Pratique de la programmation et projet
TP 4 : GNU make (suite)

Frédéric Vivien

1 Génération automatique des dépendances

Ci-dessous est cité le manuel de référence expliquant une méthode de génération automatique des dépendances et de remise à jour automatique et minimale de ces dépendances. Mettez en œuvre cette méthode sur un exemple contenant plusieurs fichiers sources incluant des fichiers d'en-têtes différents. Vous vérifierez le comportement de votre système lorsque qu'un des fichiers sources ou un des fichiers d'en-têtes est modifié.

Generating Prerequisites Automatically

In the makefile for a program, many of the rules you need to write often say only that some object file depends on some header file. For example, if 'main.c' uses 'defs.h' via an #include, you would write:

```
main.o: defs.h
```

You need this rule so that make knows that it must remake 'main.o' whenever 'defs.h' changes. You can see that for a large program you would have to write dozens of such rules in your makefile. And, you must always be very careful to update the makefile every time you add or remove an #include.

To avoid this hassle, most modern C compilers can write these rules for you, by looking at the #include lines in the source files. Usually this is done with the '-M' option to the compiler. For example, the command:

```
cc -M main.c
```

generates the output:

```
main.o : main.c defs.h
```

Thus you no longer have to write all those rules yourself. The compiler will do it for you.

Note that such a prerequisite constitutes mentioning 'main.o' in a makefile, so it can never be considered an intermediate file by implicit rule search. This means that make won't ever remove the file after using it [...].

With old make programs, it was traditional practice to use this compiler feature to generate prerequisites on demand with a command like 'make depend'. That command would create a file 'depend' containing all the automatically-generated prerequisites; then the makefile could use include to read them in [...]

In GNU make, the feature of remaking makefiles makes this practice obsolete–you need never tell make explicitly to regenerate the prerequisites, because it always regenerates any makefile that is out of date. [...]

The practice we recommend for automatic prerequisite generation is to have one makefile corresponding to each source file. For each source file 'name.c' there is a makefile 'name.d' which lists what files the object file 'name.o' depends on. That way only the source files that have changed need to be rescanned to produce the new prerequisites.

Here is the pattern rule to generate a file of prerequisites (i.e., a makefile) called 'name.d' from a C source file called 'name.c':

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< \
                  | sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
                [ -s $@ ] || rm -f $@
```

[...] The '-e' flag to the shell makes it exit immediately if the $(CC) command fails (exits with a nonzero status). Normally the shell exits with the status of the last command in the pipeline (sed in this case), so make would not notice a nonzero status from the compiler.

With the GNU C compiler, you may wish to use the '-MM' flag instead of '-M'. This omits prerequisites on system header files. [...]

The purpose of the sed command is to translate (for example):

```
main.o : main.c defs.h
```

into:

```
main.o main.d : main.c defs.h
```

This makes each '.d' file depend on all the source and header files that the corresponding '.o' file depends on. make then knows it must regenerate the prerequisites whenever any of the source or header files changes.

Once you've defined the rule to remake the '.d' files, you then use the include directive to read them all in. [...] For example:

```
sources = foo.c bar.c
include $(sources:.c=.d)
```

(This example uses a substitution variable reference to translate the list of source files 'foo.c bar.c' into a list of prerequisite makefiles, 'foo.d bar.d'. [...]) Since the '.d' files are makefiles like any others, make will remake them as necessary with no further work from you. [...]

2 Examen de juin 2001

L'objectif de cette partie est de mettre au point des makefiles pour compiler des fichiers contenant un programme écrit dans le langage *objective caml*. Aucune connaissance de ce langage n'est nécessaire pour la résolution de ce problème.

2.1 Un makefile basique pour *ocaml*

Le compilateur d'*objective caml* est `ocamlc`. Les fichiers source ont une extension `.ml` alors que les fichiers objets ont pour extension `.cmo`. Pour créer le fichier objet `toto.cmo` à partir du fichier source `toto.ml`, la commande est :

```
ocamlc -c toto.ml
```

Ensuite pour obtenir le fichier exécutable `go` à partir des fichiers objets `toto.cmo`, `titi.cmo`, `tutu.cmo` et `tata.cmo`, la commande est :

```
ocamlc toto.cmo titi.cmo tutu.cmo tata.cmo -o go
```

Vous trouverez à l'url `http://icps.u-strasbg.fr/~vivien/Enseignement/PPP-2001-2002/` une archive `ocaml.tgz` qui inclut un répertoire `ocaml/kb/` qui contient un ensemble de fichiers `prelude.ml terms.ml equation.ml order.ml kb.ml go.ml` qui représentent un programme en *objective caml*. On demande d'écrire un fichier `makefile0` satisfaisant les contraintes suivantes :

1. Dans ce premier makefile, la fabrication des fichiers objets à partir des fichiers source devra faire l'objet d'une et une seule règle.
2. Pendant le processus de la fabrication du fichier exécutable, chaque fichier source devra être compilé au plus une fois.
3. La fabrication du fichier exécutable à partir des fichiers objets devra faire l'objet d'une et une seule règle. Attention, les fichiers objets doivent apparaître dans la ligne de compilation **obligatoirement** dans l'ordre suivant :

   `prelude.cmo terms.cmo equation.cmo order.cmo kb.cmo go.cmo`
4. Éviter toutes les répétitions par l'utilisation de variables.

2.2 Un makefile amélioré

Dans une deuxième version `makefile1`, on impose que la seule liste explicite de fichiers soit celle des fichiers source, `prelude.ml terms.ml equation.ml order.ml kb.ml go.ml`, et que la liste des fichiers objet, `prelude.cmo terms.cmo equation.cmo order.cmo kb.cmo go.cmo`, soit automatiquement générée à partir de la liste des fichiers source.

2.3 La compilation optimisée

Le compilateur `ocamlopt` permet de compiler les fichiers source d'*objective caml* de façon optimisée. Les fichiers objets compilés de façon optimisée ont l'extension `.cmx` (au lieu de `.cmo`). Pour créer le fichier objet `toto.cmx` à partir du fichier source `toto.ml`, la commande est :

