Introduction au Makefile

Nicolas Kielbasiewicz *

3 mars 2009

Le développement d'un programme et plus généralement d'un logiciel demande au(x) programmeur(s) de gérer plusieurs fichiers, voire plusieurs langages. Imaginons par exemple un logiciel de calcul matriciel dont le coeur est développé en C++ (avec une dizaine de fichiers aux dépendances diverses) et l'interface graphique en Java. Sans rentrer dans les détails, un tel logiciel demande la gestion d'une interface native entre les deux langages. Cela se traduit par le chargement par java d'une librairie dynamique générée par le code C++., librairie dynamique qui va demander un fichier .h généré par une commande de compilation java et le fichier source .cpp associé.

L'étape de compilation va donc demander de compiler le code java, de générer le fichier .h, de compiler tout le code C++ avec la gestion des diverses dépendances, de générer la librairie dynamique et de la positionner au bon endroit. On comprend bien que cette étape, sans l'apport d'un outil de gestion performant, sera très fastidieuse.

Imaginons maintenant le cas d'un étudiant qui écrit un certain nombre de fichiers indépendant les uns des autres, mais qui demandent de taper de manière répétitive les mêmes commandes de compilation. Avoir la possibilité d'effectuer toutes ces commandes en une seule fois, en conservant la souplesse de modifier librement le nombre de fichiers à compiler, serait là aussi très intéressant.

Imaginons maintenant les phases de développement des deux exemples précédents. Le programmeur va probablement effectuer un certain nombre de fois ces commandes de compilation. Cela serait une fois encore fastidieux si la phase de compilation se traduit par un grand nombre de commandes à exécuter, d'autant que cela demande aussi de supprimer certains fichiers.

La réponse à tous ces problèmes de convivialité, de souplesse et d'automatisation est le Makefile . C'est la raison pour laquelle je vais présenter ici la structure et les commandes principales qui constituent un Makefile . Je tiens toutefois à préciser que ce document n'est absolument pas exhaustif. Il contient les commandes de base qui permettent de réaliser ses premiers Makefile .

Table des matières

*Unité de Mathématiques Appliquées, École Nationale Supérieure de Techniques Avancées

1 Structure d'un Makefile

Afin de simplifier les illustrations, on va considérer le cas de trois fichiers exo1.f90, exo2.f90 et exo3.f90 indépendants deux à deux, sauf mention contraire, dans un environnement où le compilateur disponible est `gfortran`. Je tiens ici à préciser que ce qui va suivre fonctionnerait toujours avec un autre compilateur, quel que soit le langage.

1.1 Définir des variables

La compilation d'un fichier se fait en deux étapes :

1. Générer les fichiers objets par la commande :

```
gfortran -c exo1.c
```

2. Générer l'exécutable par la commande :

```
gfortran -o exo1 exo1.o
```

Il pourrait y avoir davantage d'options (inclusion de librairies, options d'optimisation, ...).

Dans notre fichier dont le nom est par défaut **Makefile**, on va donc définir différentes variables, au minimum trois, à savoir une pour le compilateur utilisé (si on change le compilateur, il n'y aura qu'une modification à faire), et deux pour les éventuelles options supplémentaires (une pour chacune dezs deux commande de compilation). En général, il y aura également une variable contenant la liste des fichiers sources, une variable contenant la liste des fichiers objets, voire également une variable pour la liste des exécutables. Bien que l'on puisse mettre les noms que l'on veut, il y a des noms usuels. Concernant la variable de compilateur, on l'appelle généralement FC pour Fortran, CC pour C et C++, ... De même, les options concernant la génération de l'objet sera FFLAGS en Fortran, CFLAGS en C et C++, ..., tandis que les options liées à la génération de l'exécutable seront usuellement stockées dans la variable LDFLAGS.

```
FC = gfortran
FFLAGS =
LDFLAGS =
```

1.2 Définir des règles de compilation

Si vous avez déjà installé des applications ou compilé et exécuté le code d'un collègue ou prédécesseur, vous avez pu remarquer qu'il y a des commandes qui reviennes souvent :

make ou **make all** c'est la commande de base qui compile tous les fichiers nécessaires.

make clean c'est la commande qui permet de "nettoyer" une arborescence pour pouvoir autoriser une recompilation.

make exo1 c'est la commande usuelle qui permet de ne compiler que la cible exo1.

make install c'est la commande qui lance la procédure d'installation d'une application.

make run c'est la commande qui lance l'exécution du logiciel compilé.

