Generare documenti LaTeX con diversi linguaggi di programmazione

Roberto Giacomelli, Gianluca Pignalberi

Sommario

Questo articolo mostrerà come generare sorgenti LaTeX per mezzo di programmi scritti in diversi linguaggi di programmazione. Vogliamo con ciò presentare una soluzione generale per i progetti di documentazione basati su processi automatici a causa della complessità dei dati o per scelta applicativa, così da ottenere rapporti magnificamente composti.

Abstract

This paper will show how to generate LaTeX source code with programs written in different programming languages. We aim at presenting a general solution to those documentation projects based on automatic processes — because of data complexity or on purpose — so to obtain beautifully typeset reports.

1 Introduzione

Non diciamo niente di nuovo rimarcando il fatto che un documento LaTeX è un documento di puro testo. La scelta di questa codifica, ridondante dal punto di vista dell'efficiente occupazione di spazio su disco, offre due notevolissimi vantaggi:

1. la possibilità di leggere e modificare ogni documento da *qualunque* editor di testo, interattivo o meno;
2. la possibilità di scrivere programmi e librerie in grado di generare o modificare documenti in maniera automatica e dipendente dall'elaborazione.

Del primo vantaggio non vale la pena discutere oltre, vista la sovrabbondanza di manuali e articoli che elencano questo punto in ogni lista di vantaggi di LaTeX rispetto a ogni programma concorrente che salvi in un formato binario.

Intendiamo invece soffermarci sul secondo vantaggio, più che mera estensione del primo. Lo faremo fornendo degli esempi concreti, più o meno semplici, tramite programmi scritti in diversi linguaggi di programmazione e scripting. Ognuno di questi programmi scriverà in parte o completamente un documento LaTeX inserendogli i risultati di un'elaborazione. Vedremo diversi metodi: scrittura a partire da un modello — un *template* — con riempimento di parti bianche o con sostituzione di quelle fittizie; scrittura mediante l'uso una libreria già predisposta, scrittura completa di una tabella mediante scrittura su file di stringhe.

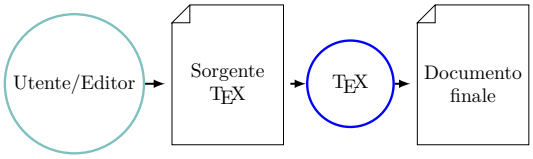


Figura 1: Flusso di lavoro manuale. L'utente crea e modifica direttamente un documento TeX tramite un editor di testi. Tale documento verrà poi composto dal sistema TeX.

Nel caso di scrittura parziale, lo farà a partire da un modello — un *template* — riempiendo le parti bianche o sostituendo quelle fittizie; nel caso di scrittura totale, scriverà tutto il documento, se necessario inserendo il preambolo.

Dopo aver chiarito il campo applicativo nel paragrafo 2, descriveremo il tipo di dato stringa e le relative operazioni nel paragrafo 3. Il paragrafo 4 introdurrà i modelli in termini generali.

Essendo i paragrafi fin qui elencati a carattere teorico, demanderemo ai successivi paragrafi 6, 7, 8 e 10 di chiarire i dettagli di funzionamento dei comandi specifici del linguaggio in esame e spiegare le tecniche usate con relativi vantaggi e svantaggi. Per ogni linguaggio presenteremo uno o più esempi in grado di fornire una panoramica sufficiente, benché non esaustiva, delle operazioni necessarie a rendere un programma *TeX-friendly*. Arriveremo a tale *TeX-friendliness* mediante programmazione bruta, dunque costruendo delle funzioni *ad hoc*, o per mezzo di librerie preesistenti.

I paragrafi 9 e 11 mostreranno due corposissimi programmi in Lua e Python per risolvere dei problemi reali quali la realizzazione della griglia di attività annuali di un gruppo di persone e un report d'inventario di un esercizio commerciale con interrogazione di database.

2 Il campo applicativo

Il flusso di lavoro normale per generare un documento LaTeX prevede che l'utente scriva manualmente i sorgenti usando un editor di testi (figura 1).

Nel caso della produzione automatica della documentazione invece, l'utente non interviene in alcun modo sui sorgenti. Interagisce però con un'applicazione che elabora i dati e ne memorizza il risultato

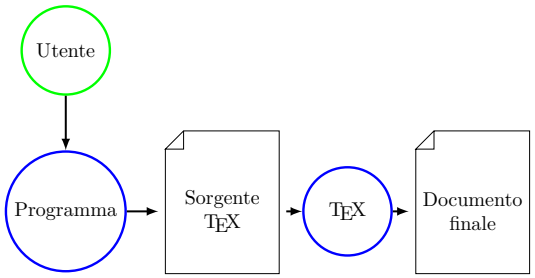


FIGURA 2: Flusso di lavoro automatico. L'utente crea e utilizza un programma che genera il sorgente TEX. Tale documento verrà poi composto dal sistema TEX.

in file perfettamente compilabili da uno dei motori di composizione della famiglia TEX (figura 2).

Stiamo entrando in un mondo completamente diverso dall'usuale: in questa prospettiva considereremo l'utente non come chi scrive il documento ma come chi scrive il programma che scriverà il documento. Un punto di vista che ci possiamo trovare ad affrontare in tutti quei casi in cui i dati del documento finale *richiedano* un'elaborazione automatica iniziale.

Si pensi per esempio alla produzione di report da database: è un campo applicativo dominato da appositi strumenti informatici programmati in un linguaggio ad alto livello da sviluppatori esperti che operano in infrastrutture di rete. Nulla però vieta di esplorare soluzioni basate su TEX perché con questo sistema il report è un testo comprensibile direttamente modificabile dagli utenti.

Il nostro terreno di esplorazione sarà quindi quello che percorrerebbero gli utenti del sistema TEX desiderosi di risolvere in proprio i problemi di reporting per raggiungere due principali obiettivi:

- report di ottima qualità tipografica;
- report che si adattino facilmente a nuove esigenze per contenuto e grafica.

Come vedremo più avanti, in confronto ai linguaggi di programmazione studiati per applicazioni di sistema come il C/C++, i linguaggi di scripting non obbligano a preoccuparsi della gestione della memoria e di molti altri dettagli, a utilizzare i puntatori oppure a usare un compilatore per ottenere gli eseguibili.

In questo articolo mostreremo esempi concreti, didattici o meno, in C e bash e in altri due linguaggi di scripting che giudichiamo importanti per la loro potenziale utilità per gli utenti TEX: Lua e Python.

In particolare, Lua (`www.lua.org`) ha una sintassi essenziale e una sola struttura dati predefinita, la tabella, che può essere impiegata sia come lista che come dizionario e incorpora una libreria standard che offre all'utente solamente le funzionalità indispensabili.

Il linguaggio Python (`www.python.org`), invece, si avvale di tre tipi fondamentali di dato (la lista, la tupla e il dizionario) e presenta una sintassi più vicina al paradigma object oriented. Sono disponibili moltissime librerie, praticamente per tutte le necessità e in particolare per la connessione a database SQLite o PostgreSQL.

Strumenti simili integrati in un flusso di lavoro possono essere le migliori soluzioni possibili in tutti quei casi che potremo definire *artigianali* dove l'utente TEX vuole produrre documenti da insiemi di dati dalle origini e dai formati più diversi.

Progetti in questo senso ne sono già stati realizzati moltissimi: si pensi all'applicazione Facile del comune di Napoli coadiuvata da Agostino De Marco (De Marco e Cresci, 2011) o al recente esempio di produzione di menù per ristorante con un'applicazione web basata su PHP di Claudio Fiandrino (Fiandrino, 2014). Ma anche la produzione di certificati di prova sui materiali secondo specifiche normative tecniche elaborato da Renato Battistin (Battistin, 2014).

Tutti questi progetti sono recenti perché le esigenze di reporting crescono sempre di più in un mondo connesso e informatizzato come quello delle nostre società. D'altro canto grazie a Internet sono disponibili strumenti software multipiattaforma, aperti e ben documentati dalle funzionalità davvero notevoli con cui gli utenti possono impegnarsi per risolvere i problemi di elaborazione dati integrando TEX nel processo.

3 L'elaborazione delle stringhe

La *stringa* è un tipo di dato di *alto livello* che consiste in una sequenza di caratteri per esprimere il concetto di testo. Attraverso le stringhe il testo può essere memorizzato in o caricato da un file (dove per file intendiamo anche il flusso a video o da tastiera e non solo un file su disco).

La locuzione "di alto livello" sta a indicare che tale dato è, di fatto, qualcosa di più complesso: il C memorizza una stringa come un vettore di caratteri gestito per mezzo del puntatore al primo elemento interpretando il carattere `NULL` (`\0`) come segnale di fine sequenza. Tale terminazione è inserita quando si preme il tasto `invio` — codificato come `\cr\lf`, `\cr` o `\lf` a seconda che il sistema operativo sia DOS/Windows, Mac pre-OS X o Unix — e non fa parte della stringa.

La *stringa* è un tipo di dato di *alto livello* che consiste in una sequenza di caratteri. La locuzione "di alto livello" sta a indicare che tale dato è, di fatto, qualcosa di più complesso: il C memorizza una stringa come un vettore di caratteri (o, meglio, un puntatore a carattere). Il tipo *stringa* esprime dunque esattamente il concetto di contenitore di testo. Il testo può essere memorizzato in una stringa che a sua volta può essere memorizzata, cioè il suo contenuto può essere memorizzato, in un file di testo e viceversa. Naturalmente non c'è una restrizione sul tipo di caratteri contenuti in

una stringa, sebbene il C standard termini una stringa col carattere `NULL` (`\0`) e tale terminazione venga segnalata premendo `invio`, il cui carattere (`\cr\lf`, `\cr` o `\lf` a seconda che il sistema operativo sia DOS/Windows, Mac pre-OS X o Unix) pertanto non farà parte della stringa.

Per scrivere un programma che generi sorgenti sembra naturale usare le stringhe per tutte le operazioni, ivi compresa la memorizzazione di quelle parti del documento non soggette ad alcuna modifica dipendente dall'elaborazione. L'uso che delle stringhe possiamo fare è in buona misura determinato dalle caratteristiche di questi dati e dalle relative operazioni di manipolazione. Vediamo brevemente le possibili operazioni su una stringa, tenendo conto che ogni linguaggio le implementerà in base alle proprie caratteristiche e con una sintassi specifica, non sempre perfettamente coerente dal punto di vista dell'utente:

creazione: operazione con cui la stringa viene creata, ed eventualmente inizializzata; di fatto le viene riservata un'area di memoria accessibile tramite un'etichetta (il nome della variabile di tipo stringa);

lettura: operazione con cui il programma legge il contenuto della stringa;

scrittura: operazione con cui il programma scrive del contenuto nell'area di memoria riservata alla stringa;

lunghezza: determinazione del numero dei caratteri della stringa;

assegnazione: scrittura del contenuto di una stringa in un'altra stringa;

concatenazione: operazione con cui il contenuto una stringa viene attaccato alla fine del contenuto di un'altra stringa;

comparazione: date due stringhe, se ne confronta il contenuto per determinare quale delle due venga alfabeticamente prima dell'altra;

conversione: una stringa può essere convertita, quando ce ne siano le condizioni, in numero, in lista di token...

interpolazione: espansione di *segnaposto* presenti in una stringa con dati corrispondenti per la creazione di una nuova stringa;

ricerca: operazione mediante la quale si ricerca la presenza, o l'assenza, di un carattere o di una stringa all'interno di un'altra stringa.

Nel paragrafo 5, durante l'esposizione dei problemi da risolvere e delle relative soluzioni, vedremo come i linguaggi esaminati rendono disponibili al programmatore alcune delle operazioni menzionate.

Naturalmente, ai fini del nostro lavoro, sarebbe inutile non poter salvare delle stringhe più o meno complesse in un file, così come lo sarebbe non poter memorizzare il contenuto di un file di testo in una o più stringhe; vedremo pertanto, sempre nel paragrafo 5, alcune istruzioni per salvare dette stringhe in maniera permanente o acquisirle da un file.

4 La tecnologia dei template

Come accennato nel paragrafo precedente, una volta che sappiamo come trattare le stringhe è immediato pensare di scrivere un programma in grado di memorizzare una documento il cui testo sia contenuto in una o più stringhe e in cui eventualmente sostituire le parti variabili con il contenuto di stringhe impostate durante l'esecuzione. Infatti tramite le operazioni sulle stringhe è possibile costruire in modo efficiente brani testuali. Man mano che si utilizzano queste tecniche con successo diviene tuttavia chiaro che nel programma non esiste una separazione concettuale tra dati e testo.

Esaminiamo il caso di un classico problema di *mail merge*, ovvero il problema di produrre una serie di lettere dal contenuto comune da personalizzare a partire dai destinatari e dai relativi indirizzi. Possiamo certamente separare in file distinti l'elenco degli indirizzi, la logica del programma e il testo della lettera ma, usando le tecniche di manipolazione delle stringhe, è proprio il concetto di *modello di lettera* che rimarrà "nascosto" nel codice, così come la sua realizzazione fisica, cioè delle variabili di tipo stringa contenenti il testo da trattare da parte del programma. Ciò rende difficoltoso apportare qualunque modifica al modello limitando l'evoluzione dei contenuti e il riuso del codice.

4.1 Cos'è un template

Il *template*, o *modello*, è un particolare tipo di documento testuale in cui una parte fissa e immutabile di contenuto si alterna a parti variabili chiamate *tag* che contengono nomi individuati da sequenze di caratteri detti delimitatori.

Il template replica nel campo informatico il concetto di *modulo da compilare* con l'ovvia differenza che gli spazi bianchi da riempire sono entità associate al contenuto prodotto da un'elaborazione. A differenza di quanto detto sulle stringhe nel paragrafo precedente, il metodo di lavoro associato ai template è differente perché:

- il template introduce il modello del testo, un nuovo concetto per rappresentare il testo finale;
- il testo finale è la composizione delle parti letterali di testo del template e l'espansione dei tag con informazioni esterne — dette *contesto* — corrispondenti per struttura e nomi;

- un template può essere annidato in altri template per rappresentare un documento strutturato;
- il codice del programma è incentrato sulla logica e diviene più semplice;
- la generazione di testo con il template può essere molto efficiente in funzione dell'implementazione che ne viene data in un linguaggio di programmazione.

La corrispondenza tra template e contesto deve essere biunivoca in termini sia di struttura che di nomi, costituendo l'*interfaccia* tra i dati e la forma che assumeranno nel documento finale.

4.2 I tag

All'interno del template i tag sono introdotti tramite i caratteri prestabiliti dei *delimitatori*. Di solito il delimitatore di apertura è una coppia di parentesi graffe aperte `{{` mentre quello di chiusura è una coppia di parentesi graffe chiuse `}}`.

Un esempio di template minimale è la seguente stringa dove un tag è associato a una variabile del contesto chiamata `name`:

```
"Hello {{ name }}!"
```

I delimitatori — che possono essere reimpostati dall'utente — e gli spazi presenti all'interno dei tag non verranno inseriti nel testo finale. Si tratta infatti di una sorta di ambiente codice. Come vedremo, i tag possono essere di diverso tipo.

4.3 Utilizzo dei template

In termini generali l'uso dei template comprende due fasi successive:

compilazione un processo d'elaborazione del template che fornirà un nuovo componente *binario*;

resa (*rendering*) una o più esecuzioni del componente binario con dati d'ingresso strutturati e corrispondenti che forniranno altrettanti testi finali.

In alcuni casi, la fase iniziale d'elaborazione del template viene svolta durante la compilazione del programma, anziché a ogni sua esecuzione, con evidente incremento dell'efficienza: il programma dovrà eseguire solamente la produzione dei documenti finali. Nei casi tipici dei sistemi basati sui linguaggi di scripting, invece, il template può essere compilato ogni volta che viene unito ai dati. È da tener presente inoltre che, per la produzione da programma di file sorgenti per il sistema TEX, non è tanto cruciale l'efficienza di elaborazione quanto la semplicità e l'espressività della sintassi dei tag del template: l'utente può modificare più agevolmente i documenti generati sia durante la fase di costruzione che in quella di affinamento o al mutare delle esigenze o delle specifiche.

Occorre un'apposita libreria per utilizzare i template con un dato linguaggio senza eccessivi sforzi programmativi. Sorprendentemente ci sono molte librerie poiché questa tecnologia è molto diffusa sui server di rete per fornire all'utente pagine web dinamiche o file di puro testo scaricabili.

Nelle prossime sezioni verranno illustrati esempi d'uso dei template sia per Lua con la libreria `lustache` (Olivine-Labs, 2015), sia per Python con la libreria `Jinja` (Jinja). Questi due *template engine* sono semplici da usare essendo scritti per un linguaggio di scripting e sono adatti alla generazione di sorgenti TEX perché consentono di cambiare i delimitatori di default a doppie parentesi graffe con altri che non interferiscono con il sorgente finale.

5 Applicazioni pratiche

I prossimi paragrafi raccolgono tutta la parte pratica dell'articolo. Ogni paragrafo verterà su uno specifico linguaggio, traducendo in pratica gli elementi teorici visti nei due precedenti paragrafi e mostrando degli esempi concreti in cui un programma genera file di testo contenente codice che uno dei motori di composizione del sistema TEX tradurrà in documenti finiti.

