GNU make

Table des matières

1 Préambule

L'essentiel de ce document est une synthèse du document en anglais que l'on peut trouver à l'adresse `http://www.gnu.org/manual/make/html_mono/make.html` sur le site officiel de GNU (`http://www.gnu.org/`).

Tous les *makes* ne sont pas des *GNU make*. Au département d'informatique de l'ULP, tous les PC ont seulement GNU make. Par contre, plusieurs makes sont installés sur le serveur ada :

1. `/usr/local/bin/make` est un make GNU.
2. `/usr/ccs/bin/make` est un make conçu par SUN.

Les extensions de fonctionnalités fournies par le GNU make sont nombreuses et souvent très importantes. C'est pourquoi nous ne nous contenterons pas de décrire les fonctionalités communes à tous les *makes*. De votre côté il vous faudra vérifier, à chaque fois que vous utiliserez un *make*, qu'il s'agit bien d'un *GNU make*.

2 L'objectif visé

L'objectif premier de l'outil `make` est de permettre de construire des fichiers, éventuellement à partir d'autres fichiers, par des commandes shell, et de manière automatique. Exemples :

- Compilation de programmes C, parce que les fichiers objets et les exécutables peuvent être construits par des commandes shell.
- Construction d'un fichier PostScript ou PDF à partir d'un fichier au format LaTeX.
- Construction de librairies, sauvegarde dans des fichiers archives, etc.

Accessoirement, l'outil `make` peut servir à exécuter un grand nombre d'opérations, quasiment à la manière d'un shell script. On peut ainsi mettre au point une procédure pour faire le ménage (détruire les fichiers inutiles) dans toute une arborescence, par exemple sur tout un compte ou, pourquoi pas, sur toute une machine, simplement en tapant « `make récursive_clean` ».

Évidemment, l'outil make ne sait pas, à l'avance, distinguer les fichiers que vous considérez comme inutiles de ceux que vous voulez conserver. Il ne sait pas non plus quels fichiers il faut construire, à partir de quels autres fichiers, ni dans quelles conditions et avec quelles commandes il faut mettre à jour ces fichiers. C'est l'utilisateur qui doit indiquer tout cela en éditant un fichier qui contiendra toutes les informations nécessaires. Par exemple pour construire le fichier `scanline` à partir des informations contenues dans le fichier `scanline.mk`, il faut taper : « `make scanline -f scanline.mk` ». Il est également possible de taper simplement « `make scanline` » Mais alors les informations qui permettent de construire le fichier `scanline` doivent être contenues dans un fichier qui s'appelle `Makefile` (ou `makefile`). Ainsi la commande précédente sera l'exact équivalent de : « `make scanline -f Makefile` » Dans les sections suivantes, les fichiers contenant ces informations seront tous appelés des *makefiles*. Nous allons voir comment écrire un fichier de type makefile et comment l'utiliser.

3 Les règles

3.1 Les composantes d'une règle

Les makefiles sont, en grande partie, constitués d'une suite de **règles**. Une règle (*rule* en anglais) est constituée de :

- **cibles** (*targets* en anglais) ;
- **prérequis** ou **dépendances** (*prerequisites* ou *dependencies* en anglais) ;
- **commandes** (*commands* en anglais).

3.2 La fonction de ces composantes

- Une cible est généralement le nom de l'un des fichiers (ou d'un des ensembles de fichiers) que l'on souhaite construire avec le makefile.
- Les prérequis désignent toujours les fichiers dont dépendent la cible ou les cibles. Plus précisément, un fichier `Prrq` donné doit être le prérequis d'une cible `Targ` donné, si on considère qu'il faut remettre à jour `Targ` à chaque fois que le fichier `Prrq` est modifié.
- Enfin les commandes sont une liste de commandes qu'il faut exécuter pour reconstruire la cible.

3.3 La syntaxe

Voici le schéma général :

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ... prerequisD
            commande1 ; commande2 ; commande3 ; ... ; commandeC
```

ou bien

```
cible1 cible2 ... cibleN : prerequis1 prerequis2 ... prerequisD ; commande1 ; commande2 ; ... ; commandeC
```

ou encore

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ... prerequisD
            commande1
            commande2
            commande3
            ...
            commandeC
```

Tous les cas intermédiaires sont possibles :

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ... prerequisD ; commande1 ;
            commande2
            commande3 ; ... ; commandeC
```

Les cibles doivent être séparées par des espaces (blancs ou tabulations), ainsi que les prérequis. Par contre, deux commandes consécutives sur une même ligne doivent être séparées par des points-virgules « `;` ».

3.3.1 Le découpage d'une ligne en plusieurs morceaux (le caractère « \ »)

Les cibles et les prérequis doivent se trouver sur une même ligne. Or pour des raisons esthétiques et de lisibilité, il est quelquefois préférable de répartir certaines expressions sur plusieurs lignes. Pour cela on utilise le caractère « \ » comme dans les programmes C et dans tout ce qui utilise le préprocesseur `cpp`. Lorsque make lit un fichier makefile, il ignore toutes les occurrences du caractère « \ » suivi d'un caractère « `\n` » (newline). Donc la règle :

```
cible1 cible2 \
cible3 ... cibleN : \
prerequis1\
prerequis2 prerequis3 \
```

```
... prerequisD ; commande1 ;
        commande2
        commande3 ; ... ; commandeB ; \
commandeC
```

est l'exact équivalent de la règle :

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2
prerequis3 ... prerequisD ; commande1 ;
        commande2
        commande3 ; ... ; commandeB; commandeC
```

3.3.2 Les commentaires

Avant de commencer effectivement à lire le makefile, une fois que les caractères « \ » ont été éliminés par make, celui-ci cherche les caractères « # » et élimine tous les caractères qui se trouvent derrière et sur la même ligne. Ceci permet d'écrire des commentaires après les caractères « # ». L'élimination des commentaires est effectuée *après* l'élimination des « \ ». Cela signifie que, si à la fin d'une ligne de commentaire vous ajoutez un caractère « \ », alors la ligne suivante sera également considérée comme faisant partie du commentaire.

3.3.3 Un point importantissime : le caractère « \t » (TAB)

Lorsqu'une ligne commence avec le caractère « `\t` », ce qui suit ce caractère est interprété par make comme une commande shell. Réciproquement, une ligne de commande ne peut pas se trouver tout à fait en début de ligne, elle doit nécessairement être précédée d'un caractère « `\t` ». La seule exception est une commande séparée de la commande précédente par un « ; » suivi d'un « \ » suivi d'un « `\n` » (cf. ce qui se passe entre `commandeB` et `commandeC` ci-dessus). Évidemment, un ensemble de caractères « espace » de même longueur ne remplit pas du tout la même fonction. Ce fonctionnement peut être parfois déroutant au départ. Il arrive souvent que l'on écrive une commande que `make` ne considère pas comme une commande parce qu'il n'a pas trouvé le caractère « `\t` » au début de la ligne. Réciproquement il arrive souvent aussi qu'un caractère « `\t` », tapé en début de ligne pour des raisons esthétiques et aussi pour éviter de taper N fois sur la touche « espace » , ait dérouté make qui a essayé d'interpréter ce qui suit comme une commande shell.

3.4 L'application d'une règle

Lorsqu'on entre « `make cible1` » make cherche dans le makefile la cible « `cible1` » et s'intéresse alors à la règle de construction correspondante. Avant toute chose, make vérifie si, parmi les prérequis de `cible1`, certains éléments ne seraient pas eux-mêmes les cibles de certaines autres règles dans le makefile.

- Si aucun des prérequis ne correspond à une cible de ce makefile, alors make ne s'intéressera plus aux autres règles. Il se contentera de déterminer s'il faut ou non exécuter les commandes associées à `cible1` et, si oui, les exécutera.
- Par contre si au moins un des prérequis de `cible1` (appelons le « `prrq` ») est la cible d'une autre des règles exprimées dans le makefile alors, avant de se demander s'il faut ou non exécuter la commande associée à `cible1`, il doit s'intéresser à la règle dont `prrq` est la cible, et il doit faire pour `prrq`, tout ce qu'il a fait pour `cible1`. Autrement dit, avant toute chose, il doit encore vérifier si des prérequis de `prrq` correspondent à la cible d'autres règles et, le cas échéant, il délaissera la règle de `prrq` pour s'intéresser aux autres règles.

