Mosè Giordano

```
target: dependencies ...
        commands
        ...
```

Introduzione ai Makefile per compilare documenti LaTeX

v.1.0 del 2012/12/19

Note su questa guida

Quest'opera è distribuita con licenza Creative Commons Attribuzione 3.0 Unported. Un riassunto in linguaggio accessibile a tutti della Licenza in versione 3.0 è reperibile all'indirizzo 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
- di usare quest'opera per fini commerciali

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.

Colophon

Lo sviluppo di questo documento è gestito mediante un repository Git disponibile all'indirizzo https://github.com/giordano/makefile-latex. È possibile scaricare un archivio compresso con il codice sorgente di questa guida all'indirizzo https://github.com/giordano/makefile-latex/tarball/master.

Introduzione

La compilazione di un semplice documento LaTeX richiede l'esecuzione di uno o due comandi, che generalmente possono essere facilmente eseguiti all'interno del proprio editor di testo preferito. All'aumentare delle dimensioni del sorgente si può decidere di suddividere il sorgente in differenti file di dimensioni più piccole, magari ciascuno contenete il codice di un intero capitolo, e più facili da gestire. Anche questa situazione non costituisce normalmente un grosso problema perché i più diffusi editor di testo sono in grado di gestire automaticamente progetti di questo tipo. L'utilizzo di alcuni pacchetti richiede, però, l'esecuzione di comandi particolari non accessibili all'interno dell'editor, oppure potrebbe esserci bisogno di eseguire dei comandi da terminale su file non legati a LaTeX. Per risolvere alcuni di questi problemi sono stati sviluppati recentemente dei programmi appositi, come `latexmk`[1] e `arara`.[2] In questa guida, invece, ci occuperemo di un programma creato nel 1977 da Stuart Feldman: `make`. Nella voce di Wikipedia in lingua italiana di `make` (https://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 usato nella compilazione dei programmi scritti in C o C++, specie nell'ambiente del software libero, ma può essere utilmente sfruttato per automatizzare e semplificare anche la compilazione di complessi

[1] Per maggiori informazioni su questo programma visita il sito http://www.phys.psu.edu/~collins/software/latexmk-jcc/.

[2] Per maggiori informazioni su questo programma visita il sito http://cereda.github.com/arara/.

documenti LaTeX. Rispetto a software come `arara`, `make` permette di svolgere, su richiesta, altre operazioni quali la cancellazione di file temporanei non necessari e la creazione di un archivio compresso per conservare i file sorgenti.[3] Non ci sono limiti alla fantasia.

Di `make` esistono alcune versioni, le più famose sono `GNU make`, sviluppata da Richard Stallman e Roland McGrath, e `BSD make`. Questa guida è incentrata su `GNU make`, il dialetto presente nei sistemi GNU/Linux e Mac OS X. Questo programma si utilizza da linea di comando, chi non dovesse avere familiarità con il terminale può leggere la guida di GIACOMELLI (2012). Il manuale di riferimento per `GNU make`, a cui si rimanda per qualsiasi approfondimento, è STALLMAN *et al.* (2010), ma è orientato soprattutto a programmatori che sviluppano software nel linguaggio C, la presente guida vuole essere invece una semplice introduzione a `make` per gli utenti di LaTeX.

Non si vuole convincere nessuno che `make` sia il programma migliore in assoluto, perché sicuramente *non* lo è, ha i suoi pregi e anche qualche difetto. Qui verranno solo illustrate le sue funzioni e potenzialità fornendo anche degli esempi d'uso, starà poi al lettore decidere se `make` si adatta alle proprie necessità.

Sono ben accetti suggerimenti per migliorare la guida, segnalazioni di errori e richieste di chiarimenti.

Mosè Giordano
`giordano dot mose at libero dot it`

[3] `arara` permette di eseguire qualsiasi operazione, però queste vengono eseguite tutte, ogni volta che viene invocato `arara`, non è possibile scegliere selettivamente quali operazioni svolgere.

