GESTION DES DEPENDANCES

Plan

1. Exemple
2. Principe
3. Variables
4. Evidences
5. Règles

Exemple

Considérons un programme `calcul` à compiler à partir de trois fichiers

```
calcul.c
   addg.c
   multg.c
```

La commande de compilation s'écrit:

```
gcc -o calcul calcul.c addg.c multg.c
```

Sous cette forme, toute modification dans l'*un* des fichiers oblige de recompiler les *trois* fichiers.

La compilation séparée permet d'écrire:

```
gcc -c calcul.c
   gcc -c multg.c
   gcc -c addg.c
   gcc -o calcul calcul.o multg.o addg.o
```

Seul le fichier modifié doit être recompilé, suivi d'une édition de lien. Ces dépendances sont exprimées dans le fichier `Makefile` de manière symbolique:

```
calcul : calcul.o multg.o addg.o
       gcc -o calcul calcul.o multg.o addg.o
   calcul.o : calcul.c
       gcc -c calcul.c
   multg.o : multg.c
       gcc -c multg.c
   addg.o : addg.c
       gcc -c addg.c
```

Principe

L'utilitaire `make` est un générateur de commande. Il aide à la maintenance d'un programme obtenu à partir de plusieurs modules. Il

- prend en compte les dates de modifications des fichiers
- utilise un graphe de dépendances, pour mettre à jour des fichiers périmés.

Le graphe de dépendances est contenu dans un fichier traditionnellement appelé `makefile` ou `Makefile`.

Chaque entrée a la forme

```
cible : source1 source2 ... sourcen
<tab> action
```

- la première ligne indique que le fichier `cible` *dépend* des fichiers `source1`,..., `sourcen`;
- la deuxième ligne décrit l'*action* à entreprendre si
  - le fichier `cible` n'existe pas ou
  - si l'un des fichiers source est *plus récent* que le fichier cible.

  L'action peut être une commande *shell* quelconque.

Si l'un des fichiers source est lui-même cible d'une dépendance, il est auparavant traité en fonction de l'action associée.

Si, dans l'exemple, le fichier `calcul.c` est modifié, l'appel de `make` produira les commandes

```
gcc -c calcul.c
gcc -o calcul calcul.o multg.o addg.o
```

Précisons:

- La commande `make` utilise un fichier dont le nom par défaut est `makefile` ou `Makefile`. Un autre fichier peut être spécifié avec l'option `-f`.

  ```
  make -f Fairefichier
  ```

- On peut choisir de réaliser une dépendance particulière en indiquant la cible (par défaut c'est la première).

  ```
  make multg.o
  ```

- Une cible peut être une simple étiquette (et non pas un fichier).

Voici un exemple:

```
all : calcul edit
   calcul : calcul.o
       gcc -o calcul calcul.o
   edit : edit.o
       gcc -o edit edit.o
   calcul.o : calcul.c
       gcc -c calcul.c
   edit.o : edit.c
       gcc -c edit.c
   install : all
       mv edit /usr/local
       mv calcul /usr/local
```

Le parcours d'un premier `make install` est :

```
install -> all
     all -> calcul
       calcul -> calcul.o
         calcul.o -> calcul.c
         <-
       <-
     all -> edit
       edit -> edit.o
         edit.o -> edit.c
         <-
       <-
     <-
   <-
```

Les parcours suivants mettent à jour les fichiers à modifier.

Variables

Une entrée d'un fichier de dépendance peut être une définition de variable, sous la forme

identificateur=chaîne

Une expression de la forme `$(`*identificateur*`)` est remplacé par la *chaîne*.

Ceci simplifie et standardise le fichier de dépendances.

```
OBJ= calcul.o multg.o addg.o
  CIB=calcul
  CC=gcc
  CFLAGS=-g
  $(CIB) : $(OBJ)
      $(CC)  -o calcul $(OBJ)
  calcul.o : calcul.c
      $(CC) $(CFLAGS) -c calcul.c
  multg.o : multg.c
      $(CC) $(CFLAGS) -c multg.c
  addg.o : addg.c
      $(CC) $(CFLAGS) -c addg.c
```

Un grand nombre de variables sont prédéfinies.

Evidences

Un certain nombre d'évidences sont offertes. Ce sont des dépendances qui sont implicites. Par exemple, pour tout fichier `x.c`, la dépendance

```
x.o : x.c
      $(CC) $(CFLAGS) -c x.c
```

est automatique. De même, pour tout fichier `x.o`, la dépendance

```
x : x.o
      $(CC) $(LDFLAGS) $(LDLIBS) -o x  x.o
```

est automatique.

Le fichier de dépendances se réduit alors singulièrement. Il devient:

```
OBJ= calcul.o multg.o addg.o
  CIB=calcul
  CC=gcc
  CFLAGS=-g
  $(CIB) : $(OBJ)
```

Règles

Les règles implicites sont de lecture assez opaque. Chaque règle implicite peut être écrasée par une règle ajoutée explicitement.

Quatre autres macros interviennent. Pour

```
cible.o : source.c
```

`$@` désigne `cible.o`, le nom complet du fichier cible

`$?` désigne `source.c`, nom du fichier source

`$*` désigne `cible`

`$<` désigne `source.c`, le fichier ayant provoqué l'action en cours (dans une règle implicite seulement).

Une règle implicite est régie par les extensions des fichiers, et appelée pour cela une *règle de suffixe*.

- Une règle de suffixe est une *description générale* de dépendance, basée sur les suffixes des fichiers.
- Les suffixes en jeu sont précisés par une ligne

  ```
  .SUFFIXES: .o .c
  ```

  La liste des suffixes par défaut est bien plus longue.
- Une règle de suffixe s'écrit sous la forme

  ```
  .c.o:
          gcc -c $(CFLAGS) $<
  ```

  signifiant que chaque fichier d'extension `.c` est transformé en fichier de même nom d'extension `.o` par l'action spécifiée.

La version GNU de `make` permet des *dépendances explicites* de la forme

```
%.o : %.c
        gcc -c -o $@ $(CFLAGS) $<
```

Plusieurs dizaines de règles existent par défaut.

Aide:

- Pour vérifier la correction syntaxique d'un fichier `Makefile`, utiliser

  ```
  cat -v -t -e Makefile
  ```

  Une tabulation est visualisée par `^I` et chaque ligne est terminée par un `$` .

- Pour tester un fichier sans l'exécuter, faire

  ```
  make -n
  ```

- Pour modifier la date de dernière modification d'un fichier, utiliser `touch`.