Ce n'est que lorsque `make` s'est intéressé récursivement à toutes les règles dont les cibles sont les prérequis de `cible1`, qu'enfin il peut déterminer s'il faut ou non exécuter les commandes associées à `cible1`. Par exemple, soient les quatre règles suivantes écrites dans un fichier makefile qui permet d'obtenir un fichier PostScript (imprimable) à partir d'un fichier texte au format LaTeX.

```
# Ceci est un exemple de makefile

coucou.dvi : coucou.tex
        latex coucou.tex
```

```
article.ps : coucou.dvi figure.eepic
        dvips coucou.dvi -o article.ps
bang :
        banner coucou
```

Suivons le parcours de make. Lorsqu'on tape :

```
make coucou.dvi
```

`make` s'intéresse à la première règle. Le seul prérequis de cette règle est `coucou.tex` qui ne correspond à la cible d'aucune autre règle. Donc make peut se désintéresser de toutes les autres règles et se demander s'il faut ou non mettre à jour le fichier coucou.dvi et, le cas échéant, de réaliser effectivement la mise à jour. Par contre, lorsqu'on tape :

```
make article.ps
```

la situation est tout à fait différente. `make` s'intéresse à la deuxième règle. Dans cette règle, un des deux prérequis (`coucou.dvi`) est la cible de la première règle. Donc, dans un premier temps, l'exécution de la commande de la deuxième règle (`dvips coucou.dvi -o article.ps`) n'est pas du tout à l'ordre du jour. `make` doit d'abord s'intéresser à la règle dont `coucou.dvi` est la cible et, le cas échéant, exécuter la commande associée. C'est seulement quand la première règle a été traitée qu'on peut s'intéresser à la commande associée à la cible `article.ps` et éventuellement l'exécuter. Si plusieurs prérequis d'une règle R sont des cibles d'autres règles, alors avant de pouvoir exécuter la commande associée à R, make devra d'abord s'intéresser à toutes ces règles, dans l'ordre où elles ont été écrites dans la liste des prérequis.

3.5 Les cibles « bidons » (*Phony* targets)

Certaines cibles n'ont aucun prérequis et ne désignent aucun fichier. Ces cibles sont appelées des *phony targets*, ce que j'ai traduit, tant bien que mal, par « cibles bidons ». Je ne connais pas l'expression française. En général, même en français, on parle de cibles « phony ». Quelquefois on parle de cibles « logiques », par opposition à des cibles correspondant à des fichiers physiques. Dans la suite, au risque de choquer les puristes de la langue française, je me contenterai de l'expression *phony*.

Ces cibles sont un moyen pratique de demander au makefile de réaliser certaines actions qui n'ont pas nécessairement trait à la construction de fichiers. Ainsi, dans l'exemple du paragraphe précédent, la troisième règle a une cible qui est *phony*. En l'occurrence la commande de cette règle n'effectue rien d'utile. Elle se contente d'écrire à l'écran « coucou » en grosses lettres. Pour exécuter cette commande, il suffit de taper : `make bang`. Le résultat est un affichage à l'écran qui dépend du système (cf. `http://coriolis.u-strasbg.fr/habibi/PPP/BannerLinux.html` pour voir le résultat obtenu sur les machines Linux du département, et `http://coriolis.u-strasbg.fr/habibi/PPP/BannerSolaris.html` pour le résultat obtenu sur `steed`).

3.6 Les conditions dans lesquelles les commandes d'une règle sont exécutées.

Dans les paragraphes précédents, nous mentionnions simplement que make « se demandait s'il faut ou non exécuter la commande d'une règle ». Quelles sont les conditions qui déclencheront l'exécution de ces commandes ? Supposons que le répertoire dans lequel se trouve le makefile s'appelle `rep` (entre nous, aucun répertoire ne devrait s'appeler ainsi, tout répertoire devant avoir un nom significatif et descriptif de son contenu). Supposons qu'on s'intéresse à une règle pour savoir si sa commande doit être exécutée ou non. `make` applique l'algorithme suivant :

```
Si la cible est le nom d'un fichier appartenant au répertoire rep,
    Alors
        Si la règle comporte des prérequis,
            Alors
                make compare les dates de dernière modification de la cible et des prérequis.
                Si au moins un des prérequis est plus jeune que la cible,
                 cela signifie que, depuis la dernière mise à jour de la cible,
                 les prérequis ont été modifiés.
                    Alors
```

```
                    la commande est exécutée pour remettre à jour la cible.
            Sinon (si la règle ne comporte aucun prérequis)
                la cible est considérée inconditionnellement
                 à jour et la commande associée n'est JAMAIS exécutée
                 (En général on cherche rarement à tomber dans ce cas;
                 on y tombe souvent par erreur.)
    Sinon (si la cible ne correspond à aucun nom de fichier appartenant au répertoire rep,
        Si la règle comporte des prérequis
            Alors
                make interprète tout de même la cible
                 comme un fichier qui n'existe pas encore ou qui a été effacé
                 et qu'il faut remettre à jour.
                La commande associée à la règle est exécutée.
            Sinon (si la règle ne comporte aucun prérequis)
                make interprète la cible comme phony, et donc
                 exécute inconditionnellement la commande de la règle
```

On peut se demander ce qui pourrait pousser un programmeur de makefile à écrire une règle pour construire un fichier dans laquelle il ne mettrait aucun prérequis. En réalité, le programmeur ne pensait sans doute pas à faire une telle règle. Il voulait sans doute faire un *phony*, mais le nom de cette cible s'est révélé être celui d'un vrai fichier. Par exemple, supposons que vous ayez un makefile comme celui de notre exemple (programme `article.ps`). Le *phony* « bang » ne fonctionne que s'il n'existe aucun fichier du nom de « bang ». À titre d'expérience, si on crée un fichier du nom de « bang », alors le fait de taper

```
make bang
```

provoque le message suivant :

```
make: 'bang' is up to date.
```

ce qui signifie clairement que make considère que par « bang » nous avons voulu désigner le fichier et non l'action à effectuer. Si cela vous empêche d'écrire « coucou » en grosses lettres, le problème n'est pas tellement affolant. Par contre, supposez qu'au lieu d'un *phony* « bang » vous ayez utilisé un *phony* « sauvegarde » qui enregistre tous les fichiers de votre répertoire (ou même de votre compte) dans un seul fichier compressé, dans le but de le graver sur un CD. C'est souvent ce qu'on fait par précaution avant de réinstaller un système, de modifier les partitions, etc. Imaginez qu'après une tentative infructueuse ou quelque chose de ce genre, vous ayez justement dans votre répertoire un fichier vide du nom de « sauvegarde ». Dans ce cas, lorsque vous taperez

```
make sauvegarde
```

rien ne sera sauvegardé. On aura simplement le message « *sauvegarde is up to date* », ce qui est souvent interprété, à tort, comme « la sauvegarde a été effectuée ». Pour éviter ce genre de problème, on peut utiliser ce qu'on appelle une *directive* du makefile (nous verrons les directives plus loin). La directive `.PHONY` indique explicitement à `make` quelles cibles sont *phonys*. Nous changeons notre makefile :

```
 # Ceci est un exemple de makefile

coucou.dvi : coucou.tex
        latex coucou.tex
article.ps : coucou.dvi figure.eepic
        dvips coucou.dvi -o article.ps
.PHONY : bang sauvegarde
# cette directive indique a make que bang et sauvegarde ne doivent pas etre
# consideres comme des noms de fichiers, meme si ces fichiers existent
bang :
        banner coucou
sauvegarde :
        tar -cvf backup.tar *
        gzip backup.tar
```

NB : `.PHONY` n'est pas une règle, mais une directive. Donc le fait d'avoir ou non dans votre répertoire un fichier du nom de « `.PHONY` » ne change rien au comportement de cette directive.

Exercice 1

Soit un programme constitué d'un fichier `main.c`, des fichiers sources `liste.c, vecteur.c, matrice.c` et des fichiers entêtes correspondants : `liste.h, vecteur.h` et `matrice.h.` Pour ce TP, vous pouvez très bien prendre des fichiers vides, sauf pour main.c qui doit comporter une fonction main. Écrire un makefile, avec une seule règle, capable de compiler ce programme. Il doit y avoir compilation si et seulement si :

- l'exécutable n'existe pas ;
- l'exécutable est plus vieux que l'un des fichiers `main.c`, `liste.c`, `vecteur.c`, `matrice.c`, `liste.h`, `vecteur.h` ou `matrice.h`.

On peut changer la date de dernière modification d'un fichier avec la commande `touch.` Donc si, après avoir reconstruit l'exécutable, on tape `make,` aucune compilation ne doit être effectuée. Par contre, si on tape `touch vecteur.h` alors la prochaine commande `make` devra relancer une compilation.

Exercice 2

Reprenons les programmes de l'exercice 1. Dans l'exercice 1, à chaque fois qu'un seul des fichiers sources ou entêtes était modifié, tous les fichiers sources étaient recompilés. Nous voulons que si seul un sous-ensemble des fichiers sources a été modifié, alors ce seul sous-ensemble soit recompilé. Ceci suppose qu'on puisse garder les fichiers intermédiaires `vecteur.o, liste.o, matrice.o` et `main.o` pour les fichiers qui n'ont pas été modifiés. Ensuite, une fois que tous les fichiers objets auront été mis à jour, on procédera à l'édition de liens pour créer l'exécutable. Écrivez un makefile avec plusieurs règles où une telle compilation séparée est réalisée lorsqu'on tape simplement `make`.

