

Introduzione ai Makefile per compilare documenti
L^AT_EX

Elrond

28 settembre 2009

Quest'opera è rilasciata con licenza Creative Commons Attribuzione 3.0 Unported. L'enunciato integrale della Licenza in versione 3.0 è reperibile all'indirizzo internet <http://creativecommons.org/licenses/by/3.0/deed.it>.

Tu sei libero:

- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
- di modificare quest'opera

Alle seguenti condizioni:

Attribuzione Devi attribuire la paternità dell'opera nei modi indicati dall'autore o da chi ti ha dato l'opera in licenza e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.

- Ogni volta che usi o distribuisi quest'opera, devi farlo secondo i termini di questa licenza, che va comunicata con chiarezza.

Indice

1	Introduzione	1
1.1	Installazione di <code>make</code>	1
2	Realizzare un Makefile	3
2.1	Le regole	3
2.2	Come funziona <code>make</code>	4
2.3	Un semplice <code>Makefile</code>	4
2.4	Un <code>Makefile</code> un po' più elaborato	5
2.5	Phony target	6
2.6	Le variabili	7
2.7	Un <code>Makefile</code> ancora più complesso	10
	Bibliografia	15

Capitolo 1

Introduzione

Quando si scrive un documento \LaTeX molto elaborato, potrebbe diventare necessario dare numerosi comandi in fase di compilazione, comandi che a volte risultano molto lunghi o di difficile memorizzazione. In questo caso viene in aiuto la utility dei sistemi UNIX `make` che si utilizza da linea di comando. `make` è molto usata nella compilazione dei programmi, specie nell'ambiente open source, ma può essere utilmente sfruttata per automatizzare e semplificare anche la compilazione di sorgenti \LaTeX e anche altre operazioni quali la cancellazione di file temporanei non necessari e la creazione di un archivio compresso per conservare i file sorgenti. Non ci sono limiti alla fantasia.

Per compilare un sorgente, `make` non fa altro che leggere le istruzioni presenti in un file di testo, chiamato necessariamente `Makefile`¹ e scritto con una particolare sintassi.

I `Makefile` sono dei file di testo puro, per scriverli c'è bisogno quindi di un normale editor di testo come Gedit o Kate per GNU/Linux, TextEdit per Mac OS X.

Di `make` esistono alcune versioni, le più famose sono GNU `make` e BSD `make`. In questa guida verrà spiegato come scrivere `Makefile` elementari per GNU `make`, cioè il dialetto presente nei sistemi GNU/Linux e Mac OS X.

Nota: da qui in poi (tranne nel Paragrafo 1.1), tutti i comandi da eseguire nel terminale devono essere lanciati trovandosi nella stessa cartella in cui è presente il `Makefile` che si desidera utilizzare, se non diversamente specificato.

1.1 Installazione di `make`

Per ottenere `make` in ambiente GNU/Linux, se non già presente nel sistema bisogna installare il pacchetto omonimo utilizzando il gestore pacchetti della propria distribuzione. Per esempio, per Debian e sistemi derivati (come Ubuntu) bisogna dare il comando

```
apt-get install make
```

con i diritti di root. In Fedora e derivate, invece, bisogna eseguire il comando

```
yum install make
```

¹In realtà potrebbe avere anche altri nomi, questo è quello consigliato, vedi [3], pag. 12.

Sui sistemi Mac OS X, `make` dovrebbe essere preinstallato. Per verificarlo si può dare il comando nel Terminale

```
make --version
```

che restituisce la versione del programma installata nel sistema. Se il programma non è presente verrà mostrato un messaggio di errore.

Capitolo 2

Realizzare un Makefile

2.1 Le regole

Un `Makefile` basilare è composto essenzialmente da *regole* (chiamate in inglese *rules*) che hanno la seguente struttura (vedi [3], pag. 3):

Codice 2.1: Struttura di una regola

```
obiettivo: prerequisiti
----->comandi
----->...
----->...
```

L'*obiettivo* (in inglese *target*), ciò che viene scritto prima del simbolo ':', è il nome del file che verrà generato con la regola descritta. Si può utilizzare come obiettivo un'azione da eseguire, come 'clean' (vedi il paragrafo 2.5 a pagina 6).

