ROYAUME DU MAROC

مكتب التكوين المهني وإنعاش الشغل

Office de la Formation Professionnelle et de la Promotion du Travail

Utilisation du shell Bash

www.ofppt.info

OFPPT

DIRECTION **R**ECHERCHE ET **I**NGENIERIE DE **F**ORMATION

SECTEUR NTIC

Sommaire

Le shell est l'environnement utilisateur en mode texte sous Linux. C'est le programme qui se charge de lire et d'exécuter les commandes que l'utilisateur saisit. Classiquement, le shell est utilisé de manière interactive, c'est-à-dire que l'utilisateur dialogue avec le système par l'intermédiaire du shell. Il saisit les commandes, et le shell les exécute et affiche les résultats. Le shell le plus couramment utilisé sous Linux est sans aucun doute bash. En tout cas, c'est le shell par défaut que la plupart des distributions utilisent. Il est donc conseillé de connaître un petit peu ce que ce shell est capable de réaliser, et comment. Le shell bash est une évolution du shell sh, utilisé par quasiment tous les systèmes Unix. Son nom provient de l'abréviation de l'anglais « Bourne Again SHell », ce qui signifie qu'il s'agit effectivement d'une nouvelle variante du shell sh.

Au temps des interfaces graphiques complexes et sophistiquées, il peut paraître archaïque de vouloir encore utiliser des lignes de commandes pour utiliser un ordinateur. C'est en partie vrai, mais il faut savoir que les shells Unix sont extrêmement puissants et que les interfaces graphiques ne permettent toujours pas, même à l'heure actuelle, de réaliser toutes les tâches faisables avec un shell. D'autre part, il est souvent plus efficace de taper une simple commande dans un shell que de rechercher un outil graphique et de parcourir les divers menus, puis de choisir les options de la commande désirée avant de valider. Des ergonomes ont démontré, et des graphistes du monde entier le confirmeront, que la souris n'est pas le périphérique d'entrée le plus précis et le plus facile à utiliser pour manipuler les objets de l'environnement utilisateur. La plupart des programmeurs utilisent encore bon nombre de ce qu'on appelle des « raccourcis clavier » pour exécuter des commandes, même dans les environnements utilisateurs graphiques.

Quoi qu'il en soit, le shell est bien plus qu'un interpréteur de commande. Il s'agit réellement d'un environnement de programmation, permettant de définir des variables, des fonctions, des instructions complexes et des programmes complets, que l'on appelle des *scripts shell*. Les sections suivantes ont pour objectif de vous montrer les principales caractéristiques du shell, sans pour autant prétendre vous apprendre la programmation des scripts shell. La lecture de cette section pourra donc être différée dans un premier temps. Toutefois, elle pourra être bénéfique à ceux qui désirent comprendre les scripts de configuration utilisés par leur distribution, ou tout simplement à ceux qui sont curieux de nature.

1. Contrôle des processus

Un des avantages des lignes de commandes par rapport aux environnements graphiques est la facilité avec laquelle elles permettent de contrôler les processus. Ce paragraphe décrit les principales méthodes pour lancer et arrêter un processus, ainsi que pour lui fournir les données sur lesquelles il doit travailler et récupérer ses résultats.

1.1 Lancement d'un programme en arrière-plan

Le lancement normal d'un programme se fait en tapant sa ligne de commande et en appuyant sur la touche de validation. Le shell ne rendra pas la main et ne permettra pas de lancer un autre programme tant que le processus en cours ne sera pas terminé. Cependant, vous pouvez fort bien désirer lancer en arrière-plan une commande dont la durée d'exécution peut être très longue et continuer à travailler. Après tout, Linux est multitâche... Eh bien, rien de plus facile !

Pour lancer une commande en arrière-plan, il faut :

- s'assurer que la commande aura toutes les informations nécessaires pour travailler sans intervention de l'utilisateur (ou, autrement dit, que la commande n'est pas interactive) ;
- ajouter une esperluette (caractère « & ») à la fin de la ligne de commande.

Par exemple, la commande suivante :

cp /cdrom/kernel/linux-2.6.6.tar.gz . &
: copiera l'archive du noyau 2.6.6 du CD-ROM vers le répertoire courant, et s'exécutera en arrière-plan.

Lorsqu'une commande est lancée en arrière-plan, le shell affiche deux nombres qui permettront de l'identifier par la suite. Le premier nombre, indiqué entre crochets, est le numéro de « job » du shell. Ce numéro sert à identifier les commandes du shell de manière unique. Un job est donc en réalité une commande du shell, simple ou complexe. Le deuxième numéro est le numéro de processus (« PID », pour « Process IDentifier ») dans le système du processus maître du job. Le PID est un numéro unique dans le système, qui permet d'identifier de manière unique les processus en cours. Ces deux nombres permettront de manipuler les processus, avec les commandes que l'on verra plus tard.

Il ne faut pas confondre les numéros de job avec les numéros de processus. Premièrement, un numéro de job n'est unique que dans un shell donné, et n'a aucune signification au niveau du système complet, alors que le numéro de processus est attribué par le système à chaque programme en cours d'exécution. Ensuite, une même commande du shell peut lancer plusieurs processus conjointement. Dans ce cas, il y a bien évidemment plusieurs numéros de processus, mais un seul et unique job. Ce genre de situation se produit par exemple lors de l'utilisation d'une redirection du flux de sortie standard d'un processus vers le flux d'entrée standard d'un autre processus.

Le numéro de processus affiché par le shell lors du lancement d'une ligne de commande complexe représente le PID du processus maître de la commande, c'est-à-dire, en pratique, le dernier processus d'une série de redirections ou le processus du shell exécutant les commandes complexes. Cela signifie que dans tous les cas de configuration, ce PID est celui du processus qui contrôle l'ensemble des opérations effectuées par la ligne de commande. C'est donc par

ce processus que l'on peut manipuler la commande complète, par exemple pour l'interrompre.

Il est possible de retrouver le PID du processus maître d'une commande à partir du numéro de job correspondant du shell. Cela se fait simplement, en utilisant l'expressions suivante :

%job
: où job est le numéro du job dont on cherche le PID. Ainsi, dans toutes les commandes décrites ci-dessous, le PID peut être utilisé directement, ou être remplacé par le numéro du job préfixé du caractère de pourcentage.

1.2. Listing des processus

Il n'est pas nécessaire de retenir tous les numéros de jobs et tous les PID des processus en cours d'exécution. Il existe en effet des commandes permettant d'obtenir la liste des processus et des jobs. La plus simple à utiliser est bien évidemment la commande du shell pour obtenir la liste des jobs avec leurs lignes de commandes. Pour obtenir cette liste, il suffit de taper la commande suivante :

jobs
: qui affiche, dans l'ordre, le numéro de job, l'état du processus correspondant, et la ligne de commande.

Une autre commande, plus bas niveau, permet d'obtenir des informations plus complètes directement à partir du système. Il s'agit de la commande **ps**, dont la syntaxe est donnée ci-dessous :

ps [options]

Les options les plus utiles sont sans doute x, qui permet de demander l'affichage de toutes les commandes en cours d'exécution et non pas seulement les processus en cours d'exécution dans le shell où la commande **ps** est exécutée, et a, qui permet d'obtenir l'affichage de toutes les commandes, pour tous les utilisateurs connectés. Ces deux options peuvent être cumulées, et la commande suivante :

ps ax
: affiche donc toutes les commandes en cours d'exécution sur le système.

Les informations les plus intéressantes affichées par **ps** sont le PID du processus, qui est donné par le premier nombre affiché, et la ligne de commande, qui est la dernière information affichée. Pour plus de détails sur la commande **ps**, veuillez consulter la page de manuel correspondante.

1.3. Notion de signal

Dans un système Unix, tous les processus peuvent recevoir des messages, envoyés soit par l'utilisateur, soit par un autre processus, soit par le système. Ces messages sont appelés *signaux*. La plupart des signaux sont envoyés par le

système pour indiquer au processus qu'il a fait une faute et qu'il va être terminé. Cependant, ce n'est pas toujours le cas : certains signaux sont envoyés uniquement dans le cadre de la communication entre les processus, et certains autres ne peuvent même pas être captés par le processus et sont traités directement par le système. Nous n'entrerons pas en détail dans la gestion des signaux ici, car cela nous emmènerait trop loin. Cependant, la manière d'envoyer un signal à un processus à partir du shell sera décrite.

L'envoi d'un signal se fait avec la commande **kill**, avec la syntaxe suivante :

kill [-signal] PID

où signal est une option qui permet de préciser le signal qui doit être envoyé, et PID est le numéro du processus qui doit le recevoir. Les numéros de signaux les plus importants sont décrits dans le tableau ci-dessous :

Tableau 1-1. Principaux signaux Unix

Numéro de signal	Signification
15	Signal de terminaison de processus.
9	Signal de destruction inconditionnelle de processus.
19	Signal de suspension de processus.
18	Signal de reprise d'exécution d'un processus suspendu.

Lorsqu'aucun signal n'est spécifié, le signal 15 de terminaison est utilisé par défaut. Ce signal demande au processus en cours d'exécution de se terminer immédiatement. Il peut être capté par le processus, pour lui donner une chance d'enregistrer les données sur lesquelles il travaillait et de libérer les ressources qu'il utilisait. Pour certains processus, cela ne fonctionne pas, et il faut utiliser le signal de destruction du processus à l'aide de la commande suivante :

kill -9 PID

Attention cependant à cette commande : le processus est immédiatement détruit, sans autre forme de procès. Il peut donc s'ensuivre une perte de données, n'en abusez donc pas trop.

1.4. Arrêt d'un processus

Tout processus lancé en ligne de commande peut être arrêté immédiatement sous Linux. Pour cela, deux méthodes sont disponibles. La première consiste à taper la combinaison de touches CTRL+C lorsque le processus est en cours d'exécution interactive (c'est-à-dire lorsqu'il n'a pas été lancé en arrière-plan). S'il a été lancé en arrière-plan, on peut soit le ramener en avant-plan (avec la commande **fg**, que l'on verra plus loin) avant d'utiliser CTRL+C, soit lui envoyer

le signal de terminaison à l'aide de la commande **kill** vue précédemment. Dans le cas d'une ligne de commande, le signal de terminaison est transmis au processus maître de la ligne de commande, et est ensuite propagé à l'ensemble des processus fils de ce processus. Cela signifie que tous les processus invoqués dans le cadre de cette commande sont également arrêtés.

1.5. Gel d'un processus

Il est possible de « geler » un processus en cours d'exécution, c'est-à-dire de le suspendre, sans pour autant l'arrêter définitivement. Cela peut être utilisé pour libérer un peu les capacités de calcul, lorsque ce processus consomme trop de ressources par exemple. Pour cela, deux méthodes sont possibles :

- soit on utilise la combinaison de touches CTRL+Z, lorsque le processus est en avant-plan ;
- soit on envoie le signal 19 au processus (signal « STOP ») à l'aide de la commande **kill**.

La première méthode est recommandée pour les processus lancés par une ligne de commande complexe, car le signal STOP est envoyé au processus maître de la commande, et est propagé à l'ensemble des processus fils de ce processus. Cela signifie que tous les processus invoqués dans le cadre de cette commande sont également gelés. La deuxième méthode est plus bas niveau, et permet de geler n'importe quel processus que l'on a lancé.

1.6. Relancement d'un processus

Un processus suspendu peut être relancé soit en avant-plan, soit en arrière-plan. Pour relancer un processus en avant-plan, il faut utiliser la commande suivante :

fg [PID]

où PID est le PID du processus à relancer en avant-plan. Si ce paramètre n'est pas spécifié, le dernier processus stoppé sera relancé en arrière-plan. **fg** est l'abréviation de l'anglais « foreground », ce qui signifie « avant-plan ». Il faut attendre que ce processus se termine pour entrer de nouvelles commandes. Par conséquent, on ne peut lancer en avant-plan qu'un seul processus.

De même, pour lancer un processus en arrière-plan, il faut utiliser la commande **bg**, qui est l'abréviation de l'anglais « background ». Cette commande s'utilise de la même manière que la commande **fg** :

bg [PID]

Le relancement d'un processus suspendu peut également se faire en lui envoyant le signal 18 à l'aide de la commande **kill**.