```
ocamlopt -c toto.ml
```

Pendant cette compilation, sont également générés de façon automatique les fichiers `toto.cmi` et `toto.o`. Ensuite pour obtenir le fichier exécutable `go_opt` à partir des fichiers objets `toto.cmx`, `titi.cmx`, `tutu.cmx` et `tata.cmx`, la commande est :

```
ocamlopt toto.cmx titi.cmx tutu.cmx tata.cmx -o go_opt
```

On demande d'écrire `makefile2` satisfaisant les contraintes suivantes :

1. Ce makefile devra permettre de compiler les fichiers soit de façon ordinaire (en tapant `make kb -f makefile2`), soit de façon optimisée (en tapant `make opt -f makefile2`), soit des deux façons (en tapant `make -f makefile2`).
2. Écrire une règle permettant de détruire tous les fichiers intermédiaires créés pendant le processus de compilation (autres que les sources et les exécutables) ainsi que les fichiers `core`.
3. Écrire une autre règle qui permet également de détruire les exécutables, ne laissant que les fichiers source.

Vous veillerez à ce qu'une erreur survenue dans l'exécution des commandes ci-dessus (par exemple le fait qu'il n'y ait aucun fichier avec une extension `.o`) ne puisse pas empêcher les commandes suivantes d'être exécutées.

2.4 Gérer les dépendances

En *objective caml*, il existe des fichiers entête dont l'extension est `.mli`. Ces fichiers jouent en quelque sorte le même rôle que les fichiers entête en langage C (extension `.h`). Ils contiennent des définitions de type et des prototypes de fonctions. Par exemple le fichier `token.mli` contient la définition des types et le prototype des fonctions définies dans `token.ml`. Lorsqu'une fonction dans un autre fichier, par exemple `parser.ml` a besoin de celles qui ont été définies dans `token.ml` alors le fichier `parser.ml`, doit contenir la ligne

```
open Token;;
```

qui est l'équivalent de la directive `#include` du préprocesseur `cpp`. Donc lorsque le fichier `token.mli` a été modifié, alors tous les fichiers contenant la ligne

```
open Token;;
```

doivent être recompilés. Les dépendances de ce genre doivent être exprimées dans le makefile qui gère la compilation du programme. Pour cela, on utilise la commande `ocamldep`. Cette commande prend en argument un ensemble de fichiers source et entête (extensions `.ml` et `.mli`) dans lesquels elle recherche des indications de dépendance (en particulier les directives `open`). Étant donnés ces arguments, la commande `ocamldep` écrit sur la sortie standard toutes les dépendances écrites dans un format compréhensible par `make`.

Dans le répertoire `ocaml` obtenu à partir de l'archive `ocaml.tgz`, vous disposez d'un répertoire `asl/` qui contient un ensemble de fichiers `asl.ml main.ml parser.ml prel.ml run.ml semant.ml token.ml typing.ml`, et les fichiers `asl.mli parser.mli token.mli main.mli prel.mli` qui représentent un programme en *objective caml*. Essayez le fonctionnement de `ocamldep` en tapant

`ocamldep parser.mli run.ml`

Écrire un fichier `makefile`

1. De même que pour la question 2.3, ce makefile devra permettre de compiler le programme de ce répertoire de façon ordinaire (en tapant `make asl`) de façon optimisée (en tapant `make opt`) et des deux façons (en tapant `make`). Là aussi, les fichiers objets doivent apparaître dans la ligne de compilation **obligatoirement** dans l'ordre suivant : `prel.cmo asl.cmo token.cmo parser.cmo semant.cmo typing.cmo main.cmo run.cmo`
2. Il devra aussi comporter une règle permettant de détruire tous les fichiers intermédiaires produits par la compilation (sauf les exécutables) et une autre règle permettant également de détruire tous les exécutables.
3. Ce makefile devra également comporter une règle permettant de générer un fichier `.depend` qui contient toutes les règles de dépendance écrites dans un format compréhensible par `make`. Ce fichier `.depend` sera inclus dans le makefile grâce à la directive `include`.

2.5 Un makefile récursif

On s'intéresse au répertoire `ocaml` qui contient les répertoires `kb` et `asl` mentionnés ci-dessus. Avant toute chose, copier tout votre répertoire `ocaml` dans un autre répertoire `ocaml_copie` avec la commande unix `cp -r` de façon à ce que les manipulations qui suivent ne puissent pas les détruire. Écrire un fichier `makefile` dans le répertoire `ocaml` permettant de gérer les deux répertoires sans y entrer. Ce makefile devra permettre de réaliser les tâches suivantes.

1. Tout compiler dans le répertoire `asl` de façon ordinaire et optimisée (en tapant `make asl`) ;
2. Tout compiler dans le répertoire `kb` de façon ordinaire et optimisée (en tapant `make kb`) ;
3. Éliminer tous les fichiers intermédiaires produits par la compilation (sauf les exécutables) en tapant `make clean`, et éliminer également les exécutables (en tapant `make veryclean`) ;
4. Sauvegarder dans le répertoire qui se trouve au dessus de `ocaml` un fichier archive compressé contenant toutes les données du répertoire `ocaml` et des répertoires sous-jacents.