1. Vérifiez que lorsqu'on tape `touch vecteur.h` ou `touch vecteur.c`, puis `make`, seul le fichier `vecteur.c` est recompilé.
2. On suppose que le fichier `matrice.h` contient les directives : `#include "vecteur.h"` et `#include "liste.h"`. Est-ce qu'en tapant `touch liste.h` et/ou `touch vecteur.h`, puis en tapant `make` le fichier `matrice.c` est recompilé comme il devrait l'être ?

Exercice 3

- Ajoutez au makefile de l'exercice précédent une règle qui vous permet d'éliminer tous les fichiers qui ne vous intéressent pas (les fichiers objets, les fichiers *core*, les *backups* (les fichiers dont le nom se termine par « ~ » ou « .bak »), etc.
- Ajoutez à ce makefile une règle qui, en plus d'éliminer les fichiers qui ne vous intéressent pas, élimine aussi l'exécutable. Utiliser la règle précédente. Que se passe-t-il lorsqu'on essaie d'éliminer les fichiers objet, core, backups, *puis* l'exécutable et que, dans le répertoire courant, il n'y a pas de fichier objet, pas de fichier core *ou* pas de fichier backup ? Est-ce que la commande d'élimination de l'exécutable est lancée ?

Indication : le fait qu'une commande lancée par make retourne avec un statut non nul (erreur) peut entraîner un arrêt de make pour cause d'erreur. Si nous voulons que les erreurs engendrées par une commande soient ignorées, il faut faire précéder la commande du caractère « - » (moins).

4 Les variables

Une variable est une chaîne de caractères qui peut désigner des listes de noms de fichiers, des options à passer à un compilateur ou à une autre commande, des noms de programmes à exécuter, des listes de répertoires où chercher d'autres fichiers, où écrire d'autres fichiers, etc. Cette chaîne de caractères peut contenir n'importe quels caractères à l'exception de « = », « : », « # » ou des espaces (« », « \t », « \n », etc). Ceci dit, il est recommandé d'éviter tous les caractères autres que les lettres, les nombres et les « _ » car les autres caractères pourront être utilisés à l'avenir comme des caractères ayant une signification. D'autres part dans certains environnements, ces autres caractères peuvent déjà avoir une signification particulière.

Il y a plusieurs types de variables pour make :

- les variables passées par commande ;

– les variables d'environnement ;
– les variables automatiques ;
– les variables définies dans le makefile.

4.1 Les variables passées par commande

Quelque soit le contenu d'un makefile, on peut définir une variable dans l'appel à `make` qui lit ce fichier makefile. Par exemple si on tape `make TOTO=objet.c` alors, dans le fichier makefile, toutes les occurrences de `$(TOTO)` ou de `${TOTO}` seront remplacées par `objet.c`. De manière générale, dans toute la suite, pour référencer les variables, on pourra utiliser les parenthèses et les accolades de façon interchangeable.

4.2 Les variables d'environnement

L'environnement shell est caractérisé par un certain nombre de variables dont on peut consulter la valeur soit en invoquant la commande `env` qui liste toutes les variables d'environnement et leurs valeurs, soit en utilisant la commande `echo` : `echo` $(VARIABLE) renvoie la valeur de la variable d'environnement VARIABLE. Par exemple les variables USER, HOSTTYPE ou OSTYPE sont des variables du shell. Si la commande make est lancée par un shell donné, (bash, sh, csh, ksh, tcsh ou autre) toutes les variables d'environnement de ce shell peuvent être utilisées dans le makefile. Autrement dit, toutes les occurrences de $(USER), $(OSTYPE), etc., dans le makefile, seront remplacées par la valeur qu'elles avaient dans le shell. La seule exception est la variable SHELL qui, par défaut, vaut toujours `/bin/sh`, mais qui peut être modifiée dans le makefile. La raison de cette exception est d'assurer qu'un makefile, appelé par une même ligne commande, ne puisse pas avoir deux comportements différents selon que celui qui lance la commande préfère ksh à bash ou à tout autre shell.

Exercice 4

– Écrivez un makefile avec une seule règle *phony*, qui ne fait qu'afficher à l'écran la valeur des variables USER, OSTYPE et SHELL.
– Appelez ce makefile en spécifiant, dans la ligne de commande, des valeurs différentes pour ces variables. Par exemple `make USER=toto`.
– Quelle est alors la valeur de la variable USER pour make ? et pour le shell lui-même ?
– Que peut-on en déduire sur la préséance de la définition des variables par commande et par héritage de l'environnement ?

4.3 Les variables automatiques

Ici, le terme automatique n'a pas du tout le même sens que dans le langage C. Dans les makefiles, il existe des variables que vous n'avez pas besoin de mettre à jour. Elles sont définies et remises à jour par make. Elles ne peuvent être utilisées que dans une règle et désignent une des cibles, les prérequis, un sous-ensemble de prérequis, etc.

$@ : Dans toutes les règles, toutes les occurrences de $@ sont remplacées par le nom du fichier cible. S'il y a plusieurs noms dans la cible, alors $@ est remplacé par celui des noms dans la cible qui a déclenché l'application de la règle.

$^ : Dans toutes les règles, toutes les occurrences de $^ sont remplacées par la liste des prérequis, chaque nom étant séparé par une espace. Si un certain prérequis a été répété plusieurs fois, alors il n'apparaît qu'une seule fois dans $^. Si on veut que l'ordre des éléments et le nombre des répétitions soient préservés, il faudra utiliser $+ au lieu de $^.

$< : Désigne le premier prérequis.

$? : Dans toutes les règles, toutes les occurrences de $? sont remplacées par la liste des prérequis qui sont plus jeunes qu'au moins une des cibles. En fait $? désigne la liste des prérequis qui sont responsable de l'application de la commande de la règle.

Il existe d'autres variables automatiques ($%, $+, $*, etc.). Pour plus de détails, consulter `http://www.gnu.org/manual/make/html_mono/make.html` .

Exercice 5

Construire une règle comportant plusieurs cibles et plusieurs prérequis. On pourra par exemple utiliser le programme des exercices 2 et 3, et en mettant les fichiers objets en cible et les fichiers sources en prérequis. Il s'agit simplement d'une expérience, car la règle en question serait applicable à chaque fois qu'un fichier source quelconque est plus jeune qu'un fichier objet quelconque, sans nécessairement que le fichier objet corresponde au fichier source (absurde). La commande associée à cette règle affichera à l'écran la valeur des différentes variables décrites ci-dessus. En particulier vérifier le comportement de la variable $? lorsque tous les fichiers sont à jour, puis quand vous modifiez un des fichiers sources. Est-ce qu'il est possible de donner une autre valeur à ces variables, par exemple en tapant « `make @=toto` » ?

Nous verrons plus loin comment utiliser ces variables de façon plus intéressante.

4.4 Les variables définies dans le makefile

Lorsqu'on écrit :

```
var=valeur1 valeur2 valeur3 ... valeurn
```

on définit une variable `var`. Dans tout le fichier makefile (que ce soit avant ou après cette définition) toutes les occurrences de `$(var)` ou de `$var` seront remplacées par `valeur1 valeur2 ... valeurn`. Cette opération est appelée une **substitution** (en anglais *expansion*) de la variable `var`. Plus précisément, lorsque `$(var)` ou `$var` doit être évalué, make parcourt le fichier makefile à la recherche des définitions de `var`. Il ne devrait y avoir qu'une seule définition pour une variable donnée. Mais s'il y en a néanmoins plusieurs, c'est la *dernière* définition dans le fichier qui prime sur les autres. Si la définition d'une telle variable `var` contient la référence à d'autres variables, alors ces autres variables sont substituées avec le même algorithme (en cherchant la définition de cette autre variable en commençant par le début du fichier et en ne gardant que la dernière définition s'il y en a plusieurs). Par exemple :

```
TITI = $(TUTU)
TOTO = $(TITI)
TUTU = coucou
all:
    echo $(TOTO)
TUTU=NonNon!!!
```

Lorsque make tente d'exécuter « `echo $(TOTO)` », il cherche à remplacer `$(TOTO)` par sa valeur. Pour cela il parcourt tout le fichier à la recherche des définitions de `TOTO` et attribue à `TOTO` la valeur `$(TITI)`. Ensuite, il parcourt le fichier à la recherche de définitions de `TITI` et, lorsqu'il trouve la première définition, attribue à `TITI` la valeur `$(TUTU)`. Enfin, il parcourt le fichier à la recherche de définitions de `TUTU` et, lorsqu'il trouve la dernière définition du makefile, attribue à `TUTU` la valeur « `NonNon!!!` ». On remarquera que la définition `TUTU=coucou` est totalement ignorée puisqu'elle se trouve avant la définition `TUTU=NonNon!!!`. Ainsi le fait de taper `make` donnera le résultat :

```
echo NonNon!!!  NonNon!!!
```

Ce type de substitution est appelée *substitution récursive* (par opposition à *substitution simple*). Dans les autres versions de make (autres que GNU make) c'est le seul type de variable que l'on puisse utiliser. Donc pour des raisons de portabilité, il est conseillé d'utiliser les variables à substitutions récursives. Le grand intérêt de ce type de variable est qu'il n'est pas nécessaire que l'endroit (dans le fichier makefile) où la variable est définie se trouve avant l'endroit où elle est utilisée. Par contre, un inconvénient est qu'on ne peut pas ajouter une variable à une liste déjà existante. En particulier une définition comme :

```
PATH = $(PATH) /usr/sbin
```

entraînerait, en principe, une boucle infinie et, en pratique, provoque une erreur. C'est la raison pour laquelle GNU make permet d'utiliser les variables à substitution simple. Pour plus de détails sur les différents types de variables et de substitutions, cf. `http://www.gnu.org/manual/make/html_mono/make.html` .

