Outils de manipulation de texte

Linux comporte un certains nombre d'utilitaires permettant de manipuler du texte. Il est important de bien définir ce terme : nous parlons ici d'utilitaires permettant de réaliser sur des textes des opérations de recherche, remplacement, extraction, comptage, analyse, formatage,.. et ceci de façon très rapide et automatisable. Il ne s'agit pas d'utilitaires de **traitement de texte** au sens de **Publication Assistée par Ordinateur**. Dans le contexte qui nous intéresse ici, le terme **texte** s'applique à un flot de caractères constitué de lignes. Tous les utilitaires décrits dans ce chapitre sont capables de traiter des textes contenus dans des fichiers, mais seront le plus souvent utilisés en tant que filtres : ils lisent un texte ligne par ligne sur leur entrée standard, le traitent et produisent un texte et/ou un résultat sur leur sortie standard. Grâce au mécanisme des tubes (voir le paragraphe 7.3), il sera possible de réaliser des traitements très complexes sur des flots de texte de taille illimitée. Le flot d'entrée peut être la sortie d'une commande ou le contenu d'un fichier, le résultat pouvant être redirigé vers un fichier. Tous ces utilitaires interprètent la même description des motifs de caractères : les **expressions régulières**.

Ces utilitaires présentent une gradation de possibilités : du plus simple au plus puissant, chacun d'entre eux peut faire ce que font les précédents. On pourrait se contenter de ne connaître que le plus complet. En pratique, il vaut mieux utiliser le plus simple dans chaque cas. Ces utilitaires sont les suivants, par ordre croissant de possibilités :

`grep` Recherche des chaînes de caractères conformes à un motif donné et les affiche.

`sed` (Stream Editor, éditeur de flot)
Recherche des chaînes conformes à un motif donné et les modifie.
Petit langage de programmation permettant l'écriture de programmes : les scripts `sed`.

`awk` Permet de rechercher, modifier, formater, compter, afficher, traduire, …
`awk` est un langage de programmation spécialisé permettant d'effectuer des traitements relativement complexes sous forme de programmes très courts.

Nous présenterons également `tr`, complémentaire des utilitaires précédents, bien qu'il ne traite pas les expressions régulières. Il rend de grands services dans la même classe d'applications.

Nous allons d'abord présenter les expressions régulières d'une façon très générale et indépendamment des utilitaires `grep`, `sed`, `awk` puis nous présenterons les utilitaires eux–même.

14.1 LES EXPRESSIONS RÉGULIÈRES

Les expressions régulières permettent de décrire des motifs formés de caractères. Ce mécanisme est très similaire au mécanisme de génération de noms de fichiers par le shell que nous avons décrit au paragraphe 7.7. Il en diffère toutefois par les aspects suivants : la génération de noms est un mécanisme rudimentaire qui décrit de façon générique des **noms** de fichiers, alors que les expressions régulières représentent une description générique très sophistiquée de chaînes de caractères **contenues dans** des fichiers (ou un flot de caractères).

Exemples

Rechercher toutes les chaînes de caractères alphabétiques dans :

- le fichier `fich1.f`
- tous les fichiers dont les noms se terminent par `.f`
- tous les fichiers dont les noms commencent par une majuscule et se terminent par `.f`

```
xstra> grep    '[a zA Z]'     fich1.f
xstra> grep    '[a zA Z]'     *.f
xstra> grep    '[a zA Z]'     [A Z]*.f
                    ↑              ↑
§     expression régulière  génération de noms de fichier
```

La syntaxe des expressions régulières est en outre différente de la syntaxe de génération de noms. Ces expressions sont utilisées par un grand nombre d'utilitaires : non seulement *grep*, *sed* et *awk*, mais également par le langage *perl* (the Practical Extraction and Report Language) et tout éditeur de texte, dont *vi* et *emacs*.

Les expressions régulières servent à décrire de façon concise et inambiguë des motifs de caractères, comme par exemple : « toutes les chaînes de 1 à 8 caractères alphanumériques, le premier étant alphabétique ».

Cette périphrase traduite en expression régulière devient :

```
[A Za z][A Za z0 9]{0,7}
```

Les expressions régulières (et les utilitaires qui les comprennent) ne sont pas limitées au monde Unix : *grep*, *sed*, *awk*, et le langage *perl* ont été portés sur d'autres systèmes d'exploitation.

Il existe deux sortes d'expressions régulières : les **expressions régulières étendues (ere)** que nous décrirons ici, et les **expressions régulières de base**, considérées comme obsolètes dans le monde linux, dont nous dirons quelques mots.

14.1.1 Conventions d'écriture

Dans ce chapitre, on dira qu'une chaîne de caractères **vérifie** une expression régulière (l'aide en ligne utilise le terme : **match**), et le terme **caractère** exclut le caractère <new-line>.

- ch désigne un caractère quelconque sauf un caractère spécial
- sp désigne un caractère spécial

Les caractères spéciaux pour les expressions régulières sont les suivants :

```
| . * + ? ^ $ ( ) [ ] {} \
```

Les expressions régulières sont construites progressivement, à partir de briques de base appelées **expressions régulières atomiques**.

