Numero 24
Ottobre
2017

ArsTEXnica

Rivista italiana di TEX e LATEX

GUIT

http://www.guitex.org/arstexnica/

GUIT – Gruppo Utilizzatori Italiani di TEX

ArsTEXnica è la pubblicazione ufficiale del GUIT

ArsTEXnica è la prima rivista italiana dedicata a TEX, a LATEX ed alla tipografia digitale. Lo scopo che la rivista si prefigge è quello di diventare uno dei principali canali italiani di diffusione di informazioni e conoscenze sul programma ideato quasi trent'anni fa da Donald Knuth.

Le uscite avranno, almeno inizialmente, cadenza semestrale e verranno pubblicate nei mesi di Aprile e Ottobre. In particolare, la seconda uscita dell'anno conterrà gli Atti del Convegno Annuale del GUIT, che si tiene in quel periodo.

La rivista è aperta al contributo di tutti coloro che vogliano partecipare con un proprio articolo. Questo dovrà essere inviato alla redazione di ArsTEXnica, per essere sottoposto alla valutazione di recensori. È necessario che gli autori utilizzino la classe di documento ufficiale della rivista; l'autore troverà raccomandazioni e istruzioni più dettagliate all'interno del file di esempio (`.tex`). Tutto il materiale è reperibile all'indirizzo web della rivista.

Gli articoli potranno trattare di qualsiasi argomento inerente al mondo di TEX e LATEX e non dovranno necessariamente essere indirizzati ad un pubblico esperto. In particolare tutorials, rassegne e analisi comparate di pacchetti di uso comune, studi di applicazioni reali, saranno bene accetti, così come articoli riguardanti l'interazione con altre tecnologie correlate.

Di volta in volta verrà fissato, e reso pubblico sulla pagina web, un termine di scadenza per la presentazione degli articoli da pubblicare nel numero in preparazione della rivista. Tuttavia gli articoli potranno essere inviati in qualsiasi momento e troveranno collocazione, eventualmente, nei numeri seguenti.

Chiunque, poi, volesse collaborare con la rivista a qualsiasi titolo (recensore, revisore di bozze, grafico, etc.) può contattare la redazione all'indirizzo:

arstexnica@guitex.org.

Nota sul Copyright

Associarsi a GUIT

Fornire il tuo contributo a quest'iniziativa come membro, e non solo come semplice utente, è un presupposto fondamentale per aiutare la diffusione di TEX e LATEX anche nel nostro paese. L'adesione al Gruppo prevede una quota di iscrizione annuale diversificata: 30,00 € soci ordinari, 20,00 (12,00) € studenti (junior), 75,00 € Enti e Istituzioni.

Indirizzi

Gruppo Utilizzatori Italiani di TEX
c/o Università degli Studi di Napoli Federico II
Dipartimento di Ingegneria Industriale
Via Claudio 21
80125 Napoli – Italia
`http://www.guitex.org`
guit@guitex.org

Redazione ArsTEXnica:
`http://www.guitex.org/arstexnica/`
arstexnica@guitex.org

Codice ISSN 1828-2369

Stampata in Italia
Napoli: 15 Ottobre 2017

Venezia-Mestre, 21 ottobre 2017

09:30 – 18:30 Via Torino 155 - Auditorium Danilo Mainardi

XIV Convegno annuale su TEX LATEX e tipografia digitale organizzato dal Gruppo Utilizzatori Italiani di TEX

PROGRAMMA

9:30
Benvenuto. Inizio dei lavori

9:45 - 10:15
La composizione tipografica di alcune lingue orientali (**Claudio Beccari**)
10:15 - 10:45
Tutorial – Yudit: un editor Unicode che "parla" (anche) LaTeX (**Gianluca Pignalberi**)

BREAK

11:15 - 11:45
Lo scriba A del Codex Sinaiticus e il font Simonides (**Claudio Vincoletto, Massimiliano Dominici**)
11:45 - 12:15
Requirements for a Music Engraving Program: a Composer's Point of View (**Jean-Michel Hufflen**)
12:15 - 12:45
Condizionali in LaTeX (**Enrico Gregorio**)

PRANZO

15:00 - 15:30
Introduzione alla composizione di testi scacchistici (**Maurizio Molinaro**)
15:30 - 16:00
Let's Connect LuaTeX to the World (**Roberto Giacomelli**)

BREAK

16:30 - 17:00
Un DTD SGML per la generazione di codice NGSpice e CircuiTikZ (**Renato Battistin**)
17:00 - 17:30
A Template Engine with TeX and Friends (**Paulo Roberto Massa Cereda**)

17:30 - 18:30
Riunione annuale & arrivederci

INFO & REGISTRAZIONE: https://www.guitex.org/home/it/meeting

Università Ca'Foscari Venezia

ArsTEXnica

Rivista italiana di TEX e LATEX

Numero 24, Ottobre 2017

Gruppo Utilizzatori Italiani di TEX

Editoriale

Claudio Beccari

Il Meeting del GUIT 2017 ha luogo a Venezia presso l'Università Ca' Foscari nella sede staccata di Mestre. Ringrazio le Autorità Accademiche che ci hanno permesso di svolgere qui il nostro incontro annuale.

In questo numero di ArsTEXnica ci troviamo di fronte ad alcuni importanti contributi e sono convinto che i lettori li troveranno molto interessanti. Li presento seguendo l'ordine del programma.

Claudio Beccari con il suo articolo *La composizione tipografica di alcune lingue orientali* si cimenta con un argomento interessante, in particolare per gli studiosi di lingue orientali di Ca' Foscari: la composizione in queste lingue mediante gli strumenti del sistema TEX. Naturalmente l'autore non può descriverle tutte dal Medio all'Estremo Oriente; ma le tre lingue di cui tratta sono abbastanza caratteristiche delle varietà esistenti: