quelque chose que nous avons déja vu : la fabrication de la lsite des yusers avec un script Awk. Cette liste sera rangée dans un fichier que nous nommerons *yusers.liste*. Son contenu est dépendant de deux choses : `/etc/passwd` et le script de fabrication, `lstyusers.awk`. Si l'un des deux change, notre cible finale va probablement changer aussi.

Exprimons cela en règles de Makefile :

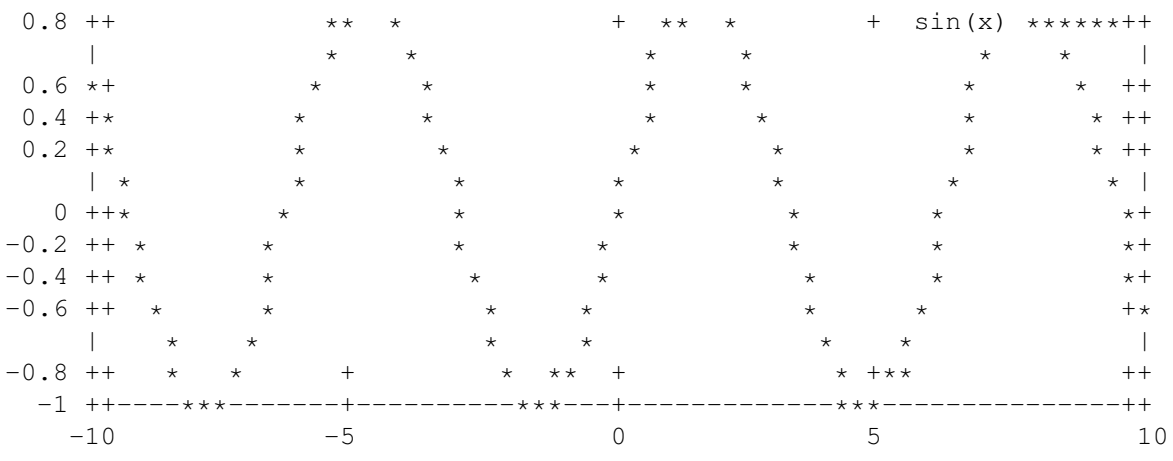
```
yusers.liste:   lstyusers.awk /etc/passwd
        ./lstyusers.awk < /etc/passwd > yusers.liste
```

Astuce dans l'astuce : les makefiles utilisent obligatoirement le caractère de tabulation pour indenter les lignes de commandes à exécuter. Vous connaissez `cat` et son option `-t`...

23.4 Gnuplot

Gnuplot est un outil de visualisation de données numériques. Vous lui fournissez un fichier texte contenant des nombres, il arrive à en faire une image. Il arrive même à en faire une image qui ne soit pas graphique. Exemple :

```
gnuplot> set term dumb 75 17
Terminal type set to 'dumb'
Options are 'feed  75 17'
gnuplot> plot sin(x)
```



Nous allons essayer de construire un exemple complet afin d'utiliser certaines notions que nous avons vues tout au long de cette formation.

23.5 Le package `netpbm`

Un ensemble très puissant d'outils en ligne de commande destinée aux manipulations d'images bitmap : conversion de format, rotation, réductions de couleurs, génération, mixage...

Cet ensemble exploite vraiment à fond la notion de pipeline de traitements. Un format générique de description d'image permet de les faire transiter d'un outil à l'autre.

23.6 Un peu d'interactivité

24 Exemples pratiques

Voilà, nous avons découvert plein de choses (même moi, en fait, puisque la rédaction de ce document m'a obligé à remettre mes connaissances à jour :) vraiment efficaces. Il nous faudrait donc maintenant imaginer quelques exemples pratiques...

25 Conclusion

Vous en savez assez maintenant pour vous rendre compte que vous n'êtes que des débutants, et que la puissance que vous offre les systèmes de la famille Unix ne connait qu'une frontière : votre imagination.

Si le document que vous êtes en train de lire vous semble inexact, inprécis, incomplet, approximatif, c'est probablement de ma faute, mais nous sommes tous à même de le rectifier. Une des devises des Perlistes est : IL Y A PLUSIEURS FAÇONS DE FAIRE LES CHOSES, et cette sage devise s'applique parfois aussi à la programmation des shells Unix.

Je reste à votre disposition...

Index