I *prerequisiti* sono i file che devono essere presenti al momento dell'esecuzione di una regola e a partire dai quali viene generato il file obiettivo. Spesso gli obiettivi dipendono da più file (che vengono elencati separati da uno spazio). Per evitare problemi in fase di compilazione è conveniente che tutti i file non abbiano degli spazi nel nome. Alcune regole, come la citata 'clean', possono non avere alcun prerequisito. Fra i prerequisiti possono comparire anche i nomi di altri obiettivi che, come detto, potrebbero essere dei comandi.

Il programma `make` conosce i comandi necessari per eseguire una regola leggendo dall'elenco dei *comandi*. I comandi di ciascuna regola *devono necessariamente* essere preceduti da una tabulazione (nel Codice 2.1 è evidenziata da una lunga linea), non da spazi altrimenti verrà segnalato un errore.

Un `Makefile` può contenere, oltre alle regole, anche dei commenti, introdotti dal simbolo `#`: tutto ciò che si trova alla destra di `#` viene trattato come commento. Questo simbolo ha un comportamento simile a `%` nel linguaggio `LATEX`, vedi [2], pag. 24.

Se necessario è possibile suddividere una riga di codice molto lunga su più righe in inserendo `\` seguito da un carattere di nuova linea (in pratica bisogna premere il tasto `[Invio]` subito dopo l'inserimento del backslash). Ciò non è comunque obbligatorio, in quanto non sono posti limiti alla lunghezza delle righe di un `Makefile`.

2.2 Come funziona make

Se si richiama `make` senza argomenti, cioè si da solo il comando

```
make
```

esso procede all'esecuzione del primo target che trova nel `Makefile`, per la precisione il primo target il cui nome non inizia con '.' (questo comportamento può essere modificato, vedi [3], pag. 5).

Quando `make` processa un `Makefile` verifica se è necessario aggiornare un obiettivo: se l'obiettivo indicato non esiste oppure ha una data di ultima modifica precedente almeno a una di quelle dei file elencati nei *prerequisiti* è considerato da aggiornare e la regola viene eseguita. In caso contrario (l'obiettivo esiste ed è più recente di tutti i prerequisiti) la regola non viene eseguita. Se si vuole forzare l'esecuzione delle regola, ignorando le date di modifica, bisogna utilizzare l'opzione `-B` in questo modo:

```
make -B obiettivo
```

Le istruzioni per aggiornare un obiettivo sono presenti nei *comandi*: questi sono passati alla shell, normalmente la 'sh'. Per scegliere una shell differente bisogna modificare la variabile `SHELL`. Per informazioni sulle variabili vedi il Paragrafo 2.6 a pagina 7.

Quindi `make` verifica se, prima di eseguire una regola, è necessario aggiornare uno o più dei file elencati fra i prerequisiti (o eseguire una o più delle regole presenti). Nel seguente esempio

```
obiettivo1: prerequisito1 prerequisito2
           comando1
           comando2

prerequisito2: prerequisito3
              comando3
```

il `prerequisito2` ha una propria regola: dando il comando

```
make obiettivo1
```

`make` verifica se è necessario aggiornare, operando il `comando3` specificato, il file `prerequisito2` prima di eseguire la regola associata all'`obiettivo1`.

Gli obiettivi dipendono dai prerequisiti: quando un prerequisito viene modificato allora probabilmente l'obiettivo deve essere ricreato. Questo meccanismo, forse un po' laborioso e difficile da comprendere inizialmente, dovrebbe risultare chiaro più avanti.

2.3 Un semplice Makefile

Passiamo ora alla pratica vedendo un esempio di `Makefile` molto semplice:

Codice 2.2: Un semplice Makefile

```
documento.dvi: documento.tex
—————→ latex documento
```

Nel Codice 2.2, l'*obiettivo*, cioè il file ottenuto dopo la compilazione è il file `documento.dvi`, il *prerequisito* è il solo file `documento.tex` e l'unico comando che deve essere eseguito è `latex documento`.

Per invocare `make`, dopo aver aperto un terminale ed essersi spostati nella cartella in cui si trova il `Makefile` con il comando `cd`, bisogna utilizzare il comando

```
make target
```

sostituendo a `target` il nome del target che specificato nella regola che si desidera eseguire.

