

Introduzione ai Makefile
per compilare documenti L^AT_EX

Mosè Giordano

17 aprile 2012

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.

È possibile scaricare un archivio compresso contenente una copia del codice sorgente di quest'opera all'URL <https://github.com/giordano/makefile-latex/tarball/master>. Chi fosse interessato a contribuire al progetto, all'URL <https://github.com/giordano/makefile-latex> può trovare il repository `git` in cui viene gestito lo sviluppo.

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 Makefile	5
2.4	Un Makefile un po' più elaborato	5
2.5	Phony target	7
2.6	Le variabili	8
2.6.1	Usò delle variabili nelle regole	11
2.6.2	Sostituzione di comando	11
2.7	Un Makefile ancora più complesso	12
2.8	Conclusione	15
	Riferimenti bibliografici	17

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. Nella voce di Wikipedia in lingua italiana di `make` (<http://it.wikipedia.org/wiki/Make>) possiamo leggere la seguente descrizione:

Il **make** è un'utility [...] che automatizza il processo di creazione di file che dipendono da altri file, risolvendo le dipendenze e invocando programmi esterni per il lavoro necessario.

`make` è molto usata nella compilazione dei programmi, specie nell'ambiente del software libero, 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

¹In realtà potrebbe avere anche altri nomi, questo è quello consigliato, vedi Stallman, McGrath e Smith [3, pagina 12].

```
apt-get install make
```

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

```
yum install make
```

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 Stallman, McGrath e Smith [3, pagina 3]):

Codice 2.1: Struttura di una regola.

```
obiettivo: prerequisiti
—————>comandi
—————>...
—————>...
```

L'*obiettivo* (in inglese *target*), ciò che viene scritto prima dei due punti `:`, di norma è il nome del file che verrà generato con la regola descritta. Vedremo nel paragrafo 2.5 in che modo sia possibile usare un obiettivo che non sia il nome di un file da creare eseguendo la corrispondente regola.

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 proprio nome. Le regole possono non avere alcun prerequisito.

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

Per scrivere correttamente un **Makefile** risulta utile disegnarsi un grafo ad albero che illustri le relazioni che ci sono fra i vari file, individuando quali dovranno essere gli obiettivi delle regole e quali saranno i prerequisiti delle regole. Uno stesso file può svolgere il ruolo di prerequisito in una regola e di obiettivo in un'altra. Nell'esempio della figura 2.1 ci sono due obiettivi, **obiettivo1** e **prerequisito2**, per le quali verranno scritte le opportune regole. I prerequisiti di **obiettivo1** sono i file **prerequisito1** e **prerequisito2**, l'unico prerequisito di **prerequisito2** è il file **prerequisito3**.

Un **Makefile** deve contenere almeno una regola, ma possono essercene anche più di una, ciascuna corrispondente a un diverso file da creare, e l'ordine con cui le regole compaiono nel **Makefile** *non* è importante.

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

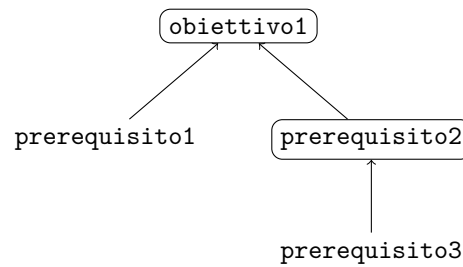


Figura 2.1: Grafo ad albero che illustra le dipendenze fra i file. I file che si trovano alla punta di una freccia, evidenziati in un rettangolo, costituiscono l'obiettivo di una regola, i file che invece si trovano alla coda di una freccia rappresentano i prerequisiti della corrispondente regola e sono necessari per la creazione dell'obiettivo.

simbolo ha lo stesso comportamento del simbolo `%` nel linguaggio `LATEX`, vedi Pantieri e Gordini [2, pagina 26].

