C'est l'histoire d'un make...

Alban Mancheron

C’est l’histoire d’un *make*…

Alban Mancheron
[Enseignant-Chercheur en bioinformatique à l’Université de Montpellier, linuxien depuis 1997 (convaincu depuis 1998)]

L’outil make permet de fabriquer des fichiers selon des règles de production décrites dans une syntaxe très simple. Dans cet article, nous allons explorer les grands principes de l’écriture de ces règles et pour ceux qui ne sont pas encore familiers de cet outil, vous faire regretter d’avoir attendu si longtemps pour le connaître…

/// Mots-clés ///

make, *Makefile*, compilation, programmation, recettes.

/// Fin Mots-clés ///

« C’est l’histoire d’un make…

Vous la connaissez ? Non ? Oui ? Non, parce que sinon… Parce que des fois y a des *makes*… Bon… Ah oui... Parce que y a des *makes*…

Vous la connaissez ? Non, dites-le parce que quand les gens y la connaissent après on a l'air d'un con.

Alors là, le *make*…

Ah oui ! Parce que y a des *makes* des fois… Non, c'est un exemple… Oui, y a des *make*…

Alors, euh… Ça dépend des *makes*, parce que y a des *makes*… Alors, bon, des fois, c'est l'histoire avec des programmes, tout ça… Et puis le *make* oui, euh… Mais là, non ! Ah oui ! Non là, c'est l'histoire d'un *make*, mais un *make* normal…

Ah oui, parce que dans les histoires, y'a deux genres de *makes*… Ah oui…

Mais là, non. Un *make* normal. Pas un *automake*.

Ah oui, parce que y a des histoires… Y a deux genres d'histoires, ah oui…

Y a des histoires, c'est plus rigolo quand c'est un *automake*… Si on est… Pas *CMake*… Ben oui, faut un minimum…

Et puis y a les histoires, c'est plus rigolo quand c'est un *CMake*… Oui… Si on est… pas *SnakeMake*… »

Mais pour ça, il faut lire les autres articles parus et à venir se rapportant à tous ces outils.

Alors le *make*… est un utilitaire qui permet, à partir d’un fichier de description, de fabriquer d’autre fichiers. Par défaut, le fichier de description peut s’appeler **`GNUmakefile`**, **`makefile`** ou **`Makefile`** mais peut porter n’importe quel autre nom, dans ce cas il faut le spécifier explicitement lors de l’invocation du programme **`make`**.

Avant de commencer, il faudra donc installer cet outil, soit *via* votre gestionnaire de paquets préféré,

```
sudo apt install make
```

soit à partir des sources [**https://www.gnu.org/software/make/**]

1 Un make créant

Pour mieux comprendre le fonctionnement de cet outil, créons un nouveau répertoire **`test-make`** dans lequel nous travaillerons, puis lançons la commande **`make`** sans option, puis avec l’option **`--debug=all`** (qui comme son nom le laisse deviner affiche tout plein d’informations sur ce que fait la commande) :

```
$ mkdir test-make
$ cd test-make/
$ ls
$ make
make: *** Pas de cible spécifiée et aucun makefile n'a été trouvé. Arrêt.
$ make -debug=all
GNU Make 4.1
```

```
Construit pour x86_64-pc-linux-gnu
Copyright (C) 1988-2014 Free Software Foundation, Inc.
Licence GPLv3+ : GNU GPL version 3 ou ultérieure <http://gnu.org/licenses/gpl.html>
Ceci est un logiciel libre : vous êtes autorisé à le modifier et à la redistribuer.
Il ne comporte AUCUNE GARANTIE, dans la mesure de ce que permet la loi.
Lecture des makefiles...
Mise à jour des makefiles....
Étude du fichier cible « GNUmakefile ».
 Le fichier « GNUmakefile » n'existe pas.
 Recherche d'une règle implicite pour « GNUmakefile ».

[ affichage supprimé ]

 Pas de règle implicite trouvée pour « GNUmakefile ».
 Fin des dépendances du fichier cible « GNUmakefile ».
Il faut refabriquer la cible « GNUmakefile ».
Échec de refabrication du fichier cible « GNUmakefile ».
Étude du fichier cible « makefile ».
 Le fichier « makefile » n'existe pas.
 Recherche d'une règle implicite pour « makefile ».

[ affichage supprimé ]

 Pas de règle implicite trouvée pour « makefile ».
 Fin des dépendances du fichier cible « makefile ».
Il faut refabriquer la cible « makefile ».
Échec de refabrication du fichier cible « makefile ».
Étude du fichier cible « Makefile ».
 Le fichier « Makefile » n'existe pas.
 Recherche d'une règle implicite pour « Makefile ».

[ affichage supprimé ]

 Pas de règle implicite trouvée pour « Makefile ».
 Fin des dépendances du fichier cible « Makefile ».
Il faut refabriquer la cible « Makefile ».
Échec de refabrication du fichier cible « Makefile ».
make: *** Pas de cible spécifiée et aucun makefile n'a été trouvé. Arrêt.
$
```

À la lecture des lignes affichées, il est aisé de comprendre que tout l'intérêt de cet outil réside donc dans le fameux fichier décrivant les règles de productions.