Al termine degli esempi didattici in Lua e Python, due ulteriori paragrafi riporteranno due applicazioni reali. Ci proponiamo con esse di dare indicazioni utili all'utente per definire il progetto nella sua fase iniziale e a svilupparlo successivamente per adattarsi ai cambiamenti nelle sorgenti dati causati da nuove o inaspettate esigenze di gestione o documentazione.

La qualità del codice molto spesso dipenderà anche dall'equilibrio tra le tre componenti principali del processo di generazione: il programma, il template e il sorgente TEX. Per esempio, non è sempre chiaro se far calcolare le somme economiche dal programma o dal template, oppure se introdurre delle macro LATEX per svolgere compiti che potrebbero essere affidati al template. È buona norma basarsi sull'esperienza lasciandosi ispirare dal criterio di buona manutenibilità e sviluppo del progetto di documentazione e da quello dell'efficienza di esecuzione.

Per la comprensione del codice d'esempio e delle applicazioni presentate nell'articolo è necessario conoscere almeno i fondamenti dei linguaggi di programmazione che chiaramente non possono essere forniti qui. Per Lua si consiglia la lettura del libro Ierusalimschy (2013) e del manuale di riferimento pubblicamente consultabile sul sito ufficiale. Per Python esistono molti libri e tutorial alcuni dei quali liberamente disponibili in rete. Per esempio, *Think Python* (Downey, 2012), con particolare attenzione riguardo alla versione 3 del linguaggio, tra

l'altro composto in LaTeX. Oppure ancora si può consultare il sito `http://getpython3.com/` dove è possibile documentarsi sulle basi del linguaggio — per esempio con il libro scaricabile *Dive into Python 3* (Pilgrim, 2009) — e su temi specifici come la gestione Unicode delle stringhe. Il riferimento per il C è senz'altro il Kernighan & Ritchie (Kernighan e Ritchie, 2004) ma vale la pena consultare anche il Klemens (2014) per tenersi aggiornati sulle modernizzazioni del linguaggio, specialmente sugli ampliamenti ai tipi di dato e all'elaborazione delle stringhe. Infine, per bash è possibile consultare i siti ABS guide, GNU Bash e il manuale in linea (`man bash` da un qualunque terminale *nix).

6 Bash

Bash è un interprete di linguaggio sh-compatibile[1] che esegue comandi letti dallo standard input o da un file. Bash incorpora anche delle caratteristiche utili mutuate da Korn e C shell (ksh e csh).

Descriveremo alcune delle operazioni realizzate da uno script bash durante la compilazione di un numero di ArsTEXnica. Avremo modo di osservare che tutte le operazioni si basano sulla creazione di una stringa di testo che poi verrà inserita in una posizione specifica di un file `.tex` predisposto. La manipolazione delle stringhe[2] in bash è senz'altro più agevole della manipolazione dei numeri, però avremo spesso bisogno di ricorrere a programmi di sistema[3] per isolare quelle parti di una stringa di nostro interesse.

Il sorgente che segue, una porzione minima dello script di compilazione di ArsTEXnica, determina l'anno della rivista, genera l'ISSN adeguato alla versione (a stampa e online) e, oltre alla generazione del codice a barre, inserisce i dati nella seconda di copertina.

```
1  #Determina l'anno corrente; serve per
2  #l'add-on dell'issn e, poi, per i dati
3  #di uscita di ArsTeXnica
4  current_year=`date +%Y`
5  ylf=`echo $current_year | cut -b 4`
6
7  #Estrae i dati dell'uscita corrente
8  #e determina l'add-on
9  current_issue=`pwd | awk -F / '{print $NF}'`
10 issue_name="arstexnica_"$current_issue
11 issue_number=`echo $current_issue | \
12     cut -d_ -f2`
13 if [[ $issue_number -lt 10 ]]; then
14   add_on=$ylf"000"$issue_number;
15 elif [[ $issue_number -lt 100 ]]; then
16   add_on=$ylf"00"$issue_number;
17 elif [[ $issue_number -lt 1000 ]]; then
18   add_on=$ylf"0"$issue_number;
19 else add_on=$ylf$issue_number;
20 fi
21
22 #Analizza la modalità di generazione,
23 #crea i file appropriati e genera
24 #l'issn corretto
25 lv=`cat latest_version`
26 if [ "$1" = "print" ]; then
27   issn="977182823500 $add_on"
28   echo "Genero issn per $issn..."
29   cat seconda_head.tex issn_print.tex \
30       seconda_tail.tex > seconda.tex;
31   barcode -E -e ean -u cm -g 4x1.5 -b \
32       "$issn" -o issn_code.eps
33   if [ "$lv" = "online" ]; then
34     sed -i arstexnica.tex -e \
35         's/%colorlinks/colorlinks/g; \
36         s/%citecolor/citecolor/g; \
37         s/%linkcolor/linkcolor/g; \
38         s/%urlcolor/urlcolor/g'
39   fi
40 elif [ "$1" = "online" ]; then
41   issn="977182823600 $add_on"
42   echo "Genero issn per $issn..."
43   cat seconda_head.tex issn_online.tex \
44       seconda_tail.tex > seconda.tex;
45   barcode -E -e ean -u cm -g 4x1.5 -b \
46       "$issn" -o issn_code.eps
47   if [ "$lv" = "print" ]; then
48     sed -i arstexnica.tex -e \
49         's/colorlinks/%colorlinks/g; \
50         s/citecolor/%citecolor/g; \
51         s/linkcolor/%linkcolor/g; \
52         s/urlcolor/%urlcolor/g'
53   fi
54 fi
55 sed -i arstexnica.tex -e "s/XX, YY/ \
56     $issue_number, $current_year/g"
```

Vediamo in dettaglio cosa fa questo codice ricordando che le righe inizianti con # sono dei commenti. La riga 4 fa sì che venga eseguito per primo il comando (esterno) tra apici inversi e questo estrae l'anno (di quattro cifre) dalla data corrente. La stringa così ottenuta (non è un numero) viene assegnata a una variabile. La riga 5 assegna a una seconda variabile l'ultima cifra dell'anno. Per ottenere questo risultato in bash occorre inviare l'output del comando esterno `echo` (che stampa a video una riga di testo) sulla variabile appena assegnata, cioè proprio la stringa-anno, al comando esterno `cut`, che rimuove delle sezioni da ogni riga di un file stampando le rimanenti; in questo caso, stampa il solo quarto byte dopo aver tolto gli altri. La semantica della riga di codice appena vista è: esegui per primo il comando entro gli apici retroversi e assegna il suo output alla variabile.

La riga 9 assegna a una variabile il nome della directory corrente, quella contenente l'u-

1. Mentre sh sta per Bourne shell, bash sta per Bourne-again shell.

2. Di fatto, le uniche operazioni immediate di manipolazione di stringhe in bash sono la concatenazione e il confronto.

3. La locuzione *programmi di sistema* indica quei filtri — semplici programmi a riga di comando che compiono un'operazione piuttosto specifica — normalmente presenti nei sistemi Unix sotto le directory `/bin`, `/usr/bin`, `/usr/local/bin`.

scita. Il nome convenzionale di tale directory è `Numero_<numero>`. Dunque la variabile, ipotizzando di parlare del numero 18, conterrà la stringa `Numero_18`. La riga 10 crea una variabile (sempre stringa) a cui assegna il nome dell'uscita così composto: `arstexnica_<contenuto della variabile assegnata alla riga 9>`. Nelle righe 11–12 estraiamo il numero dell'uscita e lo assegniamo a una variabile (c'è da ripetere "di tipo stringa"?) col solito meccanismo `echo` + `cut`. Ormai avremo notato che l'assegnazione di una variabile in bash implica la scrittura del solo nome della variabile; per recuperare il valore in essa contenuta dobbiamo anteporre il simbolo dollaro (`$`) al nome della variabile.

Ora abbiamo tutti i dati per formare quel dato variabile del codice ISSN del giornale noto come *add-on*: ultima cifra dell'anno e numero. Questi due numeri dovranno essere separati dal numero di zeri necessario per formare un numero di cinque cifre. Dunque le righe 13–20 si incaricano di confrontare (mediante conversione stringa-numero implicita nelle doppie parentesi quadre dell'`if`) il numero di uscita con 10, 100 e 1000. Per un numero, essere minore di una di quelle cifre, implica essere composto da una, due o tre cifre. Sfruttiamo questo dato per iniettare nella variabile `add-on` il numero opportuno di zeri.

Siamo pronti a iniziare la generazione del numero. Possiamo compilare la rivista in due modalità: stampa (`print`) o elettronica (`online`). Inseriamo in una variabile la modalità dell'ultima compilazione effettuata (salvata in un file di testo, riga 25). Stabiliamo la modalità di compilazione corrente analizzando il parametro fornito al comando (righe 26 e 40) e stabiliamo di conseguenza l'ISSN corretto (righe 27 e 41; l'add-on è lo stesso per entrambe le versioni). Quindi generiamo il file della seconda di copertina, dato dalla concatenazione di tre file di testo tramite il comando esterno `cat` (righe 29–30 e 43–44), e il codice a barre (comando esterno `barcode`, righe 31–32 e 45–46).

A seconda che la compilazione corrente sia per la versione a stampa o elettronica e l'ultima eseguita sia stata l'opposta (righe 33 e 47), bisogna commentare o decommentare le impostazioni relative ai link colorati nel file `arstexnica.tex`: le vogliamo nella versione elettronica ma non in quella a stampa (righe 34–38 e 48–52). Naturalmente, se la compilazione corrente ha la stessa modalità della precedente, non c'è bisogno di fare alcunché dato che le impostazioni sono già quelle necessarie. Anche quest'operazione è affidata a un comando esterno: `sed`, o Stream EDitor, un editor non interattivo.

L'ultima operazione è inserire i dati relativi al numero e all'anno della rivista nel master `arstexnica.tex`. Di nuovo grazie a `sed` sostituiamo i dati contenuti nelle due variabili costruite con bash alle due stringhe di comodo `XX` e `YY` contenute nel master `.tex` (righe 55–56), in questo frangente strutturato come un modello i cui campi variabili non sono delimitati da tag ma costituiti da testo di comodo.

C'è da chiedersi con quale difficoltà e in quanto tempo avremmo potuto realizzare questo pezzetto di script se avessimo dovuto elaborare dei file binari?

7 C

In questo paragrafo vedremo come istruire un programma C a scrivere parte di un documento LaTeX. Per questo esempio didattico scegliamo di calcolare i primi n numeri di Fibonacci e di scriverli in una tabella immagazzinata in un file. Per dare un po' più di informazioni ai lettori, forniremo un altro esempio, sempre didattico, in cui generiamo casualmente un diagramma *scatter* (a dispersione) di cui scriveremo il grafico in un template Ti*k*Z (Tantau, 2015).

7.1 C e tabelle LaTeX

Il primo esempio mostrato calcola i primi n numeri di Fibonacci, che ricordiamo essere dati dalla seguente formula (ricorsiva perché calcolata in termini di sé stessa):

$$f(n) = \begin{cases} n & \text{se } n \leqslant 1 \\ f(n-1) + f(n-2) & \text{altrimenti} \end{cases}$$

La più banale funzione C che calcoli questo algoritmo è la seguente:

```
1  long
2  fibonacci (long numero)
3  {
4    if (numero < 0)
5      {
6        printf ("Numero errato\n");
7        exit (0);
8      }
9    if (numero <= 1)
10     return numero;
11   else
12     return (fibonacci (numero - 1) +
13             fibonacci (numero - 2));
14
15 }
```

Vediamo la perfetta corrispondenza dei `return`, cioè della funzione che restituisce il valore d'uscita di una funzione (o del programma se la funzione è `main()`) con le definizioni della formula.[4] Abbiamo scritto una funzione C ricorsiva. Dal momento che le funzioni ricorsive sono molto onerose dal punto

4. Stiamo volutamente tralasciando una banale questione di efficienza: scambiando le due condizioni dell'`if` rendiamo il programma leggermente più efficiente perché saranno più numerosi i casi in cui calcoliamo la funzione di numeri superiori a 1, dunque non saltiamo una parte di codice per andare al ramo `else`.

di vista dei tempi di calcolo e dell'impiego di CPU, decidiamo di usare una versione iterativa della funzione appena vista:

```
long
fibonacci (long numero)
{
  if (numero < 0)
    {
      printf ("Numero errato\n");
      exit (0);
    }
  if (numero <= 1)
    return numero;
  else
    {
      int menouno = 1, menodue = 0, i;
      long res = 0;
      for (i = 2; i <= numero; i++)
        {
          res = (menouno + menodue);
          menodue = menouno;
          menouno = res;
        }
      return res;
    }
}
```

Non stiamo qui a discutere la correttezza[5] o l'efficienza[6] di questa versione della funzione, non essendo questo il luogo deputato. La useremo però per costruirci intorno un programma[7] che prenda dalla riga di comando il numero di numeri di Fibonacci da calcolare e il nome del file in cui salvare il codice LaTeX e termini senza output ma abbia creato una tabella perfettamente inseribile in un documento LaTeX e contenente i dati calcolati.

Vediamo il programma principale (di fatto, la funzione `main()` e poco altro):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

long fibonacci (long);
FILE *open_file (char *, char *);

int
main (argc, argv)
     int argc;
     char *argv[];
{
  long i, numero;
  char ft[50];
  FILE *filetab;

  numero = atoi (*++argv);
  strcpy (ft, *++argv);
  strcat (ft, "_tabella.tex");
  if ((filetab =
       open_file (ft, "w")) == NULL)
    exit (1);
  fprintf (filetab,
     "\\begin{tabular}{rr}\n\\hline\n");
  fprintf (filetab,
     "\\textsc{i} & \\textsc{i-esimo \
           numero di Fibonacci}\\\\\n");
  fprintf (filetab, "\\hline\n");
  printf
    ("Calcola i primi %d numeri di \
     Fibonacci\n", numero);
  for (i = 0; i < numero; i++)
    fprintf (filetab, "%ld & %ld\\\\\n",
             i + 1, fibonacci (i));
  /*printf("Scrittura del file %s\n", ft); */
  fprintf (filetab, "\\hline\n");
  fprintf (filetab, "\\end{tabular}\n");
  fclose (filetab);
  return 0;
}

long
fibonacci (long numero)
{
  if (numero < 0)
    {
      printf ("Numero errato\n");
      exit (0);
    }
  if (numero <= 1)
    return numero;
  else
    {
      int menouno = 1, menodue = 0, i;
      long res = 0;

      for (i = 2; i <= numero; i++)
        {
          res = (menouno + menodue);
          menodue = menouno;
          menouno = res;
        }
      return res;
    }
}

FILE *
open_file (char *nomefile,
           char *accesso)
{
  FILE *fp;

  if ((fp =
       fopen (nomefile,
              accesso)) == NULL)
    fprintf (stderr,
```

5. Pur non spiegando i dettagli della funzione iterativa, i risultati delle due versioni sono identici.

6. La velocità di calcolo sembra un argomento capzioso in un banale esempio didattico, eppure la versione iterativa del programma finale, scrittura del file compresa, termina l'esecuzione del calcolo dei primi 45 numeri di Fibonacci in un millisecondo sulla macchina di riferimento, quella ricorsiva in circa 21 *secondi* (circa 21 000 volte più lenta).

7. Il programma, didattico, non tiene conto di molti aspetti cruciali in un programma di produzione quali, per esempio, il controllo preventivo sull'input, il controllo degli *overflow*, l'interfaccia utente e il controllo sull'esistenza e l'eventuale volontà di sovrascrivere i file. Quello che conta, ai fini dell'articolo, è mostrare come le funzioni di libreria del C permettono di gestire le stringhe e di scriverle su file.