14.1.2 Les expressions régulières atomiques : era

On appelle expression régulière atomique (**era**) un motif constitué d'**un seul caractère** appartenant à un **ensemble** précis.

cette era	définit l'ensemble de caractères suivants
ch	l'ensemble constitué du seul caractère **ch**
\sp	l'ensemble constitué du seul caractère spécial **sp**
.	l'ensemble de tous les caractères
[gkl]	l'ensemble constitué des caractères placés entre [ ] (**g,k,l**)
[^gkl]	l'ensemble constitué des caractère autres que **g,k,l.** Le caractère ^ placé après [indique la négation.
[a-z]	l'ensemble constitué des caractères compris entre **a** et **z** (ordre alphabétique)
[^a-z]	l'ensemble constitué des caractères non compris entre **a** et **z**

Exemples

```
a       § le caractère a
[i n]   § un caractère appartenant à l'ensemble ijklmn
[ i n]  § un caractère appartenant à l'ensemble  ijklmn
.       § un caractère quelconque
\.      § le caractère .
[^0 9]  § tout caractère autre qu'un chiffre
[^0 9^] § tout caractère autre qu'un chiffre ou ^
        § car le dernier ^ n'est pas après [
```

14.1.3 La construction d'une expression régulière : er

Si l'on recherche une chaîne de caractères (un "mot") conforme à un certain motif, il faut pouvoir définir l'ensemble des caractères constituant ce "mot", le nombre de caractères dans ce mot, et à quel endroit le chercher (quoi ? combien ? où ?). Par exemple : un mot de 3 à 6 lettres minuscules au début de la ligne. Les briques élémentaires (**era**) définissent des ensembles de caractères (quoi ?) et seront combinées entre elles par trois opérations élémentaires : la concaténation, la quantification (combien ?) et l'ancrage (où ?).

La concaténation

Une expression régulière (**er**) peut être obtenue par concaténation (juxtaposition) d'expressions régulières atomiques (**era**). La concaténation est décrite sans opérateur ou caractère spécial :
abc signifie : a suivi de b suivi de c.

Exemples

```
abc             § la chaîne abc
[Oo]ui          § la chaîne Oui ou la chaîne oui
[A Z][0 9]..    § une majuscule suivie d'un chiffre suivi
```

```
                  § de deux caractères quelconques :
                  § A8b5 ou Z444 mais pas h7fu
[a z][0 9].\.     § une minuscule suivie d'un chiffre suivi
                  § d'un caractère quelconque suivi d'un
                  § point : a8b. ou h6.. mais pas Z67f
```

La quantification

Les **quantifieurs** permettent de définir combien de fois l'**era** qui précède est répétée. Si ⊗ désigne une expression régulière atomique les quantifieurs utilisables sont les suivants :

quantifieur	signification
⊗*****	tout mot de 0 à N caractères vérifiant ⊗
⊗**+**	tout mot de 1 à N caractères vérifiant ⊗
⊗**?**	tout mot de 0 à 1 caractère vérifiant ⊗
⊗**{n}**	tout mot de n caractères vérifiant ⊗
⊗**{n1,n2}**	tout mot de n1 à n2 caractères vérifiant ⊗
⊗**{n1,}**	tout mot d'au moins n1 caractères vérifiant ⊗
⊗**{,n2}**	tout mot de 0 à n2 caractères vérifiant ⊗

Dans ce tableau, la phrase « tout mot de n caractères vérifiant ⊗ » signifie exactement : « tout mot de n caractères appartenant à l'ensemble défini par l'expression régulière atomique ⊗ ».
La notation **{n,m}** est générale mais lourde pour les cas simples, c'est pourquoi :

- ***** est utilisé à la place de **{0,}**
- **+** est utilisé à la place de **{1,}**
- **?** est utilisé à la place de **{0,1}**

Exemples

```
abc       § la chaîne abc
a\.       § la chaîne a.
a.        § a suivi de n'importe quel caractère
a*        § rien ou a ou aa ou aaa ou ...
a.*       § a suivi de n'importe quelle chaîne (même vide)
          § (a ou ab ou abc ...)
a+        § a ou aa ou aa ou aaa ou ...
a?        § rien ou a
a{2}      § la chaîne aa
[a b]{2}  § aa ou ab ou bb ou ba
```

L'ancrage

Deux caractères spéciaux servent à préciser la position de la chaîne recherchée dans la ligne :
^ désigne le début de la ligne (s'il est placé au début de l'expression)
$ désigne la fin de la ligne (s'il est placé à la fin de l'expression)

Exemples

```
^linux  § linux en début de ligne
linux$  § une ligne terminée par linux
^linux$ § une ligne ne contenant que linux
^$      § une ligne vide
```

14.1.4 Combinaison d'expressions régulières

Deux opérations permettent de combiner entre elles des expressions régulières : **l'alternative** et le **groupage**.

L'alternative

Les expressions régulières permettent de désigner "ceci ou cela" en séparant deux expressions régulières (er) par le caractère **|** comme dans : "*ceci|cela*". Cette combinaison est une nouvelle expression régulière.

Exemples

```
linux|unix          § le mot linux ou le mot unix
^linux|^unix        § le mot linux ou le mot unix au début
[A Z]{8}|[0 9]{4}   § 8 lettres majuscules ou 4 chiffres
```

Le groupage par (...) et la notation \W ...

Mettre une expression régulière entre parenthèses ne change rien à cette expression : l'expression **(er)** représente toute chaîne qui vérifie **er**. Les parenthèses semblent inutiles. Cette notation offre cependant deux possibilités :

- les quantifieurs vus plus haut (voir le paragraphe 14.1.3) peuvent s'appliquer à de telles expressions,
- la notation **\N**, où **N** est un nombre de 1 à 9, désigne la chaîne de caractères qui a vérifié l'expression placée dans la N^{ème} paire de parenthèses.