Ces commandes correspondent chacune à une règle de compilation On peut en définir une infinité. Dans le Makefile , cela se traduit par un bloc de la forme :

```
target: dep
	commande
```

La tabulation devant la ligne de commande est obligatoire.

target il s 'agit du nom de la règle de compilation. Par exemple : clean, all, exo1, exo1.o, ...

dep Il s'agit de la liste des dépendances nécessaires à l'exécution de la ligne de commande.

Si on reprend le cas de notre fichier exo1.f90, on obtiendrait alors en ajoutant les règles *all* et *clean* :

```
FC = gfortran
FFLAGS =
LDFLAGS =

all: exo1

exo1.o: exo1.f90
	$(FC) $(FFLAGS) -c exo1.f90

exo1: exo1.o
	$(FC) $(LDFLAGS) -o exo1 exo1.o

clean:
	rm -f exo1.o exo1
```

Deux questions se posent ici. Premièrement, on comprend ici que la manière la plus simple est de dupliquer ces lignes pour la gestion des fichiers exo2.f90 et exo3.f90. Comment faire pour s'affranchir des noms des fichiers ? Deuxièmement, dans cet exemple, on voit des informations redondantes sur les tag et les dépendances. Comment écrire les noms une seule fois ? La réponse va venir à présent avec l'utilisation des variables réservées.

1.3 Les variables réservées

1.3.1 Les variables réservées concernant les commande de compilation

$@ Utilisée dans une commande, cette variable récupère le nom de la cible.

$< Utilisée dans une commande, cette variable récupère la dépendance courante de la cible (une seule).

$^ Utilisée dans une commande, cette variable récupère l'ensemble des dépendances de la cible.

$? Utilisée dans une commande, cette variable récupère l'ensemble des dépendances mises à jour de la cible.

Si on reprend notre exemple précédent, on a :

```
FC = gfortran
FFLAGS =
LDFLAGS =

all: exo1 exo2 exo3

exo1.o: exo1.f90
	$(FC) $(FFLAGS) -c $<

exo1: exo1.o
	$(FC) $(LDFLAGS) -o $@ $<

...

clean:
	rm -f exo1.o exo2.o exo3.o exo1 exo2 exo3
```

1.3.2 Les variables réservées concernant les tags et les dépendances

% Ce caractère signifie un fichier en particulier, ou une règle.

$* Cette variable réservée permet dans la commande de récupérer le nom de la cible sans son extension.

Dans l'exemple précédent, cela devient :

```
FC = gfortran
FFLAGS =
LDFLAGS =

all: exo1 exo2 exo3

%.o: %.f90
	$(FC) $(FFLAGS) -c $<

%: %.o
	$(FC) $(LDFLAGS) -o $@ $<

clean:
	rm -f exo1.o exo2.o exo3.o exo1 exo2 exo3
```

On a ainsi condensé les règles de compilation des 3 fichiers. Il reste néanmoins un problème dans le cas où on rajouterait un autre fichier. On peut alors souhaiter récupérer de manière automatique la liste des fichiers à compiler. Pour cela, on va utiliser de nouvelles variables :

```
FC = gfortran
FFLAGS =
LDFLAGS =

SRCS = exo1.f90 exo2.f90 exo3.f90
OBJS = $(SRCS:.f90=.o)
EXEC = $(SRCS:.f90=)

all: $(EXEC)

%.o: %.f90
	$(FC) $(FFLAGS) -c $<

%: %.o
	$(FC) $(LDFLAGS) -o $@ $<

clean :
	rm -f $(OBJS) $(EXEC)
```

Si maintenant on veut récupérer tous les fichiers sources dont l'extension est .f90, on va écrire :

```
FC = gfortran
FFLAGS =
LDFLAGS =

SRCS = $(wildcard *.f90)
...
```

2 Comment ça marche ?

L'explication de la règle "make exo1" va permettre de comprendre le mécanisme du MAKEFILE .

