LA PROGRAMMATION SHELL

Rappelons brièvement l'idée fondamentale de Unix : tout ce qui se fait en Unix est fait par des programmes ordinaires, chacun s'exécutant de la même manière. Bien sûr, quelques-uns sont privilégiés, comme le programme qui organise le login, qui vérifie que votre nom de login et votre mot de passe sont corrects et invoque ensuite l'interprète de commande shell.

Le shell lit et interprète ce que vous tapez au terminal.

Normalement vous tapez des commandes, avec des arguments, et le shell s'arrange pour que ces commandes soient exécutées et les arguments bien transmis. Les commandes sont des programmes séparés qui tournent sous la supervision de Unix. Quand elles se terminent, le shell attend une autre commande. La plupart des commandes sont simplement des programmes écrits en langage C.

Cette façon de voir l'interprète de commande et les commandes elles mêmes est très originale. Unix est un des rares systèmes qui permet aux utilisateurs d'écrire leur propre shell (qui est juste un autre programme utilisateur), ou de rajouter librement des commandes supplémentaires.

Une commande shell est formée d'un mot (le nom de la commande), éventuellement des options (des flags) et éventuellement des arguments.

Exemple :

la commande

```
ls
```

est formée d'un seul mot, sans options ni arguments, qui, comme nous l'avons vu, liste les fichiers du catalogue de travail courant.

La commande

```
ls -la
```

est formée du mot de commande ls et de deux options. Les options sont normalement précédées du signe -, et chaque option est formée d'une lettre. Ici, il y a donc l'option l, qui donne la description des fichiers dans un format long, et l'option a disant de lister tous (all) les fichiers.

Finalement, la commande

```
ls a.out
```

est formée du nom de la commande et d'un argument, le nom d'un fichier. Cette commande ne liste que le fichier a.out s'il existe.

Les Noms De Fichiers

La plupart des commandes attendent, comme arguments, des noms de fichiers, qui peuvent être donnés soit comme des noms simples, s'il s'agit d'un fichier dans le catalogue de travail courant, soit comme des pathnames complets, s'il s'agit de fichiers à un endroit quelconque dans la hiérarchie du système de fichiers.

Un **nom de fichier** consiste en une suite de caractères alphanumériques et parfois d'un ou de plusieurs "." (points). Des caractères non-alpahnumériques devraient être évités dans les noms, puisque beaucoup ont une signification particulière pour le shell. Ces caractères particuliers sont appelés métacaractères. Nous en avons déjà rencontré quelques-uns : <, >, et |.

Le point "." n'est pas considéré comme métacaractère. Il est très souvent utilisé pour séparer des parties communes à des noms de fichiers. Par exemple, il est fréquent de trouver des noms de fichiers comme :

```
prog.c
prog.o
prog.out
```

où, évidemment, il s'agit de différentes représentations d'un même fichier.

Le fichier prog.c, par exemple, peut être le fichier contenant le code source d'un programme écrit en C, le fichier prog.o peut correspondre au code objet correspondant, et le fichier prog.out peut être la version exécutable.

Le shell donne un mécanisme simple pour la génération et l'expansion des noms de fichiers donnés dans des commandes.

Des pathnames, absolus ou relatifs, peuvent être écrits en remplaçant des parties ou l'ensemble par des caractères spéciaux. Par exemple, l'expression

```
prog*
```

dénote (filtre) tous les noms de fichiers, dans le catalogue courant, qui commencent par le mot prog. Tous les exemples ci-dessus seraient filtrés par cette écriture.

Si vous donnez la commande

```
ls prog*
```

le mot prog* est, avant l'exécution de la commande, filtré avec les noms de fichiers dans le catalogue courant. Les noms qui filtrent (qui commencent par prog) sont ensuite, dans un ordre alphabétique, donnés comme des arguments à la commande.

Dans notre exemple,

```
ls prog*
```

est donc la même chose que

```
ls prog.c prog.o prog.out
```

Le métacaractère "*" filtre donc toutes les séquences (la séquence vide incluse) de caractères dans un nom de fichier. Le caractère "*" lui-même filtre tous les noms de fichiers dans le catalogue courant.

Le métacaractère "?" filtre n'importe quel caractère dans un nom de fichier, mais, à la différence de "*", il ne filtre qu'un seul caractère.