2.4 Un Makefile un po' più elaborato

Se nel nostro file di esempio `documento.tex` è presente una bibliografia realizzata con `BIBTEX`, per compilare il documento è necessario (vedi [2], pag. 151) eseguire i comandi

```
latex documento
bibtex documento
latex documento
latex documento
latex documento
```

Ripetere questi cinque comandi ogni volta che si desidera compilare un documento può diventare, quindi, un'operazione noiosa. È in questi casi che si vede l'utilità di `make`. Nel nostro `Makefile` possiamo mettere la seguente regola

```
documento.dvi: documento.tex bibliografia.bib
    latex documento
    bibtex documento
    latex documento
    latex documento
    latex documento
```

in cui i prerequisiti (cioè i file che devono essere presenti e aggiornati) sono il file principale `documento.tex` e il file in cui abbiamo scritto la nostra base di dati dei riferimenti bibliografici. Per compilare è sufficiente dare il comando

```
make documento.dvi
```

oppure solo `make` qualora quella regola fosse la prima presente nel `Makefile`.

Si può anche mettere una regola per compilare lo stesso documento con `LATEX` e un'altra per compilarlo con `PDFLATEX`, per poter scegliere fra un file di output in formato DVI o PDF. In questo caso il `Makefile` apparirebbe così:

Codice 2.3: La prima regola permette di compilare un documento con `LATEX`, la seconda con `PDFLATEX`

```
documento.dvi: documento.tex bibliografia.bib
    latex documento
    bibtex documento
    latex documento
```

```

    latex documento
    latex documento

documento.pdf: documento.tex bibliografia.bib
    pdflatex documento
    bibtex documento
    pdflatex documento
    pdflatex documento
    pdflatex documento

```

In alcune distribuzioni GNU/Linux sono presenti gli script shell `texi2dvi` e `texi2pdf` che eseguono rispettivamente \LaTeX e \PDF\TeX (e, se necessario, \BIB\TeX) sul sorgente il numero strettamente necessario di volte per la corretta compilazione del documento (vedi [1], pag. 63). Utilizzando questi due comandi il Codice 2.3 si potrebbe ridurre quindi a

```

documento.dvi: documento.tex bibliografia.bib
    texi2dvi documento

documento.pdf: documento.tex bibliografia.bib
    texi2pdf documento

```

Bisogna però evidenziare che se il documento richiede comandi particolari per la compilazione (come quando si utilizza il pacchetto `frontespizio`), `texi2dvi` e `texi2pdf` non sono più in grado di generare correttamente il documento finale.

2.5 Phony target

È possibile specificare delle regole che non hanno come obiettivo il nome del file che verrà ottenuto. Questo tipo di obiettivi vengono chiamati in inglese *phony targets* (cioè *falsi obiettivi*). Spesso i phony target hanno come nome dei comandi. Per esempio, quando si esegue la regola

```

clean :
    rm *.aux *.log *.out

```

vengono cancellati tutti i file con estensione `.aux`, `.log` e `.out` che vengono prodotti durante la compilazione di un semplice documento \LaTeX .¹ È consigliabile specificare esplicitamente quali sono i phony target utilizzati nel `Makefile`: se nella cartella in cui si trova il `Makefile` è presente un file chiamato `clean` questa regola non verrebbe mai eseguita.² Per fare ciò si usa l'obiettivo speciale `.PHONY`:

```

.PHONY: clean
clean :
    rm *.aux *.log *.out

```

¹Il comando `rm` cancella tutti i file che vengono elencati di seguito. Il metacarattere `*` sostituisce una qualsiasi sequenza di caratteri.

²Dal momento che la regola non ha prerequisiti, il file `clean` risulterebbe sempre aggiornato (vedi [3], pag. 31).

In questo modo `make` sa che ‘clean’ non è il nome del file che si deve ottenere e la regola verrà quindi sempre eseguita, indipendentemente dalla presenza di un eventuale file `clean`.

I phony target possono essere anche usati per creare una sorta di *alias* di altre regole. Per esempio, inserendo il seguente

Codice 2.4: I prerequisiti della regola dell’obiettivo `.PHONY` sono i nomi dei phony target che vengono successivamente specificati

```
.PHONY: dvi pdf
```

```
dvi: documento.dvi
```

```
pdf: documento.pdf
```

in un `Makefile`, prima del Codice 2.3, per compilare il documento in formato DVI servirà dare il comando

```
make dvi
```

Analogamente, per ottenere il file PDF bisognerà dare il comando

```
make pdf
```

L’utilità dell’uso di questi alias è che i comandi da eseguire sono indipendenti dal nome del file di output.

