Système et réseaux (SR1)

Plusieurs outils de programmation sont à disposition des utilisateurs sous Unix. Nous allons passer en revue principalement le langage de programmation système (le *shell*) et un langage de traitement des fichiers textes (*awk*). Nous verrons aussi les bases des expressions régulières (utilisées par de nombreuses instructions, dont *expr* et *grep*, ainsi que par l'éditeur de texte *awk*).

1) Les expressions régulières

Les expressions régulières sont utilisées dans de nombreux contextes (programmation *shell*, *Perl*, *awk*, …). Elles permettent d'écrire des **motifs** (fragments de texte) qui seront cherchés dans une ligne de texte. Le motif peut ne pas être exactement le fragment de texte à chercher mais une description générale. Les règles d'écriture des motifs sont assez complexes ; nous n'en utiliserons qu'une infime partie.

Les principaux éléments constituant une expression régulière sont :

- les caractères tels qu'ils doivent figurer dans le filtre, comme par exemple :

  ```
  CM3 reconnaît une ligne qui contient CM3 (par exemple, CM3, CM30, ACM345)
  ```

- un joker, le point, qui remplace n'importe quel caractère. Comme par exemple :

  ```
  P.NG reconnaît une ligne qui contient par exemple PING, PONG, P5NG, P(NG,etc.
  ```

- des listes de caractères. Elles sont placées entre crochets et contiennent éventuellement des intervalles de valeurs (valeur initiale et finale séparées par un tiret), comme par exemple :

  ```
  [AZERTY]2 reconnaît une ligne qui contient: A2 Z2 E2 R2 T2 Y2, etc.
  temps[0123] reconnaît une ligne qui contient temps0 temps1 temps2 temps3, etc.
  temps[0-9] reconnaît une ligne qui contient temps suivi d'un chiffre
  [a-zA-Z] reconnaît une ligne qui contient une lettre minuscule ou majuscule
  ```

- des facteurs de répétition (* ou + chiffres entre accolades) qui permettent de répéter plusieurs fois ce qui précède, comme par exemple :

  ```
  [A-Z]*2 reconnaît une ligne qui contient 0 ou plusieurs lettres majuscules suivies de 2
  temps[0-9]+ reconnaît une ligne qui contient temps suivi de 1 ou plusieurs chiffres
  [a-z]{4}3 reconnaît une ligne qui contient quatre lettres minuscules suivies du chiffre 3
  [a-zA-Z]{2,4} reconnaît une ligne qui contient deux, trois ou quatre lettres
  .* reconnaît n'importe quelle ligne
  ```

- des caractères spéciaux pour indiquer la position du filtre dans la ligne :

  ```
  ^[A-Z] reconnaît une ligne qui commence par une lettre majuscule
  [0-9]$ reconnaît une ligne qui se termine par un chiffre
  \<[A-Z] reconnaît les mots qui commencent par une lettre majuscule (avec egrep, awk)
  ent/> reconnaît les mots qui se terminent par ent (avec egrep, awk)
  ```

- des parenthèses et du pipe | pour indiquer des variantes possibles dans l'expression :

  ```
  (21|58|71|89)000 les codes postaux des préfectures de Bourgogne
  ```

- des parenthèses seules pour récupérer une partie d'expression :

  ```
  ([0-9]{2})[0-9]{3}) reconnaît un code postal de 5 chiffres et récupère dans une variable le numéro de département.
  ```

  La récupération peut prendre plusieurs formes. Avec awk, il faut utiliser match() qui indique le nom de la variable, en Perl les variables $1, $2 … sont utilisées.

- les signes « spéciaux » comme par exemple le point . ou les crochets [] … à condition de les préfixer par un anti-slash \. Par exemple :

  ```
  [a-z]\.{1,3}[a-z] reconnaît une ligne avec un à trois points entre deux minuscules
  ```

2) Le shell

Le shell est le langage de programmation système sous Unix. Il en existe plusieurs versions, les plus utilisées étant *sh* et *bash*. Vous avez déjà utilisé le shell « en ligne de commande », dans des terminaux, lors des essais faits en TP. Nous allons nous intéresser maintenant à la programmation proprement dite. Les programmes shell sont structurés en fichiers qui doivent être exécutables ; un fichier est donc un programme. Comme tous les autres processus, les programmes shell sont associés :

- à trois voies d'entrée-sortie par défaut,
- à un code retour (renvoyé par exit et récupérable dans la variable $?),
- à un pid (récupérable dans la variable $$).

Un programme shell peut recevoir des paramètres à l'appel (dans les variables d'énumération).

Les bases du langage

Les **variables** sont propres à chaque programme (pour les rendre disponibles aux sous-processus, vous pouvez utiliser la clause `export`). Les noms de variables sont constitués de lettres, chiffres et souligné ; le premier caractère ne peut pas être un chiffre. Le contenu d'une variable est désigné par $ suivi de son nom (par exemple, variable de nom `A` et de contenu `$A`). En cas d'ambiguïté, vous pouvez indiquer la portée du dollar par des accolades (par exemple `${A}`).
Le mécanisme de base du shell est l'**affectation** de valeurs dans des variables **combinée avec un mécanisme de substitution** (une variable est remplacée par sa valeur pour constituer une chaîne de caractères). L'affectation est notée par le signe = (par exemple A=4). Attention, il ne faut pas d'espace autour du signe =. Vous pouvez supprimer une variable par `unset` et la protéger (plus d'affectation, ni de unset) par `readonly`.

Vous disposez en shell de toutes les **instructions** des langages de haut niveau :

- les conditions sont écrites avec l'instruction `test` qui permet de comparer des valeurs entières (-ne -eq -ge -le -gt -lt) ou des chaînes de caractères (= et !=) ainsi que de tester la longueur de chaînes de caractères (-z -n) et l'existence de fichiers (-f -d, etc.). Attention, pour la comparaison des chaînes de caractères il faut des espaces autour de l'opérateur de comparaison.
  Les opérateurs logiques sont notés ! -a -o (pour not, and et or, respectivement).
  Il est possible de parenthéser les conditions avec \(et \) ;

- les instructions conditionnelles sont :

`if` condition
`then` action
`else` action
`fi`
	`case` contenuVariable `in`
	pattern) action ;;
	pattern) action ;;
	*) action ;;
	`esac`

dans les imbrications, il est possible d'utiliser `elif` à la place de `else if`	attention : les patterns ne respectent pas exactement le format des expressions régulières : **?** remplace un et un seul caractère ***** remplace toute chaîne de caractères **a|b** signifie a ou b **[abc]** et **[0-9]** : comme sur les expressions régulières

- les itérations sont :

`while` condition `do` action `done`	`until` condition `do` action `done`	`for` variable `in` liste `do` action `done`

  La liste de valeurs d'une boucle `for` utilise l'espace comme séparateur. Cette instruction peut aussi prendre la forme indiquée ci-dessous afin de traiter toutes les entrées du répertoire désigné par le chemin :

```
for variable in chemin/*
    do action
    done
```

- pour effectuer des calculs, vous pouvez utiliser l'instruction `expr` :
  - sur des entiers, avec les opérateurs + - * / %. Par exemple expr 2 + 4. Pour tout autre calcul, utilisez plutôt `dc ;`
  - sur des chaînes de caractères avec l'opérateur : comparer les chaînes et renvoyer le nombre de caractères identiques. Par exemple `expr linux : ligne` renvoie 2.
  - expr avec l'opérateur de comparaison de chaînes (:) peut être utilisée avec comme second argument une expression régulière. Si vous utilisez des parenthèses \(et \) pour récupérer une partie de l'expression, elle constituera le résultat de l'exécution de `expr`.

- Vous pouvez afficher une chaîne de caractère avec l'instruction `echo` (option -n pour ne pas passer à la ligne) et lire au clavier avec `read`. L'instruction `read` peut être utilisée dans une boucle `while` pour lire ligne à ligne un fichier : elle renvoie « faux » en fin de fichier.

- Vous pouvez découper une liste en mots (séparateur dans la variable IFS qui contient par défaut espace, ainsi que tabulation et newLine) avec l'instruction `set` qui utilise les variables systèmes :
  - **$1 $2 $3** etc. pour récupérer les valeurs dans l'ordre,
  - la variable **$#** pour avoir le nombre de variables définies,
  - l'instruction **shift** pour décaler les variables ($1 disparaît et est remplacé par $2, etc.),
  - la variable **$*** contient la liste complète.

  Par exemple, après avoir exécuté `set L3 informatique Dijon` la variable `$2` contient `informatique`. Attention, les

variables système sont utilisées par de nombreuses commandes. Il faut donc en récupérer rapidement les valeurs.

Sous-programmes et paramètres

Un sous-programme est contenu dans un fichier (qui doit être exécutable). Les paramètres passés à l'appel sont récupérables sous forme d'une liste de valeurs avec les variables systèmes.La variable **$0** contient le nom du programme. Il est important de bien vérifier tous les paramètres devant être reçus par un sous-programme. La plupart des programmes shell vont donc commencer par :

```
if test $# -ne 3
then echo "il faut trois paramètres : ..."
       exit
fi
```

Il peut être nécessaire de vérifier qu'un nom de fichier/répertoire passé en paramètre correspond bien à un fichier/répertoire existant (options `-f -d` de la commande `test`) et sur lequel vous avez les droits d'accès nécessaires (options `-r -w -x` de la commande `test`).

Le code retour est indiqué dans l'instruction de fin du programme, `exit code`. Après exécution, il est possible de récupérer dans la variable **$?** le résultat renvoyé par l'exécution du programme (un entier positif ou nul, inférieur à 256).

Attention, chaque programme est exécuté par un processus qui lui est propre. Il est possible de lancer un sous-programme en arrière-plan (avec &). Son numéro de processus est alors contenu dans la variable **$!** (rappel : le numéro du processus en cours est dans la variable **$$**).
L'instruction `wait pid` permet d'attendre la fin du processus fils lancé en arrière-plan dont le numéro est indiqué.

Les notations particulières

Plusieurs notations particulières doivent être maîtrisées pour programmer en shell :

- Un **commentaire** est introduit par # et se poursuit jusqu'en fin de ligne
- Certains blocs d'instructions ne peuvent pas être vides mais vous pouvez utiliser l'**instruction vide** indiquée par le caractère « : »
- Pour suivre l'exécution d'un programme shell, vous pouvez placer dans ce programme l'instruction **set** dont l'option -x a pour effet de faire afficher chaque ligne telle qu'elle sera exécutée avant son exécution.
- Pour garantir que l'exécution d'un programme se fera bien dans le shell pour lequel il a été écrit, il est conseillé de placer en première ligne du fichier une directive indiquant quel shell doit être utilisé

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

- Il existe plusieurs modes de lancement d'instructions :
  - les blocs avec () qui prennent la forme **(instr1; instr2; instr3)** permettent de placer plusieurs instructions sur la même ligne. Ces instructions sont exécutées dans un nouveau shell (même pid mais l'environnement peut être modifié et il disparaît en fin de bloc).
  - les blocs avec {} qui prennent la forme **{instr1; instr2; instr3; }**. Les instructions sont exécutées dans le shell courant (les modifications de l'environnement sont conservées)
- Il existe plusieurs **délimiteurs** de chaînes de caractères. Les quotes doubles " permettent la substitution alors que les quotes simples ' la rendent inopérante. Les anti-quotes ` indiquent que la chaîne doit être exécutée et « remplacée » par le résultat de son exécution. Par exemple A=`expr $A + 1` incrémente la variable A.
- L'instruction **eval** permet de donner, sous forme de variables, les différents composants d'une instruction. Par exemple si les variables A1, A2 et OP contiennent respectivement 5, 10 et +, l'appel à `eval expr $A1 $OP $A2` renvoie `15`.
- L'instruction **exec** permet de lancer un nouveau processus pour la suite du programme. Elle sera souvent utilisée pour rediriger la lecture des données depuis un fichier. Attention la redirection est active jusqu'à la fin du programme. La structure usuelle est :

```
exec < fichier
while ( read ligne )
do
done
```

- Le lancement d'un programme avec la notation **. ./nom** permet de l'exécuter dans le même shell (même environnement).
- La fin de fichier peut être simulée au clavier par `CTR-D`.

Ordre des actions en vue d'exécuter une ligne de commande :

- retrouver les mots (en tenant compte des séparateurs actifs)
- isoler les chaînes non modifiables (entre quotes simples ')
- remplacer les variables par leur contenu
- remplacer les commandes entre anti-quotes (`) par le résultat de leur interprétation
- évaluer les expressions arithmétiques
- isoler les chaînes modifiables (entre quotes doubles ")
- remplacer les caractères * [] ? ~ etc., par leur valeur
- repérer les séquences () et {}

- prendre en compte les caractères | & && || etc.
- mettre en place les redirections
- localiser les commandes (via la variable PATH) sauf pour les commandes internes

3) Awk

Awk est un filtre proposé sous Unix pour le traitement des fichiers texte mais il peut travailler sur la base d'un programme qui indique pour chaque schéma de ligne lue ce qu'il convient de faire. Chaque instruction est de la forme :

```
pattern { action }
```