Quand on lance cette commande, on demande au MAKEFILE d'exécuter la règle correspondant à la cible exo1. Dans le cas de notre exemple, cette cible demande le fichier objet exo1.o. Est-il plus récent que le fichier source ? Si ce n'est pas le cas ou s'il n'existe pas, il sera généré à l'aide de la cible correspondante. La génération du fichier objet demande le fichier source exo1.f90. Est-il plus récent que le fichier objet existant ? Si c'est le cas, le fichier objet sera régénéré. Avec un schéma, cela donnerait :

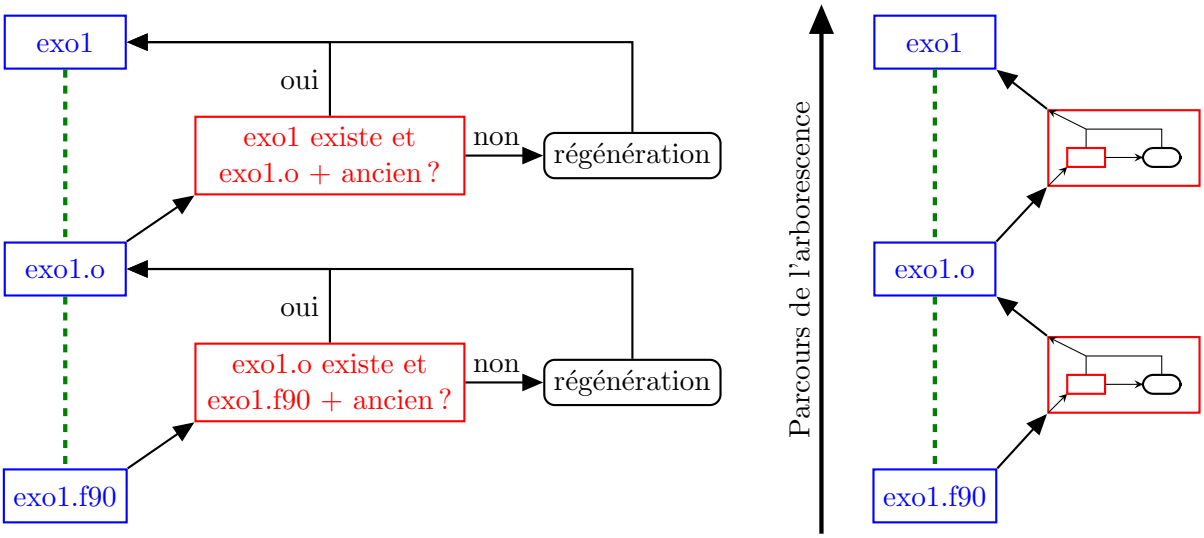

FIGURE 1 – Depuis les niveaux les plus bas de l'arbre des dépendances, les mécanismes de compilation jusqu'à l'exécutable. A droite, la version condensée qui sera très utile par la suite.

Dans le cas où la compilation n'est pas lancée car les conditions exposées précédemment sont telles que le fichier objet exo1.o est plus récent que le fichier exécutable exo1, on peut souhaiter forcer cette recompilation. C'est dans ce cas qu'on utilisera la commande **make clean**.

3 Pour aller plus loin, cas de dépendances multiples en Fortran 90

3.1 Fonctions et routines définies à part

Imaginons maintenant que exo1.f90 dépende de exo2.f90 et que ce dernier ne comporte pas de "bloc program" mais simplement des définitions de fonctions. Le MAKEFILE tel que nous l'avons écrit ne permet pas de compiler exo1.f90 correctement. En effet, la dépendance entre exo1 et exo2 n'est pas gérée, mais il faut également prendre en compte le fait que l'on ne peut définir une cible exo2, puisque l'on ne peut en faire un exécutable.