```
77              "Errore nell'apertura di %s\n",
78              nomefile);
79    return fp;
80  }
```

e discutiamone il funzionamento, con particolare riferimento alle istruzioni di manipolazione delle stringhe e gestione dei file.

Le prime tre righe includono delle librerie necessarie perché forniscono dei comandi inesistenti nel C.[8] La quinta e la sesta riga dichiarano due funzioni: `fibonacci()`, che prende un input di tipo `long int` (intero di quattro byte con segno) e restituisce un risultato di tipo `long int` (`int` è implicito e, pertanto, omesso) e `open_file()`, che prende in input due variabili di tipo puntatore a carattere (stringhe) e restituisce un puntatore a file. La definizione[9] delle funzioni avverrà dopo la funzione `main()`, cioè dopo il programma principale. Alle righe 8–9 troviamo proprio la dichiarazione della funzione `main()` che prende in input degli argomenti (e `argc` e `argv` sono i nomi standard, ma non obbligatori, delle variabili contenenti il numero di argomenti ricevuto sulla riga di comando e le stringhe contenenti gli argomenti effettivi) e restituisce in output un valore intero con segno.[10] Le righe 10 e 11 dichiarano le variabili di input di `main()`, quindi con la successiva parentesi graffa (riga 12) inizia il corpo del programma. Nelle righe 13–15 vengono dichiarate alcune variabili: due `long int`, una stringa (il C considera gli array di caratteri come stringhe. Notate che la lunghezza massima è predeterminata — il C lo prevede esplicitamente, a meno di allocare dinamicamente lo spazio durante l'esecuzione — e, cattiva pratica, è scritta numericamente nella variabile. Il prossimo esempio mostrerà una pratica migliore) e un puntatore a file.

Come abbiamo detto, i parametri d'ingresso sono visti come delle stringhe; poiché il primo parametro è un numero, bisogna effettuare una conversione. Dunque la riga 17 esegue la conversione: `atoi()` (*array [of char] to integer*) prende in input una stringa e restituisce un numero, corrispondente al numero rappresentato nella stringa. In realtà la sintassi dell'istruzione è più complessa: dice, letteralmente, "poiché `argv` punta all'inizio del primo elemento sulla riga (cioè il nome del programma in esecuzione), spostati al secondo elemento (il `++` è un incremento di 1; se postfisso, viene eseguito dopo che la funzione ha preso in input, e processato, il parametro; se prefisso, prima viene eseguito l'incremento e poi passato il parametro alla funzione e processato. Nel caso in esame, incrementare di 1 significa saltare al parametro successivo) e convertilo a numero".

La riga 18 copia il terzo elemento della riga di comando (secondo parametro, nome del file LaTeX contenente la tabella finale) nella variabile `ft` (`strcpy` sta per *string copy*). Quindi la riga 19 completa il contenuto della variabile `ft`, cioè concatena al suo contenuto (`strcat` sta per *string concatenate*) la stringa `_tabella.ft` (per una discussione più approfondita sulla tecnica della concatenazione delle stringhe, si veda più avanti il paragrafo 8.4). Naturalmente, se il nome finale eccede i 49 caratteri (il cinquantesimo serve a contenere il `\0` di fine stringa), verrà tagliato. Le righe 19–21, aprendo in scrittura il file che conterrà la tabella LaTeX, dicono letteralmente: "se il puntatore a file assegnato alla variabile `filetab` dalla funzione `open_file()` (la discuteremo alla fine) è `NULL` (cioè l'apertura del file in scrittura non è andata a buon fine) termina l'esecuzione (riga 22) e esci dando in output il valore 1". Ottenere un valore diverso da `NULL` significa che l'apertura è andata a buon fine e possiamo andare avanti con l'esecuzione.

Le righe 23–28 (una è interrotta con un `\` per motivi di impaginazione; è il modo in cui il C ci permette di mandare a capo delle sequenze di caratteri tra doppi apici, cioè delle stringhe, senza dare errore) scrivono nel file le prime righe del codice LaTeX della tabella. Essendo bufferizzata, la scrittura potrebbe non avvenire nel momento esatto in cui l'istruzione viene eseguita. Le righe 29–31 sono solo un *memento* per l'utente: stampano a video che il programma calcola i primi n numeri di Fibonacci. Tale stampa è sempre bufferizzata. Le righe 32–34 calcolano effettivamente i numeri voluti e contemporaneamente li scrivono nella tabella (nella riga 33 vediamo l'uso della tecnica dei campi variabili, discussa in maggior dettaglio più avanti, al paragrafo 8.3). Ovviamente `fibonacci(0)` rappresenta il primo numero di Fibonacci, ecco il perché dell'`i+1` nella `fprintf()`. La riga 35, sempre un *memento* per l'utente, è commentata e pertanto non eseguita.

Le righe 36 e 37 scrivono nel file tabella i comandi di chiusura della stessa; la riga 38 chiude il file e la riga 39 termina l'esecuzione del programma restituendo il valore 0 che, per convenzione, in C indica il buon fine.

Vediamo ora la funzione `open_file()`. Poiché la funzione deputata del C (`fopen()`) non segnala

8. Il C, linguaggio molto efficiente perché creato per scrivere sistemi operativi, è anche molto avaro di comandi. Moltissimi comandi usati quotidianamente sono forniti dalla libreria standard, parte integrante del linguaggio. Molti programmatori in COBOL e altri linguaggi *"storici"* polemizzano proprio su questa "avarizia" di comandi, quando i "loro" standard forniscono decine di comandi e funzioni per molte operazioni fondamentali quali l'I/O.

9. La differenza tra dichiarazione e definizione è, in sostanza, che la prima serve a indicare una forma (nome della variabile o della funzione e tipi di ingresso e uscita) mentre la seconda un contenuto (valore della variabile o serie di istruzioni della funzione e nomi delle variabili di input e output).

10. A differenza di `long int`, che è definito come un intero con segno di 4 byte, `int` è definito come un intero con segno di 2 o 4 byte. La dimensione dipende dall'implementazione.

Tabella 1: Risultato del calcolo dei primi n numeri di Fibonacci.

I	I-esimo numero di Fibonacci
1	0
2	1
3	1
4	2
5	3
6	5
7	8
8	13
9	21
10	34
11	55
12	89
13	144
14	233
15	377
16	610
17	987
18	1597
19	2584
20	4181

niente all'utente ma solo al programma tramite il parametro di ritorno, la nuova funzione colma questa lacuna stampando a video l'eventuale errore in apertura. Notiamo che, anziché una `printf()`, abbiamo usato una `fprintf()` (stampa su file) che stampi su `stderr`. Quest'ultimo è uno dei "file" standard (gli altri due sono `stdin` e `stdout`); ridirige sul terminale quello che riceve e lo fa in maniera non bufferizzata, dunque la scrittura a video avviene nel momento esatto in cui l'istruzione viene eseguita. Quindi esce restituendo il puntatore a file (o `NULL` in caso di errore).

Eseguiamo il programma, dopo averlo compilato (per compilarlo in Linux basta `gcc -o nfibonacci nfibonacci.c`), per fargli calcolare i primi 20 numeri di Fibonacci. Il risultato è visibile nella tabella 1.

7.2 C e template di grafici

Vediamo ora un secondo esempio in cui generiamo casualmente una serie di coordinate cartesiane intere comprese tra (0,0) e (10,10) e le riportiamo su un grafico Ti*k*Z la cui "ossatura" è contenuta in un modello chiamato `template.tex`. Poiché l'esempio è piuttosto limitato, decidiamo di abbandonare i tag in favore di una riga di commento e di uno spazio da riempire in maniera standard. Di seguito il contenuto del template.[11]

```
1 \begin{tikzpicture}
2 \draw[very thin,color=gray] (0,0.01) grid (10,10);
3 \draw[->] (-0.2,0) -- (10.5,0) node[right] {$x$};
4 \draw[->] (0,-0.2) -- (0,10.5) node[above] {$y$};
5 %---riga da riempire
6 \draw[color=black!30!green] () node {$\times$};
7 \end{tikzpicture}
```

È facile vedere che la seconda riga disegna la griglia del piano cartesiano e le righe 3 e 4 disegnano gli assi del piano. La riga 5 contiene il commento che controlleremo per capire qual è il punto in cui inserire i dati.[12] La riga successiva al commento è quella da riempire coi dati generati e ripetere tante volte quante sono le coordinate generate.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAXRIGA 2048
#define CONTROLLO "%---riga da riempire"

FILE *open_file (char *, char *);

int
main (argc, argv)
     int argc;
     char *argv[];
{
  long i, numero, cr;
  int x, y;
  char nomefile[MAXRIGA], fg[MAXRIGA],
    tmpl[] =
    "template.tex", riga[MAXRIGA];
  FILE *filegraf, *template;

  numero = atoi (*++argv);
  strcpy (fg, *++argv);
  strcat (fg, "_grafico.tex");
  if ((filegraf =
       open_file (fg, "w")) == NULL)
    exit (1);
  if ((template =
       open_file (tmpl, "r")) == NULL)
    exit (1);
  srand (time (NULL));
  while (fgets
         (riga, MAXRIGA - 1,
          template) != NULL)
    if (strncmp (riga, CONTROLLO,
        strlen(CONTROLLO)))
      fputs (riga, filegraf);
    else
      {
        fgets (riga, MAXRIGA - 1,
               template);
        printf
```

11. Lasciamo volutamente debordare il codice per far capire al lettore che ogni riga letta dal programma C è esattamente quella qui stampata.

12. Vista la dimensione e la semplicità del template, sarebbe stato più immediato andare direttamente al punto da modificare tramite una `fseek()`; non è però questo quello che vogliamo mostrare, ma un metodo di lavoro più generale che discuteremo più avanti.

```
44            ("Genera %d coppie di coordinate \
45              di un diagramma a dispersione\n",
46             numero);
47          for (i = 0; i < numero; i++)
48            {
49              for (cr = 0;
50                   riga[cr] != '('; cr++)
51                fputc (riga[cr],
52                       filegraf);
53              x = (int) rand () % 11;
54              y = (int) rand () % 11;
55              fprintf (filegraf, "(%d,%d",
56                       x, y);
57              for (cr++; riga[cr] != '\n';
58                   cr++)
59                fputc (riga[cr],
60                       filegraf);
61              fputc ('\n', filegraf);
62            }
63        }
64    printf ("Scrittura del file %s\n",
65            fg);
66    fclose (template);
67    fclose (filegraf);
68    return 0;
69  }
70
71  FILE *
72  open_file (char *nomefile,
73             char *accesso)
74  {
75    FILE *fp;
76
77    if ((fp =
78         fopen (nomefile,
79                accesso)) == NULL)
80      fprintf (stderr,
81               "Errore nell'apertura di %s\n",
82               nomefile);
83    return fp;
84  }
```

Passiamo alla descrizione di questo esempio saltando quelle parti di codice già discusse all'esempio precedente.

Le righe 6 e 7 contengono la direttiva `#define`, cioè un'istruzione che dice al compilatore: "ogni volta che nel sorgente trovi i termini `MAXRIGA` e `CONTROLLO`, sostituiscili con 2048 e `"%---riga da riempire"`. A differenza dell'esempio precedente, in cui il numero 50 era contenuto nel sorgente e la cui sostituzione comportava di ricercarlo in tutto il codice (e non è detto che tutte le occorrenze di 50 abbiano lo stesso significato, cioè può darsi che in alcune parti il 50 debba rimanere tale e non essere sostituito), in questo caso, invece, basta modificarlo in una sola riga contenuta all'inizio del sorgente. Le righe 15–19 contengono delle definizioni di variabili *locali* (cioè visibili solo alla funzione in cui sono definite) utili al programma.

Le righe 23–31 sono equivalenti a quanto già visto nell'esempio di Fibonacci. L'unica differenza è che ora apriamo due file: quello che conterrà il grafico in sola scrittura e quello contenente il template in sola lettura.

La riga 32 inizializza il generatore di numeri casuali prendendo come seme il valore restituito dalla funzione `time()`. Altri linguaggi forniscono una funzione equivalente senza parametri di input chiamata `randomize`.

Le righe 33–63 sono quelle che fanno tutto. Cerchiamo di raccontarle in maniera discorsiva, come se stessimo raccontando una storia. Prima di raccontare, sarà buona cosa descrivere in termini sommari alcune istruzioni introdotte qui per la prima volta:

`while` è uno dei cicli del C; esegue il corpo del `while` (cioè tutto quanto contenuto nelle parentesi graffe successive o, in mancanza di parentesi graffe, solo l'istruzione seguente) fintanto che la condizione del ciclo è *vera*. Al contrario, esce;

`fgets()` legge una riga da un file (di testo) terminata da `\n` o dalla dimensione fornita come secondo parametro;

`strncmp()` compara i primi n caratteri delle due stringhe fornite come argomento;

`strlen()` restituisce la lunghezza della stringa fornita come argomento;

`fputs()` scrive una stringa in un file di testo;

`for` è un altro ciclo del C; la sua sintassi è piuttosto articolata: all'interno delle parentesi tonde *possono* essere fornite tre espressioni, ognuna delle quali obbligatoriamente separata dal ';'. La prima espressione funge da inizializzazione del ciclo, cioè viene eseguita solo all'entrata nel ciclo; la seconda espressione viene eseguita all'inizio di ogni iterazione successiva; infine, la terza viene eseguita alla fine di ogni iterazione. L'espressione `for (;;)` è perfettamente lecita in C e viene usata per eseguire cicli infiniti;

`rand()` restituisce un valore intero, compreso tra 0 e `RAND_MAX`, (pseudo)casuale estratto da una lista di numeri eventualmente generata da `srand()`;

`%` calcola la "divisione modulo", cioè fornisce il resto di una divisione tra operandi interi.

Ora possiamo raccontare la parte di codice rimanente: per ogni riga letta dal template (33–35) verifichiamo che non sia la riga "di controllo" (36–37) per riscriverla sul file del grafico (38). Nel caso la riga sia quella "di controllo" (39), leggiamo la riga successiva (41–42) e scriviamola nel file del grafico (49–52 e 55–61) tante volte quante sono le coppie di coordinate da generare (47), "iniettando" ogni riga con la coppia di coordinate intere

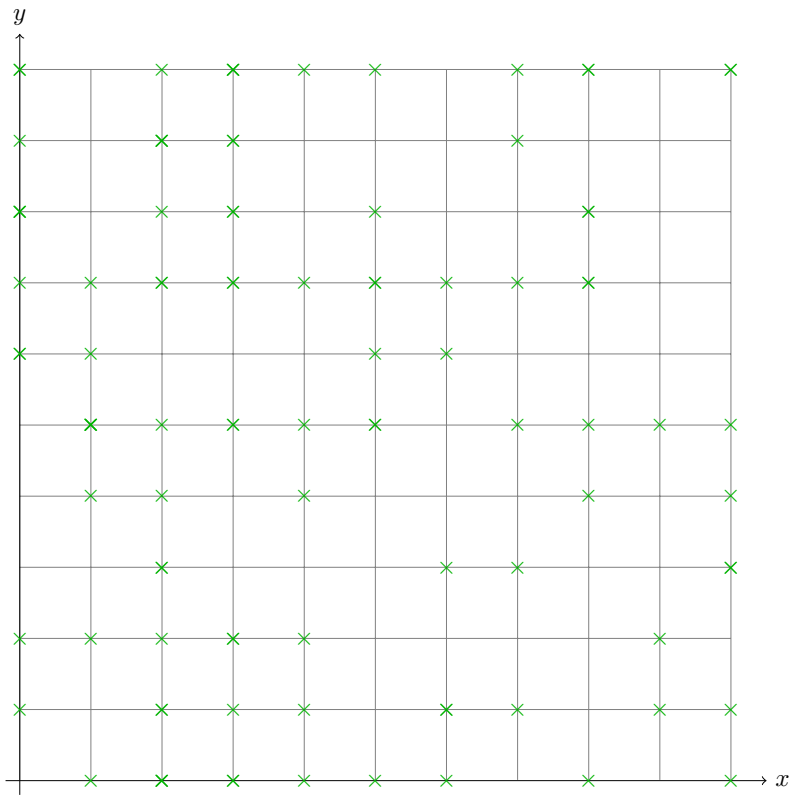


FIGURA 3: Grafico del diagramma scatter relativo alla generazione di 100 coppie di coordinate intere.

generate casualmente e comprese tra (0,0) e (10,10) (53–56) (l'iniezione avviene copiando ogni carattere di input nel file di output quando il carattere non sia '`(`', quindi scrivendo la stringa "`(`x`,`y" per poi copiare i rimanenti caratteri della riga a partire da '`)`').

Il risultato di un'esecuzione su 100 coppie di coordinate è dato nella figura 3.

8 Lua

Questo paragrafo e il prossimo, i più corposi per informazioni ed esempi di codice forniti, mostreranno diverse tecniche basate sulle stringhe di testo o sulla tecnologia dei template per generare sorgenti TEX.

8.1 Stringhe in Lua

Le stringhe in Lua sono immutabili, caratteristica comune ad altri linguaggi. Questo significa che, una volta creata, una stringa non può più essere modificata. L'unico modo, indiretto, per modificare una stringa è crearne una nuova in sostituzione dell'originale.

Esistono sostanzialmente due modi per introdurre in un programma Lua valori letterali di tipo stringa:

1. delimitando i caratteri con il simbolo di doppio apice `"` oppure con il simbolo di apice semplice `'`;
2. usando un delimitatore a doppie parentesi quadre di apertura e chiusura `[[` e `]]`.

Nel primo caso la stringa occupa un'unica linea mentre eventuali caratteri di controllo come il ritorno a capo `\n` o la tabulazione `\t` verranno convertiti nei rispettivi caratteri ASCII. Nel secondo caso, all'opposto, i caratteri non stampabili non verranno interpretati ma è possibile estendere il testo su più linee. Il codice seguente mostra l'uso di queste due diverse sintassi Lua:

```
local s1 = "This is a \\string"
local s2 = 'Literal "\\\\" means "\\"'
local s3 = [[
This is a multiline literal string
And this is a strange sequence: \n\t but
not much...
]]

print(s1)
print(s2)
print(s3)
```

Questo programma produce il seguente output:

```
This is a \string
Literal "\\" means "\"
This is a multiline literal string
And this is a strange sequence: \n\t but
not much...
```

Si noti come con il delimitatore a parentesi quadre di Lua venga ignorato l'eventuale primo carattere di ritorno a capo — cosa che succede proprio con la stringa `s3` dell'esempio formata da quattro linee l'ultima delle quali vuota corrispondente al ritorno a capo inserito dopo `not much...`

Infine, volendo inserire nel testo letterale gli stessi delimitatori, nel caso degli apici è possibile usare ancora l'escaping come in `"una \"bella\" parola"`, mentre nel caso dei delimitatori a coppie di parentesi quadre è possibile usare un qualsiasi numero di segni uguali purché bilanciati tra apertura e chiusura come in:

```
[==[una [[bella]] parola]==]
```

8.2 Creare file di testo

In Lua è molto semplice creare un file di testo ricorrendo alla libreria standard del linguaggio e in particolare alle funzioni contenute nel modulo `io`. Il metodo `io.open()` può creare oggetti `file` specificando come primo parametro il nome del file stesso e come secondo parametro una delle seguenti modalità: lettura `r`, scrittura `w` o scrittura con aggiunta `a`.