Indice

Note su questa guida ii

Introduzione iii

Indice v

1 Concetti di base 1
- 1.1 Installazione di `make` . . . 1
- 1.2 Le regole . . . 2
- 1.3 Come funziona `make` . . . 3
- 1.4 Un semplice `Makefile` . . . 5
- 1.5 Phony targets . . . 8
- 1.6 Le variabili . . . 10
  - 1.6.1 Uso delle variabili nelle regole . . . 14
  - 1.6.2 Sostituzione di comando . . . 15

2 Esempi d'uso 17
- 2.1 Convertire immagini PDF in formato EPS . . . 17
- 2.2 Creare un archivio compresso . . . 21
- 2.3 Includere immagini esterne da compilare . . . 23

3 Integrare `make` negli editor di testo 26
- 3.1 Emacs . . . 27
- 3.2 Kile . . . 27
- 3.3 Texmaker . . . 27
- 3.4 TeXstudio . . . 28
- 3.5 Texworks . . . 28

Bibliografia 30

Concetti di base 1

1.1 Installazione di make

Per verificare se make è già presente nel proprio sistema si può eseguire in un terminale il seguente comando

```
$ make --version
```

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

Per ottenere make in ambiente GNU/Linux 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 amministratore. In Fedora e derivate, invece, bisogna eseguire il comando

```
# yum install make
```

Fino a qualche anno fa, make era preinstallato sui sistemi Mac OS X. Per installare make nei sistemi Mac OS X sui quali non è già presente, si deve installare il software Xcode, disponibile nel Mac App Store, e poi scaricare gli strumenti a linea di comando dalle preferenze del programma.

Il programma di installazione di make per Windows può essere scaricato all'indirizzo http://gnuwin32.sourceforge.net/packages/make.htm. make è fornito anche insieme a Cygwin (Url: http://cygwin.com/index.html).

1.2 Le regole

Il programma `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. Un `Makefile` basilare è composto essenzialmente da *regole* (chiamate in inglese *rules*) che hanno la seguente struttura:

Codice 1.1 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.

I *prerequisiti* (in inglese *prerequisites*) 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 ogni regola sono scritti l'uno sotto l'altro nell'ordine con cui devono essere eseguiti e ciascuna riga *deve necessariamente* iniziare con una tabulazione (nel codice 1.1 è evidenziata dalla freccia ————→), non con spazi altrimenti verrà segnalato un errore. `make` invoca una nuova subshell per ogni riga dei comandi, per fare in modo che venga eseguita una sola subshell per tutto l'elenco dei comandi vedi (Stallman *et al.*, 2010, pagina 44).

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.

[1] In realtà potrebbe avere anche altri nomi, questo è quello consigliato e riconosciuto in maniera predefinita, per ulteriori informazioni vedi (Stallman *et al.*, 2010, pagina 12).

Un `Makefile` può contenere dei commenti, introdotti dal simbolo cancelletto `#`: tutto ciò che si trova alla destra di `#` sulla sua stessa riga viene trattato come commento. Questo simbolo ha lo stesso comportamento del simbolo `%` nel linguaggio LaTeX.

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 obbligatorio in quanto non sono posti limiti alla lunghezza delle righe di un `Makefile`, però può aiutare la leggibilità del codice.

Nota Da qui in poi 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.

1.3 Come funziona make

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.

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`.

Poiché tutta la logica di funzionamento di `make` è basata sulle dipendenze fra i file, per scrivere correttamente un `Makefile` risulta utile disegnare

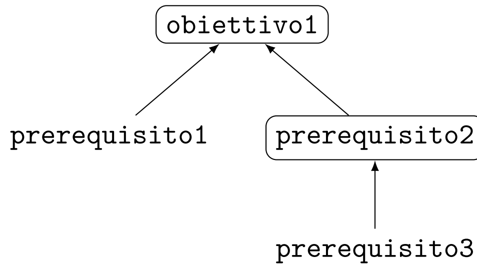

Figura 1.1 Grafo ad albero che illustra le dipendenze fra i file. I file che si trovano alla punta di una freccia, evidenziati in un rettangolo, costituiscono l'obiettivo di una regola, i file che invece si trovano alla coda di una freccia rappresentano i prerequisiti della corrispondente regola e sono necessari per la creazione dell'obiettivo.

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 1.1 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`. I file `prerequisito1` e `prerequisito3` sono modificati "manualmente" dall'utente e non generati automaticamente con un programma esterno a partire da altri file, quindi non è necessario associare a questi due file delle corrispondenti regole.