Exemples

```
tsoin           § tsoin et aucune autre chaîne
tsoin{2}        § tsoinn et aucune autre chaîne :
                § Le quantifieur {2} n'est appliqué qu'au
                § dernier n
(tsoin){2}      § tsointsoin et aucune autre chaîne :
                § Le quantifieur {2} est appliqué à
```

```
                § l'expression tsoin
(bla|tsoin){2}  § blabla ou tsointsoin
                § ou blatsoin ou tsoinbla
(bla|tsoin)\1   § blabla ou tsointsoin
                § MAIS PAS blatsoin ni tsoinbla
```

Attention

\0 désigne le caractère <**null**> et **\N** ne référence la nième paire de parenthèses que si elle existe.

Dans le groupage par **()**, la notation **\N** peut être complétée par ***** **+** ou **?** avec les significations habituelles *:*

- **\1*** signifie 0 à N fois la chaîne qui a vérifié le premier sous-motif
- **\3+** signifie 1 à N fois la chaîne qui a vérifié le troisième sous-motif
- **\2?** signifie 0 ou 1 fois la chaîne qui a vérifié le deuxième sous-motif

Cette notation est appelée *:* référence arrière à une sous-chaîne (back-reference to a sub-string). Elle permet de désigner une chaîne inconnue au moment de l'écriture de l'expression, comme par exemple :
"une chaîne identique au premier mot de la ligne ".

14.1.5 D'autres expressions régulières atomiques

Nous avons présenté les expressions régulières atomiques comme étant un moyen de définir un ensemble de caractères, par exemple :

```
[_A Za z0 9]
```

désigne un caractère alphanumérique ou le signe _.

D'autres possibilités permettent de rendre les expressions régulières atomiques plus complètes (c'est utile) et utilisables avec des jeux de caractères autre que l'ASCII US (en particulier avec des caractères accentués).

- plus complètes : comment désigner un caractère spécial,
- internationales : comment définir une classe de caractères indépendamment de la langue locale.

Les caractères spéciaux et la notation \X

Il est possible de désigner des caractères spéciaux de la façon suivante :

\0	le caractère <**null**>
\a	le caractère <**alert**> (bell)
\b	le caractère <**backspace**>
\f	le caractère <**form-feed**>
\n	le caractère <**new-line**>

\r	le caractère <**carriage-return**>
\t	le caractère <**tab**>
\v	le caractère <**vertical-tab**>
\013	le caractère dont le code ASCII est **013** en octal

Les classes de caractères

Les classes de caractères sont définies par la notation *[:code:]* et ne peuvent être utilisées que dans la définition d'une liste de caractères (era). La correspondance entre ces classes et les éléments qui les composent sont donnés ici pour la langue anglaise :

[[:alnum:]]	un alphanumérique (**[0-9a-zA-Z]**)
[[:alpha:]]	un alphabétique (**[a-zA-Z]**)
[[:cntrl:]]	un caractère de contrôle (**[\a\b\r\f\t\n\v]**)
[[:digit:]]	un digit (**[0-9]**)
[[:graph:]]	un caractère autre qu'alphanumérique ou ponctuation
[[:lower:]]	une lettre minuscule (**[a-z]**)
[[:print:]]	un caractère imprimable
[[:punct:]]	un caractère de ponctuation
[[:space:]]	un caractère d'espacement (**[\t\r\n\f]**)
[[:upper:]]	une lettre majuscule (**[A-Z]**)
[[:xdigit:]]	un digit hexa (**[0-9A-Fa-f]**)

Par exemple en français, **[[:alpha:]]** désigne toutes les lettres alphabétiques **y compris** les caractères accentués, alors que **[a-zA-Z]** ne désigne que les 26 lettres de l'alphabet en minuscule ou majuscule.

Attention

Ne pas oublier **[:** et **:]** dans les classes :

[[:print:]]	correct : un caractère imprimable
[^[:print:]]	correct : un caractère non imprimable
[^:print:]	incorrect
[:^print:]	incorrect
[[:lower:][:digit:]]	correct : une minuscule (y compris accentuée) ou un chiffre. en anglais : **[a-z0-9]**

Exemple détaillé

L'exemple suivant illustre l'utilisation des classes de caractères dans le cas des caractères accentués français : rechercher dans un texte les chaînes représentant un prénom suivi d'un nom (dans la même ligne). Un nom est une suite d'au moins deux lettres majuscules précédées d'un espace et d'un prénom. Un prénom est une suite d'au moins deux lettres, la première étant majuscule et les suivantes minuscules. Le prénom est précédé d'un espacement ou du début de la ligne. Le nom est suivi d'un espace ou d'une ponctuation ou de la fin de la ligne. Les lignes suivantes en sont des exemples :

Ce monsieur s'appelle François GUICHARD, il est assis à droite.
Frédérique OSTRÉ préfère perl : c'est bien compréhensible.
Cet exemple vous est proposé par Pierre COLIN.
Mais cette ligne ne contient PAS ce que l'on cherche.

L'expression régulière est donc construite de la façon suivante :
un nom : **`[[:upper:]]{2,}`**
un prénom : **`[[:upper:]][[:lower:]]+`**
un prénom suivi d'un nom (séparés par un espace) :

```
[[:upper:]][[:lower:]]+ [[:upper:]]{2,}
```

Il faut maintenant exprimer ce qui peut se trouver avant :

```
(^|[[:space:]])
```

... et ce qui peut se trouver après :

```
([[:space:][:punct:]]|$)
```

Et voila : on peut tout mettre ensemble en une seule ligne.
(replié en deux lignes ci-dessous pour des raisons typographiques) :

```
(^|[[:space:]])[[:upper:]][[:lower:]]
+ [[:upper:]]{2,}([[:space:][:punct:]]|$)
```

Cette expression trouvera les trois premières des quatre lignes proposées en exemple. La quatrième ligne ne sera pas prise, bien que « Mais » puisse passer pour un prénom et « PAS » pour un nom (d'après nos critères). Il est important de noter que cette expression est la même pour un utilisateur fancophone et pour un utilisateur anglais : c'est le grand avantage des classes **`[:alpha:]`**, **`[:upper:]`**, **`[:lower:]`** par rapport à une écriture comme **`[A-Za-z]`**, etc.