Se il parametro di apertura è la stringa `"w"` il file verrà aperto con i diritti di scrittura e, se già esistente, verrà cancellato. Ecco un esempio di codice completo il cui risultato di esecuzione è quello di creare il più piccolo sorgente TEX possibile:

```
-- the most simple source file in TeX
local filename = "simple.tex"
local content = "Hello world!\n\\bye"

-- a new file in the current directory
-- opened in 'write mode':
local file = assert(io.open(filename, "w"))
file:write(content)
file:close()
```

Il contenuto del file sarà esattamente il seguente:

```
Hello world!
\bye
```

Per creare un testo contenente dati variabili occorre necessariamente elaborare stringhe utilizzando funzioni di libreria o la concatenazione. Dedicheremo le prossime due sezioni a ciascuno di questi due metodi.

8.3 Campi variabili

La funzione della libreria standard di Lua che produce stringhe con campi variabili è `string.format()`. Il primo argomento della funzione è una "stringa formato" e gli argomenti successivi sono i valori corrispondenti ai campi. Questo breve esempio stampa una riga di codice LATEX in modalità matematica:

```
local fmt = "\\( \\pi = %0.5f \\)"
local s = string.format(fmt, math.pi)
print(s) --> '\( \pi = 3.14159 \)'
```

Possiamo notare che il campo variabile è relativo a un numero in virgola mobile arrotondato alla quinta cifra decimale.

I campi variabili, già visti nel paragrafo relativo al C pur senza averne specificato il nome, sono delle sequenze formate dal simbolo di percentuale e da un carattere che identifica il tipo di dato: tra gli altri, `d` i numeri interi, `s` le stringhe, `f` i numeri in virgola mobile. Come abbiamo visto è possibile inserire tra questi due elementi ulteriori specificatori per l'arrotondamento numerico o il riempimento di campi a lunghezza fissa. Tutte queste particolarità — naturalmente documentate nel manuale di riferimento del linguaggio — derivano da funzionalità già presenti nel linguaggio C e quindi immediatamente comprensibili per un buon numero di sviluppatori.

8.4 Concatenazione di stringhe

La concatenazione di stringhe comporta la creazione di una nuova stringa pertanto, per motivi di efficienza e d'utilizzo di memoria, è bene ricorrervi per la concatenazione di un numero piccolo di stringhe al di fuori di cicli iterativi. L'operatore di unione è rappresentato da due caratteri punto consecutivi:

```
local s = "Hello" .. " " .. "world!"
assert( s == "Hello world!")
```

La costruzione di un sorgente LATEX richiede l'unione di molte stringhe perciò è molto vantaggioso introdurre un'altra funzione della libreria standard chiamata `table.concat()`. Con essa possiamo concatenare un numero anche molto grande di stringhe, per esempio per formare una lunga lista di dati.

Il codice di esempio completo, dove una serie numerica di venti numeri casuali viene assemblata in un unica stringa, è il seguente:

```
local data = {}
for i=1,20 do
    data[#data + 1] = string.format(
        "%d = %0.3f",
        i,
        math.random()
    )
end

local content = table.concat(data, "\n")
print(content)
```

8.5 Considerazioni finali sulle stringhe

Basandoci sulla corrispondenza tra tipo stringa e file di testo abbiamo presentato le tecniche fondamentali per la costruzione di un sorgente. Queste tecniche sono due: la sostituzione di campi formato all'interno delle stringhe costituiti da *chiavi* segnaposto e la concatenazione efficiente di stringhe per mezzo di un buffer.

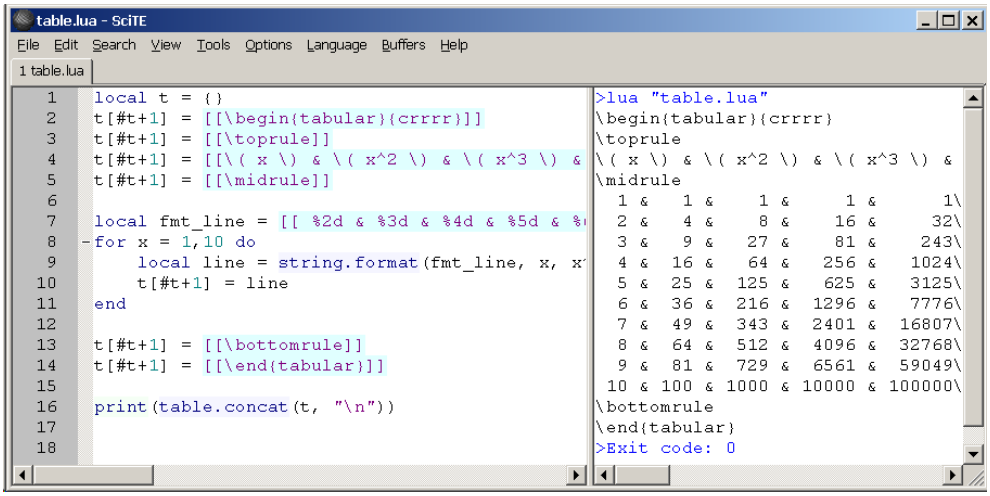


FIGURA 4: L'editor SciTE in azione mostra nel pannello laterale l'uscita del programma Lua per generare il codice LaTeX della tabella numerica descritta nel testo. Per eseguire il codice Lua è sufficiente premere il tasto F5 mentre per aprire e chiudere il pannello di output si usa il tasto F8.

Riassumendo è possibile costruire il testo attraverso il tipo stringa — tipo immutabile in molti linguaggi di programmazione — con le seguenti operazioni:

- stringhe letterali;
- campi formato (interpolazione);
- concatenazione efficiente tramite buffering.

Nelle prossime sezioni saranno presentati esempi concreti che utilizzano le operazioni sulle stringhe per generare sorgenti LaTeX. Si tratta dell'impiego di funzionalità di base dei linguaggi di programmazione o delle loro librerie standard.

8.6 Come eseguire i programmi Lua

L'esecuzione di uno script Lua avviene allo stesso modo della compilazione di un sorgente LaTeX: da riga di comando oppure da un editor predisposto tramite pressione di una combinazione di tasti.

Se non si vuole installare un interprete Lua si può far ricorso al programma `texlua` compreso in TeX Live, mentre come editor è consigliabile l'utilizzo di SciTE (figura 4), in grado di visualizzare in un pannello laterale l'uscita dei programmi lanciati premendo il tasto F5. L'editor SciTE è molto comodo e può essere programmato internamente in Lua.

8.7 Un esempio: costruire una tabella

Ci proponiamo di costruire il codice LaTeX corrispondente a una tabella numerica in cui nella prima colonna compare una variabile indipendente e nelle successive i suoi corrispondenti valori delle potenze fino alla quinta.

Ci avvarremo di una tabella Lua dove memorizzare riga per riga tutto il codice LaTeX per poi stampare il risultato in console. Il codice mostrato di seguito, la cui uscita è mostrata nella figura 4, è suddiviso in tre parti:

1. inserimento del codice di apertura dell'ambiente `tabular` e della riga d'intestazione della tabella;

2. inserimento tramite un ciclo iterativo delle righe di contenuto numerico;

3. inserimento del codice di chiusura dell'ambiente `tabular`.

```
-- a new table called 't'
local t = {[[\begin{tabular}{crrrr}]]}
t[#t+1] = [[\toprule]]
t[#t+1] = [[\(x\) & \(x^2\) & \(x^3\)]]
          ..[[ & \(x^4\) & \(x^5\)\\]]
t[#t+1] = [[\midrule]]

local f = [[ %2d & %3d & %4d & %5d & %6d\\]]
for x = 1,10 do
    local line = string.format(
        f, x, x^2, x^3, x^4, x^5
    )
        t[#t+1] = line
end

t[#t+1] = [[\bottomrule]]
t[#t+1] = [[\end{tabular}]]

print(table.concat(t, "\n"))
```

Lo scopo di questo esempio non è presentare una soluzione definitiva — anzi il codice può essere migliorato in molti modi — ma di mostrare il modello d'implementazione basato sul tipo stringa per la produzione automatica di codice: una sequenza d'inserimenti di testo — stringa letterale o dato variabile — in un *buffer*, linea per linea.

8.8 lustache un template engine per Lua

`lustache` è un'implemetazione in Lua del sistema di template *mustache*. Si tratta di specifiche generali consultabili alla pagina (Mustache, 2015) il cui nome fa riferimento alla somiglianza tra le parentesi graffe dei delimitatori e un bel paio di baffi. Tali specifiche sono disponibili in un buon numero di linguaggi grazie a progetti indipendenti. Rimandiamo il lettore al sito della libreria `lustache` (Olivine-Labs, 2015) per le istruzioni d'installazione.

Un template per un sorgente TEX minimo è mostrato nel seguente frammento di codice:

```
local template = [[
Hello {{ my_friend }}!
{{ bye }}]]
```

Per l'esecuzione di questo template occorre fornire al metodo `render()` della libreria `lustache` una tabella con chiavi uguali a `my_friend` e `bye` previste dai due tag. I valori associati a queste chiavi verranno inseriti nel testo finale. Se invece la chiave non è presente, il tag in questione verrà semplicemente ignorato.

Il codice d'esempio completo è il seguente:

```
local lustache = require "lustache"

local template = [[
Hello {{ my_friend }}!
{{ bye }}]]

local data = {
    my_friend = "John Doe",
    bye = "\\bye",
}

local doc = lustache:render(template, data)
print(doc) -- or save it on a .tex file
```

e troveremo in console il testo sperato:

```
Hello John Doe!
\bye
```

8.9 Una lista

Oltre ai tag semplici ci sono quelli di blocco concettualmente simili agli ambienti di LATEX. Per iterare il testo di una lista, per esempio, si può creare un blocco dopodiché una lista puntata:

```
local lustache = require "lustache"

local tmpl = [[
\begin{itemize}
{{#list}}
    \item {{ . }};
{{/list}}
\end{itemize}]]

local book_list = {
    "Guide to LaTeX",
    "The LaTeX Companion",
    "The LaTeX Graphics Companion",
    "The LaTeX Web Companion",
}

doc = lustache:render(
    tmpl,
    {list = book_list}
)
print(doc)
```

Il tag di blocco `list` è composto dal tag di apertura `#list` e da quello di chiusura `/list`. All'interno di questo blocco si trova un tag anonimo, rappresentato dal carattere '`.`', che riceverà uno alla volta gli elementi contenuti nella tabella/array `list`.

L'uscita del programma, riportata di seguito, mostra alcune imperfezioni:

```
\begin{itemize}

    \item Guide to LaTeX;

    \item The LaTeX Companion;

    \item The LaTeX graphics Companion;

    \item The LaTeX Web Companion;

\end{itemize}
```

8.10 Controllare i ritorni a capo

Eliminare dal template i caratteri di fine riga comporta la loro eliminazione anche dal testo finale; nell'esempio erano stati introdotti per maggiore chiarezza del codice del template. Diversamente possiamo usare i tag commento — tali quando il carattere '`!`' segue i delimitatori di apertura — perché ammettono ritorni a capo al loro interno. Ecco come potremmo riscrivere il template precedente per eliminare le righe bianche di troppo:

```
-- no newline
local tmpl = [[
\begin{itemize}
{{#list}}     \item {{ . }};
{{/list}}\end{itemize}]]
```

Forma alternative sono:

```
-- newline with comment tag
local tmpl = [[
\begin{itemize}
{{#list}}{{! comment tag
}}     \item {{ . }};
{{/list}}{{!
}}\end{itemize}]]
```

oppure:

```
-- with tag inner newline
local tmpl = [[
\begin{itemize}
{{#list
}}     \item {{ . }};
{{/list
}}\end{itemize}]]
```

8.11 Generalizzare il tipo di lista

Il modo più semplice per generalizzare il tipo di lista è renderla attraverso un tag. Ciò implica modificare i delimitatori a coppie di graffe a causa delle possibili interazioni con le graffe che circondano il nome dell'ambiente, come dimostra il seguente template:

```
local tmpl = [[
\begin{{{ env }}}
{{#list}}     \item {{ . }};
{{/list}}\end{ {{ env }} }]]
```

Il primo tag verrà interpretato come un tag delimitato a tre graffe che a differenza di quello a due evita l'escaping del contenuto — ne riparleremo più avanti nella sezione 8.15 — dando `\beginitemize` mentre l'ultimo tag darà come ci si aspetta `\end{ itemize }`.

Non è facile trovare dei delimitatori che certamente non compariranno mai in un sorgente TEX, che siano compatibili con la sintassi Lua per le stringhe e che rendano il template il più leggibile possibile. In questo esempio abbiamo pensato alla coppia `<| |>`. I delimitatori vengono modificati, direttamente nel template e ogni volta che sarà necessario, tramite un tag e il segno `=`, carattere che non può essere compreso nei delimitatori assieme al carattere spazio.

Nell'esempio di codice completo è possibile notare che i segni di uguale `=` sono esterni ai nuovi delimitatori e che questi devono essere separati almeno da un carattere spazio:

```
local lustache = require "lustache"

local tmpl = [[{{=<| |>=}}<|!
|>\begin{<| env |>}
<|#list|>     \item <| . |>;
<|/list|>\end{<| env |>}]]

local books = {
    "Guide to LaTeX",
    "The LaTeX Companion",
    "The LaTeX Graphics Companion",
    "The LaTeX Web Companion",
}

doc = lustache:render(tmpl,
    {list = books, env="itemize"})
print(doc)
```

8.12 Rendere speciale l'ultimo elemento

Le specifiche `mustache` non prevedono la gestione speciale dell'ultimo elemento della lista, per esempio per inserire il carattere '`.`' finale al posto del punto e virgola.

Nel caso in cui la chiave prevista dal tag non è presente nei dati, non verrà generato un errore ma semplicemente il testo di sostituzione sarà vuoto. Questo comportamento ci consente di risolvere il problema al prezzo di una modifica dei dati:

```
local lustache = require "lustache"

local tmpl = [[
\begin{itemize}
{{#list}}   \item {{ book }}{{!
}}{{^dot}};{{/dot}}{{#dot}}.{{/dot}}
{{/list}}\end{itemize}]]

local list = {
  {book = "Guide to LaTeX"},
  {book = "The LaTeX Companion"},
  {book = "The LaTeX graphics Companion"},
  {book = "The LaTeX Web Companion",
    dot = true},
}

doc = lustache:render(tmpl, {list=list})
print(doc)
```

Nel template sono stati inseriti due nuovi blocchi; il primo sarà eseguito per effetto del carattere `^` se la chiave `dot` non esiste, viceversa il secondo. Così, inserendo solamente nell'ultima tabella il campo `dot`, otteniamo l'effetto di aggiungere il `;` alla fine di tutte le righe tranne l'ultima dove troveremo il carattere punto.

Modificare la struttura dati per ottemperare a difetti del template è un rimedio estremo che molto probabilmente compromette lo sviluppo del progetto proprio perché l'obiettivo primario è quello di separare i dati dal modello del testo.

Per fortuna è possibile avvalerci di alcune funzionalità avanzate di Lua come le *closure* e le funzioni come oggetti di prima classe. Nel codice che segue il problema è risolto senza modificare i dati: una funzione è inserita nel template e sulla base del numero di riga aggiunge il corretto carattere finale:

```
local lustache = require "lustache"

local tmpl = [[
\begin{itemize}
{{#list
}}     \item {{ . }}{{ end_char }}
{{/list
}}\end{itemize}]]

local book_list = {
    "Guide to LaTeX",
    "The LaTeX Companion",
    "The LaTeX Graphics Companion",
    "The LaTeX Web Companion",
}

local rows = #book_list
-- closure
function end_char()
    rows = rows - 1
    if rows == 0 then
        return "."
    else
        return ";"
    end
end
```

```
doc = lustache:render(tmpl,
    {list = book_list,
     end_char = end_char,})
print(doc)
```

8.13 Una tabella numerica

Volendo replicare l'esempio della tabella numerica della sezione 8.7 possiamo scrivere il seguente template:

```
local lustache = require "lustache"

local tmpl = [[
\begin{tabular}{crrrr}
\toprule{{! package booktabs required }}
\( x \) & \( x^2 \) & \( x^3 \){{!
     }} & \( x^4 \) & \( x^5 \)\\
\midrule
{{#rows
}}{{x1}} & {{x2}} & {{x3}}{{!
      }} & {{x4}} & {{x5}}\\
{{/rows
}}\bottomrule
\end{tabular}]]

local rows = {}
for x = 1,10 do
    rows[#rows+1] = {
        x1 = x,
        x2 = x^2,
        x3 = x^3,
        x4 = x^4,
        x5 = x^5,
    }
end
local data = {rows=rows}
local doc = lustache:render(tmpl, data)
print(doc)
```

Anche in questo caso le caratteristiche avanzate di Lua e della libreria `lustache` possono essere impiegate per offrire una soluzione più elegante variando il dato associato ai nomi `x1`, ... `x5` del template: anziché dati numerici è possibile associare loro delle funzioni.

Questa volta la tabella con i dati per il template, chiamata `data`, conterrà alla chiave `rows` una tabella/array contenente la serie di tabelle con chiave `x`. Tale serie verrà passata in sequenza alle funzioni, e quindi alle chiavi `x1`, ... `x5`, le funzioni di calcolo.