Par habitude, j'utilise le nom **`Makefile`** pour créer mes fichiers, mais rien n'interdit d'utiliser un nom quelconque (*e.g.*, **`exemple-makefile-glmf.txt`**). Dans ce cas, il suffit de spécifier le fichier avec l'option **`--file`** (ou **`--makefile`** ou encore **`-f`**).

```
$ make --file exemple-makefile-glmf.txt
make: exemple-makefile-glmf.txt : Aucun fichier ou dossier de ce type
make: *** Aucune règle pour fabriquer la cible « exemple-makefile-glmf.txt ». Arrêt.
$
```

1.1 Mon premier livre de recettes

Il faut comprendre que le *Makefile* n'est ni plus ni moins qu'un livre de recettes, décrivant comment produire des fichiers à partir d'autres fichiers (existants ou à fabriquer également).

/// Début note PAO ///

Idéalement, pour tous les exemples de code, il faudrait afficher les caractères non-visibles (espaces, tabulations et retours chariots).

/// Fin note PAO ///

La syntaxe d'une recette suit le motif suivant :

```
cible: dépendances
	action₁
	…
	actionₙ
```

/// Début note : Commentaires ///

Comme tout langage de programmation, la grammaire du *Makefile* autorise l'écriture de commentaires, ce qui est plus que recommandé – même si vous ne trouverez aucun commentaire dans les exemples de cet article (mais il y a le texte autour des exemples…).

La syntaxe des commentaires est la même que pour le *shell*. Tout ce qui suit un dièse qui n'est pas dans une

chaîne de caractère ou une action est un commentaire du *Makefile*.

Bien évidemment, les actions étant des commandes de *shell*, celles-ci peuvent également bénéficier de commentaires ;-) .

/// Fin note ///

1.2 Ma première recette

Dans une recette, la **cible** est le fichier à produire, les **dépendances** sont les fichiers nécessaires pour fabriquer la cible et la **succession d'actions** correspondent aux commandes du *shell* à exécuter pour produire la cible. Chaque action doit commencer par une tabulation (et pas une suite d'espaces !!!) et être écrite sur une seule ligne. Toutefois, il est possible de continuer l'écriture d'une action sur la ligne d'après en utilisant le caractère \ qui – à l'instar du *shell* – signifie que la commande se poursuit sur la ligne suivante.

Construisons donc un exemple simple en créant un premier fichier de recette *Makefile* :

```
bonjour-GLMF.txt:
	echo "Bonjour GLMF" > bonjour-GLMF.txt
```

Lors de la première invocation de **make**, le fichier **boujour-GLMF.txt** n'existant pas, il est construit avec la recette indiquée. À l'invocation suivante, il n'est pas reconstruit puisqu'il existe déjà.

```
$ ls
Makefile
$ make
echo "Bonjour GLMF" > bonjour-GLMF.txt
$ ls
bonjour-GLMF.txt  Makefile
$ cat bonjour-GLMF.txt
Bonjour GLMF
$ make
make: « bonjour-GLMF.txt » est à jour.
$
```

1.3 Un peu de ménage

Étoffons notre exemple en ajoutant une deuxième recette :

```
$ ls
bonjour-GLMF.txt  Makefile
$ cat Makefile
bonjour-GLMF.txt:
	echo "Bonjour GLMF" > bonjour-GLMF.txt

clean:
	rm -f bonjour-GLMF.txt
	rm -f *~
$ make
make: « bonjour-GLMF.txt » est à jour.
$ make clean
rm -f bonjour-GLMF.txt
rm -f *~
$ ls
Makefile
$
```

On s'aperçoit que lors de l'invocation de **make**, il tente de reconstruire le fichier **bonjour-GLMF.txt**, mais comme celui-ci est à jour, il ne se passe rien de plus. Cependant, en spécifiant la cible **clean**, c'est cette recette qui est utilisée. L'outil *make* essaie donc de créer le fichier **clean** qui n'existe pas, et exécute les actions spécifiées. Cependant, aucune des actions ne crée le fichier **clean**. C'est une astuce plutôt sympathique pour faire du ménage sans tout effacer accidentellement…