Se lo si desidera è possibile suddividere una riga di codice molto lunga su più righe 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

Per eseguire la regola che ha per obiettivo il file `<obiettivo>` bisogna invocare `make` da terminale con il comando

```
make <obiettivo>
```

`make` verifica se è necessario aggiornare l'obiettivo della regola corrispondente in questo modo: se l'obiettivo indicato da linea di comando 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 e sul terminale si leggerà il messaggio

```
make: Nessuna operazione da eseguire per "<obiettivo>".
```

Le istruzioni per aggiornare un obiettivo sono presenti nei *comandi*: questi sono passati alla shell, normalmente la `sh`.

Come detto, uno dei prerequisiti di una regola può essere l'obiettivo di un'altra regola. In questo caso `make` verifica se, prima di eseguire una regola, è necessario aggiornare uno o più dei file elencati fra i prerequisiti. Nel seguente `Makefile`, corrispondente alla situazione illustrata nella figura 2.1,

```

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, eseguendo il `comando3` specificato, il file `prerequisito2` prima di eseguire la regola associata all'`obiettivo1`.

Tutta la logica di funzionamento di `make` si basa sul fatto che 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.

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

```
make
```

esso procede all'esecuzione della prima regola che trova nel `Makefile`, per la precisione la prima regola il cui target non inizia con un punto `.` (questo comportamento può essere modificato, vedi Stallman, McGrath e Smith [3, pagina 5]).

Se si vuole forzare l'esecuzione delle regola `<obiettivo>`, ignorando le date di modifica dell'obiettivo e dei suoi prerequisiti, bisogna utilizzare l'opzione `-B` in questo modo:

```
make -B <obiettivo>
```

2.3 Un semplice Makefile

Passiamo ora alla pratica vedendo un esempio di `Makefile` molto semplice per compilare un documento `LATEX`:

Codice 2.2: Un semplice `Makefile`.

```
documento.dvi: documento.tex
    latex documento
```

Nel codice 2.2, il file `documento.dvi` è l'*obiettivo*, cioè il file ottenuto dopo la compilazione, 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 con il comando `cd` nella cartella in cui si trova il `Makefile`, bisogna utilizzare il comando

```
make obiettivo
```

sostituendo a `obiettivo` il nome dell'obiettivo che specificato nella regola che si desidera eseguire. In questo caso dovremo eseguire il comando

```
make documento.dvi
```

oppure semplicemente `make` se la regola è la prima in ordine di apparizione nel `Makefile`.

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 Pantieri e Gordini [2, pagina 120]) eseguire i comandi

```
latex documento
bibtex 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
```

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 \PDF\LaTeX , 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 \PDF\LaTeX .

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

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

In alcune distribuzioni GNU/Linux sono presenti gli script `texi2dvi` e `texi2pdf` che eseguono rispettivamente \LaTeX e \PDF\TeX (e, se necessario, \BibTeX) sul sorgente il numero strettamente necessario di volte per la corretta compilazione del documento (vedi Caucci e Spadaccini [1, pagina 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
```

Anche lo script `latexmk`, fornito dalle distribuzioni TeX Live e MikTeX, offre funzionalità simili, bisogna però evidenziare che se il documento richiede comandi particolari

per la compilazione (come quando si utilizza il pacchetto `frontespazio`), `texi2dvi`, `texi2pdf` e `latexmk` 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 -f *.aux *.log *.out
```

vengono cancellati tutti i file con estensione `.aux`, `.log` e `.out` che vengono prodotti durante la compilazione di un semplice documento L^AT_EX.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. Infatti, dal momento che la regola non ha prerequisiti, il file `clean` risulterebbe sempre aggiornato (vedi Stallman, McGrath e Smith [3, pagina 31]). Per fare ciò bisogna mettere gli obiettivi di queste regole come prerequisiti della regola speciale `.PHONY`:

```
.PHONY: clean
clean:
    rm -f *.aux *.log *.out
```

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 si potrà eseguire il comando