Prima di eseguire una regola `make` verifica se è necessario aggiornare uno o più dei file elencati fra i prerequisiti e in questo caso esegue automaticamente le eventuali regole associate ai prerequisiti. Nel seguente `Makefile`, corrispondente alla situazione illustrata nella figura 1.1,

```
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 il file `prerequisito2`, eseguendo in maniera automatica il `comando3` specificato, prima di eseguire la regola `obiettivo1`. Uno dei pregi di `make` è che esso si occupa autonomamente di eseguire tutte le regole che servono per preparare i prerequisiti dell'obiettivo che si vuole creare, `obiettivo1` nell'esempio. Per la buona riuscita di questo processo è cruciale la corretta e completa definizione delle dipendenze fra i file.

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 *et al.*, 2010, pagina 5)). Dunque è preferibile scrivere come prima regola quella che si intende utilizzare più spesso, per esempio quella per compilare il documento completo.

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⟩
```

1.4 Un semplice Makefile

Passiamo ora alla pratica vedendo un esempio di `Makefile` molto semplice per compilare un documento LaTeX. Supponiamo che il sorgente sia interamente contenuto nel solo file `documento.tex` e che vogliamo generare il file di output in formato DVI. Se il documento non contiene bibliografia, indice, indice analitico, ecc., per fare questo è sufficiente eseguire il comando

```
$ latex documento
```

dopo essersi posizionati, usando il comando `cd`, nella cartella in cui si trova il file `documento.tex`. Il `Makefile` che descrive questa regola è il seguente

Codice 1.2 Un semplice `Makefile`.

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

Questo `Makefile` deve essere posto nella cartella in cui si trova il file `documento.tex`. Nel codice 1.2, 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 generare il file di output `documento.dvi` bisogna eseguire il comando

```
$ make documento.dvi
```

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

Se nel file `documento.tex` è presente una bibliografia realizzata con BibTeX, per compilare il documento è necessario 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 un'operazione noiosa. Nel `Makefile` si può scrivere la seguente regola

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

in cui i prerequisiti sono il file principale `documento.tex` e il file in cui abbiamo scritto la nostra base di dati dei riferimenti bibliografici. Questi file sono modificati manualmente dall'autore del documento, non devono essere ricreati da programmi esterni, quindi non sono presenti regole associate a questi prerequisiti. Per compilare il documento è sufficiente dare il comando

```
$ make documento.dvi
```

oppure solo `make` qualora quella regola fosse la prima presente nel `Makefile`. `make` eseguirà la regola solo se almeno uno dei prerequisiti è stato modificato dopo l'ultima modifica al file obiettivo, a meno di usare l'opzione `-B` illustrata in precedenza.

Si può aggiungere 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 1.3 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

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

Eseguendo `make` in un terminale, viene scritto nel terminale il comando che si sta eseguendo, seguito dall'output di quel comando. Se non si vuole che il comando venga scritto nel terminale, nel `Makefile` deve essere preceduto dalla chiocciola , per esempio

```
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 PDFTeX (e, se necessario, BibTeX) sul sorgente il numero strettamente necessario di volte per la corretta compilazione del documento (vedi (Caucci e Spadaccini, 2005, pagina 63)). Utilizzando questi due comandi il codice 1.3 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 `frontespizio`), `texi2dvi`, `texi2pdf` e `latexmk` non sono più in grado di generare correttamente il documento finale, a meno di istruirli opportunamente.

1.5 Phony targets

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 targets 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 LaTeX.[2] È consigliabile specificare esplicitamente quali sono i phony targets 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.[3] Per fare ciò bisogna mettere gli obiettivi di queste regole come prerequisiti della regola speciale `.PHONY`:

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

[2] Il comando `rm` cancella tutti i file che vengono elencati di seguito, l'opzione `-f` serve per ignorare l'eventuale assenza dei file da cancellare. Il metacarattere `*` sostituisce una qualsiasi sequenza di caratteri.

[3] Vedi (Stallman *et al.*, 2010, pagina 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 targets possono essere anche usati per creare una sorta di *alias* di altre regole. Per esempio, inserendo il seguente

Codice 1.4 I prerequisiti della regola dell'obiettivo .PHONY sono i nomi dei phony targets che vengono successivamente specificati.

```
.PHONY: dvi pdf

dvi: documento.dvi

pdf: documento.pdf
```

in un Makefile, prima del codice 1.3, 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
```

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).[4] 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 1.5 Phony targets distclean e clean.

```
.PHONY: distclean clean
```

[4]Queste sono solo delle convenzioni, diffuse in particolare nell'ambito della programmazione. L'utente è libero di creare regole diverse e con nomi differenti.

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

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

1.6 Le variabili

Un evidente svantaggio dei `Makefile` è che essi devono essere completamente riscritti per ogni nuovo progetto. Questo difetto può essere superato perché 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 (cioè con una simile struttura delle dipendenze fra i file), grazie all'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

⟨*nome della 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) dal nome della variabile racchiuso fra parentesi tonde o graffe: `$(`⟨*nome della variabile*⟩`)` oppure `${`⟨*nome della variabile*⟩`}`. Per evitare di dimenticarsi di dichiarare una variabile prima di richiamarla può essere utile abituarsi 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` 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.[5] Eventuali caratteri di spaziatura o tabulazione presenti prima o dopo il nome di una variabile vengono ignorati, come nel codice 1.6.