Le lecteur attentif aura cependant noté qu'il ne faudrait pas qu'un fichier appelé **clean** existe dans ce répertoire, sans quoi les actions de nettoyage ne seraient jamais exécutées, pour les lecteurs moins attentifs (ce n'est pas grave, mais qu'on ne vous y reprenne pas non plus !), vérifions le tout simplement :

```
$ ls
Makefile
$ make
echo "Bonjour GLMF" > bonjour-GLMF.txt
$ touch clean
$ ls
bonjour-GLMF.txt  clean  Makefile
$ make clean
```

```
make: « clean » est à jour.
$ ls
bonjour-GLMF.txt  clean  Makefile
$
```

Fort heureusement, la syntaxe du *Makefile* permet de gérer ce genre de situations facilement en listant les **cibles factices** explicitement dans une cible particulière appelée **.PHONY**.

```
$ ls
bonjour-GLMF.txt  clean  Makefile
$ cat Makefile
.PHONY: clean

bonjour-GLMF.txt:
	echo "Bonjour GLMF" > bonjour-GLMF.txt

clean:
	rm -f bonjour-GLMF.txt
	rm -f *~
$ make clean
rm -f bonjour-GLMF.txt
rm -f *~
$ ls
clean  Makefile
$
```

Maintenant que le fichier **clean** n'est plus gênant, nous pouvons le supprimer.

1.4 Une recette avec des ingrédients, c'est mieux

Créons une troisième recette qui utilise le fichier **bonjour-GLMF.txt** en substituant `"Bonjour"` par `"Bonsoir"` (et mettons nos règles de nettoyage à jour au passage).

```
.PHONY: clean

bonjour-GLMF.txt:
	echo "Bonjour GLMF" > bonjour-GLMF.txt

bonsoir-GLMF.txt:
	sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt

clean:
	rm -f bonjour-GLMF.txt bonsoir-GLMF.txt
	rm -f *~
```

Puis testons ce nouveau cahier de recettes.

```
$ ls
Makefile
$ make
echo "Bonjour GLMF" > bonjour-GLMF.txt
$ make bonsoir-GLMF.txt
sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt
$ ls
bonjour-GLMF.txt  bonsoir-GLMF.txt  Makefile
$ cat bonsoir-GLMF.txt
Bonsoir GLMF
$ make bonsoir-GLMF.txt
make: « bonsoir-GLMF.txt » est à jour.
$ make clean
rm -f bonjour-GLMF.txt bonsoir-GLMF.txt
rm -f *~
$ ls
Makefile
```

Cela semble bien fonctionner. Lors de l'invocation de *make* sans cible, la première recette rencontrée à été utilisée et à créer le fichier **bonjour-GLMF.txt** (**.PHONY** n'est pas une véritable recette). Ensuite, lorsque la cible **bonsoir-GLMF.txt** a été spécifiée, la recette demandée a été appliquée et le fichier a été correctement créé. Lorsque la cible **bonsoir-GLMF.txt** a de nouveau été spécifiée, l'outil *make* a vérifié l'existence du fichier **bonsoir-GLMF.txt** et celui-ci étant à jour, il n'a pas été régénéré. Mais que se passerait-il si le fichier **bonjour-GLMF.txt** n'existait pas ?

```
$ ls
Makefile
$ make bonsoir-GLMF.txt
sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt
sed: impossible de lire bonjour-GLMF.txt: Aucun fichier ou dossier de ce type
Makefile:7 : la recette pour la cible « bonsoir-GLMF.txt » a échouée
```

```
make: *** [bonsoir-GLMF.txt] Erreur 2
$ ls
bonsoir-GLMF.txt  Makefile
$ cat bonsoir-GLMF.txt
$
```

Mais qu' s'est-il passé (dites-le à voix haute si vous n'avez pas décelé le jeu de mot) ?

Après nettoyage, lorsque la cible **bonsoir-GLMF.txt** a été spécifiée, l'outil *make* a essayé d'appliquer la recette qui a échoué car le fichier **bonjour-GLMF.txt** n'existe plus.

En réalité, il faut expliciter la dépendance existante de **bonjour-GLMF.txt** vers **bonsoir-GLMF.txt** :

```
bonsoir-GLMF.txt: bonjour-GLMF.txt
	sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt
```

Et ainsi cela fonctionne correctement :

```
$ make clean
rm -f bonjour-GLMF.txt bonsoir-GLMF.txt
rm -f *~
$ make --debug=i bonsoir-GLMF.txt
…
Lecture des makefiles...
Mise à jour des objectifs cibles....
 Le fichier « bonsoir-GLMF.txt » n'existe pas.
   Le fichier « bonjour-GLMF.txt » n'existe pas.
  Il faut refabriquer la cible « bonjour-GLMF.txt ».
echo "Bonjour GLMF" > bonjour-GLMF.txt
  Refabrication réussie du fichier cible « bonjour-GLMF.txt ».
Il faut refabriquer la cible « bonsoir-GLMF.txt ».
sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt
Refabrication réussie du fichier cible « bonsoir-GLMF.txt ».
$ ls
bonjour-GLMF.txt  bonsoir-GLMF.txt  Makefile
$
```