Il existe plusieurs façons de lancer l'exécution de awk :

- `awk 'liste d'instructions' fichier` qui applique les instructions au fichier
- `awk -f programme.awk fichier` qui applique les instructions du programme au fichier
- `awk < fichier 'liste d'instructions'` qui travaille en redirection de stdin
- `commande | awk 'liste d'instructions'` qui applique la liste d'instructions au résultat de la commande

Le fichier est découpé en lignes et chaque ligne est découpée en mots. Les lignes sont lues une à une et les mots sont disponibles dans les variables d'énumération **$1 $2 $3** … avec **$0** pour la ligne complète. Le séparateur de lignes peut être redéfini, ainsi que le séparateur de mots, en modifiant des variables :

- FS contient le séparateur de mots en entrée (par défaut espace)
- OFS contient le séparateur de mots en sortie (pur les instructions print)
- RS contient le séparateur de lignes en entrée (par défaut \n)
- ORS contient le séparateur de lignes en sortie (par défaut \n)

Attention, si RS est vide (RS="") chaque fichier sera lu en une seule fois .

Plusieurs variables vous permettent de vous « situer » dans les fichiers en cours d'analyse :

FILENAME le nom du fichier
FNR le numéro de la ligne courante dans le fichier en cours d'analyse
NF est le nombre de champs dans la ligne courant (99 champs maximum)
NR le nombre de lignes lues depuis le début de l'exécution

La variable OFMT contient le format par défaut pour l'impression des numériques (initialement "%.6g" soit avec 6 chiffres maximum en limitant si nécessaire le nombre de décimales ou en utilisant une puissance de 10). Vous trouverez le détail des formats d'impression dans le document de H.Wertz, page 50 et suivantes.

Pour écrire les patterns, vous devez utiliser :

- les expressions régulières avec ~ pour le match et !~ pour l'absence de match,
- des comparateurs (sur les nombres ou les chaînes de caractères) :

  == != > >= < <=

Il existe aussi quelques patterns prédéfinis comme :

BEGIN pour introduire une action qui sera exécutée une seule fois avant de traiter la première ligne du fichier
END pour introduire une action qui sera exécutée une seule fois après le traitement de la dernière ligne du fichier

Pour écrire les actions vous disposez :

- des opérateurs :
  + - * / % ainsi l'incrémentation et la décrémentation ++ et - - sur les numériques,
  la concaténation des chaînes de caractères est indiquée par un espace,
- les actions sont terminées par **;**
- des instructions :
  if (*condition*) *instruction* else *instruction*
  while (*condition*) *instruction*
  do *instruction* while (*condition*)
  for (*initialisation* ; *condition* ; *incrémentation*) *instruction*
- de tableaux dont l'indice peut être une chaîne de caractères, notés par exemple A[1] ou A["un"] avec une boucle de parcours du tableau :

  for (*variable* in *tableau*) *instruction*
- de fonctions prédéfinies comme :
  - gsub(/ancien/,"nouveau") pour substituer globalement dans la ligne courante
  - sub(/ancien/,"nouveau") pour substituer (une seule fois) dans la ligne courante
    si vous utilisez des parenthèses pour récupérer une partie de l'expression régulière décrite dans ancien, vous pouvez la réutiliser dans nouveau avec la notation **\0**
  - length() pour récupérer la longueur de la ligne courante
  - tolower(chaîne) et toupper(chaîne)

- int(numérique) pour la partie entière
- rand() choix au hasard entre 0 et 1
  srand() pour initialiser le générateur aléatoire
- print sans argument affiche $0 (toute la ligne en cours d'analyse)
  print $2 $3 concatène les valeurs des variables
  print $2, $3 sépare par le caractère défini par OFS (par défaut un espace)
  sprintf(format,liste) pour afficher selon un format précis

- attention, il n'y a pas de $ devant les noms de variables.

Les notations particulières

- commentaires #
- en début de programme #!/bin/awk -f

Bibliographie

Unix sous tous les angles, A. Janssens, Eyrolles, 1991.
Maitrise des expresssions régulières, J.E.F. Friedl, O'Reilly, 2003.

Pages web

Bash Reference Manual : http://www.gnu.org/software/bash/manual/bashref.html
Shell, D. Bouillet, Telecom SudParis: http://www-inf.it-sudparis.eu/cours/UNIX/Shell/
Le langage awk, H. Wertz, www.ai.univ-paris8.fr/~hw/unx5.pdf

