MMFAI
Cours de programmation

TP numéro 5
semaine du 30 novembre 1998

Manipulation de Graphes: le programme `make`

Gestion de gros projets

Si vous avez à gérer un projet assez important, c'est nécessaire de disposer d'outils adéquats. Par exemple, il serait bien de disposer de la compilation séparée des modules qui composent votre projet. On a déjà vu, dans le corrigé du TP2 que si vous avez un fichier `essai.c` qui utilise les fonctions définies dans le fichier `matrops.c`, plutôt que de le compiler avec la commande

```
gcc -o essai matrops.c essai.c
```

qui compile les deux fichiers ensemble, on peut procéder en deux phases: d'abord

```
gcc -c matrops.c
```

qui compile `matrops.c` et produit le fichier "objet" `matrops.o` mais pas d'executable, ensuite la commande

```
gcc -o  essai matrops.o essai.c
```

qui compile `essai.c`, et ensuite le compose avec `matrops.o` pour produire l'executable `essai`. On gagne du temps si `matrops.c` est un programme stable et `essai.c` est en développement, parce que la compilation prends beaucoup plus de temps que la "composition" (techniquement appelée "édition de liens").
Et on peut gagner beaucoup de temps, dans le cas de projets complexes qui comportent plusieurs dizaines de modules séparés, si on arrive a ne recompiler, après une modification de quelques fichiers sources, que ces modules qui en ont besoin.
Pour faire ça, on dispose (sous Unix, mais aussi dans le monde Windows et MacOS) d'un outil essentiel nommé `make` qui prend en entrée un fichier (normalement nommé `makefile`) qui décrit les dépendances entre les modules de votre projet, et les actions à effectuer pour mettre à jour ces modules en cas de modifications. Par exemple, pour notre vieux projet du TP2, on peut écrire le `makefile` suivant:

```
essai: essai.c matrops.o
	gcc -o essai matrops.o essai.c

matrops.o: matrops.c matrops.h
	gcc -c matrops.c
```

Ici les dépendances sont notés *modulequidepende*: *moduledontondepend1* ... *moduledontondependn*, et les actions à faire sont des lignes qui commencent par une tabulation après la ligne qui décrit la dépendance. Ici on dit par exemple que pour obtenir l'executable `essai` il faut avoir `essai.c` et `matrops.o` et exécuter la commande `gcc -o  essai matrops.o essai.c`. Ces dépendances, qui ne doivent pas être circulaires, décrivent en effet un graphe acyclique dirigé.
Si on tape `make -f makefile essai`, le programme `make` va essayer de "construire" essai. Pour cela, il doit explorer le graphe des dépendances pour identifier tous les modules dont `essai` dépend, ensuite exécuter les actions nécessaires a les construire *dans le bon ordre*, (par exemple, on ne peut pas exécuter `gcc -o  essai matrops.o  essai.c` avant d'avoir construit `matrops.o`!), et sans

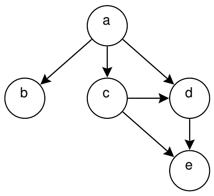
repétitions (si a dépend de b et c qui tous les deux dépendent de d, on veut quand même construire d une seule fois). De plus, `make` vérifie, avant de construire un module, si "ça vaut la peine" de le reconstruire: pour cela, il utilise la date de dernière modification du module, et si ell'est plus récente que celle des modules dont il dépend, il ne fait rien (parce-que on assume que l'opération de construction a déjà été faite, par exemple par une exécution antérieure de `make`)[1].

Un petit `make` maison

Aujourd'hui, nous allons nous intéresser à écrire notre propre version expérimentale de `make`, dans laquelle nous nous intéressons dans un premier temps seulement à l'analyse des dépendences, et non pas à comment utiliser les dates pour ne pas recompiler ce qui aurait déjà été compilé. Nous allons aussi ignorer pour l'instant les "actions". Donc, nous travaillerons sur des fichiers `makefile` sans actions, comme par exemple

```
a: b c d
b:
c: d e
d: e
e:
```

qui décrive le GAD suivant



Exercice 0.1 *Choisissez une représentation de graphes qui vous semble adéquate pour* `make`*, et définissez le correspondant type C, de nom* `GAD` *(pour Graphe Acyclique Dirigé). Reflechissez bien aux informations que vous allez garder pour chaque noeud (il y aura sûrement une chaîne de caractères pour le nom du noeud, mais quoi d'autre?).*

Exercice 0.2 *En utilisant la fonction* `int readline(FILE *fp, char **res)` *du TP4, écrivez une fonction* `int readmakefile(char *nomdufichier, GAD **graphe)` *qui lit un fichier de dépendances et produit le GAD associé, en retournant un code d'erreur s'il y a des problèmes pour lire le fichier.*
Tout ça est fait de façon à pouvoir appeler la fonction comme suit

```
GAD *mongraphe;

if (readmakefile("makefile",&mongraphe)!=0)
   {printf("Erreur en lisant makefile. J'abandonne.\n"; exit(1);};
```

Exercice 0.3 (exploration) *Écrivez maintenant une fonction qui prends un GAD et le nom d'un noeud n dans ce GAD et retourne la liste des noeuds dont n dépend, dans le bon ordre pour pouvoir l'utiliser pour construire n. Par exemple, pour le fichier d'en haut on pourrait rendre e, b, d, c, a. Si vous n'avez pas garde le code pour les listes que l'on a vu en TP3, il suffira d'imprimer les noms des noeuds dans le bon ordre, plutôt que retourner la liste.*

[1] Attention: il faut utiliser cette information sur les dates dans le bon ordre aussi!

Exercice 0.4 (optionnel) *Modifiez votre code pour prendre en compte aussi l'information sur la date: maintenant, les noms dans* `makefile` *seront des vrais noms de fichier, et on vous demande de retourner la liste des seuls noeuds qu'il vaut la peine de reconstruire. Pour cela vous pourriez avoir envie d'utiliser la fonction de librairie* `stat` *(faire* `man stat`*).*

N'oubliez pas de vous déconnecter avant d'abandonner le clavier (voir notes du stage Unix).