È semplice approfittarne di queste funzioni per aggiungere spazi bianchi in numero sufficiente per incolonnare a destra i valori; a tale proposito utilizzeremo la funzione di libreria `string.format()`.

Il listato seguente implementa quanto detto utilizzando lo stesso template del codice precedente. Il risultato del codice LaTeX compilato è riportato nella tabella 2:

```
local lustache = require "lustache"
```

Tabella 2: La tabella ricavata dal codice Lua riportato nel testo che fa uso della tecnologia dei template implementata nella libreria `lustache`.

x	x^2	x^3	x^4	x^5
1	1	1	1	1
2	4	8	16	32
3	9	27	81	243
4	16	64	256	1024
5	25	125	625	3125
6	36	216	1296	7776
7	49	343	2401	16807
8	64	512	4096	32768
9	81	729	6561	59049
10	100	1000	10000	100000

```
local tmpl = [[
\begin{tabular}{crrrr}
\toprule{{! package booktabs required }}
\( x \) & \( x^2 \) & \( x^3 \){{!
     }} & \( x^4 \) & \( x^5 \)\\
\midrule
{{#rows
}}{{x1}} & {{x2}} & {{x3}}{{!
  }} & {{x4}} & {{x5}}\\
{{/rows
}}\bottomrule
\end{tabular}]]

local rows = {}
for i=1,10 do
    rows[#rows+1] = {x=i}
end

local f = string.format
local data = {
rows = rows,
x1=function (t) return f("%2d", t.x  ) end,
x2=function (t) return f("%3d", t.x^2) end,
x3=function (t) return f("%4d", t.x^3) end,
x4=function (t) return f("%5d", t.x^4) end,
x5=function (t) return f("%6d", t.x^5) end,
}

local doc = lustache:render(tmpl, data)
print(doc)
```

8.14 Una funzione come tag di un template

La libreria `lustache` ammette che i tag di blocco possano corrispondere a funzioni. Il testo risultante sarà il valore restituito dalla funzione associata al tag.

Cominciamo col mostrare il funzionamento di questa tecnica scrivendo una funzione che restituisce il contenuto del blocco racchiuso in una macro LaTeX (il primo argomento tabella può essere ignorato):

```
local lustache = require "lustache"

local tmpl = [[
{{#bf}}a great idea{{/bf}}]]
```

```
local doc = lustache:render(
    tmpl,
    {bf = function (tab, rendered)
        return "\\textbf{"..rendered.."}"
    end
    }
)
print(doc)
```

Il risultato è il seguente:

```
\textbf{a great idea}
```

Il tag *funzione* può contenere ulteriori tag, per esempio:

```
local lustache = require "lustache"

local tmpl = [[
{{#bf}}a great {{ thing }}{{/bf}}]]

local function enclose_bf(tab, rendered)
    return "\\textbf{"..rendered.."}"
end

local doc = lustache:render(
    tmpl,
    {thing = "song", bf = enclose_bf}
)
print(doc) --> "\textbf{a great song}"
```

Questa funzionalità potrebbe essere sfruttata per formattare numeri decimali poiché, nella sua essenzialità, la libreria `lustache` non offre la possibilità di applicare un formato ai dati. L'idea di per sé è corretta perché deve essere il template il componente responsabile dell'aspetto da far assumere ai dati (principio di separazione tra modello e vista).

Nel prossimo esempio di codice questo concetto è implementato con una funzione generale chiamata `format()` perché sia semplice creare ulteriori funzioni di formato come la `num3()`:

```
local lustache = require "lustache"

local tmpl = [[
\( \pi = {{#num3}}{{pi}}{{/num3}}\).
\( e = {{#num3}}2.718281828459{{/num3}}\).
]]

local function format(x, prec)
    local fmt = string.format(
        "%%0.%df",
        prec
    )
    return string.format(fmt, x)
end

local doc = lustache:render(
    tmpl,
    {pi = math.pi,
    num3 = function (t, x)
        return format(x, 3)
    end
    }
)
print(doc)
```

8.15 Templating ricorsivo

In precedenza abbiamo accennato al fatto che un template può contenerne altri per modellare un testo strutturato. La struttura testuale espressa per mezzo di template ricorsivi è molto potente ma può risultare complessa da gestire.

Nei termini della libreria `lustache` un subtemplate viene chiamato *partials*. Esso è un normale template i cui tag devono però far riferimento diretto ai dati forniti al template principale.

Nel template di livello superiore invece, un apposito tag caratterizzato dal segno di maggiore (`>`) inserito subito dopo il delimitatore di apertura espanderà il template *figlio*. Quest'ultimo dovrà essere contenuto in una tabella passata come terzo argomento alla funzione `render()`.

La costruzione del codice LaTeX per inserire una figura come oggetto mobile è un esempio opportuno:

```
\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{A great image.}
\label{fig:animage}
\end{figure}
```

Per generare questo testo definiamo un template principale che esprime il modello generale di un ambiente LaTeX contenente un secondo template per il contenuto. Modificando al solito i delimitatori di default avremo:

```
local tmpl_env = [[
{{=<| |>=}}\begin{<| env |>}
<|> body |>\end{<| env |>}]]
```

Il template interno accoglierà i due campi variabili opzionali per la posizione centrata dell'immagine e la larghezza imposta e i tre campi obbligatori per il nome del file, il testo della didascalia e l'etichetta di riferimento.

Il fatto che un dato opzionale può non esistere si riflette nel modo in cui il tag associato è inserito nel template, cosa che lo rende meno leggibile, come è possibile constatare nel seguente script (la cosa è specialmente vera per la prima parte del template interno):

```
local lustache = require "lustache"

local tmpl_env = [[
{{=<| |>=}}\begin{<| env |>}
<|> body |>\end{<| env |>}]]

local partials = {body = [[
{{#center}}\centering
{{/center}}\includegraphics{{#width
}}[width={{width}}]{{/width}}{{=<|
```

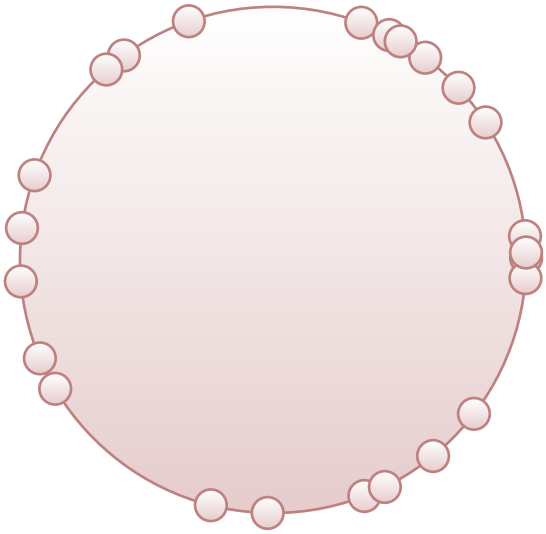


FIGURA 5: Una figura in Ti*k*Z il cui codice sorgente è stato generato da un programma che utilizza la tecnologia dei template annidati, scalata rispetto alle dimensioni originali.

```
|>=}}{<| file |>}
\caption{<| caption |>}
\label{<| label |>}
]]}

doc = lustache:render(
      tmpl_env,
      {
        env="figure",
        center = true,
        width = "\\textwidth",
        file = "image",
        caption = "A great image.",
        label = "fig:animage",
      },
      partials
)
print(doc)
```

Un'ulteriore discussione sui template annidati è quella che riguarda la realizzazione del codice che produce la figura 5 con l'utilizzo del pacchetto grafico Ti*k*Z (TANTAU, 2015). Si tratta di disporre in modo casuale 24 cerchietti su una circonferenza.

La struttura che desideriamo dare al template principale corrisponde a un ambiente con opzioni; il modello realizzato comprende due sottostrutture a cui corrisponderanno altrettanti template interni chiamati rispettivamente `option` e `body`:

```
local tmpl_env = [[
{{=<% %>=}}\begin{<% env %>}<%> option %>
<%> body %>
\end{<% env %>}]]
```

Il primo sub-template, `option`, è semplice: contiene le parentesi quadre che racchiudono un tag associato alla stringa completa il cui codice definisce lo stile Ti*k*Z dei cerchietti. Indubbiamente una soluzione che potrà essere sostituita da un'altra più sofisticata.

Poiché la definizione dello stile fa uso del carattere slash occorre cambiare il comportamento della libreria `lustache` per l'escaping HTML che trasformerebbe il carattere `/` nella sequenza `&#x2F;`. La modifica si imposta nel tag inserendo tre parentesi graffe di delimitazione oppure premettendo al nome del campo contenuto il carattere `&`:

```
local tmpl_opt = [[
[{{ &style_def }}]
]]
```

Il secondo template, `body`, contiene l'istruzione per il disegno della circonferenza di raggio dipendente da un parametro e l'inserimento di istruzioni `\node` nelle coordinate casuali del piano x e y:

```
local tmpl_body = [[
    \draw[dotstyle] (0,0) circle{{!
    }} [radius={{ radius }}cm];
{{#node_list}}{{!
}}    \node[dotstyle] at ({{x}}, {{y}}) {};
{{/node_list}}]]
```

Il codice completo dello script Lua è il seguente:

```
local lustache = require "lustache"

local tmpl_env = [[{{=<% %>=}}<%!
%>\begin{<% env %>}<%> option %><%!
%><%> body %><%!
%>\end{<% env %>}]]

local tmpl_opt = [[
[{{ &style_def }}]
]]

local style_def = [[
dotstyle/.style={
    circle, % the shape
    minimum size=5mm, % the size
    % the border:
    very thick,
    draw=red!50!black!50,
    % the filling:
    top color=white,
    bottom color=red!50!black!20
}]]

local tmpl_body = [[
    \draw[dotstyle] (0,0) circle{{!
    }} [radius={{radius}}cm];
{{#node_list}}{{!
}}    \node[dotstyle] at ({{x}}, {{y}}) {};
{{/node_list}}]]

local radius = 4 -- cm
local n = 24
local node = {}
for i=1,n do
    local alpha = 2*math.pi*math.random()
    node[#node+1] = {
        x = radius*math.cos(alpha),
        y = radius*math.sin(alpha),
    }
end
```

```
doc = lustache:render(
    tmpl_env, -- main template
    { -- data
        env="tikzpicture",
        style_def = style_def,
        node_list = node,
        radius = radius, -- cm
    },
    {-- sub template
        option = tmpl_opt,
        body = tmpl_body,
    }
)
print(doc)
```

I parametri principali come il raggio della circonferenza e il numero dei cerchietti sono memorizzati in variabili nella memoria locale del programma. Anziché con un unico template, il modello di documento è rappresentato con template specializzati per la struttura testuale associata.

9 Primo caso d'uso: griglia di attività

Questo esempio mostrerà come generare il sorgente LaTeX per comporre una tabella le cui righe rappresentano lo svolgimento di attività annuali di un gruppo di persone. Ogni cella della tabella, individuata da una colonna relativa agli anni e da una riga relativa alla persona, conterrà una barra il cui colore rappresenterà un'attività. La tabella 3 mostra un'istanza specifica di tale griglia.

L'insieme dei dati reali — fornito da Francesco Endrici — è stato opportunamente modificato sostituendo nomi e definizioni senza per questo alterare la struttura del problema. Per tradurli nella griglia utilizzeremo Lua e le sue tecniche di elaborazione delle stringhe sfruttando il costruttore delle tabelle — l'unica struttura dati di base prevista nel linguaggio — come sintassi formale per descrivere le informazioni.

9.1 Data description

Una tabella di Lua è un *dizionario*, una struttura dati formata da chiavi univoche a cui è associato un valore che può essere a sua volta una tabella. Questo tipo non solo è un contenitore molto flessibile ma ammette una sintassi letterale ispirata a BibTeX dove le chiavi stringa compaiono senza delimitatori. Questa particolarità consente al costruttore di tabelle di Lua di essere utilizzato anche come formato di descrizione dati strutturato ed espressivo. Un file di testo prodotto dall'utente tramite un editor può diventare così una sorta d'archivio con i seguenti vantaggi:

- semplice e immediata modalità d'inserimento e modifica dei dati da parte dell'utente;
- livello arbitrario di complessità;
- valutazione diretta di espressioni letterali;
- possibilità d'inserire dei commenti;
- direttamente interpretabile da Lua senza alcuna necessità di realizzare un parser per l'elaborazione successiva.

La struttura può essere disegnata per esprimere quella del problema nei termini in cui l'utente *pensa* alle informazioni. Tornando all'esempio — confronta la riga 3 della tabella 3 — il fatto che una persona di nome Sherri Castro ha svolto l'attività A nel 1972 e dal 1974 al 1976 potrebbe essere espresso scrivendo:

```
name: Sherri Castro, A: 1972, 1974->1976
```

In lua la tabella corrispondente alla sintassi letterale del costruttore sarebbe:

```
{name="Sherri Castro", A={1972,{1974,1976}}}
```

che a parole è descritta come una tabella con due chiavi, una per il nome e una per l'attività a cui è associata una seconda tabella che elenca anni e intervalli di anni (a loro volta tabelle).

Siamo liberi di indentare questa struttura per rappresentarla più chiaramente così:

```
{   name = "Sherri Castro",
    A = { 1972,
          {1974,1976}, -- triennio
    }
}
```

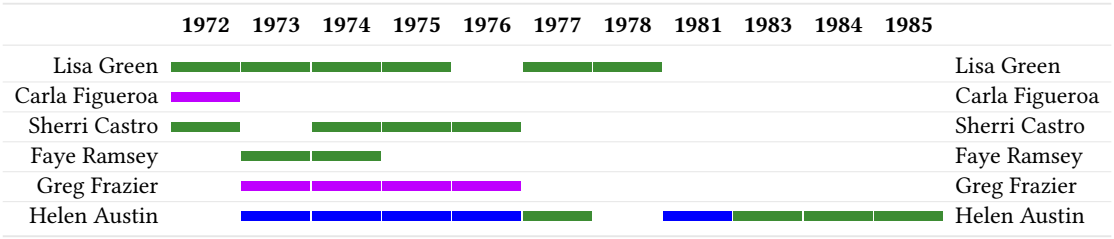
Modifichiamo la tabella nella versione definitiva per poter assegnare al membro del gruppo lo svolgimento di più attività: basta introdurre una tabella per un ulteriore livello di annidamento che associamo alla chiave `task`. In questo modo si rimuove anche la limitazione della precedente struttura per cui il codice delle attività non poteva essere identificato con la chiave `name`:

```
{   name = "Helen Austin",
    task = {-- cfr. riga 6 tabella
        A = {1977, {1983, 1985}},
        B = {{1973,1976}, 1981}
    }
}
```

9.2 Caricamento dei dati

Abbiamo due modalità per leggere le tabelle Lua: la prima è quella d'inserire i dati in un file dove un'istruzione `return` è seguita da una tabella che contiene tutte le altre e poi caricare il file nello script tramite la funzione predefinita `dofile()` che ne interpreta il contenuto come codice Lua. La seconda è premettere alle definizioni di ciascuna tabella un nome. Questa sintassi verrà interpretata da Lua come la richiesta dell'esecuzione della

Tabella 3: Porzione della tabella di grandi dimensioni che rappresenta il grafico dello svolgimento di varie attività annuali di un gruppo di persone, prodotta elaborando un file di testo contenente i dati espressi nel formato di *data description* di Lua. La tabella originale comprende le attività di circa 160 persone nell'arco di tempo di circa 65 anni.



funzione con lo stesso nome e con unico argomento la tabella stessa.[13] Nello script dovremo prima definire la funzione e poi caricare il file dati ancora tramite `dofile()`.

Poiché la sua sintassi è più significativa, useremo questo secondo modo. Il seguente è un frammento del file dati a cui corrisponde la tabella 3 in cui la funzione è stata chiamata `service()`:

```
service{name = "Lisa Green",
    task = {
        ASCI = {{1972,1974},},
        EG = {1975, {1977,1978},},
    }
}

service{name = "Carla Figueroa",
    task = {
        AsE = {1972,},
    }
}

service{name = "Sherri Castro",
    task = {
        ASCI = {1972,1974,},
        EG = {{1975,1976},},
    }
}

service{name = "Faye Ramsey",
    task = {
        ASCI = {{1973,1974},},
    }
}

service{name = "Greg Frazier",
    task = {
        AsE = {{1973,1976},},
    }
}

service{name = "Helen Austin",
    task = {
        EG = {1977, {1983,1985},},
        LC = {{1973,1976},1981,},
    }
}
```

13. In Lua le parentesi tonde della chiamata di funzione possono essere omesse se l'argomento è unico e se questo è di tipo stringa oppure di tipo tabella.

La virgola dopo l'ultimo elemento di una tabella è opzionale; inserirla facilita l'aggiunta di eventuali altri elementi, perciò considereremo il suo inserimento una buona prassi. Nel codice precedente sarebbe anche preferibile, per lo stesso motivo, dedicare una riga per ciascun anno o per ciascun intervallo di anni d'attività.

9.3 Il codice di lettura

Con la sola funzione `service()` non è possibile costruire il codice LaTeX della riga di griglia. Per farlo è necessario conoscere *tutti* gli anni di colonna per poter inserire il numero corretto di separatori di cella. Il compito di questa funzione sarà allora quello di memorizzare i dati in alcune variabili globali per la successiva elaborazione d'insieme.