Le mode *debug* avec l'option **"i"** est assez parlant…

1.5 Les recettes à refaire

Les dépendances ont une autre utilité, savoir quand un fichier construit par une recette est obsolète et doit être refabriqué. En effet, si la date de dernière modification de la cible est antérieure à la date de dernière modification d'une des dépendances, alors l'outil *make* va relancer la recette de fabrication de la cible.

```
$ ls
bonjour-GLMF.txt  bonsoir-GLMF.txt  Makefile
$ make bonsoir-GLMF.txt
make: « bonsoir-GLMF.txt » est à jour.
$ touch bonjour-GLMF.txt
$ make --debug=i bonsoir-GLMF.txt
[ affichage supprimé ]
 La dépendance « bonjour-GLMF.txt » est plus récente que la cible « bonsoir-GLMF.txt ».
Il faut refabriquer la cible « bonsoir-GLMF.txt ».
sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt
Refabrication réussie du fichier cible « bonsoir-GLMF.txt ».
$ make bonsoir-GLMF.txt
make: « bonsoir-GLMF.txt » est à jour.
$
```

1.6 Un make unique

Rapidement et avant de passer à la suite, ajoutons une cible factice qui permet, par défaut, de créer tous les fichiers :

```
.PHONY: clean all

all: bonjour-GLMF.txt bonsoir-GLMF.txt

bonjour-GLMF.txt:
...
```

Comme **all** est la première cible du fichier, c'est donc la recette qui sera appliquée par défaut.

2 Un *make* passe partout

La construction d'un *Makefile* peut sembler lourde car pour chaque fichier à fabriquer, une recette doit être définie. Cependant, il arrive souvent que les recettes soient les mêmes, aux noms des fichiers près. Fort heureusement, il est possible de définir des variables au sein du fichier de recettes, et même des recettes type.

2.1 Souvent *make* varie, bien fol qui s'y fie

La déclaration d'une variable suit la syntaxe **`clé = valeur`** et pour utiliser une variable ainsi définie, il suffit d'utiliser l'opérateur **`$(nom_de_la_variable)`**.

Pour illustrer ce concept, reprenons l'exemple précédent et déclarons la variable *FICHIERS* dans laquelle nous affectons les noms de fichiers **`bonjour-GLMF.txt`** et **`bonsoir-GLMF.txt`**, puis remplaçons ces deux noms dans la dépendance de la cible **`all`** et dans la première action de la cible **`clean`** par l'appel au contenu de la variable :

```
FICHIERS = bonjour-GLMF.txt bonsoir-GLMF.txt

.PHONY: clean all

all: $(FICHIERS)

bonjour-GLMF.txt:
	echo "Bonjour GLMF" > bonjour-GLMF.txt

bonsoir-GLMF.txt: bonjour-GLMF.txt
	sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt

clean:
	rm -f $(FICHIERS)
	rm -f *~
```

Le caractère **`$`** étant réservé à la lecture du contenu d'une variable, pour l'utiliser (par exemple dans une action), il faut donc l'échapper. Cela se fait en doublant le symbole **`$`**.

Ainsi la règle **`clean`** pourrait devenir :

```
clean:
	echo "Nettoyage des fichiers contenus dans la variable \$$(FICHIERS)(=$(FICHIERS))"
	for f in $(FICHIERS); do \
	  echo "Le fichier $${f} sera supprimé"; \
	done
	rm -f $(FICHIERS)
	rm -f *~
```

Ce qui produit la sortie suivante lorsque la recette est appliquée :

```
$ make clean
echo "Nettoyage des fichiers contenus dans la variable \$(FICHIERS)(=bonjour-GLMF.txt bonsoir-
GLMF.txt)"
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(=bonjour-GLMF.txt bonsoir-GLMF.txt)
for f in bonjour-GLMF.txt bonsoir-GLMF.txt; do \
  echo "Le fichier ${f} sera supprimé"; \
done
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
rm -f bonjour-GLMF.txt bonsoir-GLMF.txt
rm -f *~
$
```

2.2 Un *make* masqué

Vous en conviendrez, cela devient parfois désagréable de voir s'afficher la commande avant qu'elle ne soit exécutée. Pour masquer l'affichage d'une commande particulière, il suffit de commencer l'action par le symbole **`@`**. La commande sera exécutée, mais ne sera pas affichée.