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 riferimenti bibliografici 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`:[6]

Codice 1.6 Esempio di `Makefile` che utilizza le variabili.

```
PRINCIPALE               = documento
PRINCIPALE_TEX           = $(PRINCIPALE).tex
PRINCIPALE_DVI           = $(PRINCIPALE).dvi
PRINCIPALE_PDF           = $(PRINCIPALE).pdf
BIBLIOGRAFIA             = 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)
        latex $(PRINCIPALE)
```

[5] Vedi (Stallman *et al.*, 2010, pagina 57).

[6] Parte del `Makefile` è ripreso da quello presente in (Caucci e Spadaccini, 2005, pagina 61).

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

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

distclean: clean
        rm -f $(FILE_DISTCLEAN)

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

Le variabili definite 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 1.4, 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`, 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 1.5, `clean` è un prerequisito di `distclean`.

Quando si dovrà compilare un altro documento LaTeX che richiede gli stessi comandi del documento appena visto, si potrà facilmente utilizzare lo stesso `Makefile`, dopo averlo posto nell'opportuna cartella, avendo solo cura di modificare il valore delle variabili `PRINCIPALE` e `BIBLIOGRAFIA` per adattarlo alle proprie necessità.

Esistono delle cosiddette *funzioni per nomi di file.*[7] 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. Negli esempi precedenti abbiamo usato il metacarattere `*` senza bisogno di usare la funzione `wildcard` perché l'espansione dei metacaratteri nel nome dell'obiettivo e nei prerequisiti è svolta automaticamente da `make` e all'interno dei comandi è svolta dalla shell. In tutti gli altri contesti (compresa la definizione delle variabili) l'espansione dei metacaratteri avviene solo richiedendola con la funzione `wildcard`. Consideriamo il seguente `Makefile` di esempio

```
VAR1 = *.tex
VAR2 = $(wildcard *.tex)

.PHONY: prova

prova:
	@echo "Valore di 'VAR1': $(VAR1)"
	@echo "Valore di 'VAR2': $(VAR2)"
	@echo "La shell interpreta 'VAR1' come" $(VAR1)
	@echo "La shell interpreta 'VAR2' come" $(VAR2)
```

L'output della regola fittizia `prova` è qualcosa del tipo

```
Valore di 'VAR1': *.tex
Valore di 'VAR2': make.tex
La shell interpreta 'VAR1' come make.tex
La shell interpreta 'VAR2' come make.tex
```

Solo la variabile `VAR2` è stata espansa già nella sua definizione, poiché fa uso della funzione `wildcard`, comunque la shell interpreta le due variabili

[7] Per un elenco esaustivo di queste funzioni vedi (Stallman *et al.*, 2010, pagina 83).

allo stesso modo poiché espande il metacarattere `*`. È preferibile usare la funzione `wildcard` quando si vuole che l'espansione avvenga già nella definizione della variabile.

Un ultimo strumento importante sono le *funzioni per le sostituzioni di stringhe*.[8] 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`. Si ottiene questo risultato perché `img2.jpg` non ha lo stesso schema di ⟨*modello*⟩, non finisce cioè per `.png`, la sostituzione non avviene e questa parola viene soppressa dall'output della funzione `patsubst`, invece le altre due parole seguono lo schema del modello dunque `img1.png` e `img3.png` vengono sostituite rispettivamente con `img1.eps` e `img3.eps`.

1.6.1 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 *et al.*, 2010, pagina 43),

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

[8] Per un elenco esaustivo di queste funzioni vedi (Stallman *et al.*, 2010, pagina 80).

```
        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.

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 1.2) 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.

1.6.2 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

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

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

Esempi d'uso

2

Il codice 1.6 è un `Makefile` già abbastanza elaborato. In realtà, come già osservato, tutte le operazioni che esso svolge possono essere eseguite anche dai comuni editor di testo specializzati per LaTeX, quindi ci si potrebbe chiedere, a ragione, perché scrivere un `Makefile` piuttosto che affidarsi agli strumenti già disponibili. `make` è utile quando è necessario eseguire operazioni non comuni difficilmente eseguibili con altri strumenti. In questo paragrafo vedremo alcuni esempi di operazioni che un `Makefile` permette di automatizzare.

2.1 Convertire immagini PDF in formato EPS

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 PDFLaTeX. In particolare, LaTeX richiede immagini in formato EPS, PDFLaTeX accetta immagini in formato PDF, JPG e PNG (vedi (Pantieri e Gordini, 2012, 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.

Nelle distribuzioni GNU/Linux e con la distribuzione TeX Live è disponibile il programma da linea di comando `pdftops`. Con la sintassi