Ainsi la commande

```
ls prog.?
```

dans notre exemple, ne filtre que les noms prog.c et prog.o, mais non pas le fichier prog.out.

La commande

```
ls ? ?? ???
```

va donc lister tous les fichiers dans le catalogue courant, qui ont des noms de un, deux ou trois caractères.

Un autre mécanisme consiste en une séquence de caractères entre crochets ("[" et "]"). Cette métaséquence filtre des caractères uniques de l'ensemble des caractères donnés entre crochets. Par exemple :

```
ls prog.[os]*
```

ne listera que les fichiers commançant par prog qui ont ensuite soit la lettre o, soit la lettre s, suivie de n'importe quoi d'autre. Donc, cette commande ne listera que les fichiers prog.o et prog.out de notre exemple.

Il est important de rappeler que :

1. Le filtrage des métacaractères est fait avant l'exécution de la commande;

2. Tous les noms qui filtrent sont donnés comme arguments séparés, en ordre alphabétique, à la commande;

3. Si aucun nom de fichier filtre une spécification particulière, le mot lui-même, complété par les métacaractères est donné en argument.

Si toutefois vous voulez utiliser un de ces métacaractères, en tant que caractère normal, il suffit de le faire précéder du symbol "\" (encore un métacaractère !).

Les Variables Shell

Le langage de commande de Unix, shell, permet l'introduction et l'utilisation de variables d'une manière très libérale. N'importe quelle chaîne de caractères (sans métacaractères toutefois !) peut être une variable à laquelle on peut affecter des valeurs et dont les valeurs peuvent être consultées.

Par exemple, la commande

```
moi = /usr/vlisp80/hw/
```

affecte à la variable moi la chaîne /usr/vlisp80/hw/, qui est le pathname de mon catalogue sur la machine que j'utilise actuellement. Le nom moi étant librement choisi. Pour avoir accès à la valeur d'une variable, il suffit de la faire précéder par le caractère "$" (dollar). Ainsi, la commande

```
echo $moi
```

va imprimer sur la sortie standard

```
/usr/lisp80/hw/
```

Une forme plus générale de cette substitution d'une valeur de variable dans une commande est la forme

```
${nom-de-variable}
```

qui doit être utilisée obligatoirement, si le nom-de-variable est suivi par d'autres caractères.

Par exemple, la commande

```
ls ${moi}foo
```

liste le fichier

```
/usr/vlisp80/hw/foo
```

pendant que la commande

```
ls $moifoo
```

essaiera de lister le fichier dont le pathname est la valeur de la variable moifoo.

Quelques noms de variables ont une signification particulière :

PATH Cette variable contient une liste de catalogues à l'intérieur desquels shell doit chercher des commandes à exécuter. Le deux catalogues /bin et /usr/bin sont la liste par défaut. C'est dans ces catalogues que shell trouve les commandes echo, ls et cat par exemple.

Pour donner des catalogues supplémentaires, on doit affecter la variable PATH avec une liste de catalogues, chaque catalogue étant séparé du suivant par deux points (":"). Exemple :

```
PATH = :/usr/vlisp80/pg:/usr/eqdom/berry:/bin:/usr/bin
```

spécifie qu'en plus du catalogue courant (la chaîne vide avant le premier ":") et des deux catalogues standard, les commandes doivent être cherchés dans :

```
/usr/vlisp80/pg
```

et

```
/usr/eqdom/berry
```

PS1 Contient le prompt caractère du shell. C'est le caractère que shell affiche quand il attend une commande. Sur la plupart de systèmes c'est soit "$", soit "%". Si vous en voulez un autre, il suffit d'affecter cette variable avec la chaîne que vous voulez.

TERM La variable TERM contient le type de terminal sur lequel vous travaillez. Cette information est importante si vous voulez utiliser des programmes vidéos.

HOME Contient le pathname de votre catalogue de login. C'est l'argument par défaut pour la commande cd.

MAIL Est une variable consultée entre chaque commande par shell, pour voir si vous avez reçu du courrier.

Il y a encore d'autres variables spéciales (?, $, #, !) que nous verrons dans la suite.

Les Fichiers De Commandes

De même que tout autre programme, shell peut également prendre des arguments dans un fichier.