Per semplicità abbiamo scelto di non includere nel codice controlli di validità, per esempio per avvertire l'utente che nei dati relativi a una stessa attività un anno è stato specificato più volte o che per uno stesso anno la stessa persona ha più attività.

Poiché gli anni possono essere espressi come valori singoli o come intervalli, introduciamo la funzione ausiliaria `expand_year()` che linearizza gli intervalli e restituisce l'array dei valori semplici. La funzione `service()` mapperà gli anni in un dizionario a valori booleani per tener traccia delle colonne della griglia e aggiungerà una riga associando gli anni con le attività compiute dalla singola persona:

```
-- data section

local year_map = {}
local row_data = {}

-- expand the years tab in a plain array
local function expand_year(t_acty)
    local y_list = {}
    for _, elem in ipairs(t_acty) do
        if type(elem) == "table" then
            for y = elem[1], elem[2] do
                y_list[#y_list+1] = y
            end
        else
            y_list[#y_list+1] = elem
        end
    end
```

```
    return y_list
end

function service(t)
   local row = {name = t.name, years = {}}
   for acty, t_years in pairs(t.task) do
      for _, y in
           ipairs(expand_year(t_years)) do
         -- mapping global columns
         year_map[y] = year_map[y] or true
         -- insert activity
         row[y] = acty
      end
   end
   row_data[#row_data+1] = row
end
```

L'effetto di queste definizioni e la conseguente esecuzione del seguente frammento di codice scritto dall'utente:

```
service{name = "Helen Austin",
    task = {
        EG = {1977, {1983,1985},},
        LC = {{1973,1976},1981,},
    }
}
```

sarà l'aggiunta in coda alle altre righe della seguente tabella Lua contenente il nome che andrà posizionato nella prima e nell'ultima colonna della griglia e anno per anno tutte le attività:

```
{name = "Helen Austin",
 years = {
    [1977] = "EG",
    [1983] = "EG",
    [1984] = "EG",
    [1985] = "EG",
    [1973] = "LC",
    [1974] = "LC",
    [1975] = "LC",
    [1976] = "LC",
    [1981] = "LC",
}}
```

La mappa globale degli anni/colonna aggiornata per ogni esecuzione dalla funzione `service()` dovrà integrare questi dati. Nella prossima sezione vedremo infatti come la funzione `render_data()` tradurrà questa stessa tabella Lua per Helen Austin in una riga della tabella 3 nel seguente codice LaTeX qui suddiviso su due linee:

```
Helen Austin & & \LC & \LC & \LC & \LC & \EG
& & \LC & \EG & \EG & \EG & Helen Austin\\
```

9.4 Il codice di elaborazione

Al termine dell'esecuzione di tutte le funzioni `service()` contenute nel file dati, saranno disponibili in memoria tutte le informazioni necessarie alla creazione della griglia in due tabelle Lua: la mappa degli anni di attività per tutte le persone e l'array con i dati delle righe. Passando le relative variabili alla funzione `render_data()` produrremo un file contenente un ambiente tabular che sarà a sua volta inserito in un sorgente LaTeX principale, basato sulla classe standalone, che definirà i colori, le dimensioni, i font, le macro e caricherà i pacchetti necessari.

In particolare, faremo uso del pacchetto array per ottenere colonne centrate e della stessa larghezza, il pacchetto booktabs per i filetti orizzontali della prima e dell'ultima riga, il pacchetto xcolor per definire i colori delle barre nelle celle e il pacchetto colortbl per assegnare il colore dei filetti.

Il contenuto delle celle non vuote sarà una barra colorata orizzontale prodotta da una macro il cui nome corrisponderà a quello dei codici identificativi delle attività usati nei dati. Per esempio, all'interno del sorgente principale la definizione delle grandezze dimensionali in sintassi LaTeX comuni a tutte le barre, la definizione del colore e della macro per l'attività "EG" sarà:

```
% definition of the bar colors
...
\definecolor{dep}{rgb}{0.24,0.55,0.20}
...

% column width: 1.5 the length of ``2012''
\newlength{\tcellwidth}
\settowidth{\tcellwidth}{2012}
\setlength{\tcellwidth}{1.5\tcellwidth}

% definition of the bar dimensions
\newlength{\tbarheight}
\setlength{\tbarheight}{5pt}
\newlength{\tbarwidth}
\setlength{\tbarwidth}{0.97\tcellwidth}
\newlength{\tbarup}
\setlength{\tbarup}{0.5pt}

% the bar 'factory' command
\newcommand{\servicebar}[1]{{%
\color{#1}%
\rule[\tbarup]{\tbarwidth}{\tbarheight}%
}}

% the bar commands
...
\newcommand{\EG}{\servicebar{dep}}
...
```

Tornando a Lua, la funzione `render_data()` utilizzerà la mappa degli anni/colonna per generare la definizione delle colonne, parametro obbligatorio dell'ambiente tabular, e la prima riga d'intestazione, mentre utilizzerà i dati delle righe per generare la sequenza di celle separate dal carattere '`&`'. Ancora una volta il processo memorizzerà l'intero ambiente tabular riga per riga in una tabella Lua che verrà poi unita nel testo finale in modo efficiente con la funzione predefinita `table.concat()`.

Il codice completo è riportato di seguito. Esso non è né semplice né elegante perché disseminato di stringhe di codice LaTeX che lo rendono poco leggibile. Utilizzare le tecniche basate sulle stringhe

era del resto una scelta iniziale del progetto ma lasciamo al lettore per utile esercizio riscrivere il codice questa volta utilizzando il template engine `lustache`.

```
-- rendering data section

function render_data(t_years, t_data)
    -- ordered list of column years
    local col_years = {}
    for y in pairs(t_years) do
        col_years[#col_years+1] = y
    end
    table.sort(col_years)

    local rs = {}
    -- tabular header
    local cdef = "r@{\\hspace{0.5em}}"..
        "*{%d}{@{}C{\\tcellwidth}@{}}"..
        "@{\\hspace{0.5em}}l"
    rs[#rs+1] = string.format(
        "\\begin{tabular}{%s}",
        string.format(cdef, #col_years)
    )
    rs[#rs+1] = "\\toprule"
    local colfmt = {}
    for _, y in ipairs(col_years) do
        colfmt[#colfmt+1] = string.format(
            "\\textbf{%d}",
            y
        )
    end
    rs[#rs+1] = "& "..
        table.concat(colfmt, " & ")..
        " &\\\\"
    rs[#rs+1] = "\\midrule"

    -- tabular body
    for i, row in ipairs(t_data) do
        local rndrow = {}
        for _, year in ipairs(col_years) do
            if row[year] then
                rndrow[#rndrow+1] = "\\"..
                    row[year]
            else
                rndrow[#rndrow+1] = " "
            end
        end
        local hline
        if i == #t_data then
            hline = ""
        else
            hline = "\\hline"
        end

        rs[#rs+1] = row.name..
            " & "..
            table.concat(rndrow, " & ")..
            " & "..
            row.name..
            "\\\\"..
            hline
    end

    -- tabular ending
    rs[#rs+1] = "\\bottomrule"
    rs[#rs+1] = "\\end{tabular}"
    return rs
end
```

9.5 Il codice di scrittura

Non rimane che completare lo script scrivendo il codice che carica il file dati, esegue la funzione di traduzione nel codice LaTeX e salva il risultato in un file esterno:

```
-- execution section

dofile("services.lua")
local t = render_data(year_map, row_data)
-- save the LaTeX file
local tex = io.open("tabdata.tex", "w")
tex:write(table.concat(t,"\n"))
tex:close()
```

9.6 Considerazioni finali sulla griglia

Rispetto alla soluzione originale scritta in LaTeX3 quella presentata qui in Lua non è tanto più efficiente quanto è più leggibile e robusta e ciò porta a maggiori potenzialità di miglioramento per effetto dell'uso di un linguaggio ad alto livello progettato espressamente anche per questi compiti.

Per esempio, non sarebbe difficile modificare il programma per considerare anche il caso in cui una persona svolga più attività in uno stesso anno, oppure per eseguire l'ordinamento temporale delle righe della griglia evitando all'utente l'obbligo di inserire in sequenza le tabelle dati `service`, oppure per creare *griglie finestra* selezionando un intervallo di anni.

Mentre il formato di dati originale è un file `.csv` (*comma separated values*) proveniente da un foglio di calcolo, in questo esempio applicativo le stesse informazioni sono state tradotte in un formato di *data description* direttamente interpretabile da Lua sfruttando la flessibile sintassi del suo costruttore di tabelle. Con esso è stata progettata una struttura espressiva pensando alle informazioni inerenti allo svolgimento di attività annuali da parte di componenti di un gruppo che l'utente può inserire manualmente.

10 Python

L'ultimo linguaggio considerato in questo lavoro è Python, diffusissimo linguaggio di scripting per molto tempo contrapposto a Perl.

10.1 Stringhe e campi

In Python3 le stringhe sono codificate Unicode *utf-8* perciò preferiamo questa versione — in particolare la `3.4` — rispetto a quella della precedente serie stabile. Al pari di Lua, il linguaggio prevede diverse sintassi per inserire valori stringa letterali usando due diversi delimitatori: gli apici semplici o doppi per le stringhe di un'unica riga oppure tre

apici semplici o doppi consecutivi per le stringhe multiriga. Premettendo il modificatore `r` (*raw*) si disinseriscono le sequenze di escape, cosa utile, per esempio, nei casi di stringhe contenenti il carattere backslash delle macro TEX.

Nelle prossime sezioni è riportato del codice che fa uso della sintassi per le stringhe grezze sia mono che multiriga e della funzione `format()` operante come metodo del tipo stringa per testi con campi variabili.

10.2 Costruire una tabella

Diversamente dagli esempi precedenti, nel prossimo script il testo LATEX della tabella numerica già realizzata in Lua non verrà stampato a video sul canale di uscita ma sarà memorizzato in un file su disco. Questo ci darà modo di presentare le istruzioni necessarie per accedere al file system.

```
# creating a new list also called 't'
t = [r"\begin{tabular}{crrrr}"]
t.append(r"\toprule")
t.append(r"\(x\) & \(x^2\) & \(x^3\)"
         r" & \(x^4\) & \(x^5\)\\")
t.append(r"\midrule")

f = (r" {0:2d} & {1:3d} & {2:4d} &"
     r" {3:5d} & {4:6d}\\")
for x in range(1,11):
    t.append(
        f.format(x, x**2, x**3,
        x**4, x**5
    ))

t.append(r"\bottomrule")
t.append(r"\end{tabular}")

# saving content in the TeX file
fl = open("test-table.tex", "w")
fl.write("\n".join(t))
fl.close()
```

Nonostante le molte differenze concettuali tra Lua e Python, i due programmi per produrre la tabella numerica d'esempio sono molto simili. Nel caso di Python si usa una *lista* allo stesso modo della tabella/array in Lua, il metodo `format()` al posto della funzione `string.format()` e il metodo `join()` al posto della funzione `table.concat()`.

La figura 6 mostra l'editor IDLE, dedicato alla programmazione in Python, al lavoro sul codice del programma appena presentato.

10.3 `Jinja` un template engine per Python

Per proseguire nello studio della tecnologia dei template con il linguaggio Python abbiamo scelto il modulo `Jinja2` — versione 2.7.2 per Python3; la figura 7 ne mostra il logo — perché molto ricco di funzionalità; tra esse abbiamo i delimitatori adattabili per la generazione di sorgenti LATEX e la precompilazione dei template per incrementare le prestazioni.

La prima differenza che incontriamo rispetto a `lustache` è che i tag dei template non hanno un carattere marcatore che ne specifica il tipo — come il `#` per un blocco o il `!` per un commento — ma delimitatori diversi per ciascun tipo. I tipi di tag in `Jinja` sono quattro e sono qui elencati con i delimitatori di default:

- `{% ... %}` istruzioni
- `{{ ... }}` espressioni valutate e inserite nel testo finale,
- `{# ... #}` commenti non inclusi nell'uscita del template,
- `# ... ##` istruzioni in linea.

10.4 Merging PDF file

Come primo esempio con il modulo `Jinja` studieremo la creazione di un file sorgente LATEX per l'unione di più file PDF tramite il pacchetto `pdfpages`.

Nel template troveremo un tag istruzione per un ciclo iterativo che genererà la sequenza di macro `\includepdf` con una sintassi molto simile a quella in Python. La variabile d'iterazione è inserita in un tag espressione con delimitatori personalizzati:

```
tmpl = r"""
\documentclass{minimal}
\usepackage{pdfpages}
\begin{document}
{% for doc in doc_list -%}
    \includepdf[pages=-]{<<| doc |>>}
{% endfor -%}
\end{document}
"""
```

Il template è molto più leggibile che non l'equivalente per `lustache` perché possiamo inserire prima del tag di chiusura il modificatore trattino `-`, che comporta l'eliminazione dei caratteri spazio e di ritorno a capo dal testo finale successivi al delimitatore di chiusura del tag. Questo comportamento può essere reso implicito configurando opportunamente gli oggetti restituiti dal modulo.

Veniamo all'esame del codice dello script: per prima cosa si carica l'oggetto `Environment` del modulo `jinja2` istanziandone l'oggetto con il costruttore a cui vengono passate le nuove stringhe di delimitazione dei tag di tipo *variable*. La modifica dei delimitatori non è quindi locale al template ma globale nell'ambiente.

Il template binario è ottenuto tramite il metodo `from_string()` dell'ambiente mentre il testo finale viene generato chiamando il metodo `render()` con la tupla dei nomi dei file da unificare:

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-

from jinja2 import Environment
```

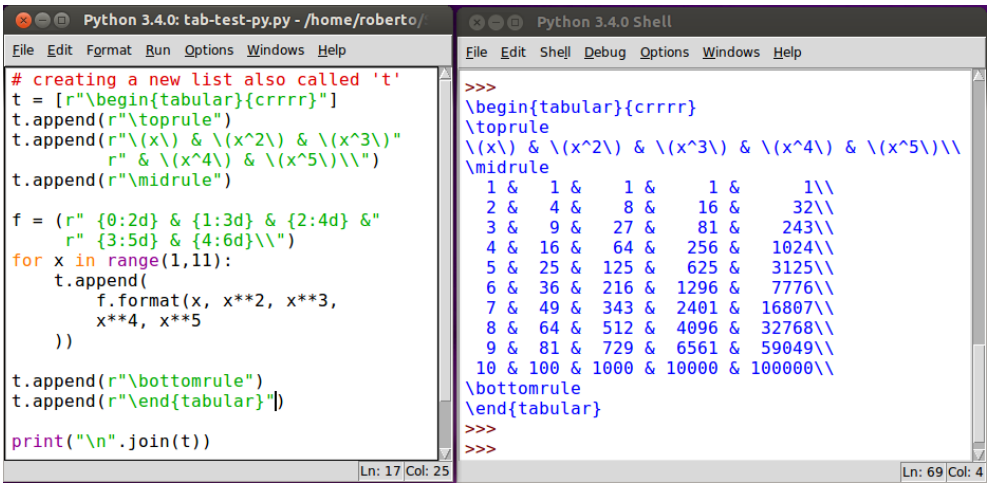


FIGURA 6: L'editor IDLE in azione mostra una finestra principale con il codice dello script in Python e una finestra di shell interattiva che raccoglie l'uscita dei programmi. Per eseguire il codice è sufficiente premere il tasto F5.

FIGURA 7: Il logo del progetto Jinja rappresenta un tempio giapponese per l'assonanza tra i termini 'template' e 'temple'.

