Introduction aux Makefiles

Un `Makefile` est un fichier, utilisé par le programme `make`, regroupant une série de commandes permettant d'exécuter un ensemble d'actions, typiquement la compilation d'un projet. Un `Makefile` peut être écrit à la main, ou généré automatiquement par un utilitaire (exemples : `automake`, `CMake`, ...). Il est constitué d'une ou de plusieurs règles de la forme :

```
cible: dépendances
	commandes
```

Quelques remarques préliminaires sur cette syntaxe. Il peut y avoir plusieurs lignes de `commandes` successives, chacune est *nécessairement* précédée d'une tabulation, faute de quoi un message abscons sera émis. Chaque ligne de `commandes` est exécutée dans un *shell* lancé par `make`, il s'agit donc d'une ou plusieurs commandes usuelles de votre *shell*. Vous pouvez mettre plusieurs commandes sur la même lignes séparées par des `;`, puisque c'est du *shell*, et continuer sur la ligne suivante en mettant un *backslash* (\) en fin de ligne. Attention dans ce cas à ne pas mettre de blanc derrière [1], car ce serait un échappement du blanc, et `make` hurlerait aussitôt.

Lors du parcours du fichier, le programme `make` évalue d'abord la première règle rencontrée, ou celle dont le nom est spécifié en argument. L'évaluation d'une règle se fait récursivement :

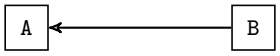
1. les dépendances sont analysées : si une dépendance est la cible d'une autre règle, cette règle est à son tour évaluée ;
2. lorsque l'ensemble des dépendances a été analysé, et si la cible est plus ancienne que les dépendances, les commandes correspondant à la règle sont exécutées.

Par exemple, sur un `Makefile` ressemblant à ceci :

```
A: B
	commande-qui-fabrique-A

B:
	commande-qui-fabrique-B
```

Puisque `A` dépend de `B`, on pourrait dessiner un graphe de dépendance entre ces deux cibles :



La commande `make` va considérer la règle `A`, mais pour cela, il faut d'abord construire les dépendances, donc sur cet exemple considérer la règle `B`, donc exécuter `commande-qui-fabrique-B` (notons que `B` n'a pas de dépendances donc peut être évaluée). Une fois les dépendances satisfaites, `commande-qui-fabrique-A` sera exécuté si nécessaire.

Ce document présente les fonctionnalités les plus importantes de `make`. Nous avancerons pas à pas, en nous basant sur des exemples. Le code complet des exemples est disponible sur la page web du cours (archive `expl-make.tar.gz`) : pour chacune des sections suivantes, un répertoire contient le nécessaire pour reproduire les manipulations décrites dans ce document. Les exemples sont complets, ce ne sont pas des exercices à trous. Pour bien comprendre comment marche l'outil, lisez en détail les `Makefile`s fournis, et essayez-les !

1 Compilation séparée, ou comment compiler plus vite

Répertoire contenant l'exemple : `01-ada/`

1. `:se list` sous *vim* permet de les identifier.

Ce répertoire contient juste un petit programme Ada. La commande `gnatmake` que l'on utilise pour le compiler fait déjà tout ce qu'il faut (compilation séparée, recompilation des fichiers seulement si nécessaire...) donc il n'est pas utile d'avoir un `Makefile` ici.

On peut compiler le programme comme ceci :

```
$ gnatmake main.adb
gcc-4.4 -c main.adb
gcc-4.4 -c au_revoir.adb
gcc-4.4 -c bonjour.adb
gnatbind -x main.ali
gnatlink main.ali
```

Dans un premier temps, les fichiers .adb sont compilés (commande `gcc`) et quand tous les fichiers ont été compilés, les morceaux sont assemblés (commandes `gnatbind` et `gnatlink`). Cet assemblage s'appelle l'édition de liens, et c'est à ce moment que le fichier exécutable est généré.