Voici la nouvelle cible **`clean`** :

```
clean:
	@echo "Nettoyage des fichiers contenus dans la variable \$$(FICHIERS)(=$(FICHIERS))"
	@for f in $(FICHIERS); do \
	  echo "Le fichier $${f} sera supprimé"; \
```

```
	done
	@rm -f $(FICHIERS)
	rm -f *~
```

Recette qui se doit d'être testée :

```
$ make clean
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(=bonjour-GLMF.txt bonsoir-GLMF.txt)
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
rm -f *~
$
```

Et si l'on veut pousser le vice un peu plus loin, une astuce consiste à déclarer une variable qui contiendrait ce symbole et de préfixer chaque action du contenu de cette variable. Ainsi, pour afficher/masquer les actions exécutées, il suffit de changer la valeur de cette variable.

```
MSQ = @

FICHIERS = bonjour-GLMF.txt bonsoir-GLMF.txt

.PHONY: clean all

all: $(FICHIERS)

bonjour-GLMF.txt:
	$(MSQ)echo "Création du fichier 'bonjour-GLMF.txt'"
	$(MSQ)echo "Bonjour GLMF" > bonjour-GLMF.txt

bonsoir-GLMF.txt: bonjour-GLMF.txt
	$(MSQ)echo "Création du fichier 'bonsoir-GLMF.txt' à partir de 'bonjour-GLMF.txt'"
	$(MSQ)sed 's,Bonjour,Bonsoir,' bonjour-GLMF.txt > bonsoir-GLMF.txt

clean:
	$(MSQ)echo "Nettoyage des fichiers contenus dans la variable \$$(FICHIERS)(=$(FICHIERS))"
	$(MSQ)for f in $(FICHIERS); do \
	  echo "Le fichier $${f} sera supprimé"; \
	done
	$(MSQ)rm -f $(FICHIERS)
	rm -f *~
```

Ce *Makefile* commence à avoir la classe !

```
$ make clean
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(=bonjour-GLMF.txt bonsoir-GLMF.txt)
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
rm -f *~
$ make
Création du fichier 'bonjour-GLMF.txt'
Création du fichier 'bonsoir-GLMF.txt' à partir de 'bonjour-GLMF.txt'
$ make
make: rien à faire pour « all ».
$
```

2.3 Des variables systématiques

Outre les variables que chacun peut déclarer, l'outil *make* vient avec un certain nombre de variables prédéfinies. Notamment, pour chaque recette, il existe 3 variables bien pratiques: *$@*, *$^* et *$<* qui contiennent respectivement la cible, les dépendances et la première des dépendances.

Amusons-nous à remplacer les noms de fichiers cible et de dépendances dans notre *Makefile*.

```
MSQ = @

FICHIERS = bonjour-GLMF.txt bonsoir-GLMF.txt

.PHONY: clean all

all: $(FICHIERS)
	$(MSQ)echo "La cible courante est '$@'"
	$(MSQ)echo "La cible '$@' dépend de '$^'"
	$(MSQ)echo "La première dépendance de '$^' est donc '$<'"

bonjour-GLMF.txt:
	$(MSQ)echo "Création du fichier '$@'"
	$(MSQ)echo "Bonjour GLMF" > $@

bonsoir-GLMF.txt: bonjour-GLMF.txt
```

```
	$(MSQ)echo "Création du fichier '$@' à partir de '$^'"
	$(MSQ)sed 's,Bonjour,Bonsoir,' $^ > $@

clean:
	$(MSQ)echo "Nettoyage des fichiers contenus dans la variable \$$(FICHIERS)(=$(FICHIERS))"
	$(MSQ)for f in $(FICHIERS); do \
	  echo "Le fichier $${f} sera supprimé"; \
	done
	$(MSQ)rm -f $(FICHIERS)
	rm -f *~
```

Cela ne change rien au résultat.

```
$ make clean
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(=bonjour-GLMF.txt bonsoir-GLMF.txt)
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
rm -f *~
$ ls
Makefile
$ make
Création du fichier 'bonjour-GLMF.txt'
Création du fichier 'bonsoir-GLMF.txt' à partir de 'bonjour-GLMF.txt'
La cible courante est 'all'
La cible 'all' dépend de 'bonjour-GLMF.txt bonsoir-GLMF.txt'
La première dépendance de 'bonjour-GLMF.txt bonsoir-GLMF.txt' est donc 'bonjour-GLMF.txt'
$
```

2.4 Des recettes systématiques

Il est possible d'aller encore plus loin avec *make* qui propose des fonctionnalités de manipulation de variables assez variées.