2. Redirections

Pour pouvoir lancer un programme en arrière-plan, il est nécessaire qu'il n'ait pas besoin de demander des données à l'utilisateur. En effet, lorsqu'il est en arrière-plan, la saisie de ces données ne peut pas se faire, puisque le shell les interpréterait comme une nouvelle commande. De plus, tout affichage en provenance d'une commande en arrière-plan apparaît sur la console tel quel, et risque de se mélanger avec l'affichage des autres programmes ou même avec la commande en cours d'édition. C'est pour résoudre ces problèmes que le mécanisme des redirections a été introduit.

2.1. Principe de base

Le mécanisme des *redirections* a pour but de transférer les données provenant d'un flux vers les données d'un autre flux. Il se base sur la notion de descripteur de fichier, utilisée par la plupart des systèmes Unix. Un *descripteur de fichier* est un numéro utilisé par les programmes pour identifier les fichiers ouverts. Les descripteurs 0, 1 et 2 sont respectivement affectés d'office au flux d'entrée standard (nommé « stdin »), au flux de sortie standard (« stdout ») et au flux d'erreur standard (« stderr »), qui en général apparaît également sur l'écran.

Les descripteurs de fichiers d'un processus sont généralement hérités par tous ses processus fils. Cela signifie que, lors de leur lancement, ces processus peuvent utiliser tous les descripteurs de fichiers mis à leur disposition par leur père. Dans le cas des lignes de commande, les processus fils sont les processus lancés par le shell, et les descripteurs de fichiers hérités sont donc les descripteurs de fichiers du shell. C'est de cette manière que le shell peut manipuler les descripteurs de fichiers des processus qu'il lance : il effectue d'abord les redirections sur ses propres descripteurs de fichiers, puis il lance le processus fils avec ces redirections actives. Le mécanisme est donc complètement transparent pour les processus fils.

Le mécanisme des redirections permet en fait d'injecter dans un descripteur de fichier des données provenant d'un autre descripteur ou d'un fichier identifié par son nom, et d'envoyer les données provenant d'un descripteur de fichier dans un autre descripteur ou dans un fichier identifié par son nom. Si l'on utilise les descripteurs de fichiers des flux d'entrée / sortie standards, on peut exécuter n'importe quelle commande interactive en arrière-plan.

2.2. Redirections de données en entrée

Pour injecter des données provenant d'un fichier dans le descripteur de fichier n d'un processus, il suffit d'ajouter la ligne suivante à la fin de la commande permettant de lancer ce processus :

n<fichier

où fichier est le nom du fichier dont les données doivent être injectées dans le descripteur n. Dans cette syntaxe, le descripteur peut ne pas être

précisé. Dans ce cas, le shell utilisera le descripteur 0, et les données du fichier seront donc envoyées dans le flux d'entrée standard du processus. Par exemple, supposons que l'on désire utiliser une commande nommée « search », et que cette commande demande un certain nombre d'informations lors de son exécution. Si l'on sait à l'avance les réponses aux questions qui vont être posées, on peut créer un fichier de réponse (nommé par exemple « answer.txt ») et alimenter la commande « search » avec ce fichier. Pour cela, on utilisera la ligne de commande suivante :

search < answer.txt

Il est également possible d'injecter des données provenant d'un autre descripteur de fichier dans un descripteur de fichier. On utilisera pour cela la syntaxe suivante :

n<&s

où n est toujours le descripteur de fichier du processus à exécuter dans lequel les données doivent être injectées, et s est un descripteur de fichier contenant les données sources à injecter. Par défaut, si n n'est pas précisé, le flux d'entrée standard du processus sera utilisé.

2.3. Redirection de données en sortie

Inversement, il est possible d'enregistrer les données écrites par un processus dans un de ses descripteurs de fichier dans un fichier. Pour cela, on utilisera l'opérateur '>' avec la syntaxe suivante :

n>fichier

où n est le numéro du descripteur de fichier du processus à enregistrer, et fichier est le nom du fichier dans lequel les données doivent être stockées. Par défaut, si n n'est pas spécifié, le descripteur du flux de sortie standard sera utilisé (descripteur 1). Par exemple, si la commande précédente affiche des résultats et que l'on désire les stocker dans le fichier « result.txt », on utilisera la ligne de commande suivante :

search < answer.txt >result.txt

Notez que cette commande détruira systématiquement le contenu du fichier « result.txt » et le remplacera par les informations provenant du flux de sortie standard du processus « search ». Il est possible de ne pas vider le fichier « result.txt » et d'ajouter les informations en fin de fichier, en utilisant l'opérateur '>>' à la place de l'opérateur '>'. Ainsi, la commande suivante :

search < answer.txt >>result.txt

aura pour effet d'ajouter à la fin du fichier « result.txt » les informations affichées par le processus « search ».

Le flux d'erreur standard, qui correspond normalement à l'écran et qui permet d'afficher les messages d'erreur, peut être redirigé avec l'opérateur '2>', de la même manière que l'opérateur '>' est utilisé pour le flux de sortie standard (puisque c'est le descripteur de fichier utilisé par défaut par l'opérateur '>'). Par exemple, si l'on veut envoyer les messages d'erreurs éventuels de la commande

précédente vers le périphérique nul (c'est-à-dire le périphérique qui n'en fait rien) pour ignorer ces messages, on utilisera la ligne de commande suivante :

search <answer.txt >result.txt 2> /dev/null

Une telle ligne de commande est complètement autonome, et peut être lancée en arrière-plan, sans aucune intervention de l'utilisateur :

search <answer.txt >result.txt 2> /dev/null &

Il est également possible d'effectuer une redirection des données provenant d'un descripteur de fichier du processus vers un autre descripteur de fichier de ce processus. On utilisera pour cela la syntaxe suivante :

n>&d

où n est le descripteur de fichier dont les données doivent être redirigées, et d le descripteur de fichier destination. Cette syntaxe est souvent utilisée pour rediriger le flux d'erreur standard vers le flux d'entrée standard, lorsqu'on veut récupérer les erreurs et les messages d'exécution normale dans un même fichier. Par exemple, si l'on veut rediriger le flux de sortie et le flux d'erreurs de la commande « search » dans un même fichier, on utilisera la ligne de commande suivante :

search <answer.txt >result.txt 2>&1

Cette ligne de commande utilise deux redirections successives pour les données affichées par la commande « search » : la première redirige le flux de sortie standard vers un fichier, et la deuxième le flux d'erreur standard vers le flux de sortie standard. Notez que l'ordre des redirections est important. Elles sont appliquées de gauche à droite. Ainsi, dans la commande précédente, le flux de sortie standard est redirigé vers le fichier « result.txt », puis le flux d'erreur standard est injecté dans le flux de sortie standard ainsi redirigé.

Note : Il est également possible d'utiliser un autre descripteur de fichier que les descripteurs des flux standards. Cependant, il est nécessaire, dans ce cas, d'ouvrir ce descripteur dans le shell avant de lancer la commande. Cela peut se faire à l'aide de la syntaxe suivante :

n<>fichier

où n est un numéro de descripteur de fichier non encore utilisé, et fichier est un nom de fichier. Ce nouveau descripteur de fichier pourra être utilisé dans les commandes précédentes, afin de faire manipuler le fichier fichier par les processus fils de manière transparente.

Les descripteurs de fichiers ouverts de cette manière le restent d'une commande sur l'autre dans le shell. Cela implique que toutes les données écrites dans ces descripteurs de fichiers sont ajoutées automatiquement à la fin des fichiers manipulés par ces descripteurs. Ce comportement est différent de la redirection vers un fichier effectuée par l'opérateur '>', qui ouvre à chaque fois le fichier en écriture à son début et qui supprime donc toutes les données déjà existantes. Il n'y a donc pas d'opérateur '>>&' pour ajouter des données à un descripteur de fichier, car cela n'a pas de sens.

Les descripteurs de fichiers peuvent également être manipulés directement, par l'intermédiaire de fichiers virtuels du répertoire /dev/fd/. À chaque descripteur de fichier (y compris les descripteurs pour les flux d'entrée/sortie standards !) y correspond un fichier dont le nom est le numéro du descripteur. Par exemple, le fichier /dev/fd/2 correspond au flux d'erreur standard.

En fait, le répertoire /dev/fd/ est un lien symbolique vers le répertoire /proc/self/fd/ du système de fichiers virtuel /proc/. Ce système de fichiers est géré par le noyau directement, et permet d'accéder aux informations sur le système et les processus. Il contient en particulier un sous-répertoire portant le nom du PID de chaque processus existant dans le système, et chacun de ces répertoires contient lui-même un sous-répertoire fd/ où sont représentés les descripteurs de fichiers ouvert par le processus correspondant. Le système de fichiers /proc/ contient également un lien symbolique self/ pointant sur le sous-répertoire du processus qui cherche à l'ouvrir. Ainsi, /proc/self/fd/ est un chemin permettant à chaque processus d'accéder à ses propres descripteurs de fichiers.

En pratique, la manipulation directe des descripteurs de fichiers n'est réellement intéressante que pour les flux standards, dont les numéros de descripteurs sont fixes et connus de tous les programmes. Pour les autres descripteurs, cette technique est souvent inutilisable ou inutile, sauf lorsqu'on utilise des programmes sachant manipuler des descripteurs de numéros bien déterminés.

2.4. Insertion de documents

Il existe un dernier opérateur de redirection, qui n'est utilisé en pratique que dans les scripts shell. Cet opérateur permet d'insérer directement un texte complet dans le flux d'entrée standard, sans avoir à placer ce document dans un fichier à part. Cette technique permet donc de stocker des données avec le code des scripts shell, et de n'avoir ainsi qu'un seul fichier contenant à la fois le script et ses données.

Cet opérateur est l'opérateur '<<', il s'utilise selon la syntaxe suivante :

```
<<EOF
texte
.
.
.
EOF
```

où texte est le contenu du texte à insérer, et EOF est un marqueur quelconque qui sera utilisé seul sur une ligne afin de signaler la fin du texte.

Par exemple, il est possible de créer un fichier test.txt de la manière suivante :

```
cat <<fin >test.txt
Ceci est un fichier texte saisi directement dans le shell.
On peut écrire tout ce que l'on veut, et utiliser les fonctions d'éditions
de ligne du shell si l'on veut.
```

Pour terminer le fichier, il faut taper le mot "fin" tout seul, au début d'une ligne vide.
fin

3. Les tubes

Les redirections sont très pratiques lorsqu'il s'agit d'injecter un fichier dans le flux d'entrée standard d'un processus, ou inversement de rediriger le flux standard d'une commande vers un fichier, mais elles ont justement le défaut de devoir utiliser des fichiers. Il est des situations où l'on désirerait injecter le résultat d'une commande dans le flux d'entrée standard d'une autre commande, sans passer par un fichier intermédiaire. Cela est heureusement réalisable, grâce à ce que l'on appelle les « tubes ».

3.1. Syntaxe des tubes

Pour rediriger le résultat d'une commande dans le flux d'entrée d'une autre commande, il faut utiliser l'opérateur '|'. Cet opérateur représente un tuyau canalisant les données issues d'une commande vers le flux d'entrée standard de la commande suivante, d'où le nom de « pipe » en anglais (ce qui signifie « tuyau » ou « tube »). L'opérateur tube s'utilise de la manière suivante :

- on écrit la première commande, qui doit fournir les données à la deuxième commande ;
- on écrit l'opérateur tube ;
- on écrit la deuxième commande, qui doit lire les données provenant de la première.

> La commande se trouvant à la gauche de l'opérateur tube doit être complète, avec ses autres redirections éventuelles. La redirection dans un tube s'effectue après les autres types de redirections vues précédemment.

Le système contrôle l'exécution des processus qui se trouvent aux deux bouts d'un tube, de telle sorte que le transfert de données puisse toujours se faire. Si le processus source a trop de données, il est figé par le système d'exploitation en attendant que le processus consommateur ait fini de traiter les données déjà présentes. Inversement, si le processus source est trop lent, c'est le processus consommateur qui attendra patiemment que les données soient disponibles.