Accanto al phony target ‘clean’ si trova spesso ‘distclean’: ‘clean’ cancella solo i file temporanei generati durante la compilazione, ‘distclean’ in più elimina anche i file di output (quindi gli eventuali file in formato `.pdf` o `.dvi`, nel caso di documenti `LATEX`).³ Poiché ‘distclean’, *oltre* a cancellare file rimossi da ‘clean’ ne cancella altri, è possibile inserire ‘clean’ come prerequisito di ‘distclean’, in modo che quella regola venga eseguita *anche* quando si dà il comando `make distclean`:

Codice 2.5: Phony target ‘distclean’ e ‘clean’

```
.PHONY: distclean clean
```

```
distclean: clean
           rm *.pdf *.dvi
```

```
clean:
           rm *.aux *.log *.out
```

2.6 Le variabili

Uno dei punti di forza dell’utilizzo di `make` per compilare documenti `LATEX` è che una volta che si possiede un `Makefile` ben organizzato per compilare un documento, con pochissime modifiche si può adattare alla compilazione di un

³Queste sono solo delle convenzioni, in uso specialmente nell’ambito della programmazione. Il redattore è libero di creare regole diverse e con nomi differenti.

altro documento, con caratteristiche simili. Ciò è reso ancora più facile dall'uso delle variabili.

Una variabile è un nome a cui è associato un *valore* che in genere è una stringa di testo. Per assegnare a una variabile il suo valore si usa la sintassi

```
VARIABILE = valore
```

Le variabili permettono di rendere il `Makefile` più compatto perché i valori delle variabili sono spesso dei lunghi elenchi di file che dovrebbero essere ripetuti più volte all'interno del file: invece di scrivere ogni volta questa lunga stringa è sufficiente richiamare il valore della variabile che sarà poi automaticamente sostituito da `make` durante la processazione del file. Inoltre quando diventa necessario modificare uno di questi elenchi, è sufficiente modificare solo una volta il valore della variabile, senza dover andare a cercare nel file tutte le occorrenze da sostituire.

Le variabili, *dopo* essere state dichiarate, possono essere referenziate usando il simbolo del dollaro seguito (senza spazi) da parentesi tonde o graffe:

```
$(VARIABILE)
${VARIABILE}
```

Per evitare di dimenticarsi di dichiarare una variabile prima di richiamarla può essere utile abituarti a definire tutte le variabili all'inizio del `Makefile`.

Le variabili possono essere referenziate in qualsiasi parte di un `Makefile`, come per esempio negli obiettivi, nei prerequisiti, nei comandi, nel valore di altre variabili. Le variabili possono rappresentare qualsiasi cosa: oltre a elenchi di file le variabili potrebbero avere come valore nomi di cartelle in cui cercare file o programmi da eseguire.

Le variabili, come qualunque altra cosa scritta nel `Makefile`, sono sensibili alle maiuscole, quindi 'Variabile', 'variabile' e 'VARIABILE' hanno significati differenti. Inoltre il nome di una variabile può essere una sequenza di qualsiasi carattere eccetto ':', '#', '=' e spazi o tabulazioni, siano essi iniziali o finali. È comunque consigliabile utilizzare per i nomi delle variabili solo lettere, numeri e trattini bassi, vedi [3], pag. 57. Eventuali caratteri di spaziatura o tabulazione presenti prima o dopo il nome di una variabile vengono ignorati, come nel Codice 2.6.

Vediamo ora un'applicazione dell'uso delle variabili. Consideriamo il caso in cui abbiamo un documento \LaTeX principale chiamato `documento.tex`, nel quale abbiamo inserito un indice analitico e una bibliografia realizzata con \BIBTeX e che l'elenco dei libri consultati si trovi nel file `bibliografia.bib`. Entrambi i file `documento.tex` e `bibliografia.bib`, inoltre, si trovano nella stessa cartella in cui è posizionato il seguente `Makefile`:⁴

Codice 2.6: Esempio di `Makefile` che utilizza le variabili

```
PRINCIPALE           = documento
PRINCIPALE_TEX      = $(PRINCIPALE).tex
PRINCIPALE_DVI      = $(PRINCIPALE).dvi
PRINCIPALE_PDF      = $(PRINCIPALE).pdf
BIBLIOGRAFIA        = bibliografia
BIBLIOGRAFIA_BIB    = $(BIBLIOGRAFIA).bib
```

