

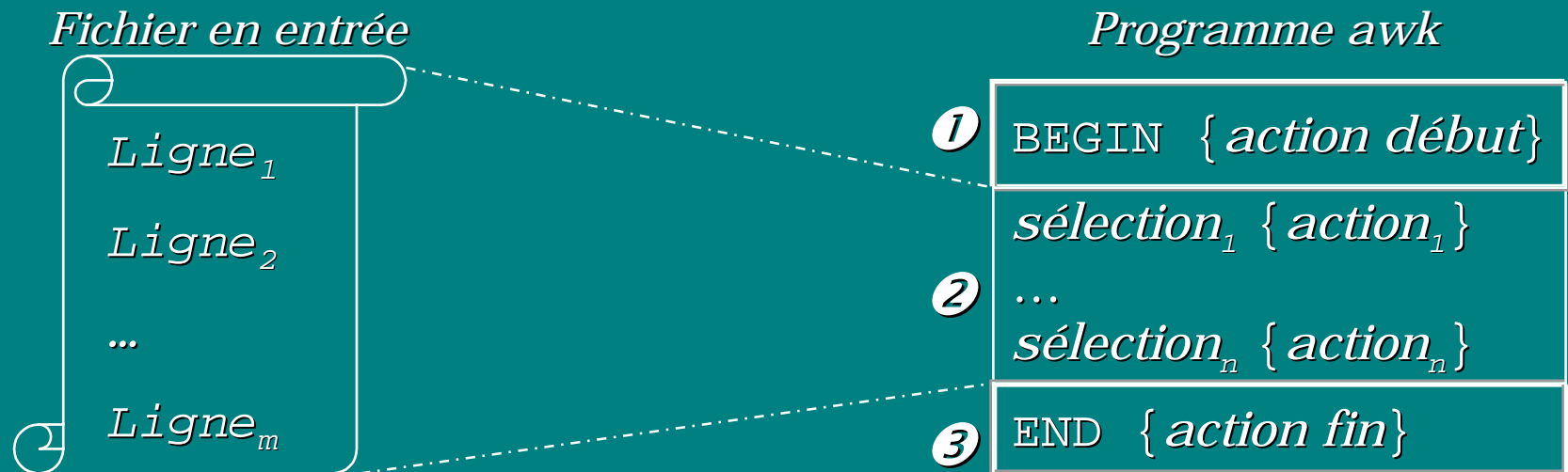
UNIX

La programmation

AWK

awk - principe de fonctionnement

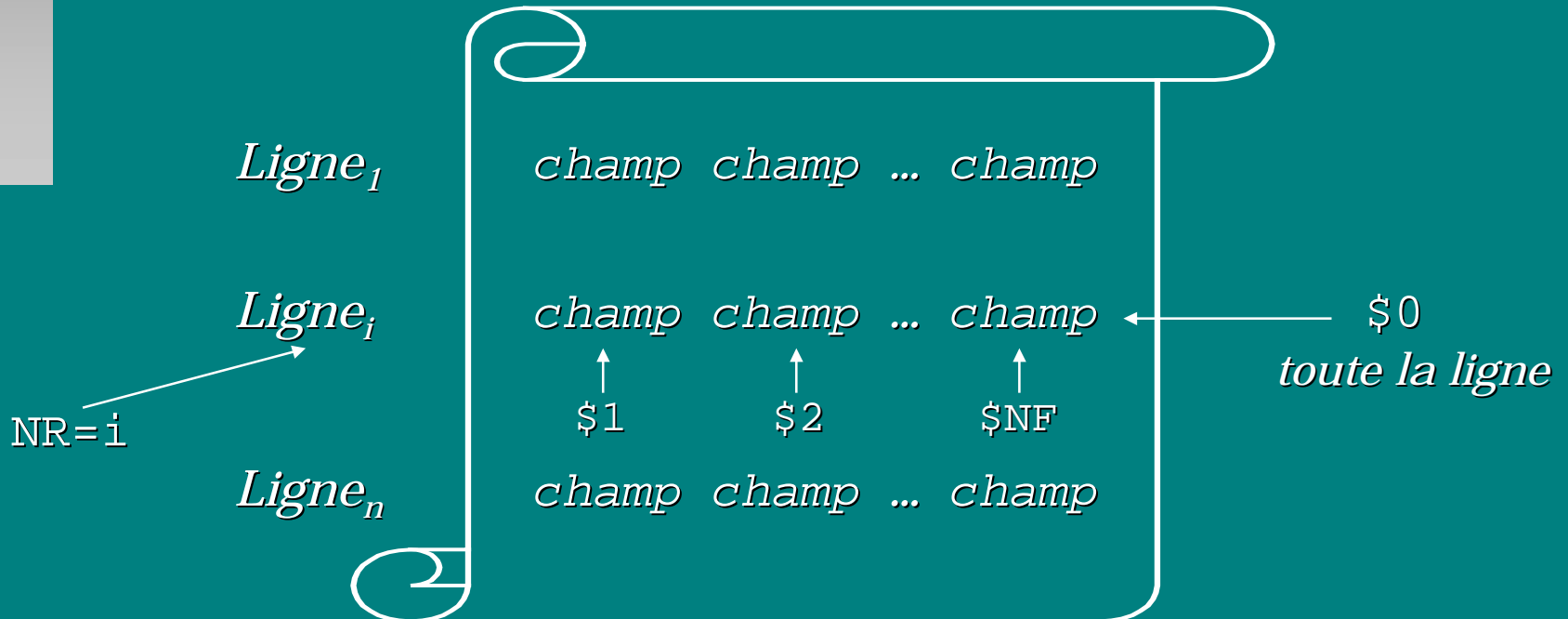
- **Forme générale d'un programme awk :**



- ① *Exécutée avant le début de lecture du fichier (sélection = BEGIN)*
- ② *Pour chaque ligne_i (i=1 à m) faire :*
pour j=1 à n faire : si sélection_j est vérifiée, exécuter action_j
- ③ *Exécutée à la fin de lecture du fichier (sélection = END)*
Remarque : si sélection_j absente, exécuter action_j pour toutes les lignes

awk - principe de fonctionnement

- awk sépare les lignes en champs selon le séparateur FS (Field Separator : par défaut *espace*)



NF : Nombre de champs d'une ligne

NR : Numéro de la ligne en court

Quelques exemples

- `$ awk 'length > 72' fich`
 - Affiche les lignes de `fich` de longueur > à 72
- `$ awk '/debut/,/fin/' fich`
 - Affiche les lignes de `fich` entre `debut` et `fin`
- `$ awk '{ print $2, $1 }' fich`
 - Affiche les champs 2 et 1 (dans cet ordre) de toutes les lignes de `fich`
- `$ awk -F: '$7 ~ /ksh/ {print $1}' /etc/passwd`
 - Affiche les noms de login (1^{er} champ) des utilisateurs utilisant `ksh` comme shell (7^{ème} champ) de connexion
- `$ ps -ef | awk '/nom/ {print $2}'`
 - Affiche les numéros des processus contenant la chaîne `nom`

Ligne de commande awk

- awk [-F *ifs*] [-f *prog_file*]... [-v *var=val*]... [*argument*]...
- awk [-F *ifs*] [-v *var=val*]... ['*prog_text*'] [*argument*]...
 - -F *ifs* : Définit *ifs* comme le séparateur de champs en entrée (*ifs* : input field separator). *ifs* peut être une expression régulière (par défaut *espace*)
 - Le programme peut être stocké dans un fichier (*prog_file*) ou écrit dans la ligne de commande entre apostrophes (*prog_text*)
 - -v *var=val* : affecte *val* à *var* avant le début du programme (avant même la partie BEGIN)
 - *argument* :
 - Fichier(s) entrée. Si pas de fichier ou le signe -, c'est l'entrée standard qui est considérée
 - *var=val* : comme -v *var=val* sauf que :
 - si avant le nom du fichier en entrée, *var* prend effet après BEGIN
 - si après le nom du fichier en entrée, *var* prend effet avant END

Script awk indépendant

- Commencer le programme awk par la chaîne :
`#!chemin absolu de awk -f (#!/usr/bin/awk -f)`
- Faire suivre par des commentaires pertinents
 - Un commentaire commence par # et se termine par la fin de ligne
 - Exemple :

```
#!/usr/bin/awk -f
# ksh_users : liste des utilisateurs du KornShell
# usage : ksh_users
# @(#) ksh_users: KSH users on local system (v 1.0)
  $7 ~ /ksh/ {print $1}
```
 - `$ what ksh_users`
`ksh_users: KSH users on local system (v 1.0)`
 - `$ ksh_users /etc/passwd`
`adam`
`durand ...`

Syntaxe des sélections

■ Expression régulière

- */ expression /*
 - Sélection vérifiée si la ligne vérifie l'expression régulière
 - Exemples :
 - */ exemple /* : ligne contenant le mot *exemple*
 - */ [a-z][a-z]* \$ /* : ligne se terminant par une suite de lettres minuscules
- *\$n ~ / expression /* (ou *\$n !~ / expression /*)
 - Sélection vérifiée si le champ *n* vérifie (ou ne vérifie pas) l'expression
 - Exemple :
 - *\$2 ~ /^[0-9]/* : le champ 2 commence par un chiffre
 - *\$1 ~ /^[0-9].[a-z]* \$ /* : le champ 1 commence par un chiffre et se termine par une lettre minuscule

Syntaxe des sélections

■ Relation

- Opérateurs de relation : $<$, $<=$, $==$, $!=$, $>=$, $>$
- Exemples :
 - $\$1 == \text{"chaîne"}$
vraie si le premier champ est "chaîne"
 - $NF != 5$
vraie si le nombre de champs de la ligne courante est $\neq 5$
 - $\$NF > 20$
vraie si le dernier champ est $>$ à 20 (ne pas confondre avec : $NF > 20$ nombre de champs $>$ à 20)
- Relation entre expressions arithmétiques :
 - $(\$1+\$2)/2 >=10$
vraie si la moyenne des deux premiers champs est ≥ 10

Combinaison de sélections

■ A l'aide des opérateurs logiques :

- && (et) : `/chaîne/ && $1 ~ /^[01]/`
 - La ligne contient chaîne et le 1er champ commence par 0 ou 1
- || (ou) : `NR == 10 || NF == 5`
 - La ligne 10 ou le nombre de champs = 5
- ! (négation) : `! /chaîne/ && NF < 5`
 - La ligne ne contient pas chaîne et le nombre de champs < 5
- () (fixer l'ordre d'évaluation) :
`(/chaîne1/ || /chaîne2/) && /chaîne3/`
 - La ligne contient chaîne₁ ou chaîne₂ en plus de chaîne₃

■ A l'aide des intervalles :

- , (virgule : définition de bloc)
`/début/,/fin/ || NR==20,NR==30`
 - Toutes les lignes entre la première qui contient début et la première qui contient fin ou les lignes entre la 20^{ème} et la 30^{ème}

Expressions régulières

- Les caractères suivants ont une signification particulière pour définir une expression régulière :

<i>Caractère</i>	<i>Interprétation</i>
.	caractère quelconque
[début de définition d'un ensemble
[^	début de définition du complément d'un ensemble
]	fin de définition d'un ensemble ou de son complément
-	marque d'intervalle dans un ensemble
^	en début d'expression, définit le début de la ligne
\$	en fin d'expression, définit la fin de la ligne
\(début de définition d'une sous-expression (accessible en suite par \1, \2, ...)
\)	fin de définition d'une sous-expression

Expressions régulières

■ Exemples

- `/[a2m]/` a, 2 ou m
- `/[a-z]/` une lettre minuscule
- `/[02-57]/` 0, 2, 3, 4, 5 ou 7
- `/[a-d5-8X-Z]/` a, b, c, d, 5, 6, 7, 8, X, Y ou Z
- `/[0-5-]/` 0, 1, 2, 3, 4, 5 ou -
- `/[^0-9]/` pas un chiffre
- `/[^a-zA-Z]/` pas une lettre
- `/[012^]/` 0, 1, 2 ou ^
- `/^abc/` ligne commençant par abc
- `/[a-z0-9]$/` ligne finissant par une lettre minuscule ou un chiffre
- `/^$/` ligne vide

Combinaisons d'expressions régulières

- Il est possible d'étendre l'expressivité d'une expression régulière en utilisant les caractères suivants

<i>Caractère</i>	<i>Interprétation</i>
*	0 ou plusieurs fois ce qui le précède (caractère ou ER)
+	1 ou plusieurs fois ce qui le précède
?	0 ou 1 fois ce qui le précède
	disjonction d'expressions régulières (ER1 ER)
(ER)	groupement de caractères (pour éviter les ambiguïtés)

- Exemples :
 - `/[a-z]*/` : 0 ou plusieurs lettres minuscules
 - `/aa*/` : au moins un a
 - `/.*/` : n'importe quelle chaîne de caractères (même vide)
 - `/^[0-9][0-9]*$/` : ligne qui ne contient que des chiffres
 - `/(fermer|ouvrir) (fenetre|porte)/` : fermer ou ouvrir suivi de fenetre ou porte
 - `(TA)+C` : reconnaît TAC, TATAC, TATATAC, etc.
 - `^[+-]?([0-9]+[.]?[0-9]*|.[0-9]+)([eE][+-]?[0-9]+)?$` : reconnaît un nombre réel seul dans une ligne

Expressions manipulées dans les actions

■ Constantes :

- Les chaînes :
 - Chaîne = constante entourées de "
 - Caractères spéciaux : \t (tabulation), \n (new line), \a (signal sonore), \b (retour arrière), \r (carriage return), \ccc (caractère représenté par son code octal (c chiffre octal)), ...
- Les nombres : 1, -2, 1.5, 0.15e-1, ...
- Les valeurs logiques : 0 \Leftrightarrow faux, \neq 0 \Leftrightarrow vrai

■ Variables :

- Variables utilisateur
 - Suite de lettres, de chiffres et du caractère '_' qui ne commence pas par un chiffre
 - Pas de déclaration préalable
 - awk détermine le type des variables par le contexte d'utilisation
- Variables prédéfinies ...

Variables prédéfinies

<i>Variable</i>	<i>Définition</i>	<i>Défaut</i>
ARGC	nombre d'arguments de la ligne de commande	-
ARGV	tableau d'arguments de la ligne de commande	-
FILENAME	nom du fichier d'entrée courant	-
NF	nombre de champs de l'enregistrement courant	-
NR	nombre d'enregistrements lus	-
FNR	nombre d'enregistrement du fichier courant	-
FS	séparateur de champs en entrée	" "
RS	séparateur d'enregistrements en entrée	"\n"
OFS	séparateur de champs en sortie	" "
ORS	séparateur d'enregistrements	"\n"
OFMT	format de sortie pour les nombres	"%.6g"
RLENGTH	longueur de la chaîne reconnue par la fonction "match"	-
RSTART	début de la chaîne reconnue par la fonction "match"	-
ENVIRON	tableau des variables d'environnement du shell	-

Opérateurs arithmétiques, logiques et d'affectation

Opération	Opérateurs	Exemple	Signification
Affectation	= += -= *= /= %= ^=	x *= 2 + \$1	x = x*(2+\$1)
Conditionnelle	? :	c ? e ₁ : e ₂	si c alors e ₁ sinon e ₂
Conjonction	&&	c ₁ && c ₂	=1 si c ₁ et c ₂ vrai, 0 sinon
Disjonction		c ₁ c ₂	=1 si c ₁ ou c ₂ , 0 sinon
Appartenance à un tableau	in	i in T	=1 si T[i] existe, 0 sinon
Correspondance (<i>matching</i>)	~ !~	\$1 ~ /ER/	=1 si champ 1 correspond à ER, 0 sinon
Comparaison	< <= == != >= >	x == y	=1 si égalité, 0 sinon
Concaténation		"mon" "awk"	"mon awk" : pas d'opérateur explicite

Opérateurs arithmétiques, logiques et d'affectation

Opération	Opérateurs	Exemple	Signification
somme, soustraction	+ -	$x + y - \$2$	
mult., div., reste	* / %	$a \% b$	Reste de la div de a par b
plus et moins unaire	+ -	-x	Opposé de x
Négation		\$1	Négation de \$1
Puissance	^	x^n	x^n
Incrément, décrétement	++ --	++x, x++	Incrémente x de 1
Accès au champ	\$	$\$NF-1$	(Valeur du dernier champ) - 1
Groupement	()	$(\$i)++$	Incrémente \$i de 1

Les actions awk - print

- La plupart des actions sont inspirées du langage C
- Les actions sont délimitées par ';' ou *fin de ligne* ou '}'

- `print [liste_expr]`

- *liste_expr* : suite d'expressions séparées par des virgules
- Affiche les expressions en les séparant par OFS (défaut: espace) et passe à la ligne
- Si *liste_expr* absent, afficher toute la ligne
- Exemples :

```
- print NR ":" $1, $NF
1:Premier Dernier (ligne 1: $1=Premier,$NF=Dernier)
2:Moi 2053          (ligne 2: $1=Moi,$NF=2053)
```

...

```
- awk -v OFS=":" '{print NR "-" $1, $NF}' fich
1-Premier:Dernier
2-Moi:2053
```

...

Les actions awk - printf

- `printf(format, liste_expr)` comme en langage C
 - Ecrit, sur la sortie standard, la liste d'expressions selon *format*
 - *format* : chaîne avec des ordres de format d'affichage :
 - %d (entier), %6d (entier sur 6 positions), %s (chaîne), %10s (chaîne sur 10 positions justifiée à droite), %-10s (idem justifiée à gauche), %f (réel), %.2f (réel avec 2 chiffres après la virgule), %6.3f (réel avec 6 chiffres avant et 3 chiffres après la virgule), %E ou %e (affichage avec exponentielle) %X ou %x (affichage hexadécimal), %% (affiche %), %o (affichage octal)
 - Exemples :
 - `printf(" |%s|--| %7s|++|%.3s|==|%-7.3s|\n", "ABCDE", "ABCDE", "ABCDE", "ABCDE")`
|ABCDE|--| ABCDE|++|ABC|==|ABC |
 - `printf("%3.2f %d %X %x\n", 45.457, 111, 111, 111)`
45.46 111 FF ff

Les actions awk - sprintf

- `sprintf(format, liste_expr)`
 - Même fonctionnement que `printf`
 - Après garnissage, *format* est renvoyée comme résultat (pas d'affichage)
 - Permet d'affecter des chaînes avec un contenu formaté
 - Exemple :
 - `X = sprintf("%3.2f %d %X %x", 45.457, 111, 111, 111)`
Met dans X la chaîne : " 45.46 111 FF ff"

Les actions awk - length

- `length(chaîne)`

- Renvoie la longueur de *chaîne* (si *chaîne* absente, \$0 pris par défaut)

- Exemple :

```
- $ echo "123456 89" |  
> awk '{ print length, length($1) }'  
9 6          longueur de la ligne : 9  
              longueur du 1er champ 123456 : 6
```

- Attention :

```
- length(15 * 35) vaut 3 :  
calcul de 15 * 35 = 525  
conversion en chaîne "525"  
length("525") = 3
```

Les actions awk - index

■ `index(chaîne, sous-chaîne)`

- Renvoie la position de *sous-chaîne* dans *chaîne*
- Si *sous-chaîne* \notin *chaîne*, renvoie 0
- Exemple :
 - `$ echo "programmation awk" |`
 - `> awk '{print index($1, "gram")}'`
 - 4

■ `match(chaîne, ER)`

- Cherche dans *chaîne*, la plus longue chaîne correspondant à l'expression régulière *ER* puis renvoie sa position
- Deux variables sont positionnées également :
 - `RSTART` : position de *ER* trouvée dans *chaîne* (0 sinon)
 - `RLENGTH` : longueur de la sous-chaîne trouvée (-1 sinon)

Les actions awk - split

- `split(chaîne, tab [, sép])`
- Décompose *chaîne* dans *tab* en plusieurs parties selon le séparateur *sép* (envoie le nombre de sous-chaînes obtenues = taille de *tab*)
- *sép* est une ER (si absent, FS par défaut)
- Exemple :
 - `L=split("le cul-de-sac", T, "-")`
L vaut 3,
T[1] vaut "le cul",
T[2] vaut "de",
T[3] vaut "sac"

Les actions awk - sub

- `sub(ER, remplacer [, cible])`
- Remplacer la première plus longue sous-chaîne correspondant à *ER* dans la chaîne *cible* par la chaîne *remplacer* (*cible* doit être une variable)
- Si *cible* absent, \$0 est pris par défaut
- Si *remplacer* contient le caractère &, ce caractère est remplacé par la sous-chaîne trouvée
- Exemples :
 - `Ch="moi, proie, toujours"`
`sub(/oi/, "e", Ch) → Ch devient "me, proie, toujours"`
 - `awk '{sub(/[Dd]urand/, "& et sa femme"); print}'`
Remplace, dans toutes les lignes, la première occurrence 'Durand' (ou 'durand') par 'Durand et sa femme' (ou 'durand et sa femme')
- `gsub` réalise la même opération que `sub`. La différence est que le remplacement est global sur toute la chaîne *cible*

Les actions awk - substr

- `substr(chaîne, début [, long])`
- Renvoie la sous-chaîne de *chaîne* de longueur *long* et qui commence à la position *début*
- Si *long* absent, tout le suffixe est renvoyé
- Exemples :
 - `substr("Bourges", 4, 2) → "rg"`
 - `substr("Ville Bourges", 4, 8) → "le Bourg"`
 - `substr("Ville Bourges", 7) → "Bourges"`

Les actions awk – tolower, toupper

■ `tolower(chaîne)`

- Renvoie la conversion de *chaîne* en minuscule
- Exemples :
 - `print tolower($1)`
Affiche \$1 après conversion en minuscule
 - `tolower($2) == "debut"`
Teste si \$2 vaut "debut" ou "Debut" ou "DEBUT" ou ...

■ `toupper(chaîne)`

- Renvoie la conversion de *chaîne* en majuscule

Les actions awk – fonctions mathématiques

- Il est possible d'utiliser les fonctions mathématiques usuelles. En voici un extrait :

<i>Fonction</i>	<i>Valeur</i>
<code>cos(x)</code>	cosinus de x (x en radians)
<code>sin(x)</code>	sinus de x (x en radians)
<code>atan2(y,x)</code>	arc-tangente de y/x renvoyée dans l'intervalle $-\pi$ et π
<code>sqrt(x)</code>	racine carrée de x
<code>log(x)</code>	logarithme népérien de x
<code>exp(x)</code>	exponentielle de x
<code>int(x)</code>	partie entière de x
<code>rand()</code>	nombre <i>r</i> aléatoire, $0 \leq r < 1$
<code>srand(x)</code>	x est le point de départ dans la génération aléatoire <code>rand()</code> . <code>srand()</code> fait débiter le générateur selon l'heure

Les actions awk – structures de contrôle

- La plupart des structures de contrôle ont la même syntaxe et la même sémantique que celles du langage C
- Résumé :
 - `if (expression) instructions [else instructions]`
Attention : `if (var = val) ...` Ne réalise pas le test `var` égale `val` : affecte à `var` la valeur `val`, la condition a pour valeur `val` (si `val = 0` \Rightarrow faux; sinon \Rightarrow vrai)
Il faut écrire : `if (var == val) ...`
 - `while (expression) instruction`
 - `for (exprInit; exprCond; exprInc) instructions`
 - `for (variable in tableau) instruction` : parcours d'un tableau...
 - `do instructions while (expression)`
 - `break` : sortie de la boucle la plus immédiate

Les actions awk – structures de contrôle

- `continue` : ne réalise pas la suite de la boucle \Rightarrow se branche directement sur la condition de boucle (dans la boucle `for`, ignore le reste du corps mais réalise *exprInc*)

- Imprimer les lignes privées de leurs I^{ème} champ :

```
- {for(x=1; x<=NF; x++)
  {
    if (x == I) continue
    printf("$x)
  }
  print " "
}
- {$I=" "; print $0}
```

Les actions awk – structures de contrôle

- `next` : stoppe le travail sur l'enregistrement courant et passe au suivant ensuite, reprend les actions à partir du début du script (abandon du reste du script pour l'enregistrement courant)
 - Afficher les lignes de tous les fichiers sauf pour les *.log où il faut afficher le 1^{er} et le 3^{ème} champ :

```
FILENAME ~ /\.log$/ { print $1, $3; next }  
{ print }
```
- `exit [expression]` : arrêt du traitement sur le fichier en cours et renvoi sur la section END. *expression* = code de retour du script (0 si absent)
- `{ instructions }` : groupement d'instructions

Les actions awk – for (var in tab)

- for (*variable* in *tableau*) *instruction*
 - *variable* parcourt les indices de *tableau*. Pour chaque passage, exécuter *instruction*
 - Accès aux éléments du tableau : *tableau*[*variable*]
 - Exemple : Le mot le plus utilisé en premier champ :

```
#Compter le nombre des différents lers mots
{NbrChamp1[$1] += 1}
#Trouver ensuite le + grand élément du tableau NbrChamp1
END {
    N=0; Mot=""
    for (m in NbrChamp1)
        if (NbrChamp1[m] > N) {
            N = NbrChamp1[m]
            Mot = m
        }
    print Mot " est le plus utilisé et figure " N " fois"
}
```

Tableaux

- Pas de déclaration, pas besoin de préciser la taille
- Tableau associatif : suite de paires (*indice*, *valeur*)
- Accès aux éléments d'un tableau : *tableau[indice]*
 - Si *indice* n'est pas dans le *tableau*, *tableau[indice]* \Leftrightarrow " "
 - Exemples :
 - `if (x in T) ...` teste si T[x] existe
 - `if (T[x] != " ") ...` teste si T[x] existe
- Affectation des valeurs dans un tableau
 - *Tableau[Indice] = Valeur*
 - *Indice* peut être n'importe quoi (chaîne ou nombre)
 - *Tableau* peut ne pas exister auparavant
 - Exemples : `T[1]="Un" ; T[12.5]="Nombre" ; T["moi"]=100`

Tableaux

- `delete tableau[indice]`
 - Ecrase l'élément indicé par *indice* dans *tableau*
 - `delete T[i]` rend inaccessible l'élément d'indice *i* :
 - `T[i]` vaut ""
 - Le test '`i in T`' renvoi faux
 - Exemples :
 - `for (i in T) delete T[i]`
écrase tous les éléments du tableau T
 - `delete T` : fait de même mais n'est pas standard POSIX
 - `split("", T)` : vide le tableau T de ses éléments
- Remarque :
 - `T[01]` et `T[1]` ne désignent pas la même chose
Tout étant ramené aux chaînes, `"01" ≠ "1"` (les éléments qu'ils indicent aussi)

Fonctions

- Objectifs

- Structurent les scripts
- Permettent la création de bibliothèques

- Définition :

- `function nom_fonction ([arg, ...]) { instructions }`
- `return [expression]` : arrêt de la fonction avec *expression* comme valeur de retour

Fonctions

■ Exemples :

- ```
function affiche(Nbr)
{ printf("%10.2f\n", Nbr) }
- $1 ~ /donnee/ { affiche($3) }
```
- ```
function inv_ch(Ch, Debut)
#Inverser Ch à partir de la position Debut
{if (Debut == 0)
    return ""
else
    return (substr(Ch,Debut,1) inv_ch(Ch,Debut-1))
}
- print inv_ch("Prog Awk", length("Prog Awk"))
kwA gorP
```

Redirections des sorties

- Les ordres de sortie peuvent être redirigés
 - `print ... > fichier`
 - Redirige le résultat vers *fichier* (en création)
 - `print ... >> fichier`
 - Redirige le résultat vers *fichier* (en ajout)
 - `print ... | commande`
 - Redirige le résultat vers l'entrée standard de *commande*
 - *fichier* et *commande* sont des chaînes
 - Si *fichier* = `"/dev/tty"` sortie terminal
 - `print ... | "cat 1>&2"` imprime dans la sortie standard des erreurs

Lecture explicite - getline

- `getline [var] [< fich]`
 - Lecture du prochain enregistrement
 - `getline` (sans argument)
 - lecture dans \$0 (\$0, NF positionnées)
 - `getline var`
 - lecture dans var (\$0, NF inchangées)
 - `getline < fich` ou `getline var < fich`
 - lecture à partir du fichier *fich*
 - Possibilité de lecture à partir d'un tube :
 - `cmd | getline` ou `cmd | getline var`

L'outil *sed* (stream editor)

- `sed [-n] [-e cmde]... [-f fich_cmdes ...]... [fichier...]`
 - Applique, pour chaque ligne des fichiers en entrée, la (les) commande(s) et affiche sur la sortie standard le résultat de ces applications
 - Options :
 - `-n` : écrit seulement les lignes spécifiées (par l'option `/p` des commandes) sur la sortie standard
 - `-e` : permet de spécifier les commandes à appliquer sur le fichier. Pour éviter que le shell interprète certains caractères, il est préférable d'encadrer les commandes avec des `'` ou des `"`.
 - `-f` : les commandes sont lues à partir du fichier *fich_cmdes*.

L'outil sed - syntaxe des lignes de commandes

- Une ligne de commande sed est de la forme :
adresse_ligne commande
 - *adresse_ligne* :
 - absente toutes les lignes
 - *num* ligne *num* (la dernière ligne : \$)
 - *num₁, num₂* lignes entre les lignes *num₁* et *num₂*
 - *ER* lignes correspondant à l'expression régulière *ER*
 - *ER₁, ER₂* lignes entre la première ligne correspondant à l'expression régulière *ER₁* et la première ligne correspondant à l'expression régulière *ER₂*

L'outil sed - les commandes

■ Substitution : s

- *s / chercher / remplacer / option*
- Remplace les expressions régulières *chercher* par la chaîne *remplacer*
- *option* :
 - *g* : remplacement global (par défaut seule la première occurrence est remplacée)
 - *p* : imprime la ligne (utile avec l'option *-n*)
 - *w fichier* : écrit la ligne dans *fichier* (en plus de la sortie standard)
- Exemples :
 - `sed "s/[Cc]omputer/COMPUTER/g" fichier`
 - `sed -e "s/\([0-9][0-9]*\) /==\1==" fichier` : encadre le premier nombre de la ligne avec des ==

L'outil sed - les commandes

- **Négation : !**
 - *! commande*
 - *commande* est appliquée à toutes les lignes qui ne correspondent pas à la caractérisation
- **Suppression : d**
 - Retire de la sortie les lignes qui correspondent à la caractérisation. Le fichier d'origine n'est pas affecté
 - Exemples :
 - `$ sed "1,5d" fich` : imprime fich à partir de la 6^{ème} ligne (équivalent à `:tail +6 fich`)
 - `$ sed "/^$/d" fich` : imprime fich sans ses lignes vides
 - `$ sed "/^Total/!d" fich` : retire toutes les lignes qui ne commencent pas par "Total" ⇒ imprime toutes les lignes qui commencent par "Total"

L'outil sed - les commandes

■ Insertion : *i*, *a*

- *i* \
texte insérer *texte* avant la ligne
 correspondant à la caractérisation
- Exemples :
 - *5i* \
 La ligne est insérée avant la ligne 5 actuelle
 - */^[0-9]+/i* \
 Un nombre au début de la ligne qui suit
- *a* \
texte insérer *texte* après la ligne
 correspondant à la caractérisation

■ Divers : *q*, *=*, *w*

- *q* quitter
- *=* écrire les numéros de lignes
- *w* *fichier* écrire dans fichier

Conseils de bon usage

- Donner l'extension `.awk` aux programmes awk
- Structurer le programme avec des fonctions
 - Mettre les plus utilisées dans des fichiers séparés et les inclure en ligne de commande par `-f` (en plus du programme awk) :

```
$ awk -f mon_prog.awk -f $HOME/lib/awk/mes_fcts.awk fich
```
- Paramétrer le programme en prévoyant options et arguments en l'incluant dans un script shell
- Commenter le programme
- Produire des messages d'erreurs standardisés :
fichier entrée : [numéro de ligne] *message court mais pertinent*

Conseils de bon usage

■ Code de retour

- Renvoyer toujours explicitement un code de retour documenté. Sa valeur est : 0 si tout s'est bien passé, ≠ 0 sinon
- Très utile si le script est appelé dans un autre

■ Où installer le script

- Dans un répertoire dédié (ex : `$HOME/bin`)
- Mettre ce répertoire dans `PATH` (au niveau du `.profile`) : `export PATH=$PATH:$HOME/bin`
- Remarque : éviter de mettre le répertoire courant `'.'` en début de `PATH` (problème de sécurité)

■ En cas de manipulation de fichiers temporaires :

- Penser à les détruire après usage (fonction *ménage*)
- Les nommer de façon unique (utiliser les valeurs `$0`, `$$`, `RANDOM`, etc. si création dans un script shell, utiliser `FILENAME`, `rand()`, etc. si création dans script AWK)
- Pour plus de clarté, les manipuler à travers des variables

L'éditeur vi

\$ vi fichier

insertion :

i niveau curseur, **a** après,
I début ligne, **A** fin ligne,
o ligne après, **O** ligne avant

ESC : revenir en mode commande

Déplacement :

caractères : **h** (gauche), **l** (droite),
j (bas), **k** (haut)

mots : **b** (gauche), **w** (droite)

lignes : **O** (début), **\$** (fin)

pages : **^d** (bas),
^u (haut)

Effacement :

x : caractère **dw** : fin mot

dd : ligne **dO** : début ligne

d\$: fin ligne

p : rétablir après
curseur

Commandes :

:f : infos sur fichier en cours

^g : *idem*

:w fich : sauvegarde dans
fich

:q : quitter

:wq : sauver et quitter

ZZ : *idem*

:q! : quitter même non sauvé

:e fich : charger *fich*

:r fich : inclure *fich* après
ligne courante

:! cmd_shell : exécuter
cmd_shell

:r! cmd_shell : exécuter et
insérer résultat de *cmd_shell*
/ motif : recherche de *motif*

^l : rafraîchit l'écran

Exemple : \$ vi -c ":r fich2" fich1 (charge le fichier fich1 en incluant à sa fin le contenu de fich2)