Les tubes sont utilisés très couramment, ne serait-ce que pour afficher page par page le contenu d'un répertoire. La commande suivante effectue un tel travail :

ls | less

Ici, le résultat de la commande **ls** est redirigé vers la commande **less**, qui permet d'afficher page par page (et de revenir en arrière dans ces pages) la liste des fichiers du répertoire courant.

Prenons un exemple un peu plus complexe. Supposons que l'on veuille archiver

et compresser un répertoire. Il est possible d'archiver ce répertoire avec la commande **tar**, puis de compresser le fichier archive résultant :

```
tar cvf archive.tar *
gzip archive.tar
```

Cette méthode est correcte, mais souffre d'un défaut : elle utilise un fichier intermédiaire, qui peut prendre beaucoup de place disque. Une méthode plus économe consiste à lancer **tar** et **gzip** en parallèle, et à rediriger la sortie standard de l'un dans le flux d'entrée de l'autre. Ainsi, il n'y a plus de fichier temporaire, et la place consommée sur le disque est minimale :

```
tar cv * | gzip > archive.tar.gz
```

La première commande demande à **tar** d'archiver tous les fichiers du répertoire et d'envoyer le résultat dans le flux standard de sortie. Le pipe redirige ce flux standard vers le flux d'entrée standard de **gzip**. Celui-ci compresse les données et les émet vers son flux standard de sortie, qui est lui-même redirigé vers le fichier archive.tar.gz. Aucun fichier temporaire n'a été utilisé, et on a ainsi économisé l'espace disque de l'archive complète non compressée, c'est-à-dire environ la taille complète du répertoire à archiver. Ce genre de considération peut être très important lorsque le disque dur commence à être plein...

Note : En fait, la commande tar de GNU permet de compresser à la volée les données à archiver, permettant d'éviter de se prendre la tête comme on vient de le faire. Pour cela, il suffit d'utiliser l'option z dans la ligne de commande de **tar**. Ainsi, la ligne de commande suivante fournit le même résultat :

```
tar cvfz archive.tar.gz *
```

Mais cette solution ne fonctionne pas avec les versions non GNU de tar, qui ne supportent pas cette option.

Un autre exemple pratique est le déplacement de toute une arborescence de fichiers d'un système de fichiers à un autre. Vous ne pourrez pas y parvenir à l'aide de la commande **mv**, car celle-ci ne fait que modifier la structure du système de fichiers pour déplacer les fichiers et les répertoires, elle ne peut donc pas fonctionner avec deux systèmes de fichiers. Vous ne pouvez pas non plus utiliser la commande **cp**, car celle-ci ne prendra pas en compte les dates des fichiers, leur propriétaire et leur groupe, ainsi que les liens symboliques et physiques. Il faut donc impérativement utiliser un programme d'archivage. La méthode à suivre est donc de créer une archive temporaire, puis de se déplacer dans le répertoire destination, et enfin d'extraire l'arborescence de l'archive :

```
cd source
tar cvf archive.tar *
cd destination
tar xvf source/archive.tar
rm source/archive.tar
```

Malheureusement, cette technique nécessite beaucoup de place disque, puisque l'archive temporaire est stockée directement sur disque. De plus, elle est assez

lente, car toutes les données à copier sont recopiées sur le disque dur, et relues ensuite, pour finalement être détruites... La vraie solution est de réaliser un tube entre les deux processus **tar** invoqués. Dans ce cas, le transfert se fait simplement via la mémoire vive :

```
cd source
tar cv * | (cd destination ; tar xvf -)
```

La commande à utiliser est cette fois un peu plus compliquée, car la commande d'extraction des fichiers nécessite un changement de répertoire. Il faut donc utiliser une commande multiple du shell. Ces commandes sont constituées de plusieurs autres commandes séparées par des points virgules. La première commande effectuée ici est le changement de répertoire, et la deuxième est l'extraction par **tar** de l'archive qui lui est transférée par le flux d'entrée standard (représenté ici par '-'). Ces deux commandes sont mises entre parenthèses, car l'opérateur '|' du tube est prioritaire sur l'opérateur ';' de concaténation des commandes du shell. Si vous trouvez que cela est un peu compliqué, je vous l'accorde. Cependant, la commande qui utilise le tube consomme deux fois moins d'espace disque et est deux fois plus rapide que la commande qui n'en utilise pas. Je vous invite à mesurer le gain de temps sur un répertoire contenant un grand nombre de données (utilisez la commande **time** !).

3.2. Les tubes nommés

Les tubes créés par l'opérateur '|' constituent ce que l'on appelle des tubes anonymes, car ils sont créés directement par le shell pour une commande donnée. Il est possible de créer manuellement des tubes en leur donnant un nom, et de les utiliser a posteriori dans plusieurs commandes. Ces tubes constituent ce que l'on appelle des tubes nommés.

En fait, les tubes nommés sont des fichiers spéciaux, que l'on crée dans un système de fichiers capable de les gérer. Les seules opérations réalisables sont l'écriture et la lecture, sachant que les données écrites en premier seront forcément les premières données lues. C'est ce comportement qui a donné leur nom à ces fichiers, que l'on appelle des « FIFO » (abréviation de l'anglais « First In First Out »). De plus, la quantité de données en transit dans ces fichiers est souvent très réduite, ce qui fait que ces données sont toujours placées dans la mémoire cache du système. Ainsi, bien qu'il s'agisse de fichiers, aucune écriture ou lecture sur disque n'a lieu lors de l'utilisation d'un pipe.

Les tubes nommés sont créés par la commande **mkfifo**, dont la syntaxe est la suivante :

```
mkfifo nom
```

où nom est le nom du tube nommé. Notez que cette commande échouera sur les systèmes de fichiers incapables de gérer les tubes nommés.

Une fois créé, le fichier de tube peut être utilisé comme n'importe quel fichier dans les redirections que l'on a vues dans la section précédente. Par exemple, la redirection suivante :

ls | less
peut être réécrite pour utiliser un tube nommé temporaire de la manière suivante :
mkfifo /tmp/tempfifo
ls > /tmp/tempfifo
less < /tmp/tempfifo

La destruction d'un tube nommé se fait comme n'importe quel fichier, à l'aide de la commande **rm**.

3.3. La commande tee

La commande **tee** est un petit programme permettant d'enregistrer les données qu'il reçoit dans son flux d'entrée standard dans un fichier et de les renvoyer simultanément vers son flux de sortie standard. Elle est couramment utilisée, en conjonction avec les tubes, pour dupliquer un flux de données. Sa syntaxe est la suivante :

tee fichier
où fichier est le nom du fichier dans lequel le flux d'entrée standard doit être enregistré.

Supposons par exemple que l'on désire rediriger tous les messages (d'erreur ou non) de la commande **ls /proc/1/*** dans un fichier result.txt, tout en continuant à les visualiser sur l'écran. Pour cela, on utilisera la commande suivante :

ls -l /proc/1 2>&1 | tee result.txt
À l'issue de cette commande, le fichier result.txt contiendra une copie des données qui ont été émises par la commande **ls -l /proc/1 2>&1**.

3.4. La commande xargs

La commande **xargs** permet d'appeler une autre commande, en passant en paramètre les données qu'elle reçoit dans le flux d'entrée standard. Sa syntaxe est la suivante :

xargs commande
où commande est la commande que **xargs** doit exécuter. **xargs** construira une ligne de commande complète pour cette commande, en utilisant comme paramètres les données issues du flux d'entrée standard. Une fois cette ligne de commande construite, **xargs** l'exécutera. Par exemple, la commande suivante :
ls -l
peut être exécutée également de la manière suivante :
xargs ls
et en tapant la chaîne de caractères « -l » suivie du caractère de fin de fichier CTRL+D.

La commande **xargs** est une commande extrêmement utile lorsqu'elle est

utilisée conjointement avec les tubes, parce qu'elle permet d'utiliser le résultat d'une commande en tant que paramètre pour une autre commande. Ce mécanisme est donc complémentaire de celui des pipes, puisque ceux-ci permettaient d'utiliser le résultat d'une commande pour alimenter le flux d'entrée standard d'une autre commande.

Un exemple plus utile que le précédent permettra de mieux comprendre comment on utilise la commande **xargs**. Supposons que l'on désire trouver tous les fichiers d'une arborescence complète dont l'extension est .txt et contenant la chaîne de caractères « test ». La liste des fichiers de l'arborescence peut être déterminée simplement à l'aide de la commande **find**, et la recherche du texte dans les fichiers se fait naturellement à l'aide de la commande **grep**. On utilisera **xargs** pour construire la ligne de commande pour **grep**, à partir du résultat fourni par la commande **find** :

find -name "*.txt" | xargs grep -l "test"
: Cette commande est plus simple et plus efficace que la commande équivalente :

find -name "*.txt" -exec grep -l "test" {} \;
: parce que **grep** n'est exécuté qu'une seule fois (alors que l'option -exec de la commande **find** l'exécute pour chaque fichier trouvé).

4. Manipulation des variables d'environnement

Les systèmes Unix permettent de définir un environnement d'exécution pour chaque programme en cours d'exécution. L'environnement est un ensemble de paramètres, que l'on appelle les *variables d'environnement*, qui permettent de modifier le comportement du programme. Ces variables contiennent une valeur de type chaîne de caractères, dont la signification est propre à chaque variable. Il est d'usage que les noms des variables d'environnement soient écrits complètement en majuscules, mais ce n'est pas une obligation.

Chaque programme est susceptible de reconnaître un certain nombre de variables d'environnement qui lui sont propres, mais il existe également des variables standards que tous les programmes utilisent. C'est notamment le cas de la variable d'environnement PATH, qui contient la liste des répertoires dans lesquels le système doit rechercher les programmes à exécuter. Cette variable permet donc de lancer les programmes en tapant simplement leur nom, et de laisser le système rechercher le fichier de ce programme dans chacun des répertoires indiqués dans cette variable.

Par défaut, les programmes sont lancés avec l'environnement du programme qui les lance, c'est-à-dire dans la plupart des cas l'environnement d'exécution du shell. Les programmes peuvent également définir de nouvelles variables d'environnement, qui seront ainsi accessibles par les programmes qu'ils lanceront eux-mêmes.

Comme tout programme, le shell dispose d'un environnement, qu'il utilise pour stocker ses propres variables. En effet, comme nous l'avons déjà signalé plus haut, le shell est bien plus qu'un interpréteur de commande : il est

complètement programmable. Et en tant qu'interpréteur d'un langage de programmation, il fournit la possibilité de définir des variables de ce langage. Les variables du shell sont donc également des variables d'environnement, mais le shell ne les communique pas par défaut aux programmes qu'il lance. Pour être plus précis, le shell utilise deux environnements différents :

- son propre environnement, qui contient les variables d'environnement locales à la session du shell en cours ;
- l'environnement d'exécution, dont les variables d'environnement sont transmises aux programmes que le shell lance.

Il est très facile de définir une variable du shell. Pour cela, il suffit de lui affecter une valeur, à l'aide de la syntaxe suivante :

variable=valeur
: où variable est le nom de la variable à définir, et valeur est la valeur que l'on désire lui affecter. Notez qu'il n'est pas nécessaire de fournir une valeur. Dans ce cas, la variable ainsi définie sera vide.

Par exemple, la ligne suivante :

BONJOUR="Bonjour tout le monde \!"
: permet de définir la variable BONJOUR. Notez que la valeur est encadrée entre guillemets, car elle contient des espaces. Notez également que le caractère point d'exclamation ('!') est précédé d'un caractère d'échappement antislash ('\'), car il a une signification particulière pour le shell. Ce caractère d'échappement permet simplement de signaler au shell qu'il ne doit pas interpréter le caractère qui le suit, et fait donc en sorte que le point d'exclamation fasse partie de la chaînes de caractères à affecter à la variable. Bien entendu, le caractère antislash étant lui-même un caractère spécial pour le shell, il doit lui-même être préfixé d'un autre antislash si l'on désire l'utiliser dans une chaîne de caractères.