4.4.1 Ajouter une chaîne de caractères à une variable

Pour pouvoir ajouter quelque chose à une variable (à substitution récursive ou simple), on utilise l'opérateur binaire +=.

```
variable = valeur
variable += davantage
```

est équivalent à

```
temp = valeur
variable = $(temp) davantage
```

sauf qu'aucune variable `temp` n'est définie.

4.4.2 La substitution est faite *verbatim*

Les variables peuvent être utilisées dans n'importe quel contexte, cible, prérequis, commande, directive, nom de nouvelle variable, etc. D'autre part les variables sont substituées *verbatim*. Par exemple le makefile suivant :

```
lettre_c = c
lettre_g = g
un_point = .

prog.o : prog$(un_point)$(lettre_c)
        $(lettre_g)$(lettre_c)$(lettre_c) - $(lettre_c) prog$(un_point)$(lettre_c)
```

est l'exact équivalent de :

```
prog.o : prog.c
        gcc -c prog.c
```

On peut même remplacer la définition

```
lettre_g = g
```

par

```
debut = lett
fin   = re_g
$(debut)$(fin) = g
```

Ce qui précède ne se veut en aucun cas un exemple de makefile lisible. Il est même fortement recommandé d'écrire les makefiles autrement. Le seul mérite de cet exemple est de montrer comment les variables peuvent fonctionner.

Exercice 6

Un des multiples intérêts des variables est de limiter la répétition d'une chaîne de caractères, ce qui est non seulement fastidieux, mais peut entraîner des erreurs. En effet, lors d'un changement, on oublie souvent de corriger une des occurrences. C'est pourquoi, le plus possible, il faut tenter d'écrire chaque chose en un seul exemplaire. Ainsi, on peut définir une variable dont la valeur est la chaîne de caractères à répéter. Ensuite, au lieu de répéter la chaîne de caractères, on répète la variable. Par exemple dans le programme de l'exercice 1, le nom de l'exécutable et celui des fichiers sources est répété au moins à deux reprises. Est-ce qu'il ne serait pas possible d'utiliser une variable pour que ces deux objets n'apparaissent qu'une seule fois ?

Exercice 7

Un autre des intérêts des variables est de permettre la communication avec make. Certaines variables ont une signification particulière pour make. Nous verrons cela plus en détail avec les règles implicites. Intéressons-nous à la variable `SHELL` qui définit avec quel interpréteur les commandes seront exécutées. Comme nous l'avons mentionné plus haut, cette variable d'environnement, contrairement aux autres variables du même type, n'est pas héritée de l'environnement et vaut « `/bin/sh` » par défaut. Définissez cette variable dans le makefile et donnez-lui une autre valeur. Que se passe-t-il lorsqu'on définit cette variable `SHELL` par commande et qu'on lui donne encore une autre valeur. Quelle est la valeur qui prévaut ? La valeur fournie par le makefile ou celle de la commande ? Même expérience avec une autre variable comme par exemple `USER`.

4.4.3 Un peu de LaTeX

Manipulons des fichiers de format LaTeX. Ce sont des fichiers comportant des caractères Ascii, mais qui peuvent coder, beaucoup d'autres caractères spéciaux en français, en espagnol, en allemand, etc., des images, des expressions mathématiques, etc. Ce genre de format s'oppose à celui des outils de type *Wysiwyg (what you see is what you get)* comme *Word* qui sont plus conviviaux. Un des avantages de LaTeX par rapport à *Word* est qu'il peut fonctionner sous n'importe quel environnement puisque les fichiers LaTeX sont des fichiers Ascii.

`http://icps.u-strasbg.fr/~vivien/Enseignement/PPP-2001-2002/planning.tex` est un fichier au format LaTeX. Pour pouvoir examiner l'aspect du document final, on peut transformer ce fichier LaTeX en un fichier au format pdf (portable document format) qu'on peut visualiser sur toutes les plates-formes. La transformation du fichier `planning.tex` se fait par la commande : « `pdflatex planning.tex` » ce qui produit un fichier `planning.pdf` qu'on peut lire avec l'outil `ghostview` (ou `gv` ou `GSView`). Pour l'impression papier, il faudrait plutôt transformer le fichier LaTeX en un fichier au format PostScript. La commande : `latex planning.tex` transforme le fichier planning.tex en un fichier planning.dvi qu'on ne peut pas éditer, mais qui donne l'aspect final du document. Un fichier dvi peut être lu par la commande : « `xdvi planning.dvi` » Enfin un fichier dvi peut être transformé en un fichier PostScript par la commande « `dvips planning.dvi -o` » qui a pour effet de construire le fichier `planning.ps` qu'on peut également regarder avec l'outil `ghostview` (ou `gv` ou `GSView`).

Exercice 8

Essayer ces commandes sur le fichier `planning.tex`, au départ sans passer par un makefile. Ensuite écrire un makefile qui puisse, au choix, construire un fichier dvi, un fichier ps ou un fichier pdf. Le fichier pdf dépend des fichiers tex et des fichiers représentant les figures (ici `Figure.eepic`). Le fichier dvi dépend du fichier tex. Le fichier ps dépend du fichier .dvi et du fichier représentant la figure.

Écrire une règle qui permette également d'éliminer les fichiers log, aux, dvi et autres backup dont le nom se termine par bak ou par « ~ ». Écrire une règle qui permette en plus d'éliminer les fichiers ps et pdf. Enfin une règle qui permette d'imprimer le fichier seulement si le fichier PostScript est à jour.

À présent, le makefile doit certainement être constellé de fichiers `planning.*`. Si l'on veut changer le nom du fichier en autre chose, ou d'utiliser le makefile pour un autre fichier LaTeX, il faut changer beaucoup d'instances de fichier. Donc on demande que le nom « `planning` » n'apparaisse qu'une seule fois dans le makefile. Pour cela on utilisera une variable `NOM` à qui on donnera la valeur planning. À partir de cette variable, on demande de construire le nom des fichiers LaTeX, dvi, pdf, PostScript, etc. et de récrire le makefile en fonction de cela.

Comment pourrait-on utiliser ce même makefile (sans aucune modification) pour appliquer les mêmes traitements à un fichier `specifications.tex` ?

5 Les makes récursifs

Soit une arborescence de fichiers. Dans un premier temps nous prendrons une arborescence limitée à trois niveaux, mais le principe peut se généraliser à un nombre quelconque de niveaux, à tout un compte, ou à toute une machine. On se place dans un répertoire `rep` et on étudie les sous-répertoires qui se trouvent

dans ce répertoire. Appelons les `subrep_1, subrep_2, subrep_3, ..., subrep_n` . Chacun de ces sous-répertoires contient un makefile qui permet de construire un certain nombre de fichiers qui permettent de faire une sauvegarde, de nettoyer le répertoire, etc., etc. Pour utiliser les makefiles dans chaque répertoire on peut aller dans chacun de ces répertoires (`cd subrep_i`) et taper les commandes make adéquats : par exemple `make arg_1 arg_2 arg_3 ... arg_N` (quelques soient les arguments `arg_i`). Mais il est également possible de taper ces commandes depuis le répertoire `rep` en tapant :

```
make -C subrep_i arg_1 arg_2 arg_3 ... arg_N
```

Lors de l'appel récursif de make, la commande « make » utilisée est celle définie *par défaut* sur le système, qui n'est pas forcément celle que vous avez invoquée. Pour contourner ce problème, plutôt que d'utiliser la commande `make` on aura recours à la variable `MAKE` qui a pour valeur le make invoqué sur le makefile :

```
${MAKE} -C subrep_i arg_1 arg_2 arg_3 ... arg_N
```

Exercice 9

On suppose que dans le répertoire `projet3`, nous avons d'une part un sous-répertoire `simulation`, un sous-répertoire `visualisation,` un sous-répertoire `rapport` et un autre sous-répertoire `planning.` Les deux premiers répertoires contiennent les fichiers source de programmes écrits en langage C. Le troisième répertoire contient la documentation que vous avez écrite (au format LaTeX) au sujet de ces programmes et enfin le quatrième répertoire contient votre planning également écrit en LaTeX. On suppose que chacun de ces sous-répertoires possède un makefile. Pour `rapport` et `planning`, les makefiles permettent de construire des documents lisibles ou imprimables à partir de fichiers LaTeX, de les imprimer, ou de faire le ménage (comme dans l'exercice 8). Construire cette arborescence dans votre compte, en prenant pour programme de simulation et de visualisation les fichiers vides utilisés dans les premiers exercices. Pour les fichiers LaTeX, prendre les fichiers de l'exercice 8. Utiliser les makefiles que vous avez vous-mêmes écrits.

