Cours Unix 5

Michel Mauny

INRIA ROCQUENCOURT

ETGL

Le cours 4 est disponible sur http://quincy.inria.fr/courses/unix/

Plan du cours 5

1. La commande `awk`
2. La commande `make`

La commande `awk`

La commande `awk`

Similaire à `sed`, mais plus puissant.

Avec `sed`:

- difficile de mémoriser la ligne précédente
- on ne peut aller en arrière dans le fichier
- on ne peut pas sélectionner une ligne par quelque chose comme `/.../+1`
- rien pour effectuer des calculs arithmétiques

La commande `awk`, dont les auteurs sont Aho, Weinberger et Kernighan, permet ce genre choses...

La commande awk

awk est un filtre programmable qui travaille aussi bien avec des chaînes qu'avec des nombres.

awk traite des *champs* alors que sed traite des lignes.

Tout comme sed et comme de nombreux outils Unix, awk traite des données émanant de

- fichiers
- redirections et pilelines
- l'entrée standard.

Tout comme sed, awk est un langage de *motifs-actions*, mais avec une syntaxe plus proche de C.

La commande awk

Un programme awk est constitué de:

1. un segment optionnel BEGIN: action exécutée avant la lecture de l'entrée
2. des paires *motif-action*: les actions sont exécutées sur chaque motif filtré
3. un segment optionnel END: action exécutée après la fin de la lecture

```
BEGIN { action }
motif { action }
...
motif { action }
END { action }
```


Invocation

Invocations:

- awk 'programme' fichier ...: le programme est donné sur la ligne de commande. Les fichier ... sont les fichiers à traiter.
- awk 'programme': idem, mais c'est l'entrée standard qui est traitée.
- awk -f pfichier fichier ...: le programme est trouvé dans pfichier.

Motifs et actions

Similaire à sed:

- Chercher des *motifs* dans un ensemble de fichiers.
- Exécuter les *actions* spécifiées sur les lignes ou champs contenant des occurrences de ces motifs.
- Traite une ligne (plus généralement un *enregistrement* – $cf.$ plus loin) à la fois.

Motifs et actions

`BEGIN { action }` optionnel, pour imprimer un message et/ou initialiser de variables, par exemple.

`motif { action }` l'un au moins de `motif` et `{ action }` doit être présent.

- si `motif` est absent, `{ action }` est éxécutée à chaque ligne;
- si `{ action }` est absent, la ligne est imprimée.

`END { action }` optionnel, pour imprimer un message.

Motifs

Les motifs agissent comme des sélecteurs:

`BEGIN` **ou** `END` : motifs spéciaux.

`/regexpr/` expression régulière.

`prédicat` par exemple `x > 0`

`motif && motif` , (ou `||`),
par exemple `/ETGL/ && name == "Cours UNIX"`

Actions

Une instruction ou séquence d'instructions à la C.

Exemple:

```
$ ls | awk '
  BEGIN    { print "Les fichiers PDF sont:" }
  /\.pdf$/ { print }
  END      { print "Et voilà!" }
  '
```

```
Les fichiers PDF sont:
cours1.pdf
...
Et voilà!
```


Définition et usage de variables

Un programme awk qui compte les lignes en entrée:

```
BEGIN { total = 0 }
      { total ++ }
END   { print total }
```

Certaines variables sont prédéfinies:

- `RS` (*record separator*). Par défaut, c'est `<NEWLINE>` (noté `\n`), mais peut être n'importe quelle expression régulière (le changer dans l'action associée à `BEGIN`);
- `NR` (*number of records*): nombre d'enregistrements lus jusqu'alors.

Champs

Chaque ligne en entrée est divisée en *champs*:

- `FS` (*field separator*), par défaut c'est `[ \t]+`
  - Peut valoir la chaîne vide (chaque caractère est alors un champ), un seul caractère, ou une expression régulière.
  - Peut être changé dans l'action du `BEGIN`
  - Peut être spécifié par `awk -F`r pour changer `FS` en r
- `$0` est la ligne entière, `$1`, `$2`, ..., les champs

Attention: seuls les noms de champs commencent par `$`, les autres variables sont utilisées comme dans un langage de programmation habituel (au contraire du shell).

Imprimer

Imprimer:

- chaque ligne:
  - `awk '{ print }' ...`
  - `awk '{ print $0 }' ...`
- certains champs:
  - `awk '{ print $1, $3 }' ...`
  - `awk '{ print $1, $NF }' ...` (le $ accepte une expression arithmétique)
  - `awk '{ print NF, $1, $(NF-2) }' ...` imprime le nombre de champs, le premier champ, et l'antépénultième

Imprimer

- le résultat de calculs
  - `{ print $1, $2 * $3 }`
  - `{ print "la dette de" $1, "est", $2 * $3 }`

Il existe aussi une commande `printf` qui permet de formatter les impressions (similaire au `printf de C`).

Sélection

Comparaison:

- `$2 >= 5 { print }`
- `$2 * $3 > 50 { printf("%6.2f pour %s\n", $2 * $3, $1) }`

Filtrage de texte:

- `$1 == "ETGL"`
- `/ETGL/`

Combinaison de motifs:

- `$2 >= 4 || $3 >= 20`
- `NR >= 10 && NR <= 20`

Variables

- Les variables de awk sont vues comme de type numérique ou texte selon le contexte.
- Elles n'ont pas besoin d'être déclarées, et valent soit 0 soit la chaîne vide, selon le contexte.

Manipulation de chaînes

- La concaténation de chaînes se note par juxtaposition:

```
    { names = names $1 " " }
END { print names }
```

- Imprimer la dernière ligne.

  NR garde sa valeur après la dernière lecture, mais pas $0:

```
    { last = $0 }
END { print last }
```


Quelques fonctions prédéfinies

- length(s)

```
     { nc = nc + length($0) + 1
       nw = nw + NF }
END { print NR, "lines,", nw, "words,", nc, "chars" }
```

- substr(s,i,len) produit la sous-chaîne de s commençant au $i^{\text{ème}}$ caractère et de longueur au plus égale à len.
- tolower(s)
- toupper(s)

Structures de contrôle

Conditionnelle:

```
$2 > 1 { n = n + 1; dette = dette + $2 * $3 }

END { if (n > 0)
        print n, "débiteurs multiples, de dette dotale:",
              dette, "-- moyenne: ", dette/n
    else
        print "aucun débiteur n'a emprunté plus d'une fois"
}
```

Structures de contrôle

Boucle *while*:

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year
     {  i = 1
       while (i <= $3) {
         printf("\t%.2f\n", $1 * (1 + $2) ^ i)
         i = i + 1
         }
      }
```


Structures de contrôle

Boucle for:

```
# interest2 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year

{ for (i = 1; i <= $3; i = i + 1)
  printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```


Structures de données: tableaux

- Les éléments de tableaux ne sont pas déclarés
- Les indices de tableaux peuvent être
  - des nombres
  - des chaînes (tableaux associatifs)
- Exemples:
  - `tab[3]="une chaine"`
  - `note["Jean"]=14.5`

Exemple d'utilisation de tableaux

```
# reverse
# - imprime l'entrée dans l'ordre inverse des lignes

{ line[NR] = $0 }           # mémorise chaque ligne

END {
      for (i=NR; (i > 0); i=i-1) {
        print line[i]
      }
    }
```

Quelques exemples courts

- `/chaine/ { nlines = nlines + 1 }`
  `END { print nlines }`
- `$1 > max { max = $1; maxline = $0 }`
  `END { print max, maxline }`
- `{ temp = $1; $1 = $2; $2 = temp; print }`
- ```
  { sum = 0
    for (i = 1; i <= NF; i = i + 1)
          sum = sum + $i
    print sum
  }
  ```


Quelques variables de awk

- `$0`, `$1`, `$2`, . . . , `$NF`
- `NR` – nombre d'enregistrements traités
- `NF` – nombre de champs dans l'enregistrement courant
- `FILENAME` – nom du fichier d'entrée courant
- `FS` – séparateur de champs
- `OFS` – séparateur de champs en sortie (espace, par défaut)

Opérateurs

- `=` affectation
- `==`, `!=` tests d'égalité
- `~`, `!~` test de filtrage (`chaîne ~ /motif/`)
- `&& ||`, conjonction, disjonction
- `!` négation
- `<`, `>`, `<=`, `>=`
- `+`, `-`, `/`, `*`, `%`, `^`
- juxtaposition: concaténation de chaîne

Functions prédéfinies

Arithmétiques

- `sin`, `cos`, `atan`, . . .

Chaînes

- `length`, `substitution`, recherche de sous-chaînes, . . .

Impression

- `print`, `printf`

Spéciales

- `system` exécute une commande Unix (`system("pwd")`)
- `exit` arrête la lecture, et exécute l'action associée à `END` si elle existe.
  ```

La commande make

Généralités

De nombreuses tâches (développement, production de documents, etc) nécessitent des opérations répétitives:

```
$ emacs progr.c &        # édition
$ cc -o ./prog prog.c    # test, édition et ...
$ cc -o ./prog prog.c    # test, édition et ...
$ ...
```

C'est encore pire si la modification d'un fichier a un impact sur d'autres:

```
$ emacs version.h &  # édition.
             # version.h contient le numéro de version
             # et prog.c mentionne version.h
$ cc -o ./prog ./prog.c
```


La commande make

Dans l'exemple précédent, on *sait* que prog dépend de version.h et de prog.c: toute modification de l'un ou de l'autre implique de regénérer prog.

La commande make permet de:

- représenter ces dépendances;
- indiquer les opérations à effectuer pour refabriquer les cibles en cas de modification (de la date) des prérequis.

La commande make*: dépendances*

Une règle de dépendance est notée comme suit:

```
cible: prérequis1 prérequis2 ...
------->action1
------->action2
------->...
```

Les lignes d'actions commencent obligatoirement par un caractère *tabulation*.

Exemple:

```
prog: version.h prog.c
------->cc -o prog prog.c
```

La commande `make`

La description des dépendances est généralement donnée dans un fichier nommé `makefile` ou `Makefile`.

```
$ cat Makefile
 prog: version.h prog.c
------->cc -o prog prog.c
$ make
 cc -o prog prog.c
```

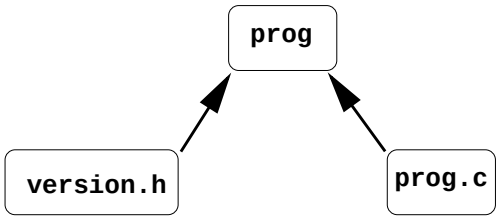
Ici, `prog` n'était pas à jour. La commande `make` l'a mis à jour.

```
$ make
 make: 'prog' is up to date.
```

Ce second appel à `make` remarque que `prog` est à jour, et n'effectue aucune opération.

Fonctionnement de `make`

Étant donné un makefile (fichier décrivant les dépendances), et une cible, `make` construit un graphe (acyclique) de dépendances pour cette cible:



Les feuilles de ce graphe ne dépendent de rien (sont toujours à jour), et les nœuds sont reconstruits (si nécessaire) en exécutant les actions spécifiées par le makefile.

Exemple

```
$ cat Makefile
 prog: f1.o f2.o f3.o
------->cc -o prog f1.o f2.o f3.o

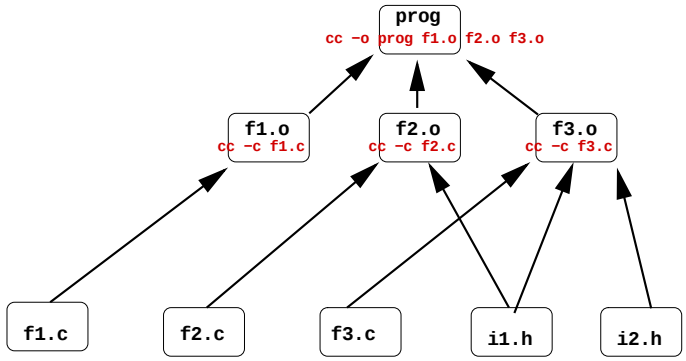
 f1.o: f1.c
------->cc -c f1.c

 f2.o: f2.c i1.h
------->cc -c f2.c

 f3.o: f3.c i1.h i2.h
------->cc -c f3.c
```


Exemple

Le graphe de dépendances associé est:



La commande `make`

La commande `make`:

- automatise ces tâches répétitives
- effectue le minimum de travail nécessaire à la mise à jour des cibles

Contenu d'un makefile

Un makefile contient:

- des commentaires (lignes commençant par #)
- des règles (cible + prérequis + actions), définissant des cibles à construire, leurs prérequis, et les actions à exécuter pour leur construction
- des définitions de variables (appelées aussi macros), destinées à être utilisées dans le makefile
- des règles de suffixes (*suffix rules*) ou règles génériques, indiquant comment produire un fichier avec un certain suffixe à partir d'un fichier de même nom avec un suffixe différent.

Make

`make` construit et traite le graphe de dépendances à partir de la première cible (ou *de la* ou *des* cibles spécifiées). Les cibles non rencontrées durant ce parcours sont ignorées.

`make` utilise pour cela:

- les règles explicites présentes dans le makefile
- les règles de suffixes présentes dans le makefile
- et des règles implicites prédéfinies

Le traitement des règles est soumis a l'expansion des variables.

Les commentaires sont ignorés.

Variables (macros)

Variables ou macros sont destinées à simplifier ou paramétrer un makefile.

Définition:

```
nom = valeur
```

Les macros sont utilisées dans les règles et dans les actions par:

```
$(nom)
```

ou bien par

```
${nom}
```

Dans le cas où le nom de la variable est monocaractère, on peut utiliser:

```
$x
```

Variables (suite)

Attention: les actions contiennent des commandes et des variables shell. Les caractères $ destinés au shell doivent être notés $$
Exemple:

```
# Un exemple de Makefile
BINDIR=/usr/local/bin

install: prog
------->test -d $(BINDIR) && cp prog $(BINDIR)/prog || \
            cp prog $$HOME/bin/prog
prog: prog.c version.h
------->...
clean:
------->/bin/rm -f *.o *.c~ prog
```


exemple

Les noms de cibles, des listes d'objets peuvent être des variables:

```
EXE = prog
OBJ = main.o f1.o f3.o

$(EXE): $(OBJ)
-------->cc -o $(EXE) $(OBJ) -lm
```

Les variables non définies sont remplacées par la chaîne vide.

Les valeurs de variables peuvent être forcées sur la ligne de commande (prenant le pas sur celles contenues dans le makefile):

```
$ make "EXE=testprog"
```


Variables prédéfinies

Les variables vues précédemment ont une valeur "globale" (uniforme), c'est-à-dire que toutes leurs occurrences dans le makefile ont la même valeur.

La valeur d'une variable est recherchée par `make`, par priorité décroissante, dans:

1. les arguments de `make`;
2. les variables définies dans le makefile;
3. les variables d'environnement (les variables du shell);
4. variables internes prédéfinies.

Il existe aussi des variables prédéfinies, telles:

```
CC=cc
```


Variables non uniformes

Il existe des variables dont la valeur n'est pas uniforme mais varie de règle en règle.

Elles permettent de désigner la cible courante ou les prérequis de la cible courante:

- $@ le nom de la cible courante;
- $? liste des prérequis plus récents que la cible.

```
prog: f1.c f2.c
-------->cc -o $@ f1.c fic2.c

prog: prog.c
-------->cc -o $@ $?
```

Règles de suffixes (dites aussi règles génériques)

On utilise souvent les mêmes actions pour passer d'un fichier avec un certain suffixe (`.c`, par ex.) à un fichier avec un autre (`.o`).

`make` permet d'exprimer ces règles génériques de la façon suivante:

```
.c.o:
------->echo Rebuilding $*.o from $*.c
------->$(CC) -c $<
```

Ici,

- `$<` est le prérequis qui a déclenché la règle (le `.c`);
- `$*` est le préfixe de la cible (c'est aussi celui du prérequis)

Règles génériques

`make` connaît un certain nombre de suffixes et de règles génériques prédéfinies.

Il est souvent plus clair d'effacer ces suffixes et règles prédéfinies pour en donner une définition explicite:

```
# On efface les suffixes connus
.SUFFIXES:
# Et on se définit les siens:
.SUFFIXES: .tex .dvi .pdf
# Règles
.tex.dvi:
------->latex $<
.dvi.pdf:
------->dvipdfm $<
```


Usage de `make`

Syntaxe:

```
make  [options] [cible ...]
```

Options:

- `-f file` utilise `file` comme makefile
- `-d` imprime des informations de *debug*
- `-k` met à jour le plus possible de cibles intermédiaires (au lieu de s'arrêter dès la première erreur)
- `-i` ignore les codes de retour des actions
- `-n` imprime les actions à exécuter (sans les exécuter)

Si aucune `cible` n'est spécifiée, la cible par défaut est celle de la première règle du makefile.

Usage de `make`

Si l'option `-f file` n'est pas présente, `make` teste d'abord la présence d'un fichier nommé `makefile`, puis celle de `Makefile`.

Exécution des actions:

Chaque action est exécutée dans un nouveau shell.

Il ne faut donc pas écrire:

```
------->cd quelque_part
------->commande à exécuter
```

mais:

```
------->cd quelque_part && commande à exécuter
```

`make`: *erreurs communes*

Erreurs courantes:

- une ligne d'action ne commence pas par `<TAB>`
- une action sur plusieurs lignes n'est pas continuée par un `\` en fin de ligne
- un `$` dans une action qui est destiné au shell n'a pas été doublé (`$$`)
- une variable `FOO` de `make` est écrite `$FOO` au lieu de `$(FOO)` ou `${FOO}`
- mauvaise utilisation des variables `$@`, `$?`, `$<`, ...