Si on relance une compilation, gnatmake se rends compte tout seul qu'il n'y a rien à faire :

```
$ gnatmake main.adb
gnatmake: "main" up to date.
```

On peut maintenant modifier l'un des fichiers, par exemple changer la chaine affichée dans la fonction `Dire_Au_Revoir` dans le fichier `au_revoir.adb`, puis relancer `gnatmake` :

```
$ emacs au_revoir.adb
$ gnatmake main.adb
gcc-4.4 -c au_revoir.adb
gnatbind -x main.ali
gnatlink main.ali
```

Seul le fichier que l'on vient de modifier est recompilé, et l'édition de liens est refaite pour régénérer le fichier exécutable `main`.

Une autre expérience intéressante : modifier le fichier `au_revoir.ads` (par exemple, ajouter un commentaire) :

```
$ emacs au_revoir.ads
$ gnatmake main.adb
gcc-4.4 -c main.adb
gcc-4.4 -c au_revoir.adb
gnatbind -x main.ali
gnatlink main.ali
```

On peut remarquer que dans ce cas, le fichier `main.adb` est également recompilé. En effet, ce fichier utilise le fichier `au_revoir.ads` pour savoir comment appeler les fonctions du paquet `Au_Revoir`. On dit que la compilation de `main.adb` *dépend de* `au_revoir.ads`.

Sur cet exemple, le gain n'est pas flagrant, mais sur un plus gros projet (par ex. le noyau Linux), recompiler l'ensemble peut prendre plusieurs minutes, voire plusieurs heures, alors que la recompilation d'un seul fichier peut se faire en quelques secondes.

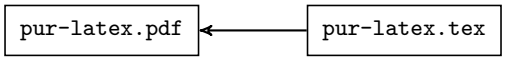
2 Compilation d'un document LaTeX simple avec `make`

Répertoire contenant l'exemple : `02-latex/`

Nous allons maintenant compiler un fichier LaTeX très simple, en utilisant un Makefile. Nous n'avons pour l'instant pas grand chose à faire, le Makefile ne comporte qu'une règle :

```
pur-latex.pdf: pur-latex.tex
	pdflatex pur-latex.tex
```

(Attention, la ligne `pdflatex` *doit* commencer par une tabulation)



La *cible* de cette règle est `pur-latex.pdf`, c'est le fichier qui va être généré. La seule *dépendance* est `pur-latex.tex`, qui est le fichier source. Le fichier `pur-latex.pdf` sera donc régénéré à chaque fois que le fichier `pur-latex.tex` sera modifié. L'*action* à exécuter pour régénérer ce fichier est la commande `pdflatex pur-latex.tex`.

On peut tester cet exemple :

```
$ make
pdflatex pur-latex.tex
This is pdfTeX, Version 3.1415926-1.40.10 (TeX Live 2009/Debian)
[...]
Output written on pur-latex.pdf (2 pages, 15993 bytes).
Transcript written on pur-latex.log.
```

La commande `make` a cherché la première cible dans le fichier (c'est facile ici, il n'y en a qu'une), et a exécuté l'action correspondante. `make` a affiché la commande (`pdflatex pur-latex.tex`) avant de l'exécuter.

Comme pour la commande `gnatmake`, une deuxième exécution de `make` ne fait rien :

```
$ make
make: 'pur-latex.pdf' is up to date.
```

En effet, `make` a trouvé la première cible, `pur-latex.pdf`, et a comparé la date de dernière modification de ce fichier sur le disque avec les dates de dernières modifications de la dépendance `pur-latex.tex`. La cible étant plus récente que la dépendance, `make` en a déduit qu'il n'y avait rien à faire.

On peut également vérifier qu'une modification de la dépendance entraîne une recompilation :

```
$ emacs pur-latex.tex
$ make
pdflatex pur-latex.tex
[...]
```

3 Plusieurs règles, dépendances, ...

Répertoire contenant l'exemple : `03-latex-chain/`

Nous allons maintenant améliorer ce Makefile en ajoutant une règle pour transformer le fichier PDF généré au format « deux par page » (pratique pour l'imprimer en économisant du papier).