En utilisant une commande shell (sans encore écrire un makefile) nettoyer tous les sous-répertoires, non pas en allant dans chacun d'eux, mais en restant dans le répertoire `projet3`. De même, toujours avec une commande shell, compiler tous les programmes, préparer tous les documents pour l'impression, faire un fichier archive compressé de tout ce que contient le répertoire `projet3` en vue d'une sauvegarde.

À présent, écrire un makefile pour le répertoire `projet3` qui permet d'effectuer ces opérations de façon automatique (nettoyage global, compilation globale, sauvegarde globale, transformation globale de tous les fichiers LaTeX en pdf ou en ps, etc.)

6 Les motifs (*wildcards)* et les règles à motifs (*pattern rules*)

6.1 Les *wildcards*

Je pense que vous devez tous connaître les *wildcards* pour les avoir utilisées plus d'une fois dans une commande shell. Les *wildcards* que nous utiliserons sont : « `*` », « `?` » et `[..]`. Utilisons-les dans des commandes shell :

```
ls af_*g*.c
```

affiche à l'écran tous les fichiers ou sous-répertoires du répertoire courant dont le nom commence par « `af_` », se termine par « `.c` », et contient par ailleurs une occurrence de la lettre « `g` ». Le caractère « * » représente une chaîne de caractères quelconque, y compris la chaîne vide. D'autre part la commande shell :

```
ls toto.?
```

affiche à l'écran tous les fichiers ou sous-répertoires du répertoire courant dont le nom commence par le préfixe « toto. » suivi d'un *unique* caractère. « ? » représente un unique caractère. De même

```
ls ??c*
```

affiche à l'écran tous les fichiers ou sous-répertoires dont le troisième caractère du nom est un « c ». Enfin la commande shell :

```
ls Version_[a-f]
```

affiche à l'écran tous les fichiers ou sous-répertoires dont le nom commence par « `Version_` » et dont la dernière lettre est une des lettres parmi `a`, `b`, `c`, `d`, `e`,et `f`. Autre exemple :

```
ls [0-5][0-9]_NavierStokes 6[0-4]_NavierStokes
```

affiche à l'écran tous les fichiers ou sous-répertoires dont le nom se termine par « `_NavierStokes` » précédé d'un nombre à deux chiffres compris entre 00 et 64. Plus précisément, « `ls [0-5][0-9]_NavierStokes` » concerne les fichiers ou sous-répertoires `00_NavierStokes`, ..., `59_NavierStokes` alors que l'expression « `ls 6[0-4]_NavierStokes` » concerne les fichiers ou répertoires `60_NavierStokes`, ..., `64_NavierStokes`.

Évidemment, ces *wildcards* peuvent être utilisées dans les commandes shells de toutes les règles, puisque ces commandes sont envoyées à un shell. Mais ces *wildcards* peuvent également être utilisées dans les cibles et les prérequis (qui, eux, ne sont pas envoyés à un shell). Lorsque make rencontre une wildcard, (dans une cible, dans une commande ou dans les prérequis d'une règle) il la remplace par la liste des noms (séparés par des espaces) de tous les fichiers et sous-répertoires du répertoire courant qui correspondent à la chaîne de caractères dans laquelle se trouve le wildcard.

Exercice 10

Prenons le code décrit dans l'exercice 1. Il faut demander à make de recompiler le programme lorsqu'un seul des fichiers sources a été modifié. Supposons que parmi les fichiers sources qui se trouvent dans le répertoire courant, il n'y a pas de déchet. Autrement dit, tous les fichiers source participent à la construction de l'exécutable. Est-ce qu'il serait possible de ne pas lister le nom de tous les fichiers source dans la règle de compilation ? Écrire le makefile correspondant.

Exercice 11

À présent, nous voudrions faire la même chose que l'exercice 2, c'est-à-dire passer par les fichiers objets. Par contre, nous voudrions, si possible, ne pas avoir à écrire une règle pour la construction de chaque fichier objet. Que pensez-vous du makefile suivant :

```
EXECUTABLE=scanline
$(EXECUTABLE) : *.o
    gcc -o $(EXECUTABLE) *.o
*.o : *.c
    gcc -c *.c
```

En particulier que se passe-t-il lorsqu'un seul des fichiers source est modifié ? (Pour vous en persuader, regardez les dates de dernière modification des fichiers objets). Que se passe-t-il quand on remplace « `gcc -c *.c` » par « `gcc -c $?` » ?

6.2 Règles à motifs (*Pattern rules*)

Les *pattern rules* servent notamment à résoudre le type de problème que nous avons rencontré à l'exercice 11. Par exemple :

```
%.toto : %.titi
    cp $^ $@
```

est une *pattern rule*. Cette règle permet de construire un fichier avec un extension « toto » à partir d'un autre fichier ayant le même nom, mais avec une extension « titi ». Par exemple, cette règle permet de construire un fichier `soleil.toto` à partir du fichier `soleil.titi`. La même règle permet de construire `pierrot.toto` à partir de `pierrot.titi`. En l'occurrence, la règle de construction est simple : on copie le contenu du fichier `nom.titi` dans le fichier`nom.toto`. (Nous avons vu au paragraphe 4.3 que `$@` représentait la cible et que `$^` représentait l'ensemble des prérequis.)

Exercice 12

Utiliser une *pattern rule* pour régler le problème rencontré à l'exercice 11. Nous voulons que, lorsqu'un sous-ensemble des fichiers sources est modifié, seul les fichiers de ce sous-ensemble soient recompilés, et qu'ils soient compilés une seule fois.

À votre avis, pourquoi est-ce que le fait de remplacer froidement la règle proposée à l'exercice 11 (`*.o : *.c ; gcc -c *.c`) par la règle proposée ci-dessus (`%.o : %.c ; gcc -c $^`) ne permet pas d'obtenir un makefile qui fonctionne correctement ?

Que se passe-t-il si on remplace les occurrences de « `*.o` » par une variable `$(OBJETS)` représentant le nom de tous les fichiers objets `(Liste.o Vecteur.o Matrice.o main.o)` ?

7 Les directives

Nous nous intéresserons très brièvement aux directives `include`, `define` et `export`, et plus longuement aux directives conditionnelles `ifeq`, `ifneq`, `ifdef` et `ifndef`.

Il existe d'autres directives comme `override`, `.IGNORE`, `.PRECIOUS`, `.DEFAULT`, `.SILENT`, `.PHONY` (vue plus haut), etc. Pour plus de détails sur ces directives, cf. `http://www.gnu.org/manual/make/html_mono/make.html`.

7.1 Quelques directives non conditionnelles

7.1.1 La directive `include`

Lorsqu'un makefile contient une directive du type

```
include <nom_fichiers>
```

make, en lisant cette directive, cesse temporairement de lire le makefile auquel appartient cette directive et lit, dans l'ordre, le ou les fichiers dont le nom appartient à la liste `<nom_fichiers>`. Lorsque cette lecture est finie, `make` recommence la lecture du makefile d'origine là où il l'avait laissée, c'est-à-dire à partir de la directive `include`. Cette liste d'arguments `<nom_fichiers>` peut contenir des motifs, des noms de variables, etc. Par exemple

```
include toto titi *.mk $(MKS)
```

En l'occurrence, tout se passe comme si `make` avait remplacé la directive `include` par le contenu du fichier `toto`, suivi de celui du fichier `titi`, de celui de tous les fichiers dont le nom se terminent par « `.mk` » et par les fichiers dont le nom est contenu dans la valeur de la variable `MKS`. Ceci peut être intéressant quand plusieurs makefiles doivent avoir un ensemble de variables ou de règles en commun. Dans ce cas, toutes ces variables et règles peuvent être écrites dans un seul fichier et inclus par tous les makefiles (un peu comme les fichiers entêtes). Autre intérêt : certaines commandes permettent de générer des dépendances automatiquement (cf. `gcc -M`) et écrivent ces dépendances dans un fichier. Une directive `include` permet d'inclure ces dépendances dans le makefile.