Par exemple, il est possible de définir la variable *FICHIERS* ainsi :

```
MODELES = bonjour-GLMF.txt
FICHIERS = $(MODELES)
FICHIERS += $(subst bonjour,bonsoir,$(MODELES))
```

Dans un premier temps, la variable *MODELES* est définie avec comme valeur **`bonjour-GLMF.txt`**. Ensuite, la variable *FICHIERS* est définie comme étant égale au contenu de la variable *MODELES*, puis on lui ajoute (opérateur **`+=`**) le résultat de la substitution de `bonjour` par `bonsoir` au contenu de la variable *MODELES*.

Poussons encore l'exemple plus loin et remplaçons la définition des variables par :

```
NOMS = GLMF
MODELES = bonjour-@NOM@.txt bonsoir-@NOM@.txt
FICHIERS = $(foreach nom, $(NOMS), $(MODELES:@NOM@.txt=$(nom).txt))
```

Cette fois-ci, les modèles représentent un motif qui est utilisé pour créer la liste des fichiers. Pour cela, une boucle est opérée sur le contenu de la variable *NOMS* avec *nom* comme variable de parcours et une substitution du suffixe `@NOM@.txt` par le contenu de la variable nom suivi de l'extension `.txt` est opérée sur la variable *MODELES*.

Vous me voyez venir avec mes gros sabots, l'objectif sera clairement d'ajouter des noms. Il faut donc modifier les règles de production des fichiers…

Nous allons pour cela utiliser deux astuces :

- la première est qu'il est possible de déclarer des variables spécifiques à une cible en utilisant la syntaxe

/// Début note PAO ///

Je ne sais pas comment mettre en emphase la ligne suivante « cible : **`clé = valeur`** » qui n'est ni du code ni une partie de fichier, mais un descriptif de la syntaxe.

/// Fin note PAO ///

cible : **`clé = valeur`**

- la seconde est l'utilisation du caractère **`%`** qui permet de faire correspondre un motif d'une cible avec ses dépendances.

Ainsi nous allons ajouter la règle qui consiste à déclarer pour tout fichier correspondant au motif **`bonjour-%.txt`** (ou **`%`** représente un motif quelconque), la variable *nom* avec pour valeur le résultat de la substitution du motif **`bonjour-%.txt`** par le motif contenu dans **`%`** à partir de la variable *$@*. En clair, pour la cible **`bonjour-GLMF.txt`**, cela signifie que *nom* = **`GLMF`**

```
bonjour-%.txt: nom = $(patsubst bonjour-%.txt,%,$@)
```

Il reste à adapter les recettes permettant de créer les fichiers répondant aux motifs **bonjour-%.txt** et **bonsoir-%.txt**

```
bonjour-%.txt:
	$(MSQ)echo "Création du fichier '$@'"
	$(MSQ)echo "Bonjour $(nom)" > $@

bonsoir-%.txt: bonjour-%.txt
	$(MSQ)echo "Création du fichier '$@' à partir de '$^'"
	$(MSQ)sed 's,Bonjour,Bonsoir,' $^ > $@
```

Nous pouvons maintenant ajouter de nouveaux noms à la variable *NOMS* (via l'opérateur **+=** par exemple), ce qui donne le *Makefile* final :

```
MSQ = @

NOMS = GLMF
NOMS += Tristan
NOMS += Clément
NOMS += Annie

MODELES = bonjour-@NOM@.txt bonsoir-@NOM@.txt

FICHIERS = $(foreach nom, $(NOMS), $(MODELES:@NOM@.txt=$(nom).txt))

.PHONY: clean all

all: $(FICHIERS)
	$(MSQ)echo "La cible courante est '$@'"
	$(MSQ)echo "La cible '$@' dépend de '$^'"
	$(MSQ)echo "La première dépendance de '$^' est donc '$<'"

bonjour-%.txt: nom = $(patsubst bonjour-%.txt,%,$@)
bonjour-%.txt:
	$(MSQ)echo "Création du fichier '$@'"
	$(MSQ)echo "Bonjour $(nom)" > $@

bonsoir-%.txt: bonjour-%.txt
	$(MSQ)echo "Création du fichier '$@' à partir de '$^'"
	$(MSQ)sed 's,Bonjour,Bonsoir,' $^ > $@

clean:
	$(MSQ)echo "Nettoyage des fichiers contenus dans la variable \$$(FICHIERS)(=$(FICHIERS))"
	$(MSQ)for f in $(FICHIERS); do \
	  echo "Le fichier $${f} sera supprimé"; \
	done
	$(MSQ)rm -f $(FICHIERS)
	rm -f *~
```