```
make dvi
```

Analogamente, per ottenere il file PDF si potrà dare il comando

```
make pdf
```

¹Il comando `rm` cancella tutti i file che vengono elencati di seguito, l'opzione `-f` serve per non stampare eventuali messaggi di errore in caso di assenza dei file da cancellare. Il metacarattere `*` sostituisce una qualsiasi sequenza di caratteri.

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 L^AT_EX).² 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 -f *.pdf *.dvi

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

2.6 Le variabili

Uno dei punti di forza dell'utilizzo di `make` per compilare documenti L^AT_EX 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 altro documento, strutturato in maniera simile. 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 abituartsi 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.

²Queste sono solo delle convenzioni, in uso specialmente nell'ambito della programmazione. L'utente è libero di creare regole diverse e con nomi differenti.

Le variabili, come qualunque altra cosa scritta nel `Makefile`, sono sensibili alle maiuscole, quindi `Variabile`, `variabile` e `VARIABLE` sono stringhe distinte. Inoltre il nome di una variabile può essere una sequenza di qualsiasi carattere eccetto spazi o tabulazioni, siano essi iniziali o finali, o uno fra i tre seguenti simboli : # =. È comunque consigliabile utilizzare per i nomi delle variabili solo lettere, numeri e trattini bassi, vedi Stallman, McGrath e Smith [3, pagina 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`.³

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

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

distclean: clean
    rm -f $(FILE_DISTCLEAN)

clean:

```

³Parte del `Makefile` è ripreso da quello presente in Caucci e Spadaccini [1, pagina 61].

```
rm -f $(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 regola `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 \LaTeX 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.

Esistono delle cosiddette funzioni per nomi di file.⁴ 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.⁵ 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)
```

viene interpretato da `make` come

```
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`.

⁴Per un elenco esaustivo di queste funzioni vedi Stallman, McGrath e Smith [3, pagina 83].

⁵Per un elenco esaustivo di queste funzioni vedi Stallman, McGrath e Smith [3, pagina 80].

2.6.1 Uso delle variabili nelle regole

Abbiamo imparato che per richiamare una variabile definita all'interno del `Makefile` bisogna usare la sintassi `$(NOME_DELLA_VARIABILE)`. Qualche volta, però, potremmo voler richiamare una variabile della shell. Nella maggior parte delle shell Unix questo si fa usando proprio il metacarattere `$`, però per far capire a `make` che in questo caso vogliamo una variabile della shell e non del `Makefile` dobbiamo raddoppiare il simbolo di dollaro `$`: `$$`.

Nel seguente esempio, tratto da Stallman, McGrath e Smith [3, pagina 43],

```
ELENCO = uno due tre
foo:
    for i in $(ELENCO); do \
        echo $$i; \
    done
```

la variabile `ELENCO` è stata richiamata normalmente con `$(ELENCO)` poiché è una variabile del `Makefile`, mentre la variabile `i` del `for` è stata richiamata con `$$i` poiché è una variabile della shell.

Osserviamo che in un `Makefile` i comandi che devono essere eseguiti in una regola vanno scritti, normalmente, uno su ciascuna riga. Anche se generalmente i cicli `for` delle shell Unix sono scritti nella forma

```
for <variabile> in <elenco>; do
    comandi
    ...
    ...
done
```

si tratta in realtà di *un unico* comando e quindi nel `Makefile` andrebbe scritto su un'unica riga in questo modo