Le lecteur intéressé pourra améliorer l'expression proposée pour tenir compte des noms composés (OSTRÉ-WAERZEGGERS), des prénoms composés (Jean-Paul), et de la possibilité d'avoir un à trois prénoms comme :

Charles-Henri Stanislas Alfred-Marie VILLEROY-BRANDT.

14.1.6 Les expressions régulières de base

La présentation précédente se veut assez générale, c'est pourquoi les exemples n'ont présenté que les expressions régulières étendues, sans mettre en œuvre les commandes *grep*, *sed* ou *awk*, qui seront présentées dans la suite de ce chapitre.

Les expressions régulières de base, considérées comme obsolètes sous linux, ont une syntaxe différente : les caractères **? + { } ( ) |** n'ont pas de signification spéciale. Il faut les précéder du caractère **** pour qu'ils soient interprétés comme dans les expressions régulières étendues. C'est un point à vérifier dans la documentation en ligne pour chaque utilitaire.

14.2 LA COMMANDE GREP

La commande *grep* est la plus simple. Elle permet de rechercher dans un flot de texte les lignes qui vérifient une expression régulière, et ne transmet sur sa sortie standard que ces lignes. Le *grep* linux comporte trois modes de fonctionnement selon l'option :

grep -G	utilise les expressions régulières de base	défaut
grep -E	utilise les expressions régulières étendues	à préférer
grep -F	recherche des chaînes littérales	**fgrep**

Sous Linux, nous conseillons d'utiliser systématiquement la commande *grep -E*, ce qui permet l'utilisation des expressions régulières étendues. La commande *fgrep* (voir ci-dessous) rend de grands services pour chercher rapidement une chaîne contenant un point ou une étoile. La commande *grep* sans option (équivalente à *grep -G*) ne devrait être utilisée que pour compatibilité avec d'anciens shell scripts. La commande *grep* est décrite en Annexe A, avec des exemples. L'option *-i* permet d'inhiber la différence majuscules/minuscules dans la recherche et l'option *-v* signifie "afficher toutes les lignes qui ne vérifient **pas** l'expression régulière".

Exemples

Le fichier *prodct.f* contient (entre autres) les lignes suivantes :

```
J J518+K
J J*5
paj 5*sqrt(alpha)
```

Les deux exemples suivants montrent la différence entre *grep* et *fgrep* :

```
xstra> grep -i 'j=j*5' *.f
  § ici j*5 est une expression régulière
prodct.f : J J518+K
prodct.f : paj 5*sqrt(alpha)
  § Attention la chaîne j j*5 ne vérifie pas
  § l'expression régulière j j*5
xstra> fgrep -i 'j=j*5' *.f
  § ici * est pris tel quel
```

```
prodct.f : J J*5
xstra>
```

14.2.1 Les caractères spéciaux dans les expressions régulières

Nous allons présenter ici l'interprétation des caractères spéciaux dans une expression régulière sur une suite d'exemples simples mettant en œuvre la commande *grep*.

Soit le fichier *fichier source* contenant les trois lignes suivantes :

```
ceci est un essai
message 1 arrive, continuons
fin de l'essai.
```

***** remplace zéro fois ou n fois le caractère qui le précède.

Exemple

```
xstra> grep 'es*a' fichier_source
ceci est un essai
message 1 arrive, continuons
fin de l'essai.
xstra>
```

. désigne un caractère quelconque.

Exemple

```
xstra> grep 'c.ci' fichier_source
ceci est un essai
xstra>
```

[...] désigne un caractère quelconque appartenant à la liste donnée entre crochets. Deux caractères séparés par un tiret (-) définissent une liste : *[c g]* équivaut à *[cdefg]*.

Exemple

Recherche de toutes les lignes contenant un caractère numérique.

```
xstra> grep '[0 9]' fichier_source
message 1 arrive, continuons
xstra>
```

^ placé en début de motif, désigne le début de la ligne.

Exemple

```
xstra> grep '^c' fichier_source
ceci est un essai
xstra>
```

$ placé en fin de motif, désigne la fin de la ligne.

Exemple

```
xstra> grep 'es*a.$' fichier_source
ceci est un essai
xstra>
```

[^...] désigne une liste de caractères à exclure.

Exemple

Recherche de toutes les lignes contenant autre chose que <espace> et chaînes alphabétiques en minuscule.

```
xstra> grep '[^ a z]' fichier_source
message 1 arrive, continuons
fin de l'essai.
xstra>
```

Le tableau 14.1 présente une synthèse des différences d'interprétation des caractères spéciaux utilisés dans la génération des noms de fichiers (voir le paragraphe 7.7) et les expressions régulières étendues.

	Génération de noms de fichier	Expressions régulières
?	un caractère quelconque, sauf <new-line>	remplace zéro ou une fois le caractère qui précède
.	le caractère point	un caractère quelconque sauf <new-line>
*	zéro ou un nombre quelconque de caractères	remplace zéro fois ou n fois le caractère qui précède
[a-i]	un caractère entre a et i	un caractère entre a et i
[!a-i]	un caractère qui n'est pas entre a et i	un caractère entre a et i, ou !
[^a-i]	un caractère entre a et i, ou ^	un caractère qui n'est pas entre a et i
\	banalise le caractère qui suit	banalise le caractère qui suit
^	le caractère ^	ce qui suit est en début de ligne
$	référence une variable	ce qui précède est en fin de ligne

TABLEAU 14.1. SYNTHÈSE DES DIFFÉRENCES D'INTERPRÉTATION DES CARACTÈRES SPÉCIAUX.