De même que pour les commandes shell, le fait d'écrire « `-include ...` » au lieu de « `include ...` » permet de ne pas arrêter make pour cause d'erreur, par exemple au cas où les fichiers à inclure n'existeraient pas.

7.1.2 La directive `define`

La directive `define` est une autre manière de définir des variables. L'intérêt de la directive `define` par rapport à la définition classique de variables (section 4) est que l'on peut inclure des retours à la ligne (« \n ») dans la variable. Ceci qui permet de définir des variables qui représentent des séquences de commandes. Voici la syntaxe de cette directive :

```
define <nom_variable>
...
...
endef
```

La directive define doit être suivie du nom de la variable (et de rien d'autre) écrit sur la même ligne. La valeur de la variable est constituée de toutes les lignes qui se trouvent entre la ligne qui contient la directive `define` et la ligne qui contient `endef`. Par exemple :

```
define sauvegarde
tar -cvf ../backup.tar *
gzip ../backup.tar
echo 'termine'
endef
```

Ensuite, pour faire référence à ces commandes, on procède comme pour les variables. Toute occurrence de `$(sauvegarde)` sera automatiquement remplacée par les trois commandes ci-dessus.

7.1.3 Les directives `export` et `unexport`

Lorsqu'on utilise des makes récursifs (des sous-makes) il est important de savoir si ces makes pourront hériter de la valeur de toutes les variables qui étaient définies dans le make global. *A priori*, il n'y a aucun héritage sauf pour les variables d'environnement. Pour forcer l'héritage dans le cas d'autres variables, on peut utiliser la directive `export`. Lorsque l'on écrit :

```
export var
```

la variable `var` sera définie dans tous les sous-makes qui pourront être lancés à partir du makefile auquel appartient la directive. De plus la valeur de `var` sera la même que dans le makefile d'origine au moment où le sous-make a été lancé. Si l'on veut exporter toutes les variables, on peut simplement écrire :

```
export
```

Lorsqu'on veut exporter toutes les variables *sauf* certaines d'entre elles, on utilise la directive `unexport` qui empêche l'héritage de variables. Par exemple si l'on veut que toutes les variables soient héritées sauf la variable `var`, on écrit :

```
export
unexport var
```

7.2 Les directives conditionnelles

Ces *directives* ne sont pas à confondre avec la *fonction* `if` qui sera vue à la section suivante (8). Dans l'ensemble, ces directives permettent d'ignorer toute une partie d'un makefile dans certaines conditions et, éventuellement, d'ignorer d'autres parties si ces conditions ne sont pas réunies. (Donc il ne s'agit pas d'exécuter certaines commandes pendant l'exécution.) Ceci permet souvent d'avoir des programmes qui sont plus portables et qui peuvent s'adapter à un plus grand nombre de situations.

7.2.1 Les directives `ifeq` et `ifneq`

Les deux directives ont la même la syntaxe :

```
ifeq (argument1,argument2)
texte1
else
texte2
endif
```

Toutes les variables dans les arguments `argument1` et `argument2` sont substituées. Si les deux arguments sont égaux, la conditionnelle est équivalente à `texte1` (`texte2` n'est pas pris en compte), et s'ils sont différents, la conditionnelle est équivalente à `texte2`. La directive `ifneq` a le même comportement, mais en intervertissant `texte1` et `texte2`.

Détail important : `ifeq, ifneq, ifdef` et `ifndef` doivent être suivis d'une espace (précédant les arguments).

Exercice 13

Écrivez un makefile pour pouvoir compiler sur `ada` comme sur les `distas` le programme `imscan` du TP 2 (`http://icps.u-strasbg.fr/~vivien/Enseignement/PPP-2001-2002/sources-exemple.tgz`) sachant que sur `ada` les librairies GMP et polylibgmp sont installées respectivement dans les répertoires `/users/prof/vivien/IciEstLaGmp` et `/users/prof/vivien/LaPolylib`, et sur les `distas` dans les répertoires `/users/prof/vivien/LinuxGMP` et `/users/prof/vivien/LinuxPolylib`. Le makefile devra déterminer de lui-même sous quel *environnement* il se trouve et qu'il en déduise la commande de compilation adéquate.

7.2.2 Les directives `ifdef` et `ifndef`

Il arrive souvent que l'on veuille utiliser la valeur d'une variable d'environnement ou d'une variable définie dans la ligne de commande, sans être certain à l'avance, que cette variable a bien été initialisée. On peut prendre des décisions différentes selon que la variable a été initialisée ou non. Le fait qu'une variable attendue n'ait pas été initialisée peut signifier une erreur. On peut réagir en initialisant la variable dans le makefile, ou en lançant d'autres commandes. Mais souvent il ne s'agit pas d'une erreur, mais simplement d'un fait qui nous informe des événements qui ont eu lieu auparavant.

Exercice 14

Il arrive que la variable d'environnement USER ou la variable PWD ne soit pas héritée de l'environnement par make. Sur `steed` c'est USER et sur les `distas` c'est PWD. Nous voulons que notre makefile puisse fonctionner dans toutes les situations. Donc avant d'utiliser USER ou PWD, nous faisons un test pour savoir si ces variables ont été définies. Si elles n'ont pas été définies, vous les définirez en conséquence. Pour cela, on pourra utiliser la fonction `shell` (cf. 8.1) de makefile.

8 Les fonctions de traitement des chaînes de caractères

Cette section présente un certain nombre de fonctions. Ici on n'étudiera pas l'écriture de nouvelles fonctions, mais l'usage de celles qui ont déjà été définies.

La syntaxe générale d'appel d'une fonction dans un makefile est la suivante :

```
$(fonction argument1, argument2, ..., argumentN)
```

le nom de la fonction est séparé des arguments par au moins une espace (plus d'éventuelles tabulations, etc). S'il y a plusieurs arguments, ils sont séparés par des virgules.

Il existe de nombreuses autres fonctions que celles décrites dans cette section et il est même possible de créer des fonctions. Pour plus de détails sur les différents types de fonctions, cf. `http://www.gnu.org/manual/make/html_mono/make.html` .

8.1 La fonction `shell`

Cette fonction peut créer des chaînes de caractères résultant de l'exécution d'une commande shell. Si une commande shell, comme `ps`, produit un résultat sur plusieurs lignes, la fonction `shell` fournit un résultat sur une seule ligne. Par exemple, la commande `ps` renvoie :

```
 PID TTY TIME CMD
 1234 pts/0 00:00:01 bash
10999 pts/0 00:00:03 emacs
11219 pts/0 00:00:00 make
11220 pts/0 00:00:00 ps
```

et dans un makefile la ligne :

```
$(shell ps)
```

a pour valeur :

```
PID TTY TIME CMD 1234 pts/0 00:00:01 bash 10999 pts/0 00:00:03 emacs 11219 pts/0 00:00:00
make 11220 pts/0 00:00:00 ps
```

c'est-à-dire la même chose, mais où les mots ont été mis bout à bout sur une seule ligne, séparés chacun par une espace.

Exercice 15

Définir une variable `SOURCES` qui contient le nom de tous les fichiers du répertoire courant qui se terminent par `*.c`.

8.2 La fonction `subst`

La fonction `subst` permet de remplacer une chaîne de caractères par une autre dans une liste de noms. Par exemple

```
$(subst ien,ienG, Tiens bien, il ne reste plus rien)
```

vaut

```
TienGs bienG, il ne reste plus rienG
```

8.3 La fonction `patsubst` et son raccourci

La fonction `patsubst` permet de reconnaître des motifs dans les mots d'une liste et de les remplacer par d'autres motifs. Le premier argument de `patsubst` peut contenir un caractère %. `patsubst` parcourt le troisième argument en cherchant les éléments qui vérifient le motif décrit par ce premier élément. Par exemple « const.% » représente tous les mots qui commencent par « const. » et qui se terminent par une chaîne quelconque X. Si le deuxième élément comprend aussi un caractère « % », ce « % » sera remplacé par la chaîne X.

```
$(patsubst %ien, %ienG, Tiens bien il ne reste plus rien)
```

donne toujours

```
Tiens bienG il ne reste plus rienG
```

Par contre, si un répertoire est composé d'autres répertoires, nommés `Version.01`, `Version.02`, `Version.03`, ..., `Version.N`, on ne peut pas utiliser la fonction `subst` pour les remplacer par `01_Version`, `02_Version`, `03_Version`, ..., `N_Version`. Avec `patsubst` c'est possible :

```
$(patsubst Version.%, %_Version, Version.01 Version.02 Version.03 ... Version.N)
```

vaut :

```
01_Version 02_Version 03_Version ... N_Version
```

Exercice 16

Étant donnée une liste de noms de fichiers source, `Vecteur.c Matrice.c Interface.c main.c,` créer une liste de fichiers objets `Vecteur.o Matrice.o Interface.o main.o.`

Raccourci :

```
$(patsubst argument1, argument2, $(variable))
```

est l'exact équivalent de :

```
$(variable:argument1=argument2)
```

Attention au fait que dans le raccourci, on a bien écrit `variable` et non `$(variable)`. Enfin, la présence d'espaces autour des opérateurs « : » et « = » peut considérablement perturber le fonctionnement de la substitution.