```
$ pdftops -eps ⟨file PDF⟩ ⟨file EPS⟩
```

si converte il cioè ⟨*file PDF*⟩ in formato EPS. Il nome del nuovo file è dedotto automaticamente dal file originale, cambiando solo l'estensione da `.pdf` a `.eps`. Se si vuole specificare un nome differente per il file di output lo si può specificare come secondo argomento, opzionale, del comando.

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 1.6:

Codice 2.1 `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.bib
TUTTI_LATEX     = $(PRINCIPALE_TEX) \
                  $(BIBLIOGRAFIA)
CARTELLA_IMG    = Immagini
IMMAGINI_PDF    = $(wildcard $(CARTELLA_IMG)/*.pdf)
IMMAGINI_EPS    = $(patsubst $(CARTELLA_IMG)/%.pdf,\
                  $(CARTELLA_IMG)/%.eps, \
                  $(IMMAGINI_PDF))

.PHONY: dvi

dvi: $(PRINCIPALE_DVI)

$(PRINCIPALE_DVI): $(TUTTI_LATEX) $(IMMAGINI_EPS)
        latex $(PRINCIPALE)
        bibtex $(PRINCIPALE)
        makeindex $(PRINCIPALE)
        latex $(PRINCIPALE)
        latex $(PRINCIPALE)

$(CARTELLA_IMG)/%.eps: $(CARTELLA_IMG)/%.pdf
        pdftops -eps $^ $@
```

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:

```
$(CARTELLA_IMG)/%.eps: $(CARTELLA_IMG)/%.pdf
	pdftops -eps $^ $@
```

Il prerequisito di questa regola è un qualsiasi file della cartella 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.[1] 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

```
	pdftops -eps $^ $@
```

Le due variabili $^ e $@ sono delle variabili dette *automatiche*[2] e che hanno un significato speciale: $^ indica tutti i prerequisiti della regola (in questo caso indica il solo file PDF che ha lo stesso nome di base dell'obiettivo), la seconda indica l'obiettivo della regola (in questo caso l'immagine EPS da generare). Un'altra variabile automatica utile è $< che è uguale al primo prerequisito. 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 2.1 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. Fra questi ci sono le figure $(IMMAGINI_EPS) che verranno generate a partire dalle corrispondenti immagini in formato PDF grazie alla regola vista in precedenza.

[1] Il metacarattere % è stato introdotto nel paragrafo 1.6.

[2] Per un elenco esaustivo di queste variabili vedi (STALLMAN *et al.*, 2010, pagina 112).

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 2.2 `Makefile` in cui la prima regola compila il documento in DVI convertendo le immagini PDF in EPS, la seconda regola compila in formato PDF.

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

dvi: $(PRINCIPALE_DVI)

pdf: $(PRINCIPALE_PDF)

$(PRINCIPALE_DVI): $(TUTTI_LATEX) $(IMMAGINI_EPS)
        latex $(PRINCIPALE)
        bibtex $(PRINCIPALE)
        makeindex $(PRINCIPALE)
        latex $(PRINCIPALE)
        latex $(PRINCIPALE)
```

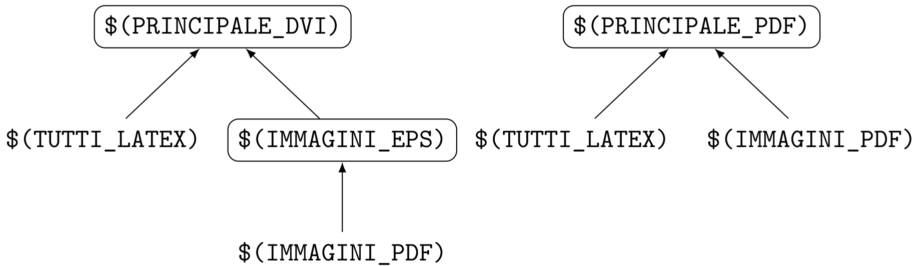

Figura 2.1 Grafo ad albero rappresentante le dipendenze fra i file del Makefile contenuto nel codice 2.2, eccetto i phony targets.

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

Immagini/%.eps: Immagini/%.pdf
        pdftops -eps $^ $@

distclean: clean
        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 2.1 è rappresentato il grafo ad albero che descrive le dipendenze fra i file del `Makefile` mostrato nel codice 2.2.

2.2 Creare un archivio compresso

Spesso si vuole creare un archivio compresso contenente il codice sorgente del proprio documento, in modo da poterlo distribuire in maniera semplice.

Questo può essere fatto con un codice come quello seguente