14.3 LA COMMANDE SED

La commande *sed* (Stream EDitor) est un éditeur non interactif. Cette commande peut être utilisée comme un filtre. Elle traite son entrée standard ligne par ligne. La forme générale de la commande *sed* est :

```
sed [ n] [ e commandes_sed] fichier_source
sed [ n] [ f fichier_commande] fichier_source
```

La commande *sed* copie le fichier *fichier source* sur la sortie standard après application des commandes d'édition. L'option *-n* a pour effet d'inhiber la sortie à l'écran du résultat des commandes d'édition. Sans cette option, *sed* copie chaque ligne d'entrée sur sa sortie standard, après l'avoir éventuellement modifiée.

Les commandes d'édition (*commandes sed*) interprétées par *sed* peuvent être stockées dans le fichier *fichier commande* ou placées directement après l'option *-e*.

14.3.1 La commande d'édition de sed

La commande d'édition de *sed (commandes sed)* est de la forme :

```
[adresse_debut [, adresse_fin] ] fonction [arguments]
```

Les adresses peuvent avoir la forme suivante :

- une valeur numérique désignant le numéro de la ligne dans le fichier,
- */motif/* désignant la première ligne contenant *motif*,
- *$* désignant la dernière ligne.

14.3.2 Quelques commandes d'édition

Impression des lignes n à m : la fonction p

```
xstra> sed  n 'n,mp' fichier_source
```

Affichage à l'écran des lignes *n* à *m* du fichier *fichier source*, du fait de la fonction *p (print)*.

Impression sur critère

```
xstra> sed  n '/motif_début/,/motif_fin/p' fichier_source
```

Affichage à l'écran des lignes du fichier *fichier source*, lignes comprises entre la première ligne contenant la chaîne de caractères *motif début* et la première ligne contenant la chaîne *motif fin*.

Substitution d'une chaîne de caractères par une autre : la fonction s

```
xstra> sed 's/ancien_motif/nouveau_motif/g' fichier_source
```

Affichage à l'écran du fichier *fichier source*. Les lignes contenant la chaîne de caractères *ancien motif* seront affichées après substitution d'*ancien motif* par *nouveau motif*, et ceci pour toutes les occurrences d'*ancien motif* ; cela est dû au modifieur *g* qui signifie global. Sans le modifieur *g*, la substitution n'est faite que pour la première occurrence de *ancien motif* pour chaque ligne. Cette commande a été décrite au chapitre traitant *vi* (paragraphe 5.2.3).

14.4 LA COMMANDE AWK

AWK a été écrit par A.V. Aho, P.J. Weinberger, B.W. Kernighan, d'où son acronyme. *awk* est un langage de programmation spécialisé dans la manipulation de texte, et optimisé dans le sens d'une extrême concision : beaucoup de programmes *awk* ne font qu'une seule ligne. La logique de fonctionnement de *awk* est celle de *sed* : la boucle sur les lignes du flot d'entrée est implicite, le programme à appliquer est passé en paramètre ou enregistré dans un fichier dont le nom est passé en paramètre.

14.4.1 Caractéristiques du langage

- *awk* supporte des opérateurs arithmétiques et de manipulation de chaînes
- *awk* supporte les expressions régulières étendues
- La boucle sur les lignes d'entrée est implicite
- Les variables *awk* sont non déclarées et non typées, normales ou spéciales

Les variables spéciales : numéro de ligne, nombre de champs, longueur, nom du fichier, etc. existent toujours et sont mises à jour automatiquement au cours de l'exécution.

L'utilisateur peut définir des variables : elles sont simultanément de type chaîne **et** numérique, le contexte d'utilisation déterminant automatiquement la conversion appliquée par *awk*. De plus, ces variables sont créées et initialisées automatiquement à zéro (ou à la chaîne vide) dès leur première apparition dans le programme *awk*.

14.4.2 La ligne de commande awk

Deux cas peuvent se présenter : soit le programme *awk* est suffisamment court pour être placé entre apostrophes sur la ligne de commandes, soit il est plus long et enregistré dans un fichier dont le nom est passé à la commande *awk* avec l'option *-f*. L'option *-Fd* permet de fixer le séparateur de champs au caractère *d*.

La première forme de la commande *awk* est la suivante :

```
awk [ Fd] 'programme_awk' [fichiers]
```

Important

Le programme *awk* est placé entre apostrophes simples, car il comporte presque toujours des caractères susceptibles d'être interprétés par le shell.

Exemple

```
xstra> awk -F: '$5 == ""' /etc/passwd
```

Ce qui signifie : dans le fichier */etc/passwd*, sélectionner les lignes dont le cinquième champ est vide, le séparateur de champs étant le caractère :

La deuxième forme de la commande *awk* est la suivante :

```
awk [ Fd]  f fichier_awk [fichiers]
```

Exemple

```
xstra> ls  l | awk -f formatls
```

Ce qui signifie : appliquer le programme *awk* enregistré dans le fichier *formatls* à toutes les lignes produites par la commande *ls l.*

awk lit une ligne sur son entrée standard, la découpe en champs selon le délimiteur de champs et lui applique toutes les instructions du programme *awk* (la copie sur la sortie standard n'est pas automatique). Dans toute la suite le terme « **sortir** » signifie : recopier sur la sortie standard.

14.4.3 Le programme awk

Un programme *awk* est une suite de lignes du type :

```
condition { action }
```

Le champ *condition* ou le champ *action* peut être omis.

Un champ *condition* manquant est vérifié par toute ligne d'entrée.