Reprenez l'exercice 16 en utilisant cette fois-ci le raccourci.

Exercice 17

Soit un répertoire contenant les répertoires `includes`, `sources` et `objects`, contenant respectivement les fichiers entêtes, les fichiers sources et les fichiers objets. Le répertoire `sources` contient `Vecteur.c`, `Matrice.c`, `Interface.c` et `main.c`. Dans un premier temps le répertoire `objects` est supposé vide, mais il sera censé contenir les fichiers `Vecteur.o Matrice.o Interface.o` et `main.o.` Enfin on suppose, pour simplifier, que le répertoire includes contient les fichiers `Vecteur.h`, `Matrice.h`, `Interface.h` et `main.h` (ce qui est assez inhabituel, mais qui simplifie notre makefile). Nous verrons plus loin comment nous débarrasser de ce fichier `main.h`.

1. Définir la variable `SOURCES` qui contient tous les fichiers `*.c` du répertoire `sources` (c'est-à-dire `sources/Vecteur.c sources/Matrice.c sources/Interface.c sources/main.c`) et en déduire la variable `NAMES` qui vaut `Vecteur Matrice Interface main`

2. À partir de `NAMES`, définir les variables `INCLUDE_NAMES` et `OBJECT_NAMES` qui contiennent les noms de fichiers correspondant (sans référence à leur répertoire). Par exemple, `OBJECT_NAMES` vaut `Vecteur.o Matrice.o Interface.o main.o`
3. À partir des variables `INCLUDE_NAMES`, `SOURCE_NAMES`, `OBJECT_NAMES`, définies ci-dessus, et des variables `SOURCE_DIR`, `INCLUDE_DIR` et `OBJECT_DIR`, définir les variables `INCLUDES`, `SOURCES`, `OBJECTS` qui représentent la liste des références absolues des fichiers. Par exemple `SOURCES` vaut :

```
sources/Vecteur.c sources/Matrice.c sources/Interface.c sources/main.c
```

8.4 La fonction `strip`

C'est une fonction qui prend en argument une chaîne de caractères et qui retourne la même chaîne de caractère, débarrassée des espaces éventuelles qui la précéderaient ou qui la suivraient. Ceci est intéressant surtout dans le cas où il faudrait comparer deux chaînes de caractères. Il serait contrariant que deux chaînes de caractères soient déclarées différentes uniquement à cause d'espaces.

8.5 Les fonctions `filter` et `filter-out`

Quelquefois il est intéressant d'extraire d'une liste de mots ceux qui vérifient (ou ne vérifient pas) un motif donné. Étant donnés une liste `texte` de mots séparés par des espaces blancs et un `motif` contenant éventuellement un caractère « `%` »,

```
$(filter motif, texte)
```

représente le texte constitué des éléments de `texte` qui vérifient `motif`

```
$(filter-out motif, texte)
```

représente le texte constitué des éléments de `texte` qui ne vérifient pas `motif`. En particulier :

```
$(filter %.c, main.c Vecteur.c Interface.o Matrice.h)
```

vaut

```
main.c Vecteur.c
```

Exercice 18

- À partir de la même liste de noms que dans l'exercice 17, définir la variable `NON_MAIN_NAMES` constituée de tous les noms de `NAMES` sauf `main` (autrement dit, `Vecteur Matrice Interface`). En déduire la nouvelle valeur de `INCLUDE_NAMES` et de `INCLUDES`.
- Écrire le makefile de manière à ce que, pour la compilation, les fichiers nécessaires soient lus où ils se trouvent, que les fichiers objects soient créés dans leur répertoire (`objects`) et que l'exécutable soit créé au même niveau que le makefile.

8.6 La fonction `sort`

Étant donnée une liste de noms, cette fonction les trie dans l'ordre lexicographique et élimine tous les doublons.

8.7 Les fonctions `dir` et `notdir`

Étant donnée une liste de références absolues de fichiers, `dir` en extrait une liste qui ne contient que les répertoires des fichiers correspondants, alors que `notdir` en extrait une liste qui ne contient que les fichiers correspondants. Prenons par exemple notre arborescence habituelle et le makefile suivant :

```
DIVERS=sources/Vecteur.c objects/main.o
      objects/Matrice.o Makefile
titi :
 @ echo répertoires : $(dir
$(DIVERS))
 @ echo fichiers    :
$(notdir $(DIVERS))
```

lorsqu'on tape `make titi` on obtient l'affichage :

```
repertoires : sources/ objects/ objects/ ./
fichiers : Vecteur.c main.o Matrice.o Makefile
```

Ainsi, aucune des deux fonctions n'élimine les doublons.

8.8 La fonction `wordlist`

Utilisation : « `$(wordlist deb, fin, text)` ». À partir d'une liste `text` de mots, `wordlist` renvoie la sous-liste qui commence par le `deb`^{e} mot de `text` et qui se termine par le `fin`^{e} mot de `text`.

Par exemple : « `$(wordlist 2, 3, Vecteur.c Matrice.c Interface.c main.c)` » renvoie la liste « `Matrice.c Interface.c` ».

8.9 La fonction `wildcard`

Cette fonction permet de remplacer explicitement les expressions qui contiennent des *wildcards* en chaînes de caractères. Par exemple si le fichier makefile qui nous intéresse se trouve dans un répertoire où on peut trouver les fichiers `Vecteur.c Matrice.c Interface.c main.c` alors « `$(wildcard *.c)` » vaut « `Vecteur.c Matrice.c Interface.c main.c` ». Évidemment, dans le makefile suivant :

```
all :
    ls *.c
    ls $(wildcard *.c)
```

les deux commandes donnent exactement le même résultat. C'est un cas où l'usage de la fonction `wildcard` ne se justifie pas. Par contre dans :

```
all :
 @ echo sans wildcard : $(words *.c)
 @ echo avec wildcard : $(words $(wildcard
*.c))
```

Les commandes n'ont pas du tout le même résultat :

```
sans wildcard : 1
    avec wildcard : 4
```

En effet `*.c` est une liste qui contient un seul mot : `*.c`, alors que `$(wildcard *.c)` est explicitement une liste de 4 mots.

8.10 La fonction `foreach`

Il s'agit d'une fonction qui permet de répéter un certain nombre de fois un texte, à chaque fois avec une substitution différente. Syntaxe : « `LIST_TEXT=$(foreach x,list,text)` ». Ici, `LIST_TEXT` sera une suite de N instances de `text`, N étant le nombre de mots dans `list`. Si `x` intervient dans la construction de `text`, dans la i^{e} instance de `text`, `(1 < i <= N)`, `$(x)` est remplacé par le i^{e} mot de `list`. Ainsi dans l'exemple suivante :

```
LIST_BIDON=$(foreach x,$(LIST),Pense a souhaiter la bonne annee a $(x).)
```

`LIST_BIDON` est simplement une suite de phrases du type « `Pense a souhaiter la bonne annee a ...` ». Simplement, à chaque fois, `$(x)` est remplacé par un des éléments de `LIST`. Par exemple pour :

```
LIST = Julien madame ton_pere le_chien le_tabouret
```

la règle : « `all : @ echo $(LIST_BIDON)` » donne :

```
Pense a souhaiter la bonne annee a Julien. Pense a souhaiter la bonne annee a madame. Pense a
souhaiter la bonne annee a ton_pere. Pense a souhaiter la bonne annee a le_chien.  Pense a
souhaiter la bonne annee a le_tabouret.
```

Lorsque votre makefile comporte un grand nombre de cibles, de dépendances ou de commandes qui sont presque identiques mais qui diffèrent seulement par un détail, alors la fonction foreach vous permet de rendre votre makefile plus lisible et plus concis.