⁴Parte del `Makefile` è ripreso da quello presente in [1], pag. 61.

```
FILE_CLEAN          = *.aux *.bbl *.blg *.brf *.idx \
                    *.ilg *.ind *.log
FILE_DISTCLEAN     = $(PRINCIPALE_DVI) \
                    $(PRINCIPALE_PDF)
```

```
.PHONY: dvi pdf distclean clean
```

```
dvi: $(PRINCIPALE_DVI)
```

```
pdf: $(PRINCIPALE_PDF)
```

```
$(PRINCIPALE_DVI): $(PRINCIPALE_TEX) $(BIBLIOGRAFIA_BIB)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    latex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
```

```
$(PRINCIPALE_PDF): $(PRINCIPALE_TEX) $(BIBLIOGRAFIA_BIB)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
```

```
distclean: clean
    rm $(FILE_DISTCLEAN)
```

```
clean:
    rm $(FILE_CLEAN)
```

Le variabili specificate all'inizio del file vengono sostituite da **make** quando viene invocato: tutte le occorrenze di `$(PRINCIPALE)` verranno lette dal programma come se ci fosse scritto `documento`, perciò la variabile `$(PRINCIPALE_TEX)` assume il valore `documento.tex`, e così via. Come nell'esempio del Codice 2.4, per compilare il documento in formato DVI è sufficiente dare il comando

```
make dvi
```

e per ottenere un documento PDF bisogna invece utilizzare il comando

```
make pdf
```

Nella variabile `$(FILE_CLEAN)` abbiamo indicato tutti i file che dovranno essere cancellati nella regole 'clean', analogamente la variabile `FILE_DISTCLEAN`

assume come valore i nomi dei file `documento.dvi` e `documento.pdf` che verranno rimossi se si esegue il comando `make distclean`. Notare che, come nel Codice 2.5, ‘clean’ è un prerequisito di ‘distclean’.

Quando si dovrà compilare un documento L^AT_EX che richiede gli stessi comandi del documento appena visto, si potrà facilmente riutilizzare lo stesso `Makefile`, avendo solo cura di modificare il valore delle variabili `PRINCIPALE` e `BIBLIOGRAFIA` a seconda delle necessità. Il `Makefile` va sempre posto o copiato nella cartella in cui si trova il nuovo documento.

Esistono delle cosiddette funzioni per nomi di file.⁵ Fra tutte ne ricordiamo una:

```
$(wildcard modello)
```

Al posto di `modello` bisogna inserire uno schema di nome di file, generalmente contenente un metacarattere. Per esempio, con

```
$(wildcard *.tex)
```

si indica un elenco di tutti i file, presenti nella cartella, con estensione `.tex`. Risulta particolarmente utile per ottenere un elenco di file che hanno tutti uno stesso formato o uno stesso schema nel nome.

Un ultimo strumento importante sono le funzioni per le sostituzioni di stringhe.⁶ In particolare la funzione

```
$(patsubst modello , sostituzione , testo)
```

permette di sostituire `modello` con `sostituzione` all’interno di `testo`. `modello` potrebbe contenere il metacarattere ‘%’ che rappresenta una qualsiasi sequenza di caratteri e numeri. Se anche `sostituzione` contiene la stessa sequenza indicata da ‘%’ allora la sostituzione viene eseguita. Per esempio

```
$(patsubst %.png,%.eps ,img1.png img2.jpg img3.png)
```

dà come risultato

```
img1.eps img3.eps
```

`img2.jpg` non ha lo stesso schema di `modello`, non finisce cioè per `.png`, e quindi la sostituzione non avviene, ma le altre due parole seguono lo schema del modello per cui `img1.png` e `img3.png` vengono sostituite rispettivamente con `img1.eps` e `img3.eps`.

2.7 Un Makefile ancora più complesso

Il Codice 2.6 è già un `Makefile` abbastanza elaborato e utile per automatizzare comandi piuttosto lunghi.

Quando in un documento si inseriscono delle immagini, queste devono avere un formato particolare a seconda che si compili il documento con L^AT_EX o con PDFL^AT_EX. In particolare, L^AT_EX richiede immagini in formato EPS, PDFL^AT_EX accetta immagini in formato PDF, JPG e PNG (vedi [2], pag. 93).

⁵Per un elenco esaustivo di queste funzioni vedi [3], pag. 81.