La valeur d'une variable peut être récupérée simplement en préfixant le nom de la variable du symbole dollar ('$'). Ainsi, la commande suivante permet d'afficher le contenu de la variable BONJOUR :

```
echo $BONJOUR
```

Les variables ainsi définies ne font partie que de l'environnement du shell, elles ne sont donc pas accessibles aux programmes que le shell lance. Donc, si l'on relance un nouveau shell avec la commande suivante :

bash
: et que l'on essaie de lire le contenu de la variable BONJOUR avec la commande **echo**, on obtient une chaîne vide. Cela est normal, puisque le deuxième shell (c'est-à-dire celui qui est en cours d'exécution) n'utilise pas le même environnement que le premier shell. Vous pouvez quitter le nouveau shell avec la commande suivante :

exit

Dès lors, vous serez à nouveau dans le shell initial, et la variable BONJOUR sera à nouveau accessible.

Pour rendre une variable du shell accessible aux programmes que celui-ci peut lancer, il faut l'exporter dans l'environnement d'exécution. Cela peut être réalisé avec la commande **export** :

export variable
où variable est le nom de la variable du shell à exporter dans l'environnement d'exécution.

La syntaxe précédente exporte de manière permanente les variables du shell. Mais il existe également une autre syntaxe, qui permet de ne définir des variables d'environnement que pour l'environnement d'exécution d'une seule commande. Cette syntaxe consiste simplement à préfixer la commande à exécuter par la définition de ladite variable. Par exemple, la commande suivante :

BONSOIR="Bonsoir tout le monde \!" bash
permet de lancer un shell et de lui communiquer la variable d'environnement BONSOIR. Cette variable ne sera définie que pour ce programme, si l'on quitte ce shell avec un **exit**, la variable BONSOIR ne sera plus définie.

Une variable peut être détruite à tout instant à l'aide de la commande **unset**. Cette commande prend en paramètre le nom de la variable à supprimer. Par exemple, la commande suivante supprime notre variable :

unset BONJOUR

Vous pouvez à tout moment visualiser l'ensemble des variables définies avec la commande **set**. Le tableau donné ci-dessous vous présentera les variables d'environnement les plus utilisées, que la plupart des programmes utilisent pour permettre à l'utilisateur de modifier leur comportement :

Tableau 4-1. Variables d'environnements courantes

Nom	Signification
HOME	Chemin du répertoire personnel de l'utilisateur.
USER	Nom de login de l'utilisateur. Cette information est également disponible au travers de la variable d'environnement LOGNAME.
TERM	Type de terminal utilisé. La valeur de cette variable sert aux applications pour déterminer les caractérisques du terminal et ses fonctionnalités afin d'optimiser leur affichage. La valeur de cette variable est souvent linux sur les consoles Linux, et xterm dans les émulateurs de

	terminal graphiques sous X11.
SHELL	Chemin sur le fichier de programme du shell actuellement utilisé. Sous Linux, il s'agit souvent du shell bash.
PATH	Liste des répertoires dans lesquels les programmes à exécuter seront recherchés. Cette liste ne doit pas contenir le répertoire courant (.) pour des raisons de sécurité de base (il suffit de placer un cheval de troie portant le nom d'une commande classique dans un répertoire pour que l'utilisateur le lance sans s'en rendre compte).
LD_LIBRARY_PATH	Liste des répertoires dans lesquels les bibliothèques dynamiques seront recherchées si elles ne sont pas trouvables dans les répertoires classiques des bibliothèques de programme du système.
C_INCLUDE_PATH	Liste des répertoires dans lesquels le compilateur C recherchera les fichiers d'en-tête lors de la compilation des fichiers sources C. Cette liste doit contenir les répertoires additionnels, qui ne sont pas déjà pris en compte automatiquement par le compilateur C.
CPLUS_INCLUDE_PATH	Liste des répertoires dans lesquels le compilateur C++ recherchera les fichiers d'en-tête lors de la compilation des fichiers sources C/C++. Cette liste doit contenir les répertoires additionnels, qui ne sont pas déjà pris en compte automatiquement par le compilateur C++.
LIBRARY_PATH	Liste des répertoires dans lesquels les bibliothèques à utiliser lors de l'édition de liens des programmes doivent être recherchées. Cette variable n'est utilisée que par les outils de développement lors de la compilation de fichiers sources et elle ne doit pas être confondue avec la variable d'environnement LD_LIBRARY_PATH, qui indique la liste des répertoires dans lequel l'éditeur de liens dynamiques recherchera les bibliothèques dynamiques utilisées par les programmes lors de leur chargement. Les notions de fichiers sources et de compilation seront détaillées dans le Chapitre 7.
TMPDIR	Répertoire des fichiers temporaires. Par défaut, le répertoire des fichiers temporaires est le répertoire /tmp/, mais il est possible d'en changer grâce à cette variable d'environnement.
TZ	Définition de la zone horaire de l'utilisateur. Le

	système travaillant exclusivement en temps universel, chaque utilisateur peut définir sa propre zone horaire pour obtenir l'affichage des dates et des heures dans son temps local. Le format de cette variable d'environnement est assez complexe. Il est constitué de plusieurs champs séparés par des espaces, représentant successivement le nom du fuseau horaire (au moins trois caractères), le décalage à ajouter à l'heure universelle pour obtenir l'heure locale, le nom du fuseau horaire pour l'heure d'été, le décalage pour l'heure d'été, et les dates de début et de fin de l'heure d'été. Les décalages horaires doivent être exprimés avec un '+' pour les fuseaux horaires placés à l'ouest de Greenwich, '-' pour ceux situés à l'est. Les dates de début et de fin de la période d'heure d'été peuvent être exprimés de deux manières différentes. La première méthode est d'indiquer le numéro du jour dans l'année après la lettre 'J'. Ce numéro ne doit pas tenir compte du 29 février, même pour les années bissextiles. La deuxième méthode est d'indiquer le mois de l'année, la semaine du mois et le jour de la semaine, séparés par des '.', et après la lettre 'M'. Les mois sont comptés de 1 à 12, les semaines de 1 à 5 et les jours de 0 à 6, 0 étant le dimanche. Seul le premier champ est obligatoire, et il est possible d'utiliser les noms de fuseaux horaires définis par la bibliothèque C. En France, on utilise normalement le fuseau CES (temps d'Europe centrale).
LANG	Nom de la locale à utiliser par défaut pour les paramètres d'internationalisation des applications. Cette valeur sera utilisée pour les paramètres qui n'en définissent pas une explicitement. Elle doit être composée de deux codes à deux caractères, le premier indiquant la langue, et le deuxième le pays (car plusieurs pays peuvent parler la même langue, et un pays peut avoir plusieurs langues nationales). Pour la France, on utilise normalement la valeur « fr_FR ». Cette valeur peut être redéfinie par l'une des variables d'environnement décrites ci-dessous.
LC_MESSAGES	Nom de la locale à utiliser pour déterminer la langue des messages. La valeur par défaut est spécifiée par la variable d'environnement LANG.
LC_TYPE	Nom de la locale à utiliser pour déterminer les règles de classification des caractères. La

	classification des caractères permet de dire si un caractère est un chiffre ou non, s'il est en majuscule ou en minuscule, etc. La valeur par défaut est spécifiée par la variable d'environnement LANG.
LC_COLLATE	Nom de la locale à utiliser pour déterminer les règles de comparaison des caractères. La comparaison des caractères est utilisée pour les tris lexicographiques (tri par ordre alphabétique par exemple). La valeur par défaut est spécifiée par la variable d'environnement LANG.
LC_MONETARY	Nom de la locale à utiliser pour déterminer l'emplacement et le caractère représentant le symbole monétaire du pays. La valeur par défaut est spécifiée par la variable d'environnement LANG.
LC_NUMERIC	Nom de la locale à utiliser pour déterminer les conventions locales d'écriture des nombres (séparateur décimal, format de la virgule, etc.). La valeur par défaut est spécifiée par la variable d'environnement LANG.

En résumé, le shell utilise les variables d'environnement du système pour gérer ses propres variables, et permet de les exporter vers l'environnement d'exécution qu'il communique aux commandes qu'il lance. Un grand nombre de variables d'environnement classiques sont reconnues par les programmes. Elles servent à paramétrer leur comportement. Nous reverrons ultérieurement quelques-unes de ces variables lors de la configuration du système de base.

5. Caractère d'échappement et chaînes de caractères

Un certain nombre de caractères sont interprétés par le shell d'une manière spéciale. Nous en avons déjà vu quelques-uns pour les redirections et les tubes, mais il en existe d'autres. Par conséquent, il faut utiliser une syntaxe particulière lorsqu'on désire utiliser un de ces caractères dans une commande du shell sans qu'il soit interprété par le shell. Pour cela, il suffit de faire précéder ces caractères du caractère d'échappement antislash (caractère de la barre oblique inverse, '\'). Ce caractère permet d'indiquer au shell que le caractère suivant doit être traité tel quel et ne doit pas être interprété avec son sens habituel. Par exemple, pour créer un répertoire nommé <, on utilisera la commande suivante :

mkdir \<

Bien entendu, le caractère antislash peut lui-même être précédé d'un autre antislash, lorsqu'on veut l'utiliser en tant que caractère normal.

Le caractère d'échappement antislash permet également, lorsqu'il est placé en fin de ligne, de supprimer le saut de ligne qui le suit. Cela signifie qu'il permet de

répartir une commande trop longue sur plusieurs lignes, à des fins de lisibilité. Vous trouverez quelques exemples de cette notation plus loin dans ce document, pour présenter des commandes trop longues pour tenir sur une page A4.

Il peut être relativement fastidieux de devoir taper des antislashs dans les chaînes de caractères qui contiennent beaucoup de caractères interprétables par le shell. C'est pour cela que le shell permet de définir des chaînes de caractères dont il ignore le contenu lors de l'analyse syntaxique. Ces chaînes de caractères sont simplement données entre guillemets simples (caractère '). Par exemple, la commande suivante :

MESSAGE='La syntaxe est A | B'

permet d'affecter la chaîne de caractères La syntaxe est A | B, contenant des espaces et le caractère | normalement utilisé par le shell pour les tubes, dans la variable d'environnement MESSAGE.

Note : Une chaîne de caractères commence par un guillemet et se termine par un guillemet. Les chaînes de caractères ne peuvent donc pas contenir de guillemet, même précédé d'un caractère d'échappement.

On veillera à ne surtout pas confondre les guillemets simples (caractère ') avec les guillemets inverses (caractère `). Ces deux caractères se ressemblent en effet énormément dans certaines polices de caractères, mais ont néanmoins une signification très différente. Le premier sert à définir des chaînes de caractères, et le deuxième à exécuter une commande et à en inclure le résultat dans une autre commande. Nous verrons plus loin comment utiliser ce type de guillemets.

Les guillemets simples sont donc très pratiques pour écrire simplement une chaîne de caractères, mais ne permettent pas de bénéficier des fonctionnalités de substitutions du shell, comme par exemple le remplacement d'une variable par sa valeur dans la chaîne de caractères. De plus, elles ne peuvent pas contenir de guillemets simples, puisque c'est leur caractère de terminaison. C'est pour ces raisons que le shell donne la possibilité de définir des chaînes de caractères plus souples, à l'aide des guillemets doubles (caractère "). Dans ces chaînes de caractères, la plupart des caractères normalement interprétés par le shell ne le sont plus, comme pour les chaînes de caractères utilisant les guillemets simples. Cependant, les caractères spéciaux $, ` et \ conservent leur signification initiale. Il est donc possible, par exemple, d'utiliser des variables d'environnement dans les chaînes de caractères de ce type :

```
echo "Mon nom est $USER"
```

Le caractère d'échappement antislash peut toujours être utilisé, en particulier pour insérer un caractère de guillemets doubles dans une chaîne de caractères. En effet, ce caractère marquerait la fin de la chaîne de caractères s'il n'était pas précédé d'un antislash.

Note : Remarquez que les guillemets et les caractères d'échappement ne sont utilisés que pour l'analyse de la ligne de commande. Une fois toutes les chaînes de caractères et toutes les substitutions traitées, les guillemets et les caractères d'échappement inutiles sont supprimés. En pratique, ce sont tous les caractères

d'échappement et les guillemets qui restent après traitement de la ligne de commande et qui ne font pas partie du résultat d'une des substitutions. Ainsi, la commande suivante :

echo "Bonjour tout le monde"
: a pour but de passer la chaîne de caractères Bonjour tout le monde en tant que premier (et unique) paramètre de la commande **echo**, puis de l'exécuter. Les guillemets ne font pas partie de la chaîne de caractères, ils ont été supprimés par le shell et seul le contenu de la chaîne sera effectivement affiché.

Notez que la commande précédente est très différente de celle-ci :

echo Bonjour tout le monde
: même si le résultat est le même. En effet, cette dernière commande passe les chaînes de caractères Bonjour, tout, le et monde en tant que paramètres (4 au total) à la commande **echo**, alors que l'utilisation des guillemets permet de passer toute la phrase en un seul paramètre. On peut voir la différence en utilisant plus d'un espace entre chaque mot : les espaces superflus ne sont conservés que dans la première commande.

6. Les substitutions

L'une des fonctionnalités les plus puissantes du shell est sans doute sa capacité à effectuer des substitutions d'expressions par leur valeur. L'une des substitutions les plus courantes est sans doute le remplacement d'une variable par sa valeur, mais le shell peut faire beaucoup plus que cela. Les lignes de commandes peuvent être écrites en utilisant différents types d'expressions spéciales, qui seront remplacées par leur valeur par le shell avant l'exécution de la commande. Ces expressions permettent de spécifier des motifs de chaîne de caractères, d'exprimer des chemins partiels sur des fichiers ou des répertoires, de récupérer la valeur des variables du shell, et de calculer des expressions mathématiques, voire d'inclure le résultat d'une autre commande dans la ligne de commande en cours.

Les mécanismes des substitutions décrits ci-dessous sont présentés par ordre de priorité décroissante. Cela signifie que si une expression substituable contient elle-même une autre expression substituable de priorité inférieure, cette expression sera remplacée après la substitution de l'expression contenante.

6.1. Génération de chaînes de caractères selon un motif

Il est possible de demander au shell de générer une série de chaînes de caractères selon un motif simple. Ce motif est toujours constitué d'un préfixe, suivi d'une partie variable, suivie d'un suffixe. La partie variable du motif est celle qui subira les substitutions pour générer une liste de chaînes de caractères commençant par le préfixe suivi du résultat de la substitution et se terminant par le suffixe. Cette partie variable doit être spécifié entre accolades, et prend la forme d'une liste de valeurs possibles pour chaque substitution, séparées par des

virgules. Par exemple, la commande suivante :

ls test{0,1,2,3,4}
sera transformée par le shell en la commande suivante :
ls test0 test1 test2 test3 test4

Note : Ceux qui se souviennent un peu de leurs mathématiques se diront qu'il s'agit là d'une factorisation. C'est rigoureusement exact.

6.2. Substitution du nom d'utilisateur

Le caractère tilde ('~') est remplacé par le nom de l'utilisateur courant ou, à défaut de nom, par le chemin sur le répertoire personnel de cet utilisateur. Il est possible de spécifier un autre utilisateur en donnant le nom de login de cet autre utilisateur immédiatement après le caractère tilde. Par exemple, la commande suivante :

`cp *.txt ~rachid`
permet de copier tous les fichiers d'extension `.txt` dans le répertoire personnel de l'utilisateur `rachid`.

6.3. Remplacements de variables

Comme il l'a déjà été indiqué plus haut, la valeur des variables du shell et des variables d'environnement peut être récupérée en préfixant le nom de la variable par le caractère dollar ('$'). En fait, cette écriture est l'une des formes les plus simples que peuvent prendre les substitutions de paramètres. En effet, il est possible de remplacer l'expression par une partie seulement de la valeur de la variable, ou une par une autre valeur calculée à partir de celle de la variable.

En pratique, les expressions utilisées par les substitutions de variables peuvent être relativement compliquées, et il peut être nécessaire de les isoler du reste de la ligne de commande à l'aide d'accolades. La syntaxe exacte complète de ce type de substitution est donc la suivante :

${expression}
où expression est l'expression qui définit la chaîne de remplacement à utiliser.

Si cette expression est un nom de variable, ce sera le contenu de cette variable qui sera utilisé pour la substitution. Il est possible de fournir une valeur par défaut pour le cas où cette variable ne contient rien ou n'est pas définie. Pour cela, on utilisera la syntaxe suivante :

${variable:-valeur}
où valeur est la valeur par défaut à utiliser dans ce cas. Notez que la variable reste indéfinie après la substitution. Pour fixer la valeur de la variable à cette valeur par défaut en plus d'effectuer la substitution, on utilisera plutôt la syntaxe suivante :
${variable:=valeur}
valeur a toujours la même signification dans cette syntaxe.

Il est parfois préférable d'afficher un message d'erreur plutôt que de donner une valeur par défaut lorsqu'une variable n'est pas définie. Cela peut se faire avec la syntaxe suivante :

${variable:?message}
où message est le message à afficher dans le cas où la variable variable est non définie ou de valeur nulle.

Si l'on veut tester si une variable est non définie et renvoyer une valeur spécifique si elle est définie, on utilisera la syntaxe suivante :

${variable:+valeur}
où valeur est la valeur à renvoyer si la variable est définie. Si la variable n'est pas définie, la substitution sera faite avec la chaîne de caractères vide (l'expression complète sera donc supprimée).

Le shell permet également de faire la substitution avec une sous-chaîne de la valeur de la variable, à partir d'une position donnée et d'une longueur. La syntaxe à utiliser est donnée ci-dessous :

${variable:position:longueur}
où position est la position à laquelle commence la sous-chaîne à extraire, et longueur est le nombre de caractères à extraire. Ce dernier champ est facultatif (on ne mettra pas non plus les deux-points précédents si on décide de ne pas spécifier de longueur). Si on ne le précise pas, la sous-chaîne extraite sera constituée du reste de la valeur de la variable à partir de la position indiquée. La position quant à elle doit être positive ou nulle. Une valeur négative indique un point de départ correspondant au nombre de caractères correspondant à partir de la droite de la valeur de la variable. Si l'on veut obtenir la longueur d'une chaîne de caractères contenue dans une variable, on utilisera cette syntaxe :
${#variable}
où variable est toujours le nom de la variable.

Il est également possible de considérer que la valeur d'une variable est une chaîne de caractère préfixée d'une autre chaîne de caractères particulière. Le shell permet d'extraire la chaîne de caractères principale, en supprimant ce préfixe. Pour réaliser cette opération, on utilisera l'une des syntaxes suivantes :

${variable#préfixe}
ou :
${variable##préfixe}
où variable est la variable contenant la chaîne de caractères à traiter, et préfixe est le préfixe à supprimer.

En fait, le préfixe peut être spécifié à l'aide d'un motif de caractères. Ce motif peut correspondre à une partie plus ou moins grande de la valeur de la variable. Dans ce cas, il y a plusieurs manières d'interpréter ce motif, et donc plusieurs choix de préfixes possibles à supprimer. La première syntaxe devra être utilisée lorsqu'on désire supprimer le plus petit préfixe possible correspondant au motif.

La deuxième syntaxe, quant à elle, permettra de supprimer le préfixe le plus long. Par exemple, si la variable VAR contient la chaîne de caractères abbbc, la commande suivante :

echo ${VAR#a*b}
affichera la chaîne de caractères bbc, car le plus petit préfixe correspondant au motif a*b est ab. Inversement, la commande :
echo ${VAR##a*b}
utilisera le préfixe le plus long, à savoir abbb. Le résultat de cette substitution sera donc la chaîne de caractères c. La syntaxe des motifs de caractères utilisés ici sera précisée dans le chapitre 7.

Le shell fournit une syntaxe similaire pour extraire des suffixes de la valeur des variables. Cette syntaxe utilise simplement le caractère % au lieu du caractère #. Comme pour les préfixes, le fait de doubler ce caractère implique que le suffixe le plus long correspondant au motif sera utilisé, alors que l'utilisation d'un seul % permet de choisir le suffixe le plus court. Ainsi, la commande :

echo ${VAR%b*c}
affichera la chaîne de caractères abb, alors que la commande :
echo ${VAR%%b*c}
n'affichera que a.

Pour terminer ce tour d'horizon des remplacements de variables, nous allons voir les possibilités de recherche et de remplacement du shell dans les chaînes de caractères contenues dans des variables. La syntaxe suivante :

${variable/motif/remplacement}
permet de rechercher la plus grande sous-chaîne de caractères correspondant au motif motif dans la chaîne contenue dans la variable variable, et de remplacer cette sous-chaîne par la chaîne de caractères remplacement. Par exemple, si la variable VAR contient la chaîne de caractères abab, la commande suivante :
echo ${VAR/b/d}
affichera la chaîne de caractères adab.

Ce remplacement n'est donc effectué qu'une seule fois. Si l'on veut que toutes les occurrences du motif soient remplacées par la chaîne de remplacement, il suffit de doubler le premier / :

${variable//motif/remplacement}
Dans les deux syntaxes, la présence du champ remplacement est facultative. Cela permet de supprimer purement et simplement les sous-chaînes de caractères qui correspondent au motif.

La syntaxe des motifs sera détaillée dans le chapitre 7. Cependant, une précision doit être signalée : si le motif commence par le caractère #, il sera obligatoirement recherché au début de la chaîne de caractères contenue dans la variable. De même, si le motif commence par le caractère %, il sera obligatoirement recherché à la fin de cette chaîne. Ces deux notations permettent d'obtenir le même effet que les suppressions de préfixes et de

suffixes présentées plus haut.

6.4. Substitution du résultat d'une commande

Le shell peut évaluer une commande apparaissant dans une expression afin de la remplacer par son résultat dans la commande appelante. Il existe deux syntaxes pour réaliser ce type de substitutions. La première, et la plus classique (voire historique), utilise des guillemets inverses :

`commande`
: où commande est la commande devant être remplacée par son résultat (c'est-à-dire ce qu'elle enverra ce résultat sur le flux standard de sortie). Pour donner un exemple, la commande suivante :

kill `cat /var/pid/p.pid`
: a pour résultat de lancer un signal SIGTERM au processus dont le PID est stocké dans le fichier /var/pid/p.pid. La commande **cat** est utilisée pour afficher le contenu de ce fichier, et elle est substituée par ce contenu. En fin de compte, la commande **kill** est appliqué au PID affiché par **cat**.

La deuxième syntaxe utilisable est la suivante :

$(commande)
: où commande est toujours la commande à exécuter et à substituer. La différence entre ces deux syntaxes est que, dans le premier cas, les caractères $, ` et \ sont toujours interprétés par le shell et doivent être précédés d'un antislash s'ils doivent apparaître tels quels dans la commande à substituer, alors que, dans le deuxième cas, on peut utiliser tous les caractères sans protection particulière (sauf, bien entendu, la parenthèse fermante, puisqu'elle marque la fin de la commande).

6.5. Évaluation d'expressions arithmétiques

En général, le shell ne manipule que des chaînes de caractères. Cependant, il est capable d'évaluer des expressions mathématiques simples faisant intervenir des entiers. Pour cela, il faut utiliser la syntaxe suivante :

$((expression))
: où expression est l'expression à évaluer.

Les expressions mathématiques peuvent contenir tous les opérateurs classiques du langage C : addition, soustraction, multiplication et division. Il existe en plus un opérateur d'élévation à la puissance, représenté par une double étoile (**). Les opérateurs de décalage binaires vers la gauche (<<) et la droite (>>) sont également utilisables, ainsi que les opérateurs de manipulation de bits & (« ET binaire »), | (« OU binaire »), ^ (« OU binaire exclusif ») et ~ (« négation binaire »).

Comme en C, les comparaisons logiques peuvent également être évaluées, elles ont la valeur 1 lorsque l'expression qui les utilise est vraie, et 0 dans le cas contraire. Les opérateurs disponibles sont ==, <, <=, >, >= et !=. Les tests

peuvent être composés à l'aide des opérateurs && (« ET logique ») et || (« OU logique »).

Les divers opérateurs d'affectation du langage C +=, -=, etc. sont également disponibles.

6.6. Substitution de commandes

Nous avons déjà vu qu'il était possible de récupérer le résultat d'une commande et de l'injecter dans le flux d'entrée standard d'un processus par l'intermédiaire d'un pipe (cf. Section 3.2). Mais le shell fournit une généralisation de cette fonctionnalité sous la forme des substitutions de commandes. Ces substitutions permettent de lancer une commande et de remplacer son expression par un pipe nommé, grâce auquel on peut communiquer avec le processus qui exécute la commande.

La syntaxe utilisée par les substitutions de commande est similaire à celle des redirections classiques :

<(command)
ou :
>(command)
où command est la commande à substituer.

Note : Attention à ne pas mettre d'espace entre le caractère de redirection et la parenthèse ouvrante, faute de quoi le shell signalera une erreur.

La première syntaxe permet de lancer une commande en arrière-plan en redirigeant son flux standard de sortie vers un descripteurs de fichiers du shell. Le résultat de cette substitution est le nom du fichier /dev/fd/n, permettant de lire les données écrites par la commande dans ce descripteur de fichier. En pratique, on utilise donc cette substitution en lieu et place d'un fichier d'entrée pour une commande normale. La deuxième commande permet de lancer également une commande en arrière-plan, mais en redirigeant le flux d'entrée standard de cette commande cette fois. Il est alors possible de fournir les données nécessaires à cette commande en écrivant dans le fichier /dev/fd/n dont le nom est fourni par le résultat de la substitution.

Ces deux commandes permettent donc de simplifier l'usage des pipes nommés, en évitant d'avoir à créer un fichier de tube nommé manuellement et d'avoir à lancer les deux commandes devant se servir de ce tube pour communiquer. Ainsi, la commande suivante :

cat <(ls)
est fonctionnellement équivalente à la série de commandes suivante :

```
mkfifo /tmp/lsfifo
ls > /tmp/lsfifo
cat /tmp/lsfifo
rm /tmp/lsfifo
```

Les substitutions de commandes sont donc nettement plus pratiques et plus sûres, car elles n'imposent pas la création d'un fichier de pipe

nommé dont le nom peut être choisi arbitrairement.

6.7. Découpage en mots

Les résultats provenant des substitutions vues précédemment sont systématiquement décomposés en série de mots par le shell avant de poursuivre le traitement de la ligne de commande. Cela signifie que les résultats de substitutions sont analysés pour identifier les mots qu'ils contiennent, en se basant sur la notion de séparateur. Par défaut, les séparateurs utilisés sont l'espace, le caractère de tabulation et le retour de ligne, mais il est possible de spécifier des séparateurs différents à l'aide de la variable d'environnement IFS (abréviation de l'anglais « Internal Field Separator »).

Par exemple, le résultat de la commande **ls** dans la commande suivante :

echo `ls`
: est une chaîne de caractères contenant la liste des fichiers du répertoire courant, chacun étant séparé du suivant par un caractère de saut de ligne. La substitution du résultat de cette commande est donc soumise au découpage en mots, et chaque caractère de retour à la ligne est interprété comme un séparateur. Par conséquent, cette chaîne de caractères est transformée en une liste de mots, chacun de ces mots étant un des noms de fichiers renvoyés par la commande **ls**. Au final, la commande **echo** est appelée, avec comme paramètres ces noms de fichiers, à raison d'un par paramètre. Les noms de fichiers sont donc affichés sur une seule ligne.

Note : Ce découpage en mot est effectué automatiquement par le shell à la suite des substitutions vues précédemment. Cela signifie en particulier que s'il n'y a pas de substitution, il n'y a pas de découpage en mots non plus.

6.8. Remplacement des caractères génériques

Si, après avoir appliqué toutes les formes de substitutions précédentes, le shell trouve des caractères génériques * et ? dans l'expression en cours de traitement, il interprétera la partie de l'expression contenant ces caractères comme un motif représentant des chemins de fichier Unix. Les caractères * et ? auront donc le comportement que l'on a déjà décrit . Ce motif sera donc remplacé par autant de chemins Unix lui correspondant que possible. Rappelons que le caractère générique * représente 0 ou plusieurs caractères quelconques, et que le caractère générique ? représente un caractère et un seul. Les chemins générés sont classés par ordre alphabétique.

Il est possible également de restreindre le jeu de caractères utilisé par le shell pour rechercher les noms de fichiers correspondants au motif. Pour cela, il faut lui indiquer un ensemble de caractères ou de plages de caractères utilisables, séparés par des virgules, et entre crochets. Les plages de caractères sont spécifiées en indiquant le premier et le dernier caractère, séparés par un tiret. Par exemple, la commande suivante :

ls [a-c,m-t]*.txt

permet d'afficher tous les fichiers dont le nom commence par les lettres a, b, c et les lettres allant de m à t, et dont l'extension est .txt. Vous trouverez de plus amples renseignements sur la syntaxe de ces motifs dans le chapitre 7.

Sauf paramétrage pour indiquer explicitement de faire le contraire, le shell ignore systématiquement les répertoires . et .. dans les substitutions. Cela est très important. En effet, une commande utilisant le caractère générique * ne s'appliquera pas, par défaut, sur le répertoire courant et le répertoire parent. Paramétrer bash pour qu'il prenne en compte ces répertoires peut être extrêmement dangereux, surtout avec une commande telle que rm -f *, qui dans ce cas effacerait également les répertoires parents en plus du contenu du répertoire courant !

7. Les expressions rationnelles

Les substitutions de variables et de noms de fichiers utilisent des motifs pour identifier des chaînes de caractères. Ces motifs peuvent être reconnus dans plusieurs chaînes de caractères différentes, car ils contiennent une ou plusieurs parties variables qui pourront représenter chacune une sous-chaîne des chaînes qui vérifient ce motif. Par exemple, le motif a*b représente toute chaîne de caractères commençant par un a et se terminant par un b. La sous-chaîne située entre ces deux caractères peut être quelconque, et constitue la partie variable du motif.

La syntaxe utilisée pour définir les motifs de chaînes de caractères dans le shell bash est un sous-ensemble d'un langage plus complexe permettant de décrire ce que l'on appelle les *expressions rationnelles* (l'usage dit également « *expressions régulières* »). Le langage des expressions rationnelles est relativement compliqué, mais extrêmement puissant. Ce langage permet d'identifier avec précision des sous-chaînes de caractères dans un chaîne de caractères à l'aide des parties variables des expressions rationnelles, et permet éventuellement de remplacer ces sous-chaînes par des chaînes de substitutions. Malheureusement, la description des expressions rationnelles pourrait prendre plusieurs pages, aussi ne verrons-nous ici que les expressions utilisables dans les substitutions du shell bash.

Comme vous l'avez sans doute déjà deviné au travers des exemples précédents, le caractère '*' permet d'identifier une quelconque chaîne de caractères, y compris la chaîne vide. Utilisé dans les expressions rationnelles, il constitue la partie variable principale de ces expressions. De la même manière, le caractère '?' représente un et un seul caractère quelconque. Ce caractère sera donc utilisé quand on désirera contrôler la taille de la partie variable d'une expression rationnelle, éventuellement en le répétant un certain nombre de fois.

Les deux caractères de substitutions précédents peuvent contenir n'importe quel caractère, ce qui peut parfois ne pas être assez restrictif dans la définition d'un motif. Le shell fournit donc une syntaxe plus évoluée, permettant de définir précisément le jeu de caractère auquel un caractère du motif doit appartenir.

Cette syntaxe consiste simplement à donner la liste des caractères du jeu de caractères entre crochets :

[...]
: Les points de suspension représentent ici l'ensemble des caractères qui peuvent apparaître dans le motif ainsi défini. Notez que dans le cas d'une suite de caractères, il suffit de spécifier le premier et le dernier caractère, et de les séparer par un trait d'union (caractère '-'). Ainsi, le motif suivant :

[a-egt]
: représente n'importe lequel des caractères de 'a' à 'e', plus les caractères 'g' et 't'.

Note : Pour spécifier le caractère - lui-même, il suffit de le placer tout seul au début ou à la fin de la liste de caractères spécifiée entre les crochets. De même, pour spécifier le caractère ']' lui-même (normalement utilisé pour marquer la fin du jeu de caractères), il faut le placer au début de la liste, juste après le crochet ouvrant.

Pour finir, sachez que le shell bash est également capable de prendre en charge des expressions rationnelles plus complexes que celles présentées ici. Cependant, ces expressions ne sont pas actives par défaut, et ne sont donc accessibles qu'en activant une option complémentaire du shell. Ces extensions ne seront pas décrites ici, mais vous pouvez consulter la page de manuel de bash si vous désirez en savoir plus à ce sujet.

8. Structures de contrôle

Tout langage de programmation digne de ce nom dispose de structures de contrôles évoluées permettant de contrôler l'exécution du programme, de réaliser des boucles et de structurer l'ensemble d'un programme. Le shell n'échappe pas à la règle, et fournit la plupart des constructions classiques. Cette section a pour but d'exposer leurs syntaxes.

8.1. Les instructions composées

Dans le langage du shell, une instruction se termine soit par un retour à la ligne (non précédé d'un antislash), soit par un point-virgule. Les instructions peuvent être pourtant très complexes, car elles peuvent contenir des tubes et des redirections. En fait, une instruction peut à peu près être définie comme étant une ligne de commande normale du shell.

Le shell permet bien entendu de réaliser des instructions composées, afin de regrouper plusieurs traitements dans un même bloc d'instructions. La méthode la plus simple pour réaliser un bloc d'instructions est tout simplement de les regrouper sur plusieurs lignes, ou de les séparer par des points-virgules, entre accolades. Par exemple, les instructions suivantes constituent un bloc d'instructions :

```
{
```

```
   cd /tmp
   rm *.bak
}
```

Notez que l'accolade fermante est considérée comme une instruction à part entière. Cela signifie que si l'on ne met pas l'accolade fermante sur une ligne indépendante, il faut faire précéder l'instruction précédente d'un point-virgule. De même, il faut le faire suivre d'un autre point-virgule s'il ne se trouve pas à la fin d'une ligne.

Les instructions des instructions composées créées à l'aide des accolades sont exécutées au sein du shell courant. Les variables qu'elles définissent, ainsi que les changements de répertoires, sont donc toujours valides à l'issue de l'exécution de ces instructions. Si cela n'est pas désirable, on pourra créer des instructions composées à l'aide de parenthèses. Les instructions seront alors exécutées dans un autre shell, lancé pour l'occasion, et elles n'auront donc pas d'effet de bord imprévu dans le shell appelant. Par exemple, le répertoire courant à l'issue de l'instruction composée précédente est le répertoire /tmp/, alors que l'instruction composée suivante :

```
(
   cd /tmp
   rm *.bak
)
```

ne change pas le répertoire courant.

Note : On ne confondra pas les instructions composées utilisant des parenthèses et les substitutions de résultat de commande. Les instructions composées renvoient le code d'erreur de la dernière instruction exécutée, alors que le résultat des substitutions est ce que la commande a écrit sur son flux de sortie standard.

Le shell permet également de réaliser des instructions composées conditionnelles, où l'exécution de chaque instruction de l'instruction composée est conditionnée par le résultat de l'instruction précédente. Ces instructions composées sont définies à l'aide des opérateurs || et &&. La syntaxe de ces opérateurs est la même :

```
command1 || command2
command1 && command2
```

où command1 et command2 sont deux commandes du shell (composées ou non). Avec l'opérateur ||, la commande command2 n'est exécutée que si le code de retour de la commande command1 est non nul, ou, autrement dit, si cette commande ne s'est pas exécutée correctement. Inversement, avec l'opérateur &&, la commande command2 n'est exécutée que si la première commande s'est exécutée correctement (et renvoie donc un code de retour nul). Par exemple, la commande suivante :

```
rm *.txt 2> /dev/null || echo "Aucun fichier à supprimer"
```

permet d'effacer tous les fichiers d'extension .txt, ou d'afficher le message d'erreur « Aucun fichier à supprimer » s'il n'existe pas de fichier ayant une telle extension.

Les instructions composées peuvent être utilisées comme n'importe quelle commande normale. En particulier, elles peuvent être utilisées dans des commandes plus complexes, par exemple comme destination d'un tube.

8.2. Les tests

Sous Unix, chaque processus reçoit plusieurs valeurs en paramètres et renvoie un code de retour. La plupart des paramètres sont passés en ligne de commande, et sont récupérés directement par le processus, mais d'autres paramètres peuvent être fournis par le processus appelant par l'intermédiaire de variables d'environnement et de descripteurs de fichiers. Le code de retour, quant à lui, est un entier signalant si l'exécution du processus s'est terminée correctement ou si des erreurs ont eu lieu. Si les codes d'erreurs varient grandement d'un programme à un autre, la valeur 0 signifie toujours, et ce quel que soit le programme, que l'exécution s'est déroulée correctement.

Il est possible de tester le code de retour d'une commande avec l'instruction if. La syntaxe la plus simple pour un test est la suivante :

```
if commande ; then
   action
fi
```

où commande est la commande dont on désire tester le code de retour, et action est la commande à exécuter si ce code vaut 0 (c'est-à-dire, si la commande commande s'est exécutée correctement).

Il peut paraître réducteur de ne pouvoir tester que le code de retour d'une commande. Mais en fait, c'est une fonctionnalité très puissante du shell, car elle permet de réaliser tous les types de tests imaginables. En effet, il existe une commande spéciale, **[**, qui permet de réaliser divers types de tests sur les paramètres qu'on lui passe, et d'ajuster son code d'erreur en conséquence. Par exemple, pour tester l'égalité d'une variable d'environnement avec une chaîne de caractères, on utilisera la syntaxe suivante :

```
if [ $variable == valeur ] ; then
   action
fi
```

Notez que dans cette syntaxe, le test effectué est une commande complète. Cela implique qu'il faut mettre une espace entre chaque paramètre, et en particulier entre le nom de la commande ([), le premier opérande ($variable), l'opérateur utilisé (==), le deuxième opérande (valeur) et le caractère de marque de fin de test (]).

La commande **[** est capable d'effectuer tous les tests standards. Par défaut, elle considère que les deux opérandes du test sont des chaînes de caractères, et elle utilise l'ordre lexicographique pour les comparer. Les tests d'égalité et d'inégalité sont effectués respectivement avec les opérateurs == et !=. Les opérateurs d'antériorité dans l'ordre lexicographique sont < et <=, et les opérateurs de postériorité sont > et >=. Notez que l'utilisation de ces opérateurs peut être

relativement pénible, parce que les caractères < et > sont interprétés par le shell en tant que redirections. Par conséquent, il faut souvent les précéder du caractère d'échappement antislash.

L'ordre lexicographique convient dans la plupart des cas, mais il n'est pas très approprié pour la comparaison de valeurs numériques. Par exemple, le test suivant :

```
if [ -1 \< -2 ] ; then
   echo "-1 est plus petit que -2"
fi
```

est vérifié, car le caractère 1 précède le caractère 2 dans l'ordre lexicographique. La commande **[** fournit donc la possibilité d'utiliser une autre syntaxe pour comparer les entiers. Cette syntaxe utilise les options lt et gt respectivement pour les tests d'infériorité stricte et de supériorité stricte, et les options le et ge respectivement pour les tests d'infériorité et de supériorité ou d'égalité. Ainsi, le test :

```
if [ $i -gt 3 ] ; then
   echo "$i est supérieur à 3"
fi
```

permet de comparer la valeur entière de la variable i avec le nombre 3.

Nous avons vu dans la Section 8.1 que les opérateurs || et && permettent de tester le code de retour d'une commande, et qu'en fonction de la valeur de ce code de retour, d'exécuter ou non la commande suivante. La syntaxe de ces opérateurs provient en fait de la possibilité de les employer pour effectuer des tests complexes avec l'instruction if. Par exemple, pour effectuer un ET logique entre deux tests, on utilisera la syntaxe suivante :

```
if [ $i == "A" ] && [ $j -lt 3 ] ; then
   echo "i contient la lettre \"A\" et j contient un nombre inférieur à 3"
fi
```

Notez que la deuxième commande **[** n'est exécutée que si le premier test est vérifié. L'utilisation de l'opérateur || se fait selon le même principe. Il est bien entendu possible de regrouper plusieurs commandes de test ensemble, à l'aide de parenthèses.

Comme dans la plupart des langages informatiques, l'instruction if peut prendre une forme plus complexe pour traiter les cas où le test n'est pas vérifié. Ainsi, pour exécuter une action spécifique pour le cas où le test serait faux, on peut utiliser la syntaxe suivante :

```
if commande ; then
   action1
else
   action2
fi
```

où commande est toujours la commande dont le code de retour sera testé, action1 est l'action qui doit être réalisée si cette commande a renvoyé le code de retour 0, et action2 la commande à exécuter dans le cas contraire. De même, si l'on veut enchaîner des tests, on utilisera le

mot clé elif. La syntaxe générale du test est donc la suivante :

```
if commande1 ; then
   action1
elif commande2 ; then
   action2
elif commande3 ; then
   ...
else
   actionn
fi
```

Note : Pour des raisons d'optimisation, le shell peut simuler le comportement du programme **[**, et éviter ainsi de le lancer à chaque fois qu'il a à faire un test. Cependant, le principe originel était bien celui décrit ci-dessus. Cette description, bien que n'étant plus tout à fait exact, permet de mieux comprendre la syntaxe du shell.

Il est possible de récupérer la valeur du code de retour de la dernière commande exécutée grâce à la variable spéciale $?. Cependant, il est très rare d'avoir à manipuler cette valeur directement, car les structures de contrôle du shell telles que if permettent d'effectuer les actions qui s'imposent sans avoir à la connaître.

Pour ceux qui savent programmer en C, sachez que le code de retour est la valeur renvoyée par la fonction C exit ou par l'instruction return de la fonction principale main. Les paramètres de la ligne de commande, quant à eux, sont récupérables par l'intermédiaire des paramètres de la fonction principale main.

Il ne faut pas oublier que la fonction première du shell est de permettre les manipulations de fichiers. Il n'est donc pas étonnant que la commande **[** permette également de réaliser tous les tests imaginables sur les fichiers. Ces tests vont de l'existence d'un fichier au test de sa nature et de ses attributs, en passant par les tests sur l'identité de son propriétaire et de son groupe. La syntaxe générale de ces tests est la suivante :

```
if [ option fichier ] ; then
   .
   .
   .
fi
```

où option est une option de la commande **[** décrivant la propriété testée, et fichier est le nom du fichier sur lequel le test doit porter.

Les principales options utilisables dans les tests sur les fichiers sont récapitulées dans le tableau ci-dessous :

Tableau 8-1. Tests sur les fichiers

Option	Signification
-e	Test d'existence d'un fichier ou d'un répertoire.
-d	Test d'existence d'un répertoire.
-f	Test d'existence d'un fichier normal.
-s	Test d'existence d'un fichier et vérification que sa taille est non nulle.
-L	Test d'existence d'un lien symbolique.
-b	Test d'existence d'un fichier spécial de périphérique de type bloc (disque dur, CD-ROM, lecteur de cassettes, etc.).
-c	Test d'existence d'un fichier spécial de périphérique de type caractère (port série, port parallèle, carte son...).
-p	Test d'existence d'un tube.
-r	Test d'existence du fichier et d'accessibilité en lecture de ce fichier.
-w	Test d'existence du fichier et d'accessibilité en écriture de ce fichier
-x	Test d'existence du fichier et de possibilité d'exécution de ce fichier.
-g	Test d'existence du fichier et de présence du bit setgid sur ce fichier.
-u	Test d'existence du fichier et de présence du bit setuid sur ce fichier
-k	Test d'existence du fichier et de présence du bit sticky sur ce fichier.
-O	Test d'existence du fichier et d'appartenance de ce fichier à l'utilisateur effectif courant.
-G	Test d'existence du fichier et d'appartenance de ce fichier au groupe effectif courant.
-N	Test d'existence du fichier et de modification de ce fichier depuis la dernière fois qu'il a été lu.

Note : Ce tableau n'est pas exhaustif, mais les options les plus importantes et les plus utilisées s'y trouvent.

La commande **[** accepte également les options -nt et -ot, qui permettent respectivement de tester si un fichier est plus récent ou plus vieux qu'un autre, en se basant sur les dates de dernière modification de ces fichiers. Ces deux opérateurs s'utilisent avec la syntaxe suivante :

```
if [ fichier1 option fichier2 ] ; then
    .
    .
    .
fi
```

où fichier1 et fichier2 sont les deux fichiers sur lesquels la comparaison doit porter, et option est l'une des options -nt ou -ot.

8.3. Le branchement conditionnel

Lorsqu'on veut effectuer différentes opérations selon la valeur d'une variable, l'instruction if peut devenir très lourde à utiliser. En effet, si le nombre de valeurs différentes est grand, elle peut conduire à écrire un grand nombre de tests. Le shell fournit donc une instruction de branchement conditionnel, qui permet de spécifier quelle action doit être prise pour chaque valeur de la variable.

Le branchement conditionnel s'utilise de la manière suivante :

```
case valeur in
   ( motif1 | motif2 | ... ) commande1 ;;
   ( motifn | motifn+1 | ... ) commande2 ;;
    .
    .
    .
esac
```

où motif1, motif2... motifn+1 sont des motifs spécifiant les valeurs possibles pour la valeur valeur, et commande1, commande2, etc. sont les commandes à exécuter pour les valeurs de ces motifs.

La commande exécutée est la première commande pour laquelle la variable correspond à l'un de ses motifs correspondants. Une fois exécutée, la recherche se termine, et l'exécution reprend à la suite du branchement conditionnel. Par exemple ce branchement conditionnel :

```
case $i in
   ( *.txt ) echo "$i est un fichier texte" ;;
   ( *.gz ) echo "$i est compressé avec gzip" ;;
   ( *.tar ) echo "$i est une archive" ;;
esac
```

affiche la nature du fichier dont le nom est stocké dans la variable i à partir de son extension.

Le code de retour du branchement conditionnel est 0 si la variable ne correspond à aucun des motifs, ou le code de retour de la commande exécutée sinon.

8.4. Les boucles

Il existe deux types de boucles : le while et le until. La syntaxe des boucles while est la suivante :

```
while commande ; do
   action
done
```

où commande est une commande dont le code de retour est utilisé comme critère de la fin de la boucle, et action est l'instruction (composée ou non) exécutée à chaque itération de la boucle. Comme on le voit, le shell utilise le même principe pour les boucles que pour les tests pour évaluer une condition. Tant que la commande commande renvoie un code de retour égal à 0, l'instruction action est exécutée.

L'instruction until utilise la même syntaxe que l'instruction while :

```
until commande ; do
   action
done
```

à ceci près que l'instruction action est exécutée tant que la commande commande renvoie un code de retour non nul. L'instruction until utilise donc simplement le test inverse de celui de l'instruction while.

Bien entendu, il est possible d'utiliser la commande **[** pour effectuer des tests plus complexes que le simple test du code de retour d'une commande. Par exemple, la boucle suivante calcule la somme des dix premiers entiers :

```
result=0
i=0
while [ $i -le 10 ] ; do
   result=$(($result + $i))
   i=$(($i + 1))
done
echo $result
```

8.5. Les itérations

Les itérations sont des boucles qui s'exécutent pour chaque élément d'un ensemble donné. Le shell gère les itérations par l'intermédiaire de l'instruction for. La syntaxe de cette instruction est la suivante :

```
for variable [ in ensemble ] ; do
   action
done
```

où variable est un nom de la variable utilisée pour l'itération, ensemble est l'ensemble des valeurs que peut prendre cette variable, et action est la commande (simple ou composée) à exécuter pour chaque valeur de cette variable.

Le principe des itérations est très simple. Pour chaque valeur indiquée dans l'ensemble des valeurs, la commande est exécutée, avec la valeur en question accessible dans la variable utilisée pour l'itération. Par exemple, la commande suivante :

```
for i in *.txt ; do
   mv $i ${i/%.txt/.doc}
done
```

permet de renommer tous les fichiers portant l'extension .txt en fichier du même nom, mais avec l'extension .doc.

Il n'est pas nécessaire de préciser l'ensemble des valeurs que peut prendre la variable. Dans ce cas, l'ensemble utilisé sera celui de tous les paramètres du script ou de la fonction. Nous verrons plus loin comment réaliser des fonctions et des scripts, ainsi que la manière de récupérer leurs paramètres.

8.6. Les ruptures de séquence

Il est parfois nécessaire de modifier l'ordre d'exécution dans les boucles et les itérations du shell. Par exemple, il est souvent nécessaire de sortir de la boucle courante, soit parce qu'on ne peut plus la continuer dans de bonnes conditions, soit parce que le traitement est terminé. C'est notamment le cas lorsqu'une erreur se produit, ou lorsqu'on recherche une valeur spécifique en itérant sur les valeurs possibles d'un ensemble.

Le shell fournit donc les instructions break et continue, qui permettent respectivement de sortir de la boucle courante et de passer directement à l'itération suivante. Ces deux commandes peuvent être utilisées aussi bien à l'intérieur des boucles while et until que dans les itérations écrites avec l'instruction for. Par exemple, le calcul de la somme des dix premiers entiers aurait pu être écrit de la manière suivante :

```
result=0
i=0
while true ; do
   result=$(($result + $i))
   i=$(($i + 1))
   if [ $i ==11 ] ; then break ; fi
done
echo $result
```

Les instructions break et continue peuvent prendre un paramètre entier indiquant le niveau d'imbrication de la boucle sur laquelle elles s'appliquent. Ce paramètre doit impérativement être supérieur à sa valeur par défaut, c'est-à-dire 1. Ainsi, pour sortir directement d'une double boucle lorsqu'on est dans le corps de la boucle la plus imbriquée, on utilisera la commande suivante :

```
break 2
```

8.7. Les fonctions

Le langage du shell est un langage procédural. Cela signifie que l'on peut créer des fonctions pour regrouper des séries d'instructions couramment exécutées. La syntaxe permettant d'écrire de telles fonctions est la suivante :

```
function nom () {
   instructions
}
```

où nom est le nom de la fonction, et instructions est la liste des commandes à exécuter dans cette fonction.

Vous constaterez qu'il n'y a pas de déclaration des paramètres de cette fonction. C'est normal : les paramètres des fonctions sont passés implicitement dans les variables d'environnement $1, $2, $3, etc. En fait, comme nous le verrons après cette syntaxe est également celle utilisée pour récupérer les paramètres de la ligne de commande des scripts shell. Cela signifie que les paramètres du script ne sont pas accessibles dans le corps d'une fonction, puisqu'ils sont masqués par les paramètres de la fonction.

Les autres variables utilisées dans les fonctions sont des variables globales. Celles qui sont déclarées dans une fonction sont donc également globales, et restent accessibles même après l'exécution de cette fonction. Si l'on veut définir des variables locales, on précédera la définition de la variable du mot clé local :

```
local variable=valeur
```

où variable est le nom de la variable locale, et valeur est sa valeur.

Les fonctions peuvent retourner une valeur numérique en code de retour. Cette valeur peut être indiquée à l'aide de l'instruction return. Par exemple, la fonction suivante calcule la somme des entiers de 0 à la valeur de l'entier qu'elle reçoit en paramètre :

```
function somme () {
   local result=0
   local i=0
   while [ $i -le $1 ] ; do
      result=$(($result + $i))
      i=$(($i + 1))
   done
   return $result
}
```

Ce code d'erreur pourra être récupéré par l'appelant dans la variable d'environnement $? :

```
somme 10
echo $?
```

8.8. Les entrées / sorties de données

Tout langage de programmation qui se respecte dispose de possibilités d'entrée / sortie pour permettre la communication avec l'utilisateur de manière interactive, et le shell n'échappe pas à la règle.

Nous avons déjà vu la commande **echo** dans bon nombre des exemples qui précédaient, et vous avez sans doute deviné qu'il s'agissait là de la commande qui permet d'afficher du texte à l'écran. Son utilisation est des plus simples, puisqu'elle se contente d'envoyer sur le flux de sortie standard une chaîne de caractères contenant tous les paramètres qu'elle reçoit, séparés par des espaces. Nous ne nous attarderons donc pas sur cette commande, qui n'a pas dû vous poser de problèmes jusqu'à présent.

Il ne nous reste donc plus qu'à voir la manière de demander à l'utilisateur de saisir une valeur. Avec bash, la demande de saisie des données se fait classiquement à l'aide de la commande **read**. Cette commande lit une ligne sur le flux d'entrée standard, la découpe en une ou plusieurs données et place les résultats dans les variables d'environnement qu'elle reçoit en paramètre. La syntaxe de **read** est donc la suivante :

read variable1 variable2 ... variablen

où variable1, variable2, etc. sont les noms des variables d'environnement dans lesquelles les résultats de la saisie doivent être placés.

La commande **read** utilise les séparateurs indiqués dans la variable d'environnement IFS pour découper la ligne lue dans le flux d'entrée standard. Si le nombre de variables spécifié est inférieur au nombre de mots de cette ligne après découpage, les premières variables d'environnement reçoivent les premiers mots, et la dernière reçoit le reste de la commande. Par exemple, la commande suivante :

read MOT RESTE

permet de lire le premier mot d'une ligne dans la variable d'environnement MOT et de placer le reste dans la variable RESTE.

La commande read dispose d'une syntaxe simplifiée, qui ne prend aucun paramètre. Dans ce cas, la ligne lue dans le flux d'entrée standard est placée telle quelle dans la variable d'environnement REPLY. Il est à la charge du programmeur d'analyser son contenu.

Le shell dispose également d'une instruction évoluée permettant de réaliser des menus simplifiés : l'instruction select. Cette instruction construit un menu à partir d'un certain nombre de choix, chaque choix étant précédé par un numéro, et demande à l'utilisateur de taper le numéro de son choix. Elle affecte alors la valeur du choix correspondant à une variable d'environnement, et exécute une commande pour le traitement du choix. La syntaxe générale de l'instruction select est donnée ci-dessous :

```
select variable in liste ; do
   action
done
```

où variable est le nom de la variable devant recevoir la valeur choisie par l'utilisateur, liste est la liste des valeurs que cette variable peut prendre, et action est la liste des instructions à exécuter pour chaque choix effectué.

Si le choix de l'utilisateur est incorrect, la variable de contrôle reçoit la valeur nulle. Le programmeur peut récupérer la valeur saisie par l'utilisateur dans la variable d'environnement REPLY et effectuer un traitement d'erreur approprié.

L'instruction select est une boucle. Le menu est reproposé après chaque exécution de l'action action. La sortie de cette boucle ne peut se faire que si un caractère de fin de fichier (CTRL + D) est lu sur le flux d'entrée standard, ou si une option de menu spécifique est proposée pour quitter cette boucle. Vous trouverez un exemple de menu simplifié ci-dessous :

```
select LU in A B C D Sortir; do
   case  $LU in
      ("A") echo "Vous avez choisi A" ;;
      ("B") echo "Vous avez choisi B" ;;
      ("C") echo "Vous avez choisi C" ;;
      ("D") echo "Vous avez choisi D" ;;
      ("Sortir") break ;;
   esac
done
```

9. Les alias

Il est incontestable que certaines commandes peuvent avoir une grande complexité, et il peut être fastidieux de les retaper complètement à chaque fois que l'on en a besoin. D'autre part, la saisie d'une longue ligne de commande multiplie les risques de faire une faute de frappe et d'avoir à corriger la commande. Cela peut au mieux faire perdre son temps à l'utilisateur, et au pire l'énerver.

Le shell fournit donc un mécanisme pour donner un nom simplifié aux commandes complexes : le mécanisme des *alias*. Les alias représentent en fait des chaînes de caractères complexes, et sont remplacés automatiquement par le shell lorsqu'il analyse les lignes de commandes. C'est un mécanisme plus souple que celui des variables d'environnement, et qui permet de définir des macro-commandes plus facilement qu'avec les fonctions du shell.

Pour créer un alias, vous devrez utiliser la syntaxe suivante :

```
alias nom=chaîne
```

où nom est le nom de l'alias, et chaîne est la chaîne de caractères

représentée par cet alias. Par exemple, pour faire un alias nommé beep permettant de faire un bip sonore, on pourra utiliser la commande suivante :

alias beep="echo $'\a'"

Cet alias pourra être utilisé simplement en tapant beep en ligne de commande.

Vous pouvez visualiser la liste des alias existant simplement à l'aide de la commande **alias**, appelée sans paramètres. Je vous recommande de consulter cette liste, pour vous donner une idée des alias courants, qui se révèlent généralement très utiles.

La suppression des alias se fait à l'aide de la commande **unalias**. Sa syntaxe est la suivante :

unalias nom

où nom est le nom de l'alias à supprimer.

Note : Les alias ne sont remplacés par la chaîne de caractères qu'ils représentent que lorsque le shell analyse la ligne de commande. Cela signifie que les définitions d'alias ne sont valides qu'après validation de cette ligne. On évitera donc de définir des alias dans la déclaration d'une instruction composée, car cet alias ne sera pas disponible à l'intérieur de son propre bloc d'instructions.

Par défaut, les alias ne sont disponibles que dans les shells interactifs. Ils ne peuvent donc pas être utilisés dans les scripts shell. La notion de script shell est détaillée dans le chapitre 10.

10. Les scripts shell

Pour l'instant, toutes les fonctionnalités de bash, aussi puissantes soient-elles, ne constituent que l'interface d'un interpréteur de commandes puissant. Mais nous avons dit que le shell était véritablement un langage de programmation. Cela signifie qu'il est possible d'écrire des programmes complexes en langage shell, simplement en stockant plusieurs commandes dans un fichier. On appelle ces fichiers des *scripts shell*.

L'écriture d'un script shell n'est pas plus compliquée que de taper les commandes du programme les unes à la suite des autres dans un shell interactif. La seule différence est que les scripts shell peuvent être rejoués plusieurs fois, recevoir des paramètres en ligne de commande et renvoyer un code de retour.

Tout script shell est en fait un fichier texte sur lequel on a mis les droits d'exécution. Il contient les différentes commandes qu'il doit exécuter. Sa première ligne est très importante, elle permet d'indiquer au shell exécutant quelle est la nature du fichier. La syntaxe de cette ligne est la suivante :

#!shell

où shell est le chemin absolu sur le shell ou l'interpréteur de commande capable d'exécuter ce script. En pratique, pour bash, on utilisera toujours

la ligne suivante :

```
#!/bin/bash
```

Les paramètres des scripts shell sont accessibles exactement comme des paramètres de fonction. On récupérera donc le premier paramètre avec l'expression $1, le deuxième avec l'expression $2, le troisième avec l'expression $3, etc.

Le code de retour d'un shell pourra être fixé à l'aide de la commande **exit**. Par exemple :

```
exit 0
```

Ce code de retour pourra être récupéré par l'appelant à l'aide de l'expression $?.

Nous n'irons pas plus loin dans la description du shell bash, car ce n'est pas le but de ce document. Vous pouvez vous référer à un bon livre d'Unix ou aux pages de manuel si vous désirez approfondir le sujet. Comme vous avez dû vous en rendre compte dans cette section, les shells Unix sont des véritables langages de programmation, qui dépassent de très loin les interpréteurs de commandes du type DOS. De plus, il existe plusieurs autres langages dont nous n'avons pas parlé ici, chacun étant conçu souvent pour réaliser un certain type de tâche (administration système, manipulation de fichiers textes, création de pages Web dynamiques, création d'interfaces utilisateur en mode fenêtré, pilotage d'applications, etc.). Si vous vous y intéressez, vous verrez que le sujet est réellement vaste et passionnant.