La commande `psnup -2` permet de faire cette transformation sur des fichiers PostScript (`.ps`), une variante du format PDF. Une solution est donc de transformer notre fichier PDF en fichier PostScript, d'appeler `psnup -2`, puis de retransformer le résultat au format PDF. On peut écrire tout cela avec un Makefile :

Pour transformer le fichier PDF en fichier PostScript, on utilise la commande `pdf2ps` :

```
pur-latex.ps: pur-latex.pdf
	pdf2ps pur-latex.pdf
```

L'application de la commande `pdfnup` proprement dite se fait comme ceci :

```
pur-latex-nup.ps: pur-latex.ps
	psnup -2 pur-latex.ps > pur-latex-nup.ps
```

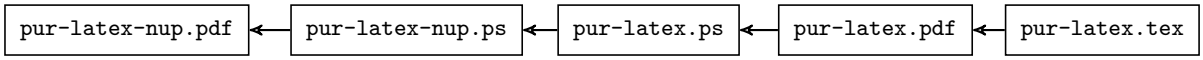
Nous avons choisi d'appeler le fichier résultat `pur-latex-nup.ps`. Le nom de ce fichier importe peu pourvu qu'il termine par `.ps`, et bien sûr qu'il n'ait pas le même nom que le fichier source.

Par défaut, la commande `psnup` écrit le résultat sur sa sortie standard. On a donc redirigé cette sortie standard vers le fichier cible. L'*action* d'une règle de Makefile est en fait un morceau de script shell, donc on a le droit d'y écrire tout ce qu'on aurait écrit dans un script shell, en particulier le `>` pour la redirection de la sortie.

La transformation du fichier PostScript « deux par pages » en PDF se fait de manière similaire à la transformation inverse :

```
pur-latex-nup.pdf: pur-latex-nup.ps
        ps2pdf pur-latex-nup.ps
```

Le graphe de dépendance entre les différents fichiers est alors le suivant :



Ici, chaque cible n'a qu'une dépendance directe, mais la modification de `pur-latex.tex` impose de regénérer `pur-latex.pdf`, qui lui même impose de regénérer `pur-latex.ps`, etc., jusqu'à `pur-latex-nup.pdf`.

On peut tester le résultat. Par défaut, la commande `make` va toujours prendre la première cible du Makefile, qui est la version simple générée par `pdflatex` :

```
$ make
pdflatex pur-latex.tex
[...]
Output written on pur-latex.pdf (2 pages, 15993 bytes).
Transcript written on pur-latex.log.
```

On peut passer un argument à `make` pour spécifier la cible à générer :

```
$ make pur-latex-nup.pdf
pdf2ps pur-latex.pdf
psnup -2 pur-latex.ps > pur-latex-nup.ps
[1] Wrote 1 pages, 108108 bytes
ps2pdf pur-latex-nup.ps
```

`make` a affiché les commandes qu'il a exécuté. On peut remarquer qu'il a exécuté seulement la partie nécessaire de la chaine, et n'a pas eu besoin de relancer `pdflatex`, puisque nous venions de le faire. La ligne `[1] Wrote 1 pages, 108108 bytes` est affichée par `psnup` : ce message est affiché sur la sortie d'erreur, et n'est donc pas redirigé par le `> pur-latex-nup.ps`.

Bien sûr, nous aurions aussi pu utiliser les commandes `latex` et `dvips` pour générer un fichier PostScript.

4 Makefiles et variables

Répertoire contenant l'exemple : `04-latex-implicit/`

Ce premier Makefile était très « ad-hoc » : le nom du fichier source est répété à trois endroits, donc pour réutiliser ce Makefile sur un autre fichier, il faudra modifier ces trois endroits. On peut améliorer la situation en utilisant une variable :

```
FILE=pur-latex

$(FILE).pdf: $(FILE).tex
        pdflatex $(FILE).tex
```

La syntaxe des variable est subtilement différente de celle du shell Unix : une variable est définie par une affectation du type `VARIABLE=valeur`, et utilisée avec la syntaxe `$(VARIABLE)`. Attention, les parenthèses sont nécessaires ici (alors que `$(...)` veut dire tout autre chose en shell).