⁶Per un elenco esaustivo di queste funzioni vedi [3], pag. 78.

Se si possiede un'immagine in un formato diverso da quello necessario per l'inserimento nel documento, si deve quindi procedere alla conversione da un formato all'altro. Il programma `ImageMagick` fornisce il comando da terminale `convert`. Con la sintassi

```
convert fileinput fileoutput
```

si converte il primo file specificato, cioè `fileinput`, nel secondo file, chiamato in questo caso `fileoutput`. Per esempio supponiamo di avere nella sottocartella `Immagini` della cartella in cui si trovano il documento `LATEX` principale e il `Makefile` tutte le immagini in formato PNG. Per compilare in DVI è quindi necessario convertire tutte le immagini. Per automatizzare la compilazione e la conversione delle immagini è possibile scrivere queste regole utilizzando le funzioni apprese nel Paragrafo 2.6:

Codice 2.7: Makefile in cui le immagini PNG vengono convertite in EPS nella compilazione con `LATEX`

```
PRINCIPALE           = documento
PRINCIPALE_TEX       = $(PRINCIPALE).tex
PRINCIPALE_DVI       = $(PRINCIPALE).dvi
BIBLIOGRAFIA         = bibliografia
BIBLIOGRAFIA_BIB     = $(BIBLIOGRAFIA).bib
TUTTLLATEX           = $(PRINCIPALE_TEX) \
                      $(BIBLIOGRAFIA_BIB)

CARTELLA_IMG         = Immagini
IMMAGINI_PNG         = $(wildcard $(CARTELLA_IMG)/*.png)
IMMAGINI_EPS         = $(patsubst $(CARTELLA_IMG)/%.eps, \
                      $(CARTELLA_IMG)/%.png, $(IMMAGINI_PNG))
```

```
.PHONY: dvi immagini-eps
```

```
dvi: $(PRINCIPALE_DVI)
```

```
$(PRINCIPALE_DVI): $(TUTTLLATEX) immagini-eps
```

```
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    latex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
```

```
immagini-eps: $(IMMAGINI_EPS)
```

```
Immagini/%.eps: Immagini/%.png
    convert $^ $@
```

La variabile `IMMAGINI_PNG` contiene l'elenco di tutti i file in formato PNG presenti nella sottocartella `Immagini` (il nome della sottocartella è salvato nella variabile `CARTELLA_IMG` in modo che sarà sufficiente cambiare solo questo valore per cartelle con nomi differenti). La variabile `IMMAGINI_EPS`, invece, contiene

l'elenco dei degli stessi file in formato PNG, ma questa volta con estensione `.eps` (abbiamo utilizzato la funzione `patsubst` per eseguire la sostituzione). In questo `Makefile` è presente una regola con una definizione un po' particolare:

```
Immagini/%.eps : Immagini/%.png
    convert $^ $@
```

Il prerequisito di questa regola è un qualsiasi file della sottocartella `Immagini` con estensione `.png`, l'obiettivo, però, non è un qualsiasi file della sottocartella `Immagini` con estensione `.eps`, bensì il file che ha lo stesso nome base del prerequisito ed estensione `.eps`.⁷ Con questa regola verranno dunque generati solo file con estensione `.eps` e nome base uguale a quello di file in formato PNG presente nella sottocartella `Immagini`. Il comando eseguito nella regola è

```
convert $^ $@
```

Le due variabili `$^` e `$@` sono delle variabili dette automatiche⁸ e che hanno un significato speciale: la prima indica tutti i prerequisiti della regola (in questo caso indica solo il file PNG che ha lo stesso nome dell'obiettivo), la seconda indica l'obiettivo della regola (in questo caso l'immagine EPS da generare). Se volessimo convertire un determinato file chiamato, supponiamo, `immagine.png` in formato EPS potremmo utilizzare il comando nel terminale:

```
make immagine.eps
```

Per come è scritta la regola con obiettivo `$(PRINCIPALE_DVI)` nel `Makefile` del Codice 2.7 non è però necessario convertire una a una le immagini quando si vuole generare il documento in formato DVI. Infatti, compilando con il comando

```
make dvi
```

il programma `make` si preoccupa di verificare se tutti i prerequisiti elencati in questa regola sono aggiornati. Uno di essi è `immagini-eps` che è un phony target: `immagini-eps` ha il compito di convertire tutte le immagini in formato PNG presenti nella sottocartella `Immagini` nel formato EPS, adatto per l'inclusione di immagini nel documento finale DVI.