Un champ *action* manquant sort toute ligne qui vérifie *condition.*

Exemple

```
xstra> awk '$1 == 5 {print $2}' fich1
  §  condition : si le premier champ vaut 5
  §  action : sortir le deuxième champ
```

Le champ condition

Le champ *condition* peut être soit une condition de base, soit une condition composée.

Les conditions de base sont les suivantes :

Condition	À quel moment l'action correspondante est appliquée

BEGIN	Avant lecture de la première ligne d'entrée.
END	Après lecture de la dernière ligne d'entrée.
expr	À toute ligne d'entrée pour laquelle l'expression **expr** est vraie, c'est à dire différente de zéro ou non vide (selon le contexte, évaluation numérique ou alphanumérique).
/er/	À toute ligne d'entrée contenant une chaîne vérifiant l'expression régulière **er**.
expr ~/er/	À toute ligne d'entrée pour laquelle la valeur alphanumérique de **expr** contient une chaîne vérifiant l'expression régulière **er**.
expr !~/er/	À toute ligne d'entrée pour laquelle la valeur alphanumérique de **expr** ne contient pas une chaîne vérifiant l'expression régulière **er**.

Les conditions de base, sauf **BEGIN** et **END**, peuvent être composées entre elles selon les combinaisons suivantes :

Condition	**À quel moment l'action correspondante est appliquée**
cond1 && cond2	À toute ligne d'entrée vérifiant **cond1 et cond2**.
cond1 || cond2	À toute ligne d'entrée vérifiant **cond1 ou cond2**.
!cond	À toute ligne d'entrée ne vérifiant **pas cond**.
cond1, cond2	À toute ligne d'entrée à partir de la première ligne vérifiant **cond1** et jusqu'à la prochaine ligne vérifiant **cond2** (incluses).

Les variables prédéfinies

Nom de variable	**Contenu**
FILENAME	Nom du fichier d'entrée courant
FNR	Numéro de ligne dans le fichier d'entrée courant
FS	Séparateur de champs en entrée
IGNORECASE	Si différent de 0, ne jamais faire de différence entre majuscule et minuscule
NF	Nombre de champs dans la ligne courante
NR	Numéro de ligne dans le flot d'entrée
OFS	Séparateur de champs en sortie
ORS	Séparateur de lignes en sortie
RS	Séparateur de lignes en entrée
$0	Ligne d'entrée courante
$1,$2,...,$NF	1er, 2ème, ..., dernier champ de la ligne d'entrée courante

Les expressions

Les expressions dont il est question ici ne doivent pas être confondues avec des expressions régulières. Il s'agit d'expressions composées à partir de variables et d'opérateurs, et que *awk* peut évaluer numériquement ou sous forme de chaîne selon le contexte. Toute expression peut également être considérée comme un booléen, la valeur vrai signifiant différent de zéro dans un contexte numérique et non vide dans un contexte de chaîne.

Exemples

Expression	Signification
$1+$3	La somme des 1er et 3ème champs.
NF > 5	Vrai si le nombre de champs est supérieur à 5.
$4 == ""	Vrai si le 4ème champ est vide.
($1+$2)/NF	La somme des 1er et 2ème champs divisée par le nombre de champs.
$7 ~/ksh/	Vrai si le 7ème champ contient la chaîne `ksh`.

Attention

Ne pas confondre NF et $NF

Pour un habitué de la programmation en shell débutant en *awk*, une erreur fréquente consiste à référencer une variable par *$variable* au lieu de *variable*. Tout particulièrement pour la variable *NF*, nombre de champs, *NF* représente le nombre de champs alors que *$NF* représente le dernier champ. En effet, si la ligne courante comporte 5 champs, *NF* vaut 5 et *$NF* vaut *$5* donc le 5ème (et dernier) champ.

Les opérateurs de comparaison s'appliquent aussi bien à des comparaisons numériques que lexicographiques, sauf **~** et **!~**. Ces opérateurs sont les suivants :

Opérateur	Signification	Exemples numériques et / ou lexicographiques
==	Egal à	**1** est égal à **1** et **linux** est égal à **linux**
!=	Différent de	**linux** est différent de **DOS**
>	Supérieur à	**linux** est supérieur à **dos**
<	Inférieur à	**dos** est inférieur à **dosread**
>=	Supérieur ou égal à	
<=	Inférieur ou égal à	
~	Vérifie l'ere	**/usr/bin/ksh** vérifie **/ksh/**
!~	Ne vérifie PAS l'ere	**/home/yannick** ne vérifie pas **/[0-9]+/**

Les actions

L'action par défaut est *print* : copier la ligne d'entrée courante vers la sortie standard. Les actions possibles sont celles d'un langage de programmation conventionnel, et la syntaxe est voisine de celle du langage C.

Très brièvement, les actions possibles sont des combinaisons de :

- affectation de variables, incrémentation, décrémentation
- opérations arithmétiques dont un certain nombre de fonctions prédéfinies telles que *sin*, *cos*, *atan2*, *exp*, *log*, *sqrt*, *rand*,..
- concaténation de chaînes *(var1 var2)*
- fonctions prédéfinies sur les chaînes telles que *index*, *length*, *split*, *match*, *sub*, *substr*, *sprintf*,..
- opérations de sortie telles que *print*, *printf*, *getline*,.. avec redirection possible vers un fichier ou un tube
- structures de contrôle telles que *if*, *else*, *do*, *while*, *for*,..

Une description complète de *awk* nécessiterait un livre entier, et le lecteur intéressé est invité à se référer au *man* de *awk* ou à un ouvrage plus spécialisé. Nous préférons présenter des exemples mettant en évidence la simplicité et l'extrême concision des programmes *awk*.