Voilà ce que donnerait les commande de compilation nécessaires à l'obtention de exo1 :

```
gfortran -c exo2.f90
gfortran -c exo1.f90
gfortran -o exo1 exo1.o exo2.o
```

Pour les prendre en compte, il y a simplement quelques correctifs mineurs à apporter au MAKEFILE précédent :

```
FC = gfortran
FFLAGS =
LDFLAGS =

SRCS = exo1.f90 exo3.f90
OBJS = $(SRCS:.f90=.o)
EXEC = $(SRCS:.f90=)

all: $(EXEC)

%.o: %.f90
	$(FC) $(FFLAGS) -c $<

%: %.o
	$(FC) $(LDFLAGS) -o $@ $^

exo1: exo2.o

clean :
	rm -f $(OBJS) $(EXEC) exo2.o
```

1. Puisque exo2.f90 ne comporte pas de bloc program, on n'en fera pas d'exécutable. On l'enlève donc de la liste des fichiers source.
2. Puisque $^ comprend toutes les dépendances et pas simplement celles de la cible courante (%.o), on va remplacer $< par $^ dans notre exemple simple.
3. Il faut rajouter la dépendance entre la cible exo1 et le fichier objet exo2.o
4. Il faut enfin rajouter exo2.o dans la liste des fichiers à effacer dans la cible "clean".

A présent, voilà le schéma de dépendance et de compilation :

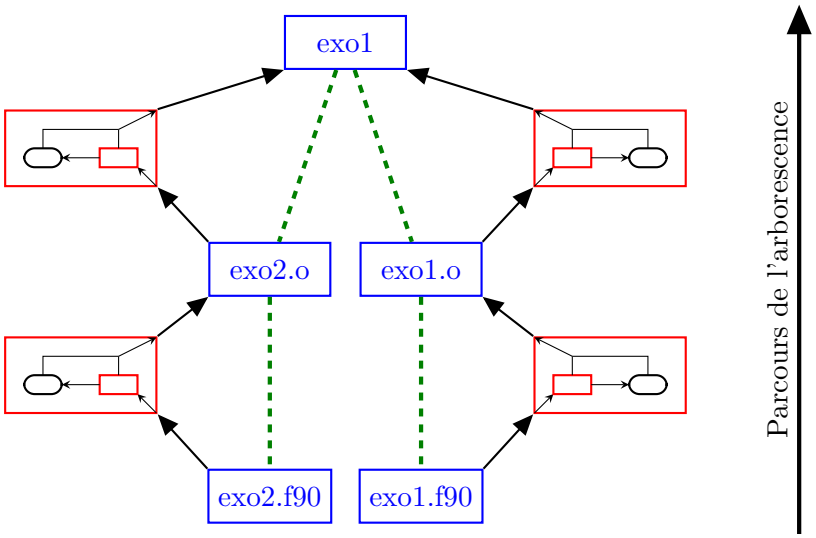

FIGURE 2 – Depuis les niveaux les plus bas de l'arbre des dépendances, les mécanismes de compilation jusqu'à l'exécutable.

3.2 Modules définis à part

Considérons maintenant que exo2.f90 comporte la définition d'un module appelé `exo2`. Il y a deux nouveautés à prendre en compte ici, même si les commandes de compilations restent les mêmes que dans la sous-section précédente.

1. Le génération de l'objet exo2.o va s'accompagner de la génération d'un fichier qui s'appellera exo2.mod
2. La génération de l'objet exo1.o va dépendre de exo2.mod, même si ce fichier n'intervient pas explicitement dans les commandes de compilation.
3. Il faut donc aussi ajouter exo2.mod à la liste des fichiers à effacer dans la cible "clean".

Il existe enfin un problème que ne peut pas résoudre le Makefile (et oui, ce n'est pas parfait). En effet, comme la notion de mise à jour est liée à la date, il est possible qu'à la génération de exo2.o et exo2.mod, celle-ci ne soit pas mise à jour avec suffisamment de précision pour voir une différence. La solution proposée ici est de forcer la mise à jour de cette date à l'aide de la commande Unix **touch**.

```
FC = gfortran
FFLAGS =
LDFLAGS =

SRC = $(wildcard *.f90)
SRCS = $(filter-out exo2.f90, $(SRC))
OBJS = ${SRC:.f90=.o}
EXEC = ${SRCS:.f90=}

all: $(EXEC)

exo2.o exo2.mod: exo2.f90
	$(FC) $(FFLAGS) -c $<
	touch $*.o $*.mod

exo1.o: exo2.mod

%.o: %.f90
	$(FC) $(FFLAGS) -c $<

exo1: exo2.o

%: %.o
	$(FC) $(LDFLAGS) -o $@ $^

clean :
	rm -f $(OBJS) $(EXEC) exo2.mod
```