Reprenons notre exemple de la commande shell donnant le nombre d'utilisateurs du système :

```
who | wc -l
```

Si nous voulons très souvent savoir combien d'utilisateurs sont sur le système, il serait adéquat de construire une nouvelle commande. Pour cela, nous créons un fichier avec la ligne de commande ci-dessous

```
echo 'who | wc -l' > combien
```

Les quotes ici étaient nécessaires pour prévenir que le "|" ne divise pas cette commande en deux : une première qui écho le mot who, et une deuxième qui applique la commande "wc -l" sur cela, tout en mettant la sortie dans le fichier combien.

Pour exécuter ce petit programme shell (on dit d'un programme shell que c'est un script) il suffit d'appeler le shell et de donner le fichier combien en argument :

```
sh combien
```

Pour vraiment pouvoir considérer le fichier combien comme une commande shell nouvelle, nous pouvons spécifier que combien est un fichier exécutable avec la commande :

```
chmod a+x combien
```

qui dit de rajouter au mode de combien que ce fichier est exécutable pour tout le monde. "chmod" veut dire change mode.

Naturellement, des shell-scripts (procédures shell) peuvent, comme tout autre programme, avoir des arguments. A l'intérieur du script, ces arguments peuvent être référés par des paramètres spéciaux (connus sous le nom de paramètres positionnels). Le métacaractère "$" est utilisé pour référer aux arguments d'une commande :

"$1" dénote le premier argument;
"$2" dénote le deuxième argument;
"$3" dénote le troisième argument;
...
"$9" dénote le neuvième argument.

Le nom de la commande même est accessible dans le paramètre "$0".

Par exemple, si le fichier ou contient la ligne

```
who | grep $1
```

et si ce fichier est exécutable, alors

```
ou hw
```

imprimera une ligne montrant sur quel terminal hw est loggé. Si hw n'est pas loggé, rien ne sera imprimé.

La commande grep cherche à l'intérieur d'un texte l'occurrence des expressions données en argument. Ici l'expression cherchée est juste la chaîne de caractères hw. Dans cet exemple, grep fonctionne principalement comme un filtre lisant l'entrée standard et imprimant toutes les lignes contenant la chaîne hw.
Pendant l'interprétation du script ou, le paramètre positionnel "$1" sera remplacé par le mot hw. Si hw est loggé, l'écran devrait contenir quelque chose comme :

```
$ ou hw
hw tty21 May 07 14:37
$
```

(où le "$" au début de la ligne est juste le prompt de shell).

En plus des paramètres positionnels, deux paramètres supplémentaires sont possibles : "$#" et "$*".

"$#" donne le nombre d'arguments donnés à l'appel et "$*" est utilisé si l'on veut substituer tous les arguments sauf le nom de commande ("$0").

Voici à présent une petite commande qui imprime tous ces arguments ainsi que leur nombre :

```
echo '('$*')' à $# éléments
```

Si cette commande se trouve dans le fichier exécutable nommé bidon, une exécution de la commande devrait donner ceci :

```
$ bidon 1 2 3
(1 2 3) à 3 éléments
```

et

```
$ bidon que c est beau
(que c est beau) à 4 éléments
```

Naturellement, "$*" est très utile si vous avez besoin de plus de 9 arguments.

Finalement, une ligne commençant par un ":" (double point) est ignorée par shell et peut donc contenir des commentaires.

Les Structures De Contrôle

Comme dans tous les langages, une programmation effective n'est pas possible sans un minimum de structures de contrôle. Shell donne des structures telles que :

for
case
if then else
while
until

Les commandes peuvent être données à partir d'un fichier de script où à partir d'un terminal. Dans ce deuxième cas, le prompt de shell change après la frappe de la première ligne d'une commande. Ce deuxième prompt-caractère peut être modifié à travers la variables "PS2".

La commande "test"

Beaucoup des utilitaires de Unix ont un exit-status qui est égal à 0 si la commande s'est bien terminée et différent de 0 sinon. La commande test, qui n'a d'utilité qu'à l'intérieur de shell-scripts, ne donne rien d'autre qu'un tel exit-status. Naturellement, cette commande est utilisée pour diriger le flux de contrôle du shell-script.

Par exemple :

test -f nom-de-fichier

donne la valeur 0 (vrai) si nom-de-fichier est un fichier qui existe et n'est pas un catalogue.