```
env = Environment(
    variable_start_string="<<|",
    variable_end_string="|>>"
)

tmpl = r"""
\documentclass{minimal}
\usepackage{pdfpages}
\begin{document}
{% for doc in doc_list -%}
    \includepdf[pages=-]{<<|doc|>>}
{% endfor -%}
\end{document}
"""

t_eng = env.from_string(tmpl)

docs = ("as541","as656","as163",
        "as048","as525","as854")
print(t_eng.render(doc_list=docs))
```

10.5 Pipeline e filtri

In `Jinja` esiste il concetto di *pipeline* di comandi: l'uscita di un comando verrà passata come ingresso alla funzione successiva separata da un carattere `|` che simboleggia un condotto. Questa funzionalità è chiamata *filtro*, una scelta non particolarmente felice perché essa è un'operazione di mappatura e non di filtraggio.

Come esempio, nel prossimo script applicheremo un filtro al titolo di sezione per renderlo in caratteri maiuscoli e per arrotondare un numero decimale a sei cifre significative:

```
from jinja2 import Environment
import math

env = Environment(
    variable_start_string="<<|",
    variable_end_string="|>>"
)

tmpl = r"""
\section{<<| sec_tit|upper |>>}

A rounded version of the \( \pi \) value:
<<| "%0.6f"|format(pi) |>>.
"""

t_eng = env.from_string(tmpl)
print(t_eng.render(sec_tit="Important",
    pi = math.pi))
```

Si tratta di dettagli di *presentazione* dei dati perciò il template non viola troppo la regola fondamentale di separazione tra modello e vista — in questo caso tra dati e testo finale — includendone la logica. Le operazioni di elaborazione dovranno riguardare esclusivamente proprietà di visualizzazione e mai la costruzione dei dati. In altre parole, al template è deputata la responsabilità di presentare i dati elaborandoli opportunamente.

Nell'esempio precedente il template non conosce il valore di π e si occupa solamente di arrotondare un numero decimale.

Tabella 4: Una semplice tabella esaminata nel testo come vista sui dati. Il template si occupa dei dettagli come la numerazione delle righe e il calcolo del totale di colonna.

N.	Description	Cost
1	Site preparation	72030.00
2	Excavation work	242000.00
3	Foundations	5078000.00
4	Main framework	8925000.00
	Total	14317030.00

10.6 Materiale tabellare

Consideriamo la tabella 4 come *vista* sui dati. Vorremmo che il template si occupasse della formattazione dei numeri a due posizioni decimali fisse, del calcolo del totale di colonna e della numerazione automatica delle righe.

Tutto ciò è piuttosto semplice in `Jinja` grazie ad alcune funzioni predefinite. Abbiamo già incontrato la funzione `format()` messa in pipeline con la stringa di formato. Un'altra utile funzione è `sum()`, che riceve una lista per sommarne gli elementi e opzionalmente anche il nome dell'attributo associato al valore da considerare nel calcolo se il dato è, per esempio, un dizionario.

Per la numerazione delle righe è sufficiente usare l'oggetto `loop` che assieme a molte altre informazioni contiene il contatore delle iterazioni nel campo `index`.

Il manuale di riferimento online del *template engine* presenta tutte le funzioni con esempi di codice e non è difficile trovare proprio la funzione necessaria a risolvere un dato problema. Di seguito il codice completo dello script che genera la tabella 4.

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-

from jinja2 import Environment
env = Environment()

tmpl = r"""
\begin{tabular}{clrr}
\toprule
N. & Description & Cost\\
\midrule
{% for w in data_list -%}
    {{ loop.index -}}
    & {{ w.descr -}}
    & {{ "%0.2f"|format(w.cost) }}\\
{% endfor -%}
\midrule
  & Total & {{
  "%0.2f"
  |format(
      data_list
      |sum(attribute='val')) }}\\
\bottomrule
\end{tabular}
"""
t_eng = env.from_string(tmpl)
data = (
{'descr':"Site preparation",'val':  72030},
{'descr':"Excavation work", 'val': 242000},
{'descr':"Foundations",     'val':5078000},
{'descr':"Main framework",  'val':8925000},
)
print(t_eng.render(data_list=data))
```

Nel listato 1 è riportato il codice esteso dello script precedente in cui viene aggiunta una colonna i cui valori sono le percentuali date dei valori corrispondenti di quella dei costi.

La particolarità sta nel calcolo della somma di queste percentuali a opera del template: nonostante la possibilità di utilizzare variabili tramite il tag `set`, non è possibile leggerne il valore modificato all'interno del ciclo `for` perché gli incrementi riga per riga rimangono *locali* al blocco.

Abbiamo quindi utilizzato la lista `vatlist`, gestita in memoria come oggetto globale, appendendovi i singoli valori all'interno di un blocco `if` vuoto. All'esterno del ciclo che produce le righe della tabella questa lista viene poi passata alla funzione `sum()` che calcola il totale desiderato.

Se nel modulo fosse previsto un tag `global set`, il codice ritornerebbe a essere quello che ci si aspetterebbe ma probabilmente le variabili globali invoglierebbero a inserire la logica dati nel template.

10.7 Funzioni custom

In questo esempio vedremo come utilizzare nel template le funzioni personalizzate. Lo scopo è, lo ricordiamo ancora, modificare solo l'aspetto dei dati per crearne una *vista*.

Lo script costruirà il codice LaTeX della seguente tabella composta da una prima colonna con i numeri non formattati e da una seconda colonna con i numeri formattati con la nuova funzione personalizzata.

No formatting	Formatting
45613854.156	45 613 854,16
546447.1	546 447,10
450215	450 215,00
454	454,00
7899.3	7899,30

Per poter chiamare una funzione in un tag istruzione del template è sufficiente registrarla nel dizionario `globals` dell'oggetto `Environment`. La chiave dovrà coincidere con il nome della funzione utilizzata internamente nel template.

Nello script si procede col definire una semplice funzione di formato numerico a precisione fissa a due decimali, che inserisce un piccolo spazio tra i gruppi di tre cifre se la parte intera del numero ne ha almeno cinque, secondo le norme del Sistema

Listato 1: Il codice dello script che genera una tabella compiendo calcoli percentuali e somme sui dati.

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-

from jinja2 import Environment

env = Environment()

tmpl = r"""
\begin{tabular}{clrr}
\toprule
N. & Description & Cost & \textsc{vat}\\
\midrule
{% set vatlist = [] -%}
{% for w in data_list -%}
    {# just a trick...
    #}{% if vatlist.append(w.cost*w.vat) -%}{%endif -%}
    {{ loop.index               -}}
    & {{ "%-18s"|format(w.descr) -}}
    & {{ "%10.2f "|format(w.cost) -}}
    & {{ "%8.2f"|format(w.cost*w.vat) }}\\
{% endfor -%}
\midrule
 & Total & {{
 "%0.2f"
 |format(data_list|sum(attribute='cost')) -}}
 & {{ "%0.2f"|format(vatlist|sum) }}\\
\bottomrule
\end{tabular}
"""

t_eng = env.from_string(tmpl)
data = (
    {'descr':"Site preparation", 'cost':  720, 'vat':.10},
    {'descr':"Excavation work",  'cost': 2420, 'vat':.16},
    {'descr':"Foundations",      'cost':50780, 'vat':.16},
    {'descr':"Main framework",   'cost':89250, 'vat':.16},
)
print(t_eng.render(data_list=data))

# program's output:
'''

\begin{tabular}{clrr}
\toprule
N. & Description & Cost & \textsc{vat}\\
\midrule
1& Site preparation  &     720.00 &    72.00\\
2& Excavation work   &    2420.00 &   387.20\\
3& Foundations       &   50780.00 &  8124.80\\
4& Main framework    &   89250.00 & 14280.00\\
\midrule
 & Total & 143170.00& 22864.00\\
\bottomrule
\end{tabular}
'''
```

Internazionale delle unità di misura.[14] La funzione tiene conto anche di un parametro opzionale che controlla il separatore decimale. Il codice completo dello script è il seguente:

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-

from jinja2 import Environment

def fmt_num(x, comma_sep=False):
    if int(x) > 9999:
        res = "{:,.2f}".format(x)
        res = res.replace(",","\\,")
    else:
        res = "{:.2f}".format(x)

    if comma_sep:
        return res.replace(".", ",")
    else:
        return res

env = Environment()
env.globals['fmt_num'] = fmt_num

tmpl = r"""
\begin{tabular}{rr}
\toprule
No formatting & Formatting\\
\midrule
{% for x in nums -%}
    {{x}} & {{ fmt_num(x, True) }}\\
{% endfor -%}
\bottomrule
\end{tabular}
"""
nums = (45613854.156,
        546447.1,
        450215,
        454,
        7899.3)
t_eng = env.from_string(tmpl)

print(t_eng.render(nums=nums))
```

10.8 Template ricorsivi

L'effetto è sostanzialmente simile a quello dei template figli di `lustache` esaminati alla sezione precedente: in un template possono essere inclusi altri template secondari per mezzo del tag `include`. La struttura che ne deriva è una gerarchia ad albero dall'alto verso il basso che corrisponde a un unico documento strutturato in parti e sottoparti.

Poiché i sorgenti LaTeX sono composti da parti e sottoparti questa funzionalità è particolarmente utile. Un esempio di strutturazione dei template è riportato nella sezione 11.

14. Ulteriori informazioni possono essere trovate nella documentazione del pacchetto `siunitx` oppure nella pagina web `http://physics.nist.gov/cuu/Units/checklist.html` al punto 16.

10.9 Ereditarietà dei template

L'ereditarietà è uno dei concetti più importanti della programmazione a oggetti. Consente di creare una classe contenente le funzionalità di base comuni a una famiglia di oggetti sempre più specializzati. Nel caso dei modelli di documento un template di base può essere esteso da un nuovo template indipendente.

Se desideriamo produrre documenti diversi che tuttavia abbiano in comune una serie di elementi, non potremo riusare il codice con i template ricorsivi perché l'inclusione dei moduli secondari sarà riferita a un template fisso e quindi a sempre e solo un unico documento. La ricorsione esprime cioè la struttura di un unico documento mentre l'ereditarietà rappresenta la struttura di una *famiglia* di documenti.

Nel template base si troverà almeno un tag di tipo istruzione di nome `block` che verrà poi, per così dire, *implementato* o *esteso* da un secondo template. Per esempio:

```
# base template:
t_m = r"""
\begin{document}
{% block content -%}
{% endblock -%}
\end{document}
"""
```

Il template che eredita un template base dichiarerà di estenderlo con il tag di nome `extends` e poi definirà i blocchi base. Per esempio:

```
# child template
t_c = """\
{% extends "main" -%}
{% block content -%}
Great {{ thing }}!
{% endblock -%}
 """
```

L'ereditarietà in `Jinja` è una funzionalità molto potente. Per esempio anche un singolo blocco può essere esteso chiamando la funzione `super()` o può essere stampato più di una volta tramite la chiave `self.<block-name>()` oppure ancora può essere inserito in un ciclo iterativo. Vediamo brevemente un esempio di ciascuna di queste funzionalità.

La parola chiave `super()`

La dimensione della pagina di una famiglia di documenti basati sulla classe article può essere fissata tra le opzioni nel template di base; tali opzioni potranno essere estese nel template derivato come in questo codice:

```
# base template
t_m = r"""
\documentclass[{% block option -%}
a4paper{% endblock %}]{article}
"""
```

```
t_c = """\
{% extends "main" -%}
{% block option -%}
{{ super() -}}
{% if opt %},{% endif -%}
    {{ opt|join(',') -}}
{% endblock -%}
 """
```

Se la lista `opt` contiene le stringhe `'12pt'` e `'draft'` il risultato sarà:

```
\documentclass[a4paper,12pt,draft]{article}
```

La parola chiave `self.<block-name>()`

Con lo stesso esempio dimostriamo la capacità d'inserire copie del testo di un blocco con la funzione `self.<block-name>()` riportata nel template di base:

```
# base template
t_m = r"""
\documentclass[{% block option -%}
a4paper{% endblock %}]{article}
% le opzioni di classe sono:
% {{ self.option() }}
"""
```

e il risultato sarà:

```
\documentclass[a4paper,12pt,draft]{article}
% le opzioni di classe sono:
% a4paper,12pt,draft
```

Blocchi innestati e la parola chiave `scoped`

Nel template di base un blocco è incluso all'interno di un ciclo iterativo, per esempio perché contiene il testo di una riga di una tabella composta con l'ambiente tabular. Perché le variabili di ciclo siano visibili all'interno del blocco è necessario specificare la chiave `scoped` per modificare il comportamento *difensivo* del template:

```
t_m = r"""
\begin{tabular}{cc}
{% for key, value in mydict.items() -%}
{% block row scoped -%}{% endblock -%}
{% endfor -%}
\end{tabular}
"""

t_c = """\
{% extends "main" -%}
{% block row -%}
 {{key}} & {{value}}\\\\
{% endblock -%}
 """
```

Se il dizionario contiene le seguenti informazioni `{'font':'12pt', 'paper':'a4paper', 'version':'draft'` il risultato sarà:

```
\begin{tabular}{cc}
paper & a4paper\\
version & draft\\
font & 12pt\\
\end{tabular}
```

10.10 Un esempio di ereditarietà

Memorizzando i template in file esterni avremo il vantaggio che le modifiche non interessanti un cambiamento della struttura dei dati non riguarderanno l'applicazione ma solamente questi file di testo. Negli esempi di questa sezione i template sono inclusi nel sorgente per scopi esplicativi e continueremo a farlo anche se sarà necessario introdurre funzioni diverse da quelle che avremmo utilizzato in un'applicazione reale.

In `Jinja` la sequenza di passi effettiva infatti è quella di utilizzare un apposito *loader* per leggere i template da disco — anche in forma precompilata. Questo è il motivo per cui il metodo `from_string()`, usato fino a ora negli esempi per generare oggetti `Template`, non prevede che i template possano rappresentare una gerarchia.

Tuttavia non è obbligatorio che i template si trovino memorizzati nel file system perché il modulo `Jinja` mette a disposizione l'oggetto `DictLoader` che elabora un dizionario e restituisce il loader con cui caricare le stringhe del template base e del template d'estensione con i nomi associati.

C'è da notare che il metodo `get_template()` dovrà richiedere all'oggetto `Environment` il template figlio e non quello base perché è il primo che ne eredita ed estende il contenuto.

Nell'esempio presentato il risultato atteso dei template sono i due sorgenti LaTeX che hanno lo stesso preambolo ma un diverso corpo:

```
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage[italian]{babel}
\begin{document}
Great job!
\end{document}

\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage[italian]{babel}
\begin{document}
Hello Albert!
\end{document}
```

Il codice completo dello script che li realizza è il seguente:

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-

from jinja2 import Environment, DictLoader

t_m = r"""
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage[italian]{babel}
\begin{document}
{% block content -%}
{% endblock -%}
```

```
\end{document}
"""

t_c1 = """\
{% extends "main" -%}
{% block content -%}
Great {{ thing }}!
{% endblock -%}
 """

t_c2 = """\
{% extends "main" -%}
{% block content -%}
Hello {{ name }}!
{% endblock -%}
 """

loader = DictLoader(
        {'main':t_m,
         'content1':t_c1,
         'content2':t_c2}
)
env = Environment(loader=loader)

tmpl1 = env.get_template('content1')
print(tmpl1.render(thing="job"))

tmpl2 = env.get_template('content2')
print(tmpl2.render(name='Albert'))
```

11 Secondo caso d'uso: l'inventario

In questo esempio reale mostreremo come produrre un report d'inventario di un esercizio commerciale interrogando un database relazionale. I dati sono memorizzati in un database SQLite3 (SQLite3). Libreria libera estremamente diffusa e collaudata, progettata per l'utilizzo efficiente su postazioni locali, SQLite3 non si basa sulla classica architettura *client/server* distribuita su una rete e non c'è quindi la necessità di effettuare alcuna configurazione d'accesso mentre è immediata la gestione dei dati salvati in un singolo, ordinario file locale. Si tratta comunque di un *database engine* transazionale ACID[15] *compliant*, multipiattaforma e disponibile praticamente per qualsiasi sistema operativo nella forma di un file sorgente scritto in linguaggio C per l'inclusione diretta in applicazioni o in librerie a caricamento dinamico.

Il programma di creazione del report sarà suddiviso in due principali fasi d'esecuzione: il reperimento delle informazioni dal database e la conseguente produzione del sorgente LaTeX a partire dalle informazioni ottenute; in dettaglio:

1. consultazione del database SQL:
   (a) connessione al database;
   (b) esecuzione delle query e inserimento dati in memoria;
   (c) chiusura della connessione.
2. costruzione del sorgente LaTeX:
   (a) definizione delle funzioni accessorie per la presentazione dei dati;
   (b) caricamento del template;
   (c) unione dei dati;
   (d) memorizzazione del risultato su file.

15. Per maggiori informazioni si consulti la pagina web https://en.wikipedia.org/wiki/ACID.

Tabella 5: Tabella con dati di esempio rappresentativa dell'aspetto del report d'inventario secondo le specifiche definite nel testo.

Supplier: *Good Shirts*

Description	Q.ty	Cost ($)	Total ($)
AQ12 PL21 — T-Shirt Pluton series	12	12.50	150.00
SZ02 PL24 — T-Shirt Pluton series	8	19.10	152.80
W964 JP02 — T-Shirt Jupiter series	6	12.90	77.40
BP01 JP10 — T-Shirt Jupiter series	20	15.80	316.00
XVYY E5 — T-Shirt Earth series	5	14.80	74.00
XVYY E19 — T-Shirt Earth series	3	12.20	36.60
XVYY E41 — T-Shirt Earth series	19	11.30	214.70
NEAR 956 — T-Shirt Moon series	5	14.80	74.00
	78		1 095.50

Come sarà ormai chiaro, scriveremo il programma in Python3, che fornisce con i propri moduli di libreria l'accesso immediato a database SQLite3, e produrremo il sorgente con il template engine `Jinja`. Questa procedura è più efficiente e affidabile che non quella basata sull'utilizzo di fogli di calcolo perché l'uso di un database assicura la coerenza e l'integrità dei dati e al contempo l'evoluzione futura della struttura relazionale.

Per dare un'idea delle prestazioni del processo descritto, il tempo di esecuzione per creare il sorgente di un inventario reale di circa 8000 articoli è risultato essere di 0,17 s sulla macchina di riferimento.

11.1 Specifiche del report

L'inventario annuale è l'elenco della merce presente in magazzino alla data del 31 dicembre. Questo elenco dovrà essere suddiviso per reparto e per fornitore riportando i totali parziali sia di quantità che di costo. Al termine dell'elenco dovrà essere presentata una tabella riassuntiva per reparto per dare le proporzioni generali di consistenza del magazzino.

Nell'elenco ogni riga sarà formata dalle seguenti informazioni: un breve testo descrittivo che definisce univocamente l'articolo, la quantità inventariata, il costo del singolo pezzo e il totale risultante. L'elenco dovrà apparire su due colonne per pagina e dovrà essere presente un frontespizio.

Nella tabella 5 è riportato un breve possibile estratto dell'elenco d'inventario composto secondo le specifiche decise per il documento finale.

11.2 Gerarchia dei template

Per produrre il report dell'inventario, anziché utilizzare un unico template ne introdurremo uno

principale che definirà il preambolo del sorgente e il frontespizio e altri due sub-template, uno per inserire l'elenco degli articoli e l'altro per comporre la tabella riassuntiva finale. In questo modo è più semplice gestire e mettere a punto le singole parti rispettando la struttura effettiva del documento finale. I template saranno salvati su file e potranno essere modificati indipendentemente dal codice dello script. Faremo uso di funzionalità della libreria `Jinja` non ancora presentate negli esempi già visti; questi ultimi includono i template come semplici variabili stringa definite nello script.

11.3 Il template principale

Useremo la classe LaTeX report con i pacchetti di base per impostare codifiche, dimensione della pagina, gabbia del testo, testatine, font, numero di colonne; useremo poi il pacchetto supertabular per comporre l'elenco degli articoli inventariati sotto forma di tabella a più pagine per ciascun reparto. Per brevità riporteremo solamente il corpo del sorgente LaTeX inserito nel template principale. Dopo aver definito le righe d'intestazione e di coda previste dal pacchetto supertabular, troviamo le due istruzioni per l'inclusione dei sub-template tramite il nome del corrispondente file racchiuso tra doppi apici:

```
\begin{document}
\maketitle
\twocolumn

\scriptsize
\sffamily

% intestazione tabella d'inizio pagina
\tablehead{
\hline
Descrizione articolo & Q.tà & Costo \euro &
Esist. \euro\\\hline}

\tabletail{% coda a fine pagina tabella
\hline
\multicolumn{4}{r}{\textit{segue\dots}}\\
}

% ultima riga tabella
\tablelasttail{\hline
\multicolumn{4}{r}{%
\textit{Fine tabella di reparto.}}\\}

<<% include "supertab-tmpl.tex" -%>>

\newpage
\onecolumn
\normalsize

<<% include "finaltable-tmpl.tex" -%>>

\bigskip
\rule[.5ex]{180pt}{1pt}
Fine documento
\rule[.5ex]{180pt}{1pt}
\end{document}
```

11.4 Il sub-template elenco

Nel seguente codice è riportato l'intero contenuto del sub-template che formerà l'elenco degli articoli secondo le specifiche del report. Tutto il testo è racchiuso in un ciclo eseguito per ogni reparto, il che significa che verranno prodotti altrettanti ambienti supertabular.

La funzione Python `items()`, definita per il tipo dizionario, restituisce un iteratore a due valori: il primo è la chiave e il secondo è il valore a essa associato. Corrispondentemente lo script definirà il campo `data` come un dizionario {⟨*nome reparto*⟩/⟨*dizionario fornitori*⟩}.

Per ogni reparto definiamo la prima riga della tabella dell'elenco e l'ambiente supertabular contenente un ulteriore ciclo annidato per ciascun fornitore sulle variabili {⟨*codice fornitore*⟩/⟨*lista articoli*⟩}.

Per ogni fornitore il ciclo di ultimo livello inserirà le righe della tabella con descrizione, quantità, costo e costo totale, a rappresentare la struttura riportata schematicamente in figura 8. Tutti i campi sono formattati per essere lunghi lo stesso numero di caratteri in modo che nel sorgente i dati risultino allineati in verticale — in particolare i valori di costo sono arrotondati a due cifre decimali — per mezzo delle funzioni `format()` e `truncate()`.

Il template prevede anche che sia definita una funzione di nome `get_sup_name()` la quale, dato il codice, restituisca il nome completo del fornitore e che i valori delle somme parziali di quantità e costi siano contenute in un dizionario strutturato per livelli ⟨*reparto*⟩/⟨*fornitore*⟩.

```
<<% for dep_name, sup_list in
                    data.items()-%>>
\tablefirsthead{
\hline
\multicolumn{4}{c}{%
--- REPARTO <<| dep_name|upper |>> ---}\\
\hline
}

\begin{supertabular}{lrrr}
\\\hline
Descrizione articolo & Q.tà & Costo \euro
& Esist. \euro\\\hline
<<% for sup_code, artlist
                  in sup_list.items() -%>>
\\
\multicolumn{4}{l}{%
Fornitore <<|get_sup_name(sup_code) |>>}\\
\hline
<<% for art in artlist -%>>
<<| "%-38s"|format(art.descr)
          |truncate(38, end="")
|>> & <<|   "%5d"|format(art.qty)
|>> & <<| "%6.2f"|format(art.cost)
|>> & <<| "%7.2f"|format(art.qty*art.cost)
```

```
|>> \\
<<% endfor -%>>
\hline
<<| "%-38s"|format("Tot.")
|>> & <<| "%5d"|format(sums[dep_name]
                          ['suptot']
                          [sup_code]
                          ['qty'])
|>> & & <<| "%8.2f"|format(sums[dep_name]
                          ['suptot']
                          [sup_code]
                          ['tot'])
|>> \\
\hline
<<% endfor -%>>
\\
\\
\hline
<<| "Tot. Rep. %-32s"|format(dep_name)
|>> & <<| "%5d"|format(sums[dep_name]
                          ['qtytot'])
|>> & & <<| "%8.2f"|format(sums[dep_name]
                          ['tottot'])
|>> \\
\hline
\end{supertabular}
<<% endfor -%>>
```

11.5 Il sub-template quadro

Per costruire il quadro riassuntivo utilizziamo un ambiente tabular centrato nella pagina. Poiché le cifre saranno elevate, per aumentarne la leggibilità faremo uso della macro `\num` del pacchetto siunitx.

Il seguente codice è l'intero contenuto del sub-template. Occorre un ciclo sulla variabile `sums` per produrre tutte le righe della rilevazione complessiva dei singoli reparti.

```
\begin{center}
\begin{tabular}{lcr}
\hline
\multicolumn{3}{c}{%
Inventario al 31/12/<<| year |>>}\\
\hline
\\
\multicolumn{3}{l}{%
Tabella riassuntiva per reparto}\\[2ex]
\hline
Nome reparto &Esistenza &Valore Esistenza
\euro\\\hline
<<% for dep_name, dep_data
                  in sums.items() -%>>
Reparto <<| "%-18s"|format(dep_name)
|>> & \num{<<|
    "%10d"|format(dep_data.qtytot)
|>>} & \num{<<|
   "%0.2f"|format(dep_data.tottot)
|>>}\\
<<% endfor -%>>
\hline
Totale inventario & \num{<<|
   "%10d"|format(sums.values()
         |sum(attribute='qtytot'))
|>>} & \num{<<|
   "%0.2f"|format(sums.values()
         |sum(attribute='tottot'))
|>>}\\\hline
\end{tabular}
\end{center}
```

Osservando il codice dei sub-template si nota che alcune operazioni di calcolo sono effettuate dal programma mentre altre dal template. Il compito di trovare il migliore equilibrio tra un template completamente privo di logica e uno che effettua tutte le elaborazioni sull'insieme minimo dei dati è lasciato al lettore come ulteriore esercizio. L'equilibrio intermedio dei template qui proposti è stato guidato da semplici considerazioni di efficienza nei calcoli dimostrando forse che le due situazioni estreme peccano la prima di complessità del codice e la seconda di efficienza del template.

Proprio le ultime righe del sub-template per generare il quadro riassuntivo provano come le somme parziali per fornitore e reparto sono lasciate allo script mentre i totali d'inventario sono lasciati al template attraverso la funzione filtro `sum()` che opera sulla lista dei valori del dizionario ottenuta con la funzione `values()`.

11.6 Il database

Ipotizziamo che il database esponga una vista che semplifichi la query degli articoli di magazzino. Una *vista* appare come una normale tabella di sola lettura del database mentre invece è ricavata al momento da una query prestabilita salvata permanentemente nella base di dati. A noi permetterà di non entrare nei dettagli della struttura dei dati, probabilmente articolata in ordini, fornitori, reparti, stagioni, articoli, rilevazioni d'inventario, eccetera. Del resto, la vista serve proprio a questo: costituire l'interfaccia ad alto livello ai dati nascondendone i dettagli. Se cambiasse la struttura interna la vista continuerebbe a essere la stessa una volta modificata adeguatamente la query sottostante.

11.7 Lo script

L'iterazione sulle tuple restituite dalla query crea il dizionario che memorizza l'intero insieme dei dati d'inventario chiamato `inv_data`. La sua struttura è rappresentata schematicamente nella figura 8 e corrisponde esattamente a quella attesa dal template. Completata la fase di reperimento delle informazioni dal database la funzione `calc_sums()` visita la struttura dati d'inventario per generare un secondo dizionario contenente i subtotali per fornitore e reparto.

Il codice del template principale è contenuto nel file `inventory.tex` nella sottocartella `templates`. Daremo quest'informazione all'oggetto `Environment` (tramite il *loader* di `Jinja`) in aggiunta alle stringhe utilizzate come delimitatori per le istruzioni e i campi espressione. Chiamando poi il template principale tramite il metodo `get_template()`, implicitamente verranno an-

```
{'dep-key': {'sup-key':[{'descr':<val>, 'qty':<val>, 'cost':<val>},
                        {'descr':<val>, 'qty':<val>, 'cost':<val>},
                        {'descr':<val>, 'qty':<val>, 'cost':<val>},
                        {'descr':<val>, 'qty':<val>, 'cost':<val>},
                        {'descr':<val>, 'qty':<val>, 'cost':<val>},
                        {'descr':<val>, 'qty':<val>, 'cost':<val>},
                        ...
                       ]
             ...
            }
  ...
}
```

FIGURA 8: Schema della struttura a più livelli in cui sono memorizzati tutti gli articoli d'inventario. I simboli sono gli stessi dei tipi in Python ovvero le parentesi graffe definiscono un dizionario e le parentesi quadre una lista. I tre puntini indicano la ripetibilità dell'elemento dello stesso livello. Nel template questo dizionario è facilmente iterato su tre livelli di annidamento per generare il codice LATEX di ambienti tabular multi-pagina dell'elenco degli articoli in inventario. Una struttura analoga è utilizzata nel codice applicativo per memorizzare i valori dei subtotali per fornitore e reparto. L'utente può disegnare schemi simili per facilitarsi la scrittura del codice.

che caricati i sub-template posizionati nella stessa directory.

Il codice si conclude chiamando il metodo `render()` dell'oggetto template e salvando il risultato testuale su disco pronto per essere compilato dal motore di composizione:

```
#! /usr/bin/python3.4
# -*- coding: utf-8 -*-

from jinja2 import Environment
from jinja2 import FileSystemLoader
import sqlite3

year = 2014

# database section
conn = sqlite3.connect("../db/archive.sqlite")

# a dictionary for suppliers info
sups = dict() # key code => long name
for row in conn.execute("SELECT * FROM suppliers;"):
    sups[row[0]] = row[1]

# a dictionary with all warehouse items
inv_data = dict()

# the 'inv' table is a view
# data are ordered by code
query = """
SELECT code, descr, qty, cost, sup, dep
FROM inv
ORDER BY code;
"""
for item in conn.execute(query):
    dep = item[5]
    if not dep in inv_data:
        inv_data[dep] = dict()
    sup = item[4]

    if not sup in inv_data[dep]:
        inv_data[dep][sup] = []
    inv_data[dep][sup].append(
        {"descr":item[1],
         "qty":item[2],
         "cost":item[3]})

conn.close()

# utility template functions

# return supplier long name by code
def get_sup_name(sup):
    return sups[sup]

# a function to calculate the sums
def calc_sums(data):
    res = dict()
    for dep_name, sup_dict in data.items():
        if not dep_name in res:
            res[dep_name] = {'suptot':{}}
        qtytot = 0
        tottot = 0.0

        for sup_code, item_list
                    in sup_dict.items():
            if not sup_code in res[dep_name]:
                res[dep_name][sup_code] = {}
            qty = 0.0
            tot = 0.0
            for a in item_list:
                qty += a['qty']
                tot += a['qty']*a['cost']
            res[dep_name]
               ['suptot']
               [sup_code] = {"qty":qty,
                             "tot":tot}
            qtytot += qty
            tottot += tot

        res[dep_name]['qtytot'] = qtytot
        res[dep_name]['tottot'] = tottot
```

```
    return res

# template section
env = Environment(
    loader=FileSystemLoader('./templates'),
    variable_start_string="<<|",
    variable_end_string="|>>",
    block_start_string="<<%",
    block_end_string="%>>",
)

tmpl = env.get_template("inventory.tex")
res = tmpl.render({
    "year":year,
    "data":inv_data,
    "sums":calc_sums(inv_data)
})

# saving the content in a TeX file
f = open("inv-{0:d}.tex".format(year), "w")
f.write(res)
f.close()
```

12 Conclusioni

I problemi di elaborazione dati per la produzione di documentazione sono molto frequenti in ogni ambito e non è difficile acquisire le conoscenze per risolverli sviluppando script o programmi che producano in uscita i sorgenti per la composizione TEX. Il vantaggio principale è che, tenendo separata la parte di elaborazione dati dalla parte della composizione tipografica, possiamo far evolvere e/o correggere singolarmente le due parti senza per questo dover stravolgere l'intero processo. L'elaborazione dei dati sarà influenzata dalla, e influenzerà la, scelta del linguaggio di programmazione in base alle strutture dati fornite e alle librerie disponibili, specialmente per il *templating*.

Nell'articolo sono stati presentati in dettaglio alcuni esempi di programmi scritti in bash, C, Lua e Python, privilegiando gli ultimi due linguaggi in ragione della loro diffusione e della loro semplicità rispetto al C. Si spera che la lettura di questo articolo aiuti l'utente a valutare se e come modificare l'approccio ai propri problemi di produzione della documentazione ogni volta che i dati abbiano un grado di complessità significativo.

L'ostacolo forse più grande per l'utente TEX all'approccio della generazione automatica di sorgenti è acquisire i concetti di base del linguaggio di programmazione scelto. Esiste però almeno un metodo quasi alternativo: utilizzare LuaTEX per elaborare i dati direttamente nel motore tipografico per poi comporre il documento con le tecnologie di interazione Lua-TEX. Nemmeno così l'utente è esentato dall'imparare un linguaggio di programmazione — in questo caso obbligatoriamente Lua — per scrivere le funzioni di accesso ai database o tradurre i dati dai fogli elettronici, tuttavia nuove librerie di reporting in Lua per LuaTEX potrebbero facilitare molto il compito di sviluppo.

Impostare l'intero processo in LuaTEX è un metodo ancora tutto da valutare ma potenzialmente interessante perché la connessione con TEX può essere molto più stretta di quanto non accada con la produzione separata dei sorgenti. Si spera che in futuro anche questo metodo si aggiunga pienamente alle soluzioni già disponibili e qui sinteticamente esposte.

13 Ringraziamenti

Ringraziamo l'amico Francesco Endrici, membro attivo della comunità del GUIT, per averci fornito un problema reale molto articolato di produzione di un documento a partire da un insieme di dati, perfetto per mostrare l'uso delle tabelle di Lua per descrivere informazioni strutturate.

Ringraziamo infine la Redazione di ArsTEXnica per tutto il prezioso lavoro fatto su questo articolo, dalla valutazione e revisione della bozza fino alla compilazione finale.

Riferimenti bibliografici

ABS guide (2015). «Advanced bash-scripting guide». Web site http://www.tldp.org/LDP/abs/html/index.html.

Battistin, R. (2014). «LATEX per la stesura dei rapporti di prova». *ArsTEXnica*, (18), pp. 55–63. URL http://www.guitex.org/home/numero-18.

De Marco, A. e Cresci, P. E. (2011). «LATEX nella pubblica amministrazione. La web application FACILE per la produzione automatica di comunicazioni interne del Comune di Napoli». *ArsTEXnica*, (12), pp. 39–56. URL http://www.guitex.org/home/it/numero-12.

Downey, A. B. (2012). *Think Python: How To Think Like a Computer Scientist*. O'Reilly Media. URL http://faculty.stedwards.edu/mikek/python/thinkpython.pdf.

Fiandrino, C. (2014). «MenùTEX: una piattaforma per realizzare menù». *ArsTEXnica*, (18), pp. 145–151. URL http://www.guitex.org/home/numero-18.

GNU Bash (2015). «Bourne-again shell manual». http://www.gnu.org/software/bash/manual/.

Ierusalimschy, R. (2013). *Programming in Lua, Third Edition*. Lua.Org.

Jinja (2015). «Jinja is a full featured template engine for python». Official web site http://jinja.pocoo.org/.

Kernighan, B. W. e Ritchie, D. M. (2004). *Il linguaggio C*. Pearson.

Klemens, B. (2014). *21st century C*. O'Reilly and Associates.

Mustache (2015). «{{mustache}} logic-less templates». Official web site https://mustache.github.io/.

Olivine-Labs (2015). «lustache - logic-less {{mustache}} templates with lua». Official web site http://olivinelabs.com/lustache/.

Pilgrim, M. (2009). *Dive Into Python 3*. Apress. URL http://getpython3.com/diveintopython3/.

SQLite3 (2015). «The sqlite consortium». Web site https://www.sqlite.org/.

Tantau, T. (2015). «pgf manual version 3.0.1». Available on www.ctan.org.

▷ Roberto Giacomelli
giaconet dot mailbox at gmail dot com

▷ Gianluca Pignalberi
g dot pignalberi at alice dot it