Exercice 19

Vous trouverez dans l'archive `http://icps.u-strasbg.fr/~vivien/Enseignement/PPP-2001-2002/Exercice19.tgz` un répertoire `Exercice19` qui contient 12 sous-répertoires :

`00_AutomateDEtat` `01_Simulation` `02_Visualisation` `03_MancheABalai` `04_PlusieursAvions` `05_JoueurDistant` `06_Protocole` `07_SynchroTempReel` `08_Inference` `09_MakeMovie` `10_EtapeTests` `11_Final`

On peut imaginer que chaque répertoire correspond aux différentes étapes de développement d'un même programme, ou qu'il s'agit des différentes composantes d'un seul grand programme. (En réalité ces programmes ne font strictement rien.) Chaque nom de répertoire commence par un chiffre, un *underscore*, puis un nom qui qualifie le contenu supposé du répertoire. Dans chaque répertoire, l'exécutable et le fichier qui contient la fonction `main` portent également ce nom (avec les extensions adéquates). Par exemple dans `06_Protocole`, l'exécutable s'appelle `Protocole` et le fichier qui contient la fonction `main` s'appelle `Protocole.c`. Le nom des autres fichiers n'a pas d'importance (ici, ils s'appellent `rien.c` et `bidon.c` dans tous les cas). Chaque répertoire contient un makefile qui permet de tout compiler et de tout nettoyer. Mais tous ces fichiers makefile sont strictement identiques. Pour les spécialiser, il faut leur donner, sur la ligne de commande, une valeur pour la variable `ETAPE`. Par exemple pour tout compiler dans le fichier `06_Protocole`, il faut taper :

```
${MAKE} -C 06_Protocole ETAPE=Protocole
```

Pour tout nettoyer dans le même répertoire, il faut taper :

```
${MAKE} clean -C 06_Protocole ETAPE=Protocole
```

L'objectif de cet exercice est de créer un makefile global situé au dessus des 12 répertoires et qui permet de compiler tous les programmes, de nettoyer les 12 répertoires et de les sauvegarder. Ainsi, qualitativement, l'objectif est le même que dans l'exercice 9. Seulement ici, on suppose que le nombre de sous-répertoires est trop grand pour qu'on puisse taper toutes les commandes une par une à la main. Nous voulons créer de façon automatique toutes les commandes de compilation et de nettoyage pour les 12 sous-répertoires.

9 Les règles implicites

Il existe un grand nombre d'opérations qui sont extrêmement courantes. Entre autres, les compilations de fichiers contenant du code C, C++, Java, ou même Fortran. La compilation de fichiers LaTeX est également une opération courante. L'outil make permet de réaliser ces opérations (et bien d'autres) de façon *implicite*, c'est-à-dire sans que vous ayez explicitement écrit une règle qui permette de réaliser ces tâches.

9.1 L'exemple de la compilation des fichiers individuels en langage C.

Il existe en particulier une règle :

```
%.o : %.c
 $(CC) -c $(CPPFLAGS) $(CFLAGS)
```

qui, même lorsqu'elle n'est pas écrite dans votre makefile, sera tout de même appliquée. On peut néanmoins changer la commande en écrivant cette règle de façon explicite et en y associant la commande voulue. Si l'on souhaite simplement empêcher la règle ci-dessus de s'appliquer, sans nécessairement y associer d'autres commandes, on peut écrire cette règle, mais lui associer de commande.

Par défaut, `$(CC)` vaut `cc`, mais on peut en changer la valeur. D'autre part `$(CPPFLAGS)` n'a aucun rapport avec le langage C++. C'est un *flag* qui a trait au préprocesseur `cpp`. Par exemple, dans un répertoire où nous avons les fichiers `Vecteur.c` `Matrice.c` `Interface.c` et `main.c`, le makefile suivant :

```
SOURCES = $(shell ls *.c)
OBJETS = $(SOURCES:%.c=%.o)
EXECUTABLE=scanline
MES_OPTIONS = -Wall -I/usr/local/include -I$(HOME)/include
$(EXECUTABLE) : $(OBJETS)
    gcc $(OBJETS) -o $(EXECUTABLE)
Vecteur.o : Vecteur.h Vecteur.c
    gcc $(MES_OPTIONS) -c Vecteur.c
```

```
Matrice.o : Vecteur.h Matrice.h Matrice.c
    gcc $(MES_OPTIONS) -c Matrice.c
Interface.o : Vecteur.h Matrice.h Interface.h Interface.c
    gcc $(MES_OPTIONS) -c Interface.c
main.o : Vecteur.h Matrice.h Interface.h main.c
    gcc $(MES_OPTIONS) -c main.c
```

produit exactement la même séquence de commandes (lorsqu'on tape `make`) que le makefile suivant :

```
CC=gcc
CFLAGS = -Wall
CPPFLAGS = -I/usr/local/include -I$(HOME)/include
SOURCES = $(shell ls *.c)
OBJETS = $(SOURCES:%.c=%.o)
EXECUTABLE=scanline
$(EXECUTABLE) : $(OBJETS)
    $(CC) $(OBJETS) -o $(EXECUTABLE)
Vecteur.o   : Vecteur.h
Matrice.o   : Vecteur.h Matrice.h
Interface.o : Vecteur.h Matrice.h Interface.h
main.o      : Vecteur.h Matrice.h Interface.h
```

qui lance les commandes suivantes lorsqu'on tape `make` :

```
gcc -Wall -I/usr/local/include -I/home/habibi/include -c Liste.c
gcc -Wall -I/usr/local/include -I/home/habibi/include -c Matrice.c
gcc -Wall -I/usr/local/include -I/home/habibi/include -c Vecteur.c
gcc -Wall -I/usr/local/include -I/home/habibi/include -c main.c
gcc Liste.o Matrice.o Vecteur.o main.o -o scanline
```

Pourtant aucune règle ne spécifie comment il faut transformer les fichiers de type `*.c` en fichiers de type `*.o`. De plus, le makefile ci-dessus indique simplement que les fichiers objets dépendent des fichiers entêtes, sans spécifier ce qu'il faut faire si un des fichiers entête est modifié. Pourtant lorsqu'on modifie, par exemple, `Interface.h` make recompile bien le fichier `Interface.c` pour tenir compte de ce changement. Ainsi, du moment que les fichiers sources et les fichiers objets ont les mêmes noms (à l'extension près) make peut automatiquement déduire les fichiers qu'il faut recompiler.

9.2 makedepend

Les quatre dernières lignes du dernier makefile peuvent paraître fastidieuses à écrire, d'autant plus qu'il s'agit d'une information redondante. En effet, la raison d'être de ces quatre règles est que

- le fichier `Matrice.c (ou plutôt Matrice.h)` contient la directive `#include "Vecteur.h";`
- le fichier `Interface.c` (ou plutôt `Interface.h`) contient les directives :
  ```
  #include "Vecteur.h"
  #include "Matrice.h"
  ```
  car on suppose que l'interface doit manipuler aussi bien des vecteurs que des matrices ;
- le fichier `main.c` contient la directive `#include "Interface.h"` car les fonctions dans `main.c` n'ont affaire qu'avec l'interface (même s'il faut recompiler `Interface.c` si `Vecteur.h` ou `Matrice.h` changent).

Il existe un outil qui permet de lire les fichiers source à la recherche des directives d'inclusion et d'en déduire les lignes de dépendance entre fichiers objets et fichiers entête. Par exemple la commande :

```
makedepend Vecteur.c Matrice.c Interface.c main.c
```

ajoute à la fin du makefile le texte :

```
# DO NOT DELETE
Interface.o: Interface.h Matrice.h Vecteur.h
main.o: Interface.h Matrice.h Vecteur.h
Matrice.o: Matrice.h Vecteur.h
Vecteur.o: Vecteur.h
```

La ligne contenant « `DO NOT DELETE` » est utilisée par makedepend pour savoir à quel endroit il peut écrire ces dépendances. Si aucune chaîne de ce type n'est trouvé dans le fichier, alors les dépendances sont ajoutées à la fin du fichier, même si elles s'y trouvent déjà.

Exercice 20

Ajouter au makefile ci-dessus une règle *phony* dont la cible est `depend`. L'objectif est qu'avant la première compilation, ou après avoir ajouté ou supprimé une directive d'inclusion, il suffise de taper `make depend` pour que toutes les dépendances soient ajoutées à la fin du fichier.

9.3 Autres règles implicites

D'autres règles implicites gèrent :
- la production de librairies : commande `$(AR)` options `$(ARFLAGS)` ;
- la compilation de programmes en assembleurs : commande `$(AS)` options `$(ASFLAGS)` ;
- la compilation de programmes en C++ : commande `$(CXX)` options `$(CXXFLAGS)` (à ne pas confondre avec `$(CPPFLAGS)`) ;
- le prétraitement de fichiers avec le préprocesseur `cpp` : commande `$(CPP)` options `$(CPPFLAGS)` ;
- la compilation de programmes en fortran : commande `$(FC)` options `$(FFLAGS)` ;
- la compilation de programmes en pascal : commande `$(PC)` options `$(PFLAGS)` ;
- la compilation de fichiers LaTeX : commande `$(TEX)` options `$(TEXFLAGS)` ;
- etc., etc., cf. `http://www.gnu.org/manual/make/html_mono/make.html` .