Riportiamo ora un `Makefile` ancora più complesso di quelli visti fino ad adesso. Immaginiamo di avere in una cartella il file `documento.tex` come file \LaTeX principale, `bibliografia.bib` come raccolta dei riferimenti bibliografici. Nelle sottocartelle `Capitolo1` e `Capitolo2` sono presenti vari file `.tex` inclusi in `documento.tex`; nella sottocartella `Immagini`, invece, sono presenti immagini nel solo formato PNG che saranno eventualmente convertite nel formato EPS nel caso di compilazione con \LaTeX . Un `Makefile` per compilare questo progetto potrebbe apparire così:

```
PRINCIPALE           = documento
PRINCIPALE_TEX      = $(PRINCIPALE).tex
PRINCIPALE_DVI      = $(PRINCIPALE).dvi
PRINCIPALE_PDF      = $(PRINCIPALE).pdf
CAPITOLO1_TEX       = $(wildcard Capitolo1/*.tex)
CAPITOLO2_TEX       = $(wildcard Capitolo2/*.tex)
```

⁷Il metacarattere '%' è stato introdotto nel Paragrafo 2.6.

⁸Per un elenco esaustivo di queste variabili vedi [3], pag. 110.

```

BIBLIOGRAFIA           = bibliografia
BIBLIOGRAFIA_BIB       = $(BIBLIOGRAFIA).bib
TUTTLLATEX             = $(PRINCIPALE_TEX) \
                        $(BIBLIOGRAFIA_BIB) \
                        $(CAPITOLO1_TEX) \
                        $(CAPITOLO2_TEX)

FILE_CLEAN             = *.aux *.bbl *.blg *.brf *.idx \
                        *.ilg *.ind *.log
FILE_DISTCLEAN         = $(PRINCIPALE_DVI) \
                        $(PRINCIPALE_PDF) \
                        $(IMMAGINI_EPS)

IMMAGINI_PNG           = $(wildcard Immagini/*.png)
IMMAGINI_EPS           = $(patsubst Immagini/%.eps,\
                        Immagini/%.png, $(IMMAGINI_PNG))

```

```
.PHONY: dvi pdf distclean clean immagini-eps
```

```
dvi: $(PRINCIPALE_DVI)
```

```
pdf: $(PRINCIPALE_PDF)
```

```
$(PRINCIPALE_DVI): $(TUTTLLATEX) immagini-eps
```

```

    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    latex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)

```

```
$(PRINCIPALE_PDF): $(TUTTLLATEX)
```

```

    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)

```

```
immagini-eps: $(IMMAGINI_EPS)
```

```

Immagini/%.eps: Immagini/%.png
    convert $^ $@

```

```

distclean: clean
    rm $(FILE_DISTCLEAN)

```

```
clean:
```

```
rm $(FILE_CLEAN)
```

Con gli strumenti acquisiti fino a questo punto dovrebbe essere comprensibile come funziona questo `Makefile`, cosa compie ciascuna regola e quali comandi bisogna dare nel terminale per compilare il documento.

I `Makefile` dei programmi spesso raggiungono svariate migliaia di righe di codice, per quanto riguarda un documento `LATEX` piuttosto complesso ne possono bastare anche poche decine. Si possono comunque realizzare dei `Makefile` molto più lunghi e avanzati di quelli visti in questa guida. In Internet è possibile trovare molta documentazione riguardo ai `Makefile` (in particolare però per compilare programmi). Una guida sicuramente importante è la più volte citata *GNU Make*, di Richard M. Stallman, Roland McGrath e Paul D. Smith che si può trovare all'indirizzo <http://www.gnu.org/software/make/manual/make.pdf>.

Bibliografia

- [1] Luca Caucci and Mariano Spadaccini. *Gestione di Figure e Tabelle con \LaTeX* , 2005. http://www.guit.sssup.it/downloads/fig_tut.pdf. (Citato alle pagine 6 e 8.)
- [2] Lorenzo Pantieri. *L'arte di scrivere con \LaTeX* , 2009. http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf. (Citato alle pagine 3, 5 e 10.)
- [3] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*, 2006. <http://www.gnu.org/software/make/manual/make.pdf>. (Citato alle pagine 1, 3, 4, 6, 8, 10 e 12.)