```
CARTELLA        = $(shell basename $$(pwd))
PRINCIPALE      = documento
DATA            = $(shell date "+%Y%m%d-%H%M%S")

.PHONY: dist distclean

# Questo Makefile è incompleto, per la definizione del
# phony target 'distclean' vedi gli esempi precedenti

dist: distclean
        cd ..; tar -cvaf $(PRINCIPALE)-$(DATA).tar.xz \
          --exclude-vcs $(CARTELLA)/
```

La variabile `CARTELLA` ha come valore il nome della cartella corrente: nella shell il percorso della cartella corrente si ottiene con il comando `pwd`, il suo nome base si estrae con `basename $(pwd)`. La sostituzione di comando con `$$` è stata spiegata nel paragrafo 1.6.2, la funzione `$(shell ...)` è stata introdotta nel paragrafo 1.6.1. Il phony target `dist` esegue queste operazione: cambia la cartella attuale di lavoro nella cartella superiore (con il comando `cd ..`) e crea in questa posizione un archivio, compresso con il programma `xz`, della cartella `$(CARTELLA)`. Il nome dell'archivio è `$(PRINCIPALE)-$(DATA).tar.xz`, in modo da individuare facilmente la data in cui è stato creato, e con l'opzione `-exclude-vcs` si escludono dall'archivio gli eventuali file legati al sistema di controllo delle revisioni usato. Per maggiori informazioni sulle opzioni di `tar` si rimanda al suo manuale consultabile con il comando

```
$ man tar
```

Si noti che nell'elenco dei comandi `cd` e `tar` devono trovarsi sulla stessa riga, separati da un punto e virgola `;` per separarli, e non uno per riga come ci si potrebbe aspettare, perché, come detto nel paragrafo 1.2, `make` in maniera predefinita invoca una nuova subshell per ogni riga dell'elenco dei comandi. Quindi se si scrivesse

```
dist: distclean
        cd ..
        tar -cvaf $(PRINCIPALE)-$(DATA).tar.xz \
          --exclude-vcs $(CARTELLA)/
```

il comando `tar` sarebbe eseguito nella stessa cartella del codice sorgente e non nella cartella superiore, come vorremmo.

In questo esempio il phony target `distclean`, già mostrata negli esempi precedenti, è un prerequisito di `dist`, quindi eseguendo da terminale il comando

```
$ make dist
```

verranno prima cancellati tutti i file generati dalla compilazione del documento e poi verrà creato l'archivio, che in questo conterrà esclusivamente il codice sorgente. Se invece si vuole che l'archivio contenga, per esempio, il file di output in formato PDF ma non tutti i file temporanei generati, i prerequisiti della regola `dist` possono essere cambiati in questo modo

```
dist: pdf clean
```

I phony targets `pdf` e `clean` sono gli stessi già illustrati in precedenza e devono essere elencati fra i prerequisiti nel preciso ordine riportato qui, in modo che prima venga generato il PDF e successivamente vengano cancellati i file temporanei.

2.3 Includere immagini esterne da compilare

Ti*k*Z è un potente pacchetto LaTeX per la realizzazione di disegni. Se in un documento si inseriscono numerosi disegni realizzati con Ti*k*Z e anche molto elaborati, i tempi di compilazione potrebbero aumentare sensibilmente. Una soluzione a questo problema potrebbe essere la seguente: mettere il codice di ciascuna figura Ti*k*Z in un file con estensione `.tikz` usando la classe `standalone`, compilare tutti questi file con PDFLaTeX per ottenere il documento in formato PDF e poi includere i file così prodotti nel documento principale con la macro `\includegraphics{}`. In questa situazione diventa importante assicurarsi che le figure PDF siano aggiornate, infatti è facile dimenticarsi di ricompilare la figura dopo aver fatto una piccola modifica al suo codice sorgente. È evidente che in questo caso `make` può risultare molto utile per controllare in maniera efficiente che le figure Ti*k*Z siano sempre aggiornate.

Supponiamo che tutti i file `*.tikz` siano posti nella sottocartella `img-tikz` della directory in cui si trovano il `Makefile` e il file principale del documento, chiamato `documento.tex`. Per i codici delle figure si

potrebbe più semplicemente usare la classica estensione `.tex`, si è preferito `.tikz` per evidenziare che si tratta di codici di figure Ti*k*Z. Inoltre questa scelta permette di riservare l'estensione `.tex` a un eventuale file, da non compilare, posto nella cartella `img-tikz` e contenente solo il preambolo comune per tutti i file `*.tikz`, all'interno dei quali viene incluso con la macro `\input{}`. Nell'esempio seguente il file di preambolo è chiamato `preambolo.tex`.

Un `Makefile` che permette di compilare il file `documento.tex` in formato PDF è il seguente