Exemple

Tout d'abord quelques exemples d'analyse du fichier */etc/passwd*. Dans ce fichier, les différents champs sont séparés par le caractère :, *awk* est donc invoqué avec l'option *-F:* pour plus de simplicité. Une solution équivalente, mais moins élégante, consisterait à commencer le programme *awk* par une ligne *BEGIN{FS=":"}*. Rappelons que chaque ligne du fichier */etc/passwd* décrit un utilisateur du système selon les champs :

```
login:!:uid:gid:gecos:home_dir:login_shell
1    :2: 3 : 4 :  5  :    6   :     7
yannick:!:202:245:perl
guru:/home/yannick:/usr/bin/ksh
```

Parmi les *login* décrits dans le fichier */etc/passwd*, certains correspondent à des utilisateurs et d'autres sont des *login* d'administration ne correspondant pas à des personnes. Voici quelques exemples de sélection faciles avec *awk* et qui seraient impossible avec *grep* : chercher les lignes comportant un certain motif dans un certain champ (la notion de champ n'existe pas pour *grep*).

```
#lister les login ayant un login shell
xstra> awk -F: '$7 != "" {print $1}' /etc/passwd
#peut s'écrire (plus bref, mais moins lisible)
xstra> awk -F: '$7 {print $1}' /etc/passwd

# lister les login et uid pour lesquels
# le login shell est un shell bash
xstra> awk -F: '$7 ~/bash/ {print $1,$3}' /etc/passwd
```

```
# lister les logins dont l'UID est inférieur à 100
# et qui n'ont pas de login shell
# l'action par défaut est print
xstra> awk -F: '$3 <= 100 && $7 == ""' /etc/passwd
```

Exemple

Voici un autre exemple de programme *awk* très court, et néanmoins très utile. Il sera abondamment commenté. Il s'agit de compter le nombre de mots et de lignes dans un fichier texte.

```
xstra> awk '{mot=mot+NF}END{print NR,mot}' fichtext
```

Explications pas à pas :

La variable mot est créée et initialisée à zéro par *awk*, puis pour chaque ligne (condition manquante) exécuter l'action : mot = mot + nombre de champs dans la ligne courante. Après avoir lu et traité la dernière ligne (condition END) exécuter l'action : sortir le nombre de lignes et le contenu de la variable mot.

Attention à la virgule

`print NR,mot`	sortir NR et mot séparés par un espace : 12 63
`print NR mot`	sortir la chaîne obtenue par concaténation de NR et mot : 1263

Exemples

Dans les exemples suivants, nous ne présenterons que le programme *awk* et non plus la ligne de commandes toute entière comme dans l'exemple précédent.

Sortir chaque ligne précédée de son numéro :

```
{ print NR,$0}
```

Sortir chaque ligne précédée de son numéro, ignorer les lignes commençant par le caractère # :

```
!/^#/{ n++;print n,$0}
```

Sortir les 15 premières lignes :

```
 NR <  15
```

Sortir le 1er champ, le 2ème champ, leur somme et leur différence :

```
{ print $1,$2,$1+$2,$1 $2}
```

Compter combien de lignes dépassent 80 caractères :

```
length($0) > 80 { n++}
END { print n, "lignes de plus de 80 car."}
```

Sortir les lignes dans lesquelles la somme de tous les champs est inférieure à 100 :

```
{ som 0
  for (i 1;i< NF;i++) som som+$i
  if (som < 100) print
}
# ce qui peut aussi s'écrire
{ som 0;for(i 1;i< NF;i++)som som+$i;if(som < 100)print}
```

14.5 LA COMMANDE TR

La commande *tr* (TRanslate) existe en deux versions légèrement différentes selon l'origine du système, la version system V se trouvant généralement dans */usr/bin/tr* et la version BSD dans */usr/ucb/tr*. La commande *tr* est un filtre strict : elle traite son entrée standard et produit le résultat sur sa sortie standard. Pour traiter le contenu d'un fichier, la redirection de l'entrée standard sur le fichier doit être explicite. Cette commande peut être utilisée sur des données non textuelles et ne connaît pas les expressions régulières. Selon les options et les paramètres qui lui sont passés, *tr* effectue trois types de transformation sur le flot d'entrée :

- transcodage : *tr* sans option : *tr string1 string2*
- suppression de certains caractères : option *-d* : *tr d string1*
- suppression de répétitions : option *-s* : *tr s string1*

Dans le premier cas, le transcodage, *tr* remplace chaque caractère du flot d'entrée appartenant à la chaîne *string1* par le caractère de même rang dans la chaîne *string2*. Dans toutes les utilisations de *tr*, la notation *a z* est autorisée pour représenter la suite des caractères de *a* à *z* (inclus) et les caractères non imprimables sont représentés par leur code ASCII en octal (\015 : le caractère <carriage-return>.

Les exemples suivants montrent le transcodage :

```
# remplacer "a" par "A", "," par ";" et "/" par "_"
xstra> tr 'a,/' 'A;_' <fich1 >fich2
# remplacer $ par F, () par {} et <tab> par <espace>
# <tab> vaut 011 en octal
xstra> tr '$()\011' 'F{} ' <fich1 >fich2
# transformation majuscules vers minuscules
xstra> tr 'A Z' 'a z' <fich1 >fich2
```

Dans le deuxième cas, suppression de certains caractères, *tr* supprime tout caractère du flot d'entrée appartenant à la chaîne *string1*. Voici un exemple très utile :

```
# supprimer les caractères <carriage return> et Ctrl Z
# ce qui convertit un fichier texte ASCII de
# DOS vers Unix
xstra> tr  d '\015\032' <fich1 >fich2
```