Voici la liste presque complète des arguments possibles pour test :

test -f fichier
vrai si fichier existe et n'est pas un catalogue;

test -r fichier
vrai si fichier existe et est accessible en lecture;

test -w fichier
vrai si fichier existe et est accessible en écriture;

test -d fichier
vrai si fichier existe et est un catalogue;

test chaîne

vrai si chaîne est une chaîne de caractères non vide;

test -n chaîne
même chose que “test chaîne”;

test -z chaîne
vrai si chaîne est une chaîne de caractères vide

test $chaîne_1$ = $chaîne_2$
vrai si la chaîne de caractères $chaîne_1$ est identique à la chaîne de caractères $chaîne_2$

test $chaîne_1$!= $chaîne_2$
vrai si la chaîne de caractères $chaîne_1$ est différente de la chaîne de caractères $chaîne_2$

test n_1 -eq n_2
vrai si le nombre n_1 est égal au nombre n_2

test n_1 -ne n_2
vrai si le nombre n_1 est différent du nombre n_2

test n_1 -gt n_2
vrai si le nombre n_1 est supérieur au nombre n_2

test n_1 -ge n_2
vrai si le nombre n_1 est supérieur ou égal au nombre n_2

test n_1 -lt n_2
vrai si le nombre n_1 est inférieur au nombre n_2

test n_1 -le n_2

vrai si le nombre n_1 est inférieur ou égal au nombre n_2

“!” est l’opérateur logique non, “-a” est l’opérateur logique binaire et et “-o” est l’opérateur logique binaire ou.

La boucle “for”

La forme générale de la boucle for est :

```
for nom in liste-de-mots
do liste-de-commande
done
```

A chaque itération de la boucle for, la variable shell nom prend comme valeur le mot suivant de la liste-de-mots. Si la liste-de-mots est omise, la boucle est exécutée

pour chacun des paramètres positionnels (des arguments donnés à la commande). Ainsi :

```
for i
```

est équivalent à :

```
for i in $*
```

L'exécution de la boucle s'arrête s'il ne reste plus de mots dans la liste-de-mots. La liste-de-commandes est une séquence de une ou plusieurs commandes, séparées ou terminées par un semicolon (";") ou un line feed.

Exemple :

```
for i in eins zwei drei
do echo $i
done
```

imprimera, à l'exécution

```
eins
zwei
drei
```

Si l'on veut avoir une commande create qui crée des fichiers avec les noms de ses arguments, elle serait écrite de la façon suivante :

```
for i in $*
do >$i
done
```

ou, en plus compacté :

```
for i do >$i; done
```

et l'appel :

```
create foo bar
```

créerait les fichiers foo et bar.

La sélection "case"

La forme générale de la commande case est :

```
case mot in
filtre) liste-de-commande;;
...
esac
```

La commande case donne la possibilité de séléctionner entre différentes possibilités. Pendant l'exécution, la liste-de-commandes associée au filtre qui correspond à mot sera exécutée, toutes les autres liste-de-commandes seront ignorées.

Exemple :

```
for i in eins zwei drei
do case $i in
     eins) N=un;;
     zwei) N=deux;;
     drei) N=trois;;
   esac
   echo $N
done
```

Cette commande imprime, à l'exécution :

```
un
deux
trois
```

Puisque le métacaractère "*" filtre n'importe quelle chaîne de caractères, il peut être utilisé pour le cas par défaut (c'est-à-dire pour le cas où aucun des filtres ne s'applique).

Notez que chaque cas est terminé par deux points-virgules et que esac (case à l'envers, dans le bon vieux style de Algol-68) termine la commande case.

Le filtre peut être n'importe quelle chaîne de caractères combinée avec des métacaractères. Ainsi le filtre

```
*.c)
```

par exemple, sera choisi si le mot (le premier argument de la commande case) se termine par la chaîne ".c", et le filtre

```
-[xy])
```

sera sélectionné pour "-x" et "-y". Ce filtre peut également être écrit comme :

```
-x | -y)
```

ce qui veut dire que la barre verticale est un séparateur entre plusieurs filtres, et l'on peut toujours écrire un filtre comme

$$filtre_1 \mid filtre_2 \mid filtre_3 \mid ... \mid filtre_n)$$

si l'on veut pouvoir choisir entre plusieurs alternatives.