Il est important de noter que la cible **bonjour-%.txt** apparaît bel et bien deux fois (ce n'est pas une erreur). La première fois, c'est pour pouvoir extraire la variable *nom*, la seconde fois pour décrire la recette de fabrication. Voyons ce qui se passe avec ce livre de recette :

```
$ ls
Makefile
$ make
Création du fichier 'bonjour-GLMF.txt'
Création du fichier 'bonsoir-GLMF.txt' à partir de 'bonjour-GLMF.txt'
Création du fichier 'bonjour-Tristan.txt'
Création du fichier 'bonsoir-Tristan.txt' à partir de 'bonjour-Tristan.txt'
Création du fichier 'bonjour-Clément.txt'
Création du fichier 'bonsoir-Clément.txt' à partir de 'bonjour-Clément.txt'
Création du fichier 'bonjour-Annie.txt'
Création du fichier 'bonsoir-Annie.txt' à partir de 'bonjour-Annie.txt'
La cible courante est 'all'
La cible 'all' dépend de 'bonjour-GLMF.txt bonsoir-GLMF.txt bonjour-Tristan.txt bonsoir-Tristan.txt
bonjour-Clément.txt bonsoir-Clément.txt bonjour-Annie.txt bonsoir-Annie.txt'
La première dépendance de 'bonjour-GLMF.txt bonsoir-GLMF.txt bonjour-Tristan.txt bonsoir-Tristan.txt
bonjour-Clément.txt bonsoir-Clément.txt bonjour-Annie.txt bonsoir-Annie.txt' est donc 'bonjour-
GLMF.txt'
$ ls
bonjour-Annie.txt   bonjour-Clément.txt  bonsoir-Annie.txt  bonsoir-Clément.txt   Makefile
bonjour-GLMF.txt    bonjour-Tristan.txt  bonsoir-GLMF.txt   bonsoir-Tristan.txt
$ make
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(= bonjour-GLMF.txt bonsoir-GLMF.txt
bonjour-Tristan.txt bonsoir-Tristan.txt  bonjour-Clément.txt bonsoir-Clément.txt  bonjour-Annie.txt
```

```
bonsoir-Annie.txt)
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
Le fichier bonjour-Tristan.txt sera supprimé
Le fichier bonsoir-Tristan.txt sera supprimé
Le fichier bonjour-Clément.txt sera supprimé
Le fichier bonsoir-Clément.txt sera supprimé
Le fichier bonjour-Annie.txt sera supprimé
Le fichier bonsoir-Annie.txt sera supprimé
rm -f *~
$ ls
Makefile
```

3 Un *make* content

En guise de conclusion, regardons ce qui se passe avec notre livre de recettes si l'on demande la création du fichier **bonsoir-make.txt**.

```
$ ls
Makefile
$ make bonsoir-make.txt
Création du fichier 'bonjour-make.txt'
Création du fichier 'bonsoir-make.txt' à partir de 'bonjour-make.txt'
rm bonjour-make.txt
$ ls
bonsoir-make.txt   Makefile
$ make clean
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(= bonjour-GLMF.txt bonsoir-GLMF.txt
bonjour-Tristan.txt bonsoir-Tristan.txt  bonjour-Clément.txt bonsoir-Clément.txt  bonjour-Annie.txt
bonsoir-Annie.txt)
Le fichier bonjour-GLMF.txt sera supprimé
Le fichier bonsoir-GLMF.txt sera supprimé
Le fichier bonjour-Tristan.txt sera supprimé
Le fichier bonsoir-Tristan.txt sera supprimé
Le fichier bonjour-Clément.txt sera supprimé
Le fichier bonsoir-Clément.txt sera supprimé
Le fichier bonjour-Annie.txt sera supprimé
Le fichier bonsoir-Annie.txt sera supprimé
rm -f *~
$ ls
bonsoir-make.txt   Makefile
$ make clean NOMS=make
Nettoyage des fichiers contenus dans la variable $(FICHIERS)(= bonjour-make.txt bonsoir-
make.txt)
Le fichier bonjour-make.txt sera supprimé
Le fichier bonsoir-make.txt sera supprimé
rm -f *~
$ ls
Makefile
$
```

Le lecteur attentif aura remarqué – l'emphase aidant – que l'outil *make* sait créer ce fichier à partir du fichier **bonjour-make.txt**. Donc il crée ce premier fichier en suivant les recettes, mais comme celui-ci a été créé à titre temporaire, *make* le supprime automatiquement. Vous aurez remarqué également que la cible **clean** ne nettoie pas le fichier **bonsoir-make.txt** car il ne fait pas partie des fichiers « connus » du *Makefile*. Toutefois, on peut également passer en argument à **make** des valeurs de variables, le cas échéant celles-ci prévalent sur celles définies dans le *Makefile*.

Il est trop fort ce *make* !

Envie d'aller plus loin ?