```
for <variabile> in <elenco>; do comandi ; ... ; ... ; done
```

Per rendere il codice più leggibile si può effettuare l'escape del carattere di nuova linea (vedi paragrafo 2.1) come fatto nell'esempio precedente. Bisogna prendere questo accorgimento anche quando in una regola si vogliono eseguire gli altri tipi di cicli e verifiche condizionali delle shell quali `if`, `while` e `until`.

2.6.2 Sostituzione di comando

Nella principali shell Unix si può effettuare la sostituzione di comando con la sintassi `'...'` e questa può essere usata anche all'interno di un `Makefile`. Inoltre nella `bash` e nella `ksh` la sostituzione di comando può essere eseguita con la sintassi `$(...)` che però nei `Makefile` abbiamo visto che si usa per richiamare il valore delle variabili. La sostituzione di comando può essere eseguita, invece, sfruttando la funzione `shell` in questo modo:

```
$(shell <comando>)
```

sostituendo a `<comando>` l'effettivo comando da eseguire. Il comando viene passato alla shell impostata con la variabile `SHELL` che in maniera predefinita è la `sh`.

Per esempio, se si vuole impostare una variabile che sia uguale alla data attuale si può usare una delle due seguenti alternative

```
ORA = `date "+%Y%m%d-%H%M%S" `
ORA = $(shell date "+%Y%m%d-%H%M%S ")
```

L'argomento di `date` può, naturalmente, essere cambiato in modo da adattarlo alle proprie esigenze.

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 \LaTeX o con \PDF\LaTeX . In particolare, \LaTeX richiede immagini in formato EPS, \PDF\LaTeX accetta immagini in formato PDF, JPG e PNG (vedi Pantieri e Gordini [2, pagina 105]). 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`. Il formato dei due file, di input e output, viene riconosciuto dalle estensioni dei file. Per esempio, il comando

```
convert nomefile.eps nomefile.pdf
```

converte il file `nomefile.eps`, in formato EPS, nel file `nomefile.pdf`, in formato PDF. Supponiamo di avere nella sottocartella `Immagini` della cartella in cui si trovano il documento \LaTeX principale e il `Makefile` tutte le immagini in formato PDF. 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 PDF vengono convertite in EPS nella compilazione con \LaTeX .

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

CARTELLA_IMG         = Immagini
IMMAGINI_PDF         = $(wildcard $(CARTELLA_IMG)/*.pdf)
IMMAGINI_EPS         = $(patsubst $(CARTELLA_IMG)/%.eps,\
                      $(CARTELLA_IMG)/%.pdf, $(IMMAGINI_PDF))

.PHONY: dvi immagini-eps

dvi: $(PRINCIPALE_DVI)

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

```

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

immagini-eps: $(IMMAGINI_EPS)

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

```

La variabile `IMMAGINI_PDF` contiene l'elenco di tutti i file in formato PDF 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 PDF, 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/%.pdf
    convert $^ $@

```

Il prerequisito di questa regola è un qualsiasi file della sottocartella `Immagini` con estensione `.pdf`, 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`.⁶ Con questa regola verranno dunque generati solo file con estensione `.eps` e nome base uguale a quello di file in formato PDF presente nella sottocartella `Immagini`. Il comando eseguito nella regola è

```

convert $^ $@

```

Le due variabili `$^` e `$@` sono delle variabili dette automatiche⁷ e che hanno un significato speciale: la prima indica tutti i prerequisiti della regola (in questo caso indica solo il file PDF 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.pdf` 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 PDF presenti nella sottocartella `Immagini` nel formato EPS, adatto per l'inclusione di immagini nel documento finale DVI.

⁶Il metacarattere `%` è stato introdotto nel paragrafo 2.6.

⁷Per un elenco esaustivo di queste variabili vedi Stallman, McGrath e Smith [3, pagina 112].