Dans le troisième cas, suppression de répétitions de certains caractères, *tr* recherche dans le flot d'entrée toute séquence constituée de plusieurs exemplaires consécutifs d'un caractère appartenant à la chaîne *string1* et remplace cette séquence par un seul exemplaire du même caractère. Voici quelques exemples :

```
# remplacer plusieurs espaces consécutifs par un seul
xstra> tr  s ' ' <fich1 >fich2
# remplacer plusieurs <new line> consécutifs par un seul
xstra> tr  s '\012' <fich1 >fich2
# ce qui peut se combiner comme ici pour les séquences
# de <espace> ou <tab> ou <new line>
xstra> tr  s ' \011\012' <fich1 >fich2
```

Pour terminer, voici un exemple mettant en évidence l'élégance du mécanisme des tubes Unix permettant d'associer très simplement des briques de base. Le problème posé consiste à construire le dictionnaire alphabétique de tous les mots d'un texte. Les étapes du traitement sont les suivantes :

tr coupe le texte en mots,

sort trie ces mots par ordre alphabétique et élimine les doublons

Le résultat final est redirigé vers le fichier résultat.

```
# dictionnaire alphabétique de tous les mots d'un texte
xstra> tr  cs 'a zA Z' '\012' < text | sort  u > dico
```

14.6 EXERCICES

Exercice 14.6.1

Recherchez dans le fichier */etc/passwd* :

- L'entrée correspondant à votre login (solution indépendante du login).
- Le nombre de lignes contenant la chaîne de caractères : MICHEL ou michel.
- Le nombre de lignes contenant la chaîne de caractères : */home/*.

Exercice 14.6.2

Recherchez dans le fichier */etc/inetd.conf* :

- Le nombre de lignes contenant un point .
- Le nombre de lignes ne commençant pas par # .
- Toutes les lignes de plus de 50 caractères.
- Toutes les lignes de plus de 50 caractères (hors commentaires).

Exercice 14.6.3

Écrivez une commande *awk* permettant de lister le contenu d'un fichier sans les lignes vides et en les numérotant. Testez votre solution avec l'entrée standard.

Exercice 14.6.4

Le fichier annuaire.txt contient des adresses sous la forme suivante :
Prénom NOM Date-de-naissance Sexe Numéro-de-téléphone
Le fichier est supposé très long, et il peut contenir des erreurs comme dans l'exemple ci-dessous :

```
Georges TARTEMPION 12 2 1967 M 038898422524
Hortense HYABLANT 7 4 1944 F 0146872556
Adrien DUBOUCHON 27 11 1938 M 0154896372
Ludwig SCHMILLBLEERCK 28 12 1957 M 0258459887
Antonio VAVILDA 16 5 1937 M
Hughes CARPETTE 8 11 1958 M 0123568629
J'en ai assez de ce travail fastidieux!
Dagobert ELOY 14 7 1947 M 0225415487
      ligne vide
Anatole SUISSE 1 2 1965 n 4
Antonino SZWPRESWKY 16 5 8937 M 0298358745
```

Écrire les trois commandes *grep* permettant de :
- supprimer les lignes vides ;
- trouver les gens qui n'ont pas le téléphone ;
- trouver les lignes non conformes ;

Écrire une solution *awk* pour le dernier cas.

Exercice 14.6.5

Listez les noms de login de tous les utilisateurs de votre système qui n'ont pas de programme de démarrage (dans */etc/passwd*).

Exercice 14.6.6

Les messages électroniques SMTP sont constitués exclusivement de caractères ASCII imprimables (pas de caractères spéciaux). Leur structure est simple : un en-tête précède le corps du message.
L'en-tête commence toujours par une ligne : "From " (From suivi d'un espace). La fin de l'en-tête est toujours signalée par une ligne vide. Vous savez que votre boite d'entrée est le fichier */var/spool/mail/$LOGNAME*. Vous savez donc tout ce qu'il faut savoir pour écrire un shell script qui envoie sur sa sortie standard :
- le nombre de messages en attente dans votre boite d'entrée ;
- les champs "From: " et "Subject: " du dernier message.

Exercice 14.6.7

Améliorez le script précédent pour qu'il fasse son travail deux fois par minute.

Exercice 14.6.8

Améliorer le script précédent pour qu'il reste silencieux si aucun message n'est arrivé dans la période qui précède.

Exercice 14.6.9

Si vous travaillez dans l'environnement X11, lancez ce script en background dans une petite fenêtre dans un coin de votre écran. Il est supérieur au programme `xbiff` à deux points de vue :
- vous savez si le message qui vient d'arriver mérite le dérangement ;
- vous avez la satisfaction de l'avoir fait vous-même.

Exercice 14.6.10

Écrire un script qui lit un message sur son entrée standard et recopie :
- l'en-tête du message dans le fichier "./_header" ;
- le corps du message dans le fichier "./_body".

Un tel script est l'outil de base permettant d'automatiser le traitement des messages à l'arrivée. Écrire une solution `sed` et une solution `awk`.

Exercice 14.6.11

Chercher un motif dans un fichier avec `egrep`, c'est facile :

```
egrep 'ere' fichier
```

Remplacer une chaîne qui vérifie une expression régulière par une nouvelle chaîne, c'est facile aussi, mais moins, et c'est surtout long à taper, et il faut rediriger, ... Ce qu'il faudrait, c'est un sed simplifié "à la grep", nommé `remplace`, à utiliser dans les cas tous simples, comme par exemple :

```
remplace 'pays' 'paysage' *.txt
```

Ce qui signifie : dans tous les fichiers d'extension `.txt`, remplacer toute occurrence de la chaîne `pays` par `paysag`e.

Bien sûr, comme pour `egrep`, le premier argument est une expression régulière.