Bien évidemment, l'objectif de cette introduction n'est pas de dévoiler toutes les fonctionnalités de l'outil *make* (le manuel fait 220 pages [**https://www.gnu.org/software/make/manual/**], je n'ai pas la prétention de faire mieux). Il est évident que pour des petits développements, faire un *Makefile* est entre judicieux et indispensable – ne serait-ce que pour faire le ménage sans se tromper.

Toutefois, pour des développements plus importants, la génération des *Makefiles* devient lourde et – dans la mesure où beaucoup de recettes se ressemblent – il existe des outils permettant de générer des *Makefiles* (*autoconf*/*automake* [**https://www.gnu.org/software/autoconf/**, **https://www.gnu.org/software/automake**], *CMake* [**https://cmake.org/**], module *ExtUtils ::MakeMaker* pour Perl

[**https://metacpan.org/pod/ExtUtils::MakeMaker**], …) ou s'en inspirant fortement (*SnakeMake* [**https://snakemake.readthedocs.io/**]).

/// Note : Cadeau bonus ///

Voici un *Makefile* « simple » qui permet de compiler un document LaTeX un peu complexe :

```
TEX = pdflatex
TEX_OPTS = -interaction=nonstopmode -file-line-error
BIB = bibtex
OPEN = xdg-open
MAX_TRY = 5

### Main LaTeX file
MAIN = document.tex

### Files that are included (and that produce a dedicated .aux file)
INCLUDED_FILES =              \
  Annexes/annexe1.tex         \
  Annexes/annexe2.tex         \
  IntroCcl/ccl.tex            \
  IntroCcl/intro.tex          \
  Corps/chap1.tex             \
  Corps/chap2.tex             \
  Corps/chap3.tex

### Files that are included/input (and that don't produce a dedicated .aux file)
OTHER_FILES =         \
  Annexes/metadata.tex        \
  Meta/infos.tex      \
  Meta/macros.tex

SOURCES =             \
  $(MAIN)             \
  $(INCLUDED_FILES)   \
  $(OTHER_FILES)

### Files that are not included/input but should be in the distributed
EXTRA_DIST =          \
  ChangeLog           \
  Makefile            \
  TODO

VERSION=$(shell git describe --always --dirty)
DIST = $(MAIN:.tex=-$(VERSION).tar.gz)

PDF = $(MAIN:.tex=.pdf)

DIRTY_FILES =                 \
  $(MAIN:.tex=.aux)           \
  $(INCLUDED_FILES:.tex=.aux) \
  $(MAIN:.tex=.bbl)           \
  $(INCLUDED_FILES:.tex=.bbl) \
  $(MAIN:.tex=.blg)           \
  $(INCLUDED_FILES:.tex=.blg) \
  $(MAIN:.tex=.done)          \
  $(MAIN:.tex=.lof)           \
  $(MAIN:.tex=.lot)           \
  $(MAIN:.tex=.out)           \
  $(MAIN:.tex=.toc)

NEED_RUN_MSG_REGEX=\(Rerun to get citations correct\)
NEED_RUN_MSG_REGEX:=$(NEED_RUN_MSG_REGEX)\|\(Label(s) may have changed. Rerun to get cross-
references right.\)
NEED_RUN_MSG_REGEX:=$(NEED_RUN_MSG_REGEX)\|\(Package rerunfilecheck Warning: File .* has changed.\)

.PHONY: all pdf view clean clean-all
.SUFFIXES: .tex .aux .bbl .pdf .done

all: pdf

pdf: $(PDF)

view: $(PDF)
	$(OPEN) $(PDF)

dist: $(DIST)

clean:
	rm -f *~ */*~ $(DIRTY_FILES) $(DIST)
```

```
clean-all: clean
	rm -f $(PDF)

run_tex: $(MAIN)
	CPT=0; \
	while \
	  [ $${CPT} -lt $(MAX_TRY) ] \
	  && $(TEX) $(TEX_OPTS) "$<" \
	  && grep -q "$(NEED_RUN_MSG_REGEX)" "$(<:.tex=.log)"; \
	do CPT=$$((CPT+1)) ; done

.aux.bbl:
	$(BIB) "$<" || true # since chapterbib provokes an error code return

$(PDF): $(MAIN:.tex=.done)

$(MAIN:.tex=.done): $(SOURCES) $(EXTRA_DIST)
	make run_tex
	make $(INCLUDED_FILES:.tex=.bbl) $(MAIN:.tex=.bbl) $(MAIN:.tex=.ind) $(MAIN:.tex=.gls)
	make run_tex
	make $(INCLUDED_FILES:.tex=.bbl) $(MAIN:.tex=.bbl) $(MAIN:.tex=.ind) $(MAIN:.tex=.gls)
	make run_tex
	touch "$@"

$(DIST): $(SOURCES) $(EXTRA_DIST)
	tar -czf "$@" $^
```

/// Fin Note ///