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. Nella sottocartella `Capitoli` sono presenti vari file `.tex` inclusi in `documento.tex`; nella sottocartella `Immagini`, invece, sono presenti immagini nel solo formato PDF che saranno eventualmente convertite nel formato EPS nel caso di compilazione con \LaTeX . Un `Makefile` per compilare questo progetto potrebbe apparire così:

Codice 2.8: Un `Makefile` complesso.

```

PRINCIPALE                = documento
PRINCIPALE_TEX            = $(PRINCIPALE).tex
PRINCIPALE_DVI            = $(PRINCIPALE).dvi
PRINCIPALE_PDF            = $(PRINCIPALE).pdf
CAPITOLI_TEX              = $(wildcard Capitoli/*.tex)
BIBLIOGRAFIA              = bibliografia
BIBLIOGRAFIA_BIB          = $(BIBLIOGRAFIA).bib
TUTTI_LATEX               = $(PRINCIPALE_TEX) \
                           $(BIBLIOGRAFIA_BIB) \
                           $(CAPITOLI_TEX)

IMMAGINI_PDF              = $(wildcard Immagini/*.pdf)
IMMAGINI_EPS              = $(patsubst Immagini/%.eps,\
                           Immagini/%.pdf, $(IMMAGINI_PDF))
FILE_CLEAN                = *.aux *.bbl *.blg *.brf *.idx \
                           *.ilg *.ind *.log
FILE_DISTCLEAN            = $(PRINCIPALE_DVI) \
                           $(PRINCIPALE_PDF) \
                           $(IMMAGINI_EPS)

.PHONY: dvi pdf distclean clean immagini-eps

dvi: $(PRINCIPALE_DVI)

pdf: $(PRINCIPALE_PDF)

$(PRINCIPALE_DVI): $(TUTTI_LATEX) immagini-eps
    latex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    latex $(PRINCIPALE)
    latex $(PRINCIPALE)

$(PRINCIPALE_PDF): $(TUTTI_LATEX) $(IMMAGINI_PDF)
    pdflatex $(PRINCIPALE)
    bibtex $(PRINCIPALE)
    makeindex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)
    pdflatex $(PRINCIPALE)

immagini-eps: $(IMMAGINI_EPS)

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

distclean: clean

```

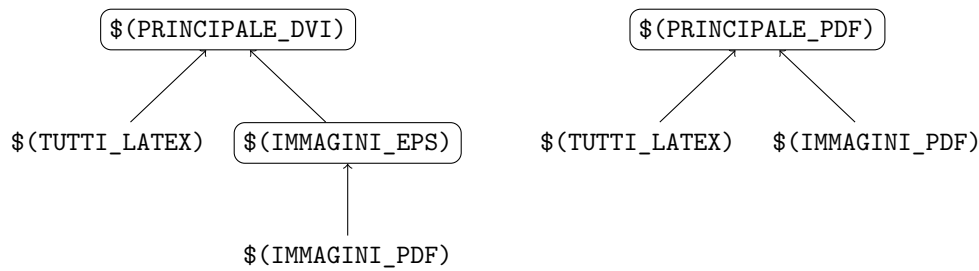


Figura 2.2: Grafo ad albero che rappresenta le dipendenze fra i file del `Makefile` del codice 2.8, eccetto i phony target.

```

    rm -f $(FILE_DISTCLEAN)

clean:
    rm -f $(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. Per maggiore chiarezza, nella figura 2.2 è rappresentato il grafo ad albero che descrive le dipendenze fra i file del `Makefile` mostrato nel codice 2.8.

2.8 Conclusione

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>.

Riferimenti bibliografici

- [1] Luca Caucci e Mariano Spadaccini. *Gestione di Figure e Tabelle con L^AT_EX*. 2005.
URL: http://www.guit.sssup.it/downloads/fig_tut.pdf (citato alle pagine 6, 9).
- [2] Lorenzo Pantieri e Tommaso Gordini. *L'arte di scrivere con L^AT_EX*. Feb. 2012.
URL: http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf (citato alle pagine 4, 5, 12).
- [3] Richard M. Stallman, Roland McGrath e Paul D. Smith. *GNU Make*. Lug. 2010.
URL: <http://www.gnu.org/software/make/manual/make.pdf> (citato alle pagine 1, 3, 5, 7, 9–11, 13).