En fait, `make` connait même un raccourci pour dire « la première dépendance » (i.e. « ce qui suit directement les deux points dans la règle », c'est à dire `$(FILE).tex` dans notre exemple) : `$<`. La règle peut donc s'écrire

```
FILE=pur-latex

$(FILE).pdf: $(FILE).tex
	pdflatex $<
```

Un autre raccourci bien pratique est `$@`, qui veut dire « la cible de la règle courante ». La règle appelant `psnup` peut donc maintenant s'écrire :

```
$(FILE)-nup.ps: $(FILE).ps
	psnup -2 $< > $@
```

Les variables implicites les plus utiles sont les suivantes :

— `$@` : nom de la cible ;
— `$<` : nom de la première dépendance ;
— `$^` : liste des dépendances ;
— `$?` : liste des dépendances plus récentes que la cible ;
— `$*` : la chaîne qui correspond au `%` dans l'instantiation d'une règle implicite (en général, le nom d'un fichier sans son suffixe).

Il est bien sur possible d'utiliser des variables du *shell* dans les commandes, ou les commandes du *shell* qui font usage du `$`, mais alors il faut précéder leur utilisation de `$`, ce qui fait un double dollar `$$`. Il est façile de l'oublier, et le comportement est alors relativement imprévisible. Un exemple qui affiche sur la sortie standard les entiers de 1 à 10 :

```
boucle:
	for i in $$(seq 1 10); do \
		echo "$$i"; \
	done
```

5 Règles ne générant pas de fichier (`.PHONY`)

Répertoire contenant l'exemple : `05-latex-phony/`

Nous allons maintenant ajouter une règle pour visualiser le fichier PDF directement après avoir été compilé. Une fois n'est pas coutume, cette règle ne génère pas de fichier, et son action sera exécutée dans tous les cas. On va simplement donner un nom à la règle (`view` dans notre exemple), ce qui permettra d'appeler `make view` pour exécuter l'action correspondante, après avoir éventuellement exécuté les actions nécessaires pour les dépendances.

Avec GNU Make, c'est ce qu'on appelle une « Phony target », et la syntaxe est la suivante :

```
.PHONY: view
view: $(FILE).pdf
	evince $<
```

Si on lance `make view`, `make` va donc recompiler le fichier si nécessaire pour générer le fichier PDF, puis lancer `evince` dessus. Si on relance `make view`, make va sauter l'étape de compilation (parce que la cible est déjà plus récente que les dépendances), mais va tout de même relancer `evince` (parce que la règle est déclarée `.PHONY`).

Une utilisation de ces « Phony target » est de spécifier simplement la règle par défaut : la convention est d'avoir une règle nommée `all` en début de fichier et qui spécifie dans ses dépendances les règles à

considérer par défaut. Dans tous les cas, la règle par défaut est la première du fichier, et si on respecte cette convention, alors la cible par défaut sera toujours `all`. Dans notre cas, on peut écrire ceci :

```
.PHONY: all
all: $(FILE).pdf
```

On remarquera que cette règle n'a pas d'action associée (la ligne suivante est une ligne vide, pas une ligne commençant par une tabulation). L'exécution de `make all`, ou plus simplement `make` va néanmoins vérifier les dépendances de la cible `all`, donc transitivement vérifier les dépendances de `pur-latex.pdf` et se rendre compte éventuellement qu'elles ne sont pas satisfaites. Si c'est le cas, l'action `pdflatex pur-latex.tex` sera exécutée.

Une autre utilisation classique est d'avoir une cible `clean` qui permette d'effacer les fichiers générés. On peut par exemple faire une règle `clean` qui efface les fichiers `.ps` intermédiaires et les fichiers générés par LaTeX :

```
.PHONY: clean
clean:
	rm -f $(FILE).ps $(FILE)-nup.ps $(FILE).log $(FILE).aux
```

On peut ajouter une règle `realclean` qui dépende de `clean` et qui supprime également les fichiers PDF générés :

```
.PHONY: realclean
realclean: clean
	rm $(FILE).pdf $(FILE)-nup.pdf
```