Voici un deuxième exemple d'utilisation de la commande case. C'est le prototype pour une commande compile qui appelle le compilateur approprié dépendant de la terminaison du nom de fichier donné en argument :

```
: initialise les "options" à la chaîne vide
options=
: ensuite on répète pour chaque argument
for i in $*
do case $i in
: on concatène les options permises (+ un espace)
    -[ocsSO]) options = $options' '$i;;
    -*) echo 'option inconnu $i' ;;
:     et on compile chacun des fichiers
    *.c) cc $options $i; options = ;;
    *.s) as $options $i; options = ;;
    *.f)f77 $options $i; options = ;;
:    sinon c'est un mauvais argument
    *) echo 'que faire avec $i\?';;
     esac
done
```

La sélection "if - then - else"

Cette construction peut être utilisée comme instruction de branchement. Sa forme générale est :

```
if liste-de-commandes
then liste-de-commandes
[else liste-de-commande]
fi
```

La liste-de-commande suivant le if est exécutée et si la dernière commande de cette liste ramène un status de 0 (vrai), alors la liste-de-commande suivant le then est exécutée. La partie else est optionnelle : si elle est présente, la liste-de-commande correspondante est exécutée pour un status différent de 0 (faux) de la liste-de-commande du if.

Nous pouvons, par exemple, reécrire notre commande create en ne créant des fichiers que si le nom du fichier qu'on veut créer n'existe pas encore :

```
for i
```

```
do
if test -f  $1
then echo 'le fichier $i existe déjà'
else >$i
fi
done
```

Voici un autre exemple : une commande qui nous dit pour chaque fichier dans le catalogue courant si c'est un catalogue ou un fichier normal et si le fichier est ouvert en lecture ou en écriture :

```
for a in *
do if test -d $a
   then echo '$a est un directory'
   else echo -n '$a est un fichier accessible en'
            if test -w $a
            then echo écriture
            else if test -r $a
                 then echo lecture
                 else echo rien
                 fi
             fi
   fi
done
```

Pour des test if imbriqués du style :

```
if ...
then ...
else if ...
     then ...
     else if ...
          ...
          fi
     fi
fi
```

existe une abréviation, et vous pouvez également les écrire de la manière suivante:

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

A vous de décider laquelle des deux formes d'écriture vous préférez.

Remarquons que les opérateurs du et logique, &&, et du ou logique, ||, peuvent être écrits en termes de if :

Les boucles “while” et “until”

while et until sont probablement les deux formes de boucles les plus usuelles dans les langages de programmation. while répète une action tant qu’une certaine condition est vraie, pendant que until répète une action jusqu’à ce qu’une certaine condition devienne vraie.

La forme générale de ces deux commandes est :

```
while liste-de-commandes
do liste-de-commandes
done
```

et

```
until liste-de-commandes
do liste-de-commandes
done
```

Les Documents In-line

Supposons que nous voulions construire un petit script shell qui donne le numéro de téléphone des personnes dont les noms sont donnés en argument au script. Pour cela, nous allons d’abord créer un fichier contenant des lignes du style :

```
...
Patrick Müller 230-00-01
Eva 777-10-50 (home) 888-10-50 (office)
Fred Dubois (16)(33) 4240-97-86
...
```

Ce fichier est une sorte de carnet d’adresses, et chaque ligne contient des noms et des numéros de téléphone.

Le texte de la procédure qui nous donne pour un nom donné le(s) numéro(s) de téléphone correspondant, est relativement aisé à écrire :

```
for i
do grep $i $HOME/agenda; done
```

Evidemment, nous utilisons la commande grep qui donne toutes les lignes d’un fichier contenant l’expression donnée en argument.
Remarquez que nous avons supposé que le fichier contenant les numéros de téléphone se trouve dans votre catalogue de login et s’appelle agenda. Afin d’y accéder,

nous avons utilisé la variable shell HOME qui contient toujours le pathname de ce catalogue.

Si la seule utilisation du fichier agenda est faite par cette procédure de recherche, il semble qu'éventuellement il serait plus aisé d'inclure ces données directement dans ce script. Cela économise un fichier !

Des données directement incluses dans une procédure shell s'appellent, en jargon Unix, des documents in-line ou des here documents.