Cet exemple est déjà plus complexe que les précédents et le schéma de dépendance et de compilation l'est tout autant. Voilà néanmoins ce que cela donnerait :

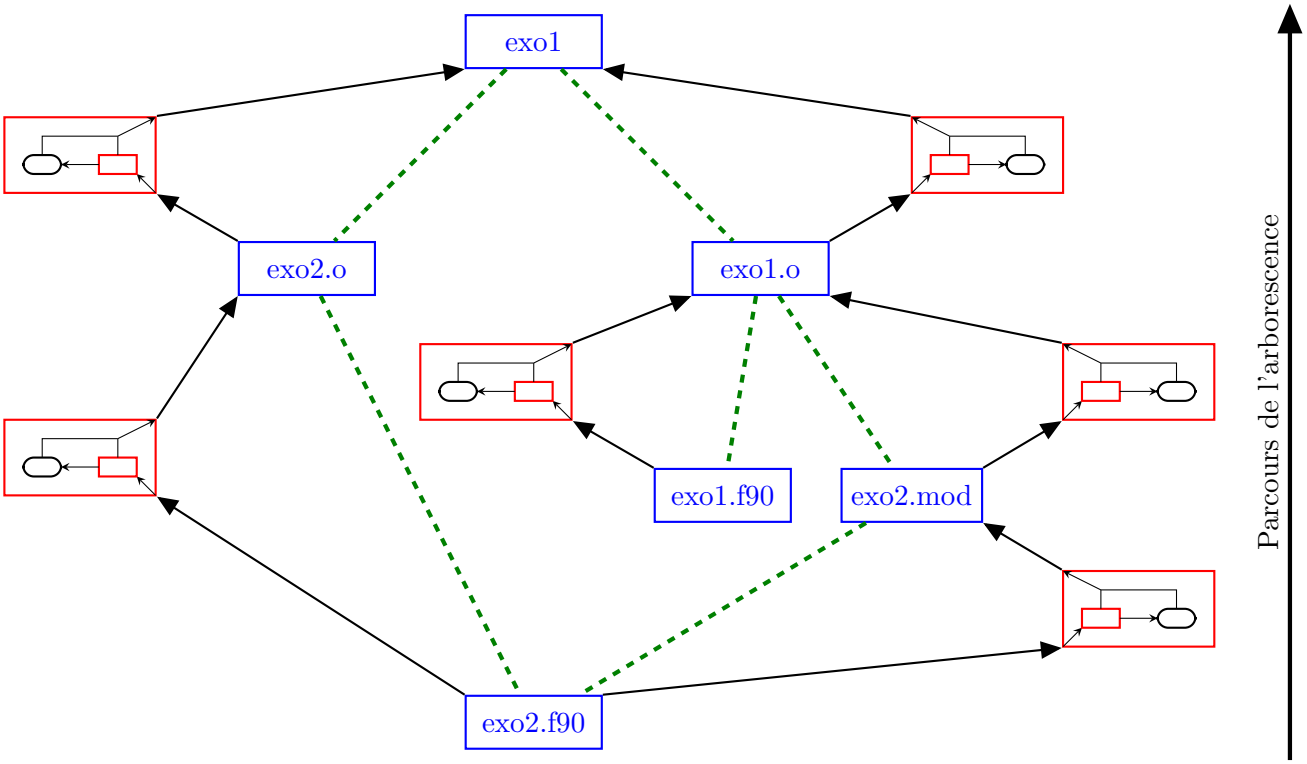

FIGURE 3 – Depuis les niveaux les plus bas de l'arbre des dépendances, les mécanismes de compilation jusqu'à l'exécutable.