```
PRINCIPALE          = documento
PRINCIPALE_TEX      = $(PRINCIPALE).tex
PRINCIPALE_PDF      = $(PRINCIPALE).pdf
CAPITOLI_TEX        = $(wildcard Capitoli/*.tex)
TUTTI_LATEX         = $(PRINCIPALE_TEX) \
                      $(CAPITOLI_TEX)
IMMAGINI_TIKZ       = $(wildcard img-tikz/*.tikz)
IMMAGINI_TIKZ_PDF   = $(patsubst img-tikz/%.tikz,\
                      img-tikz/%.pdf, $(IMMAGINI_TIKZ))
FILE_CLEAN          = *.aux *.log \
                      $(wildcard img-tikz/*.aux) \
                      $(wildcard img-tikz/*.log)
FILE_DISTCLEAN      = $(PRINCIPALE_PDF) \
                      $(IMMAGINI_TIKZ_PDF)

.PHONY: pdf distclean clean

pdf: $(PRINCIPALE_PDF)

# Supponiamo per brevità che per compilare il
# documento principale sia sufficiente eseguire
# due volte pdflatex
$(PRINCIPALE_PDF): $(TUTTI_LATEX) $(IMMAGINI_TIKZ_PDF)
        pdflatex $(PRINCIPALE)
        pdflatex $(PRINCIPALE)

img-tikz/%.pdf: img-tikz/%.tikz img-tikz/prambolo.tex
        cd img-tikz ; pdflatex $(shell basename $<)

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

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

La duplice utilità di un simile `Makefile` è che le figure PDF realizzate con Ti*k*Z sono sicuramente sempre aggiornate quando si genera il documento finale con `make` ed esse vengono ricompilate solo se il corrispondente codice sorgente è stato modificato, riducendo al minimo indispensabile il tempo di compilazione totale.

Integrare make negli editor di testo 3

Può essere utile poter richiamare make direttamente dal proprio editor di testo preferito. In generale si consiglia di aggiungere il comando make all'elenco dei comandi di composizione dell'editor, in modo da poter eseguire la regola predefinita del Makefile. Di seguito sono riportate le istruzioni per aggiungere make all'elenco dei programmi di composizione di alcuni dei più diffusi editor di testo specifici per LaTeX. Affinché tutti gli editor di testo possano effettuare correttamente l'analisi dell'output di make è necessario compilare in modalità nonstopmode. Questo può essere fatto aggiungendo nel Makefile ai comandi latex o pdflatex (o l'eventuale altro compilatore usato) l'opzione -interaction=nonstopmode.

```
# Variabili contenenti i nomi dei comandi da eseguire
# e le relative opzioni.  Per cambiare le opzioni dei
# comandi sarà sufficiente cambiare le variabili
LATEX      = latex -interaction=nonstopmode
PDFLATEX   = pdflatex -interaction=nonstopmode

$(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)
```

3.1 Emacs

Aggiungere al proprio file di inizializzazione `.emacs` o il seguente codice

```
(eval-after-load "tex"
  (add-to-list 'TeX-command-list
               '("Make" "make" TeX-run-TeX nil t)))
```

È necessario utilizzare il pacchetto AUCTEX. Con questa modifica sarà possibile eseguire `make` con la combinazione di tasti `C-c C-c Make RET`.