6 Règles génériques

Répertoire contenant l'exemple : `06-latex-pattern/`

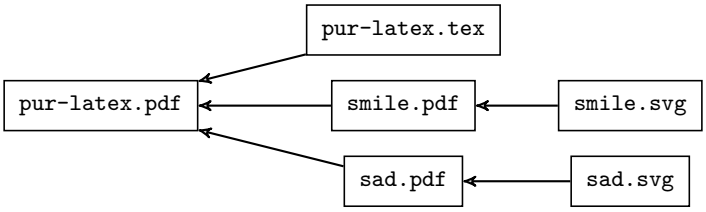
On va maintenant compiler un fichier LaTeX contenant des figures. Les figures sont au format SVG. Comme `pdflatex` ne sait pas inclure de fichier SVG, il va falloir faire la conversion nous-mêmes, avant d'appeler `pdflatex`. La commande `inkscape` *`fichier`*`.svg -export-pdf` *`fichier`*`.pdf` permet de faire exactement cela. Nous avons deux figures, `smile.svg` et `sad.svg`, donc nous pouvons écrire les deux règles :

```
smile.pdf: smile.svg
	inkscape $< --export-pdf $@

sad.pdf: sad.svg
	inkscape $< --export-pdf $@
```

Puis modifier la règle appelant `pdflatex` en ajoutant `smile.pdf` et `sad.pdf` aux dépendances :

```
$(FILE).pdf: $(FILE).tex smile.pdf sad.pdf
	pdflatex $<
```



Cette version marche, mais sera fastidieuse si nous avons beaucoup de figures à inclure. Il serait en effet agréable de dire à `make` *une seule* fois comment générer un `.pdf` à partir d'un `.svg`. Les « pattern-rules » sont faites exactement pour ça ! On peut en effet remplacer nos deux règles ci-dessus par :

```
%.pdf: %.svg
	inkscape $< --export-pdf $@
```

et garder la règle `$(FILE).pdf` comme nous l'avions écrite. Quand `make` va trouver les dépendances `smile.pdf` et `sad.pdf`, il va instancier la règle en remplaçant le `%` une première fois par `smile`, et la seconde fois par `sad`.

En fait, on peut même faire encore mieux : si on suppose que tous les fichiers `.svg` du répertoire courant sont utilisés par notre fichier LaTeX, on peut dire à `make` d'ajouter tous les fichiers `.pdf` correspondants aux dépendances de la règle principale. On peut le faire en deux temps : `$(wildcard *.svg)` permet d'obtenir la liste des fichiers `.svg` du répertoire courant, et la substitution `$(VARIABLE:`*motif*`=`*motif*`)` permet de remplacer une extension par une autre. On peut donc récupérer la liste des fichiers comme ceci :

```
SVG_SOURCES=$(wildcard *.svg)
SVG_PDFFILES=$(SVG_SOURCES:%.svg=%.pdf)
```

Puis modifier la règle principale comme cela :

```
$(FILE).pdf: $(FILE).tex $(SVG_PDFFILES)
	pdflatex $<
```

(Vous n'avez pas besoin de comprendre les substitutions de variables pour faire le TP en temps libre)

7 Un bonus : le parallélisme

Maintenant que l'on a bien défini les dépendances entre les règles, et les actions pour chaque règle, on peut demander à `make` d'exécuter les actions en *parallèle* (i.e. en même temps) quand cela est possible. Dans notre dernier exemple, les deux appels à inkscape peuvent être parallélisés (en effet, `smile.pdf` et `sad.pdf` ne dépendent pas l'un de l'autre), ce qui permettra d'accélérer un petit peu la compilation.

Concrètement, on peut faire ceci en utilisant la commande `make -j 2` au lieu de `make`, pour laisser la commande lancer 2 processus en même temps. Vous approfondirez ce point pendant le TP en libre-service.

8 Pour les curieux : LaTeX et `make` dans la vraie vie ...

Nous avons utilisé la compilation de document LaTeX pour illustrer le fonctionnement de `make`, et pour simplifier, nous avons ignoré certaines spécificités de LaTeX : certains documents ont besoin de plusieurs compilations, et éventuellement d'appels à `bibtex` et `makeindex`. Écrire un bon Makefile pour ces documents n'est pas chose simple, il est souvent préférable de réutiliser un outil tout fait. On pourra citer `LaTeX.mk` du paquet latex-utils [2] (une belle démonstration de Makefile complexe... et peu lisible !) ou bien les commandes `latex-mk` [3] et `rubber` [4].

2. `https://gforge.inria.fr/projects/latex-utils/`
3. `http://latex-mk.sourceforge.net/`
4. `https://launchpad.net/rubber`