L'écriture alternative de notre procédure sera alors:

```
for i
do grep $i <<!
...
    Patrick Müller 230-00-01
    Eva 777-10-50 (home) 888-10-50 (office)
    Fred Dubois (16)(33) 4240-97-86
...
!
done
```

Dans cet exemple, shell prend les lignes entre "<<!" et "!" comme entrée standard pour grep. Le caractère "!" est choisi arbitrairement; le document est terminé par une ligne qui n'est formée que du caractère suivant immédiatement le signe "<<".

Naturellement, puisque ce document se trouve à l'intérieur d'un script shell, avant de donner le document à grep, il y aura substitution de paramètres (si vous avez des paramètres à l'intérieur du document) ou des variables utilisées.

Pour inhiber cette substitution totalement, il suffit de quoter le caractère délimitateur du document comme:

```
...
do grep $i <<\!
      ...
      ...
!
```

Voici un exemple d'un script utilisant des here documents et ayant besoin de la substitution :

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

Si cette procédure s'appelle edg, l'appel

edg chaîne$_1$ chaîne$_2$ fichier

est alors équivalent à

```
ed fichier ≪%
g/chaîne1/s//chaîne2/g
w
%
```

et change toutes les occurrences de la chaîne chaîne$_1$ dans le fichier fichier en chaîne$_2$.

Naturellement, on peut également inhiber individuellement la substitution en quotant avec le caractère "\" les occurrences du caractère "$" qu'on veut garder tel quel :

```
ed $3 ≪ +
1,\$s/$1/$2/g
w
+
```

Cette version du script edg est plus ou moins équivalente à la précédente version.

La Substitution De Paramètres

Si un paramètre de shell n'a pas reçu une valeur, shell substitue à l'occurrence de ce paramètre la chaîne vide. Par exemple : si la variable "d" n'a pas reçu de valeurs, alors

```
echo $d
```

ou

```
echo ${d}
```

imprime la chaîne vide, c'est-à-dire rien.

On peut donner des valeurs "par défaut" comme dans :

```
echo ${d-.}
```

qui imprime la valeur de la variable "d" si "d" a une valeur et qui imprime la chaîne "." sinon. La chaîne donnée comme valeur par défaut est évaluée. Ainsi

```
echo ${d-$1}
```

imprimera la valeur de la variable "d" si elle existe, sinon cela imprimera la valeur du paramètre "$1".

Voyez-vous la différence entre

```
echo ${d-*}
```

et

```
echo ${d-'*'}  ?
```

On peut également affecter aux variables des valeurs par défaut. Ainsi

```
echo ${d=.}
```

substitue la même chaîne que

```
echo ${d-.}
```

mais, en plus, si "d" n'avait pas de valeur, après substitution, la valeur de "d" sera la chaîne ".".

Remarquons que cette affectation par défaut n'est pas possible avec des paramètres positionnels.

Une dernière possibilité est d'interrompre le calcul en imprimant un message si une variable n'a pas de valeur :

```
echo ${d?message}
```

imprime la valeur de "d" si elle existe, sinon message est imprimé par le shell et l'exécution de la procédure est abandonnée. Si message n'est pas donné, comme dans

```
echo ${d?}
```

un message standard est imprimé.

La Substitution De Commandes

La sortie standard de commandes shell peut être substituée de manière similaire à la substitution de paramètres.

Si, par exemple, le catalogue courant est /usr/fred/bin, la commande

```
d = `pwd`
```

est équivalent à

```
d = /usr/fred/bin
```

Toute la chaîne entre les accents graves est prise comme commande à exécuter et est remplacée par la sortie standard de cette commande. C'est encore un métacaractère !

La substitution de commandes a lieu dans tous les contextes (y compris les here documents) où normalement une substitution de paramètres a lieu. Le mécanisme permet, par exemple, l'utilisation de commandes de traitement de chaînes à l'intérieur de scripts shell.

Prenons comme exemple la commande basename qui enlève un suffixe spécifié d'une chaîne. Par exemple :

```
basename main.c .c
```

imprime la chaîne main. Voici une utilisation de cette commande telle qu'elle a lieu dans la commande cc :

```
case $A in
    ...
    *.c) `B= basename $A .c`
    ...
esac
```

qui donne à la variable B la chaîne A sans le ".c" à la fin.

Voici encore quelques exemple :

```
for i in `ls -t`; do ...
```

donnera à "i" les noms de tous les fichiers du catalogue dans l'ordre de leurs dernières modifications.

```
set `date `; echo $6 $2 $3, $4
```

imprimera, par exemple : 88 Nov 1, 23:59:59.

Evaluation Et Inhibition De L'évaluation

Shell est un langage qui permet la substitution de paramètres, la substitution de commandes et la génération de noms de fichiers dans les arguments pour les commandes. Dans ce paragraphe nous en donnerons un court résumé.

Avant l'exécution d'une commande les substitutions suivantes ont lieu :

- substitution de paramètres, par exemple: $user

- substitution de commandes, par exemple : `pwd`. Il n'y a qu'une seule substitution ! Si, par exemple, la valeur de la variable "$X" est la chaîne "$Y", la commande

  echo $X

  imprimera $Y.

- génération de noms de fichiers : après substitution, chaque mot est analysé pour des occurrences des caractères *, ?, [,] et . et une liste de noms de fichiers (en ordre alphabétique) est générée pour remplacer ce mot. Chacun de ces noms de fichiers est un argument séparé. Cette évaluation a lieu également dans les mots associés à une boucle for. Dans les mots utilisés dans une branche d'un case, il n'y a que substitution et non pas génération de noms de fichiers.

En plus des mécanismes pour quoter, utilisant les caractères "\" et "", il existe un mécanisme pour quoter qui utilise les guillemets ("). Entre guillemets, la substitution a lieu, mais la génération de noms de fichiers et l'interprétation de séparateurs vides (espace, tabulation et retour-chariot) sont inhibés. Donc, entre guillemets, les caractères suivants ont une signification particulière :

$	substitution de paramètre
`	substitution de commande
"	fin de chaîne quotée
\	quote les caractères spéciaux $, `, " et \.

Si plus d'une seule évaluation d'une chaîne est nécessaire, on peut utiliser la commande eval. Par exemple, si la valeur de X est la chaîne $Y et si la valeur de Y est la chaîne voici,

echo $X

imprime $Y. Mais

eval echo $X

imprime la chaîne voici

Grammaire Du Shell

item :

mot
entrée/sortie

nom = valeur

commande-simple :
item
commande-simple item

commande :
commande-simple
(liste-de-commandes)
{liste-de-commandes}
for nom do liste-de-commandes done
for nom in mot ... do liste-de-commandes done
while liste-de-commandes do liste-de-commandes done
until liste-de-commandes do liste-de-commandes done
case mot in case-part ... esac
if liste-de-commandes then liste-de-commandes else-part

else-part :
elif liste-de-commandes then liste-de-commandes else-part
else liste-de-commandes
rien

pipeline :
commande
pipeline | commande

andor :
pipeline
andor && pipeline
andor || pipeline

liste-de-commandes :
andor
liste-de-commandes;
liste-de-commandes&
liste-de-commandes; andor
liste-de-commandes & andor

entrée/sortie :
> fichier
< fichier
≫ mot
≪ mot

fichier :

mot
& digit
&-

case-part :

filtre) liste-de-commandes;;

filtre :

mot
filtre | mot

rien :

mot :

une séquence de caractères (sans espace, tabulation, line-feed)

nom :

une séquence de caractères, digits ou soulignés commençant par une lettre

digit :

0 1 2 3 4 5 6 7 8 9

Métacaractères Et Mots Réservés

a) Syntactique

|	symbole pour les pipes
&&	et logique
||	ou logique
;	séparateur de commandes
;;	délimitateur de case
&	commandes background
( )	groupement de commandes
<	redirection de l'entrée
<<	entrée en here-document
>	création de sortie
>>	rajout de sortie

b) Filtres

*	filtre tout caractère(s) ou rien
?	filtre un seul caractère
[...]	filtre chacun des caractères inclus

c) Substitution

${...}	substitution d'une variable shell
`...`	substitution d'une commande

d) Quotations

\	quote le caractère suivant
'...'	quote tous les caractères inclus, excepté ' lui-même
"..."	quote tous les caractères inclus, excepté $, \ et "

e) Mots réservés

if, then, else, elif, fi, case, in, esac, for, while, until, do, done