3.2 Kile

Andare nel menu Impostazioni › Configura Kile, nella scheda "Costruzione", sotto "Strumenti", della finestra che verrà aperta, fare clic sul pulsante Nuovo... sotto l'elenco degli strumenti disponibili. Nella nuova finestra scrivere "Make" nel campo "Nome dello strumento", fare clic sul pulsante Successivo e selezionare "PDFLaTeX" nel menu a tendina della classe a cui fare riferimento, quindi fare clic sul pulsante Completa. A questo punto bisogna configurare lo strumento: nel campo "Comando" della scheda "Generale" scrivere "make", assicurarsi che le opzioni "Controlla se il documento radice" e "Salta al primo errore" siano attive e deselezionare l'opzione "Esegui automaticamente gli strumenti aggiuntivi" (se nei `Makefile` ci sono già le istruzioni necessaria per eseguire automaticamente tutti gli eventuali strumenti aggiuntivi). Nella scheda "Avanzate" assicurarsi che il campo "Tipo" sia impostato a "Esegui al di fuori di Kile", il campo "Classe" a "LaTeX" e il campo "Stato" a "Editor". Nella scheda "Menu" selezionare "Compila" nel menu a tendina relativo alla voce "Aggiungi strumento al menu di costruzione", in questo modo sarà possibile eseguire `make` dall'elenco di tutti gli strumenti di compilazione.

3.3 Texmaker

Seguendo il menu Utente › Comandi personalizzati › Modifica comandi utente si aprirà una finestra come quella riportata nella figura 3.1 e si deve scrivere "Make" nel campo "Voce di menu" e "make" nel campo "Comando". In questo modo `make` sarà eseguibile dall'elenco dei comandi personalizzati.

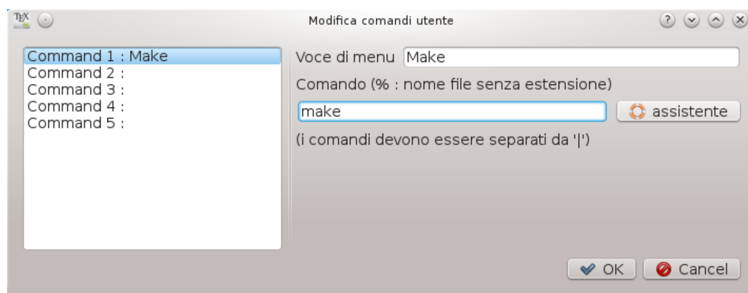

Figura 3.1 Finestra di configurazione di Texmaker.

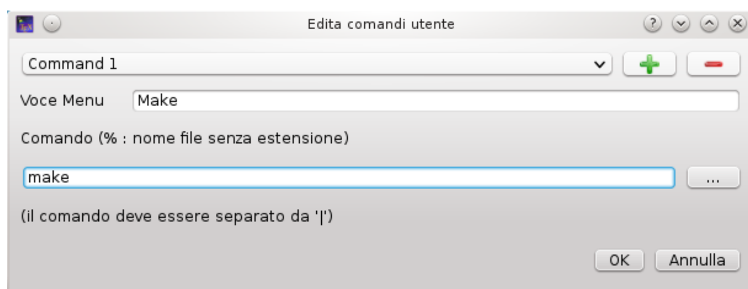

Figura 3.2 Finestra di configurazione di TeXstudio.

3.4 TeXstudio

TeXstudio è un fork di Texmaker e la procedura è simile a quella descritta qui sopra. Dal menu Utente › Comandi utente › Edit comandi utente si aprirà una finestra come quella riportata nella figura 3.2 e nel campo "Voce menu" si scriverà "Make", nel campo "Comando" si scriverà "make". Dopo di ciò sarà possibile eseguire `make` dall'elenco dei comandi utente.

3.5 Texworks

Andare nel menu Modifica › Preferenze e nella scheda "Composizione" della finestra che si aprirà fare clic sul pulsante + vicino all'elenco degli strumenti di composizione. Nella finestra che si aprirà, come quella riportata nella figura 3.3, scrivere "Make" nel campo "Nome" e "make" nel campo

Figura 3.3 Finestra di configurazione di Texworks.

"Programma". Dopo di ciò `make` sarà disponibile nell'elenco dei programmi di composizione di questo editor di testo.

Bibliografia

Caucci, L. e Spadaccini, M. (2005). *Gestione di Figure e Tabelle con LaTeX*. URL http://www.guit.sssup.it/downloads/fig_tut.pdf.

Giacomelli, R. (2012). *Guida tematica alla riga di comando. Utilizzo della console a riga di comando*. URL http://www.guitex.org/home/images/doc/GuideGuIT/guidaconsole.pdf.

Pantieri, L. e Gordini, T. (2012). *L'arte di scrivere con LaTeX*. URL http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf.

Stallman, R. M., McGrath, R. e Smith, P. D. (2010). *GNU Make*. URL http://www.gnu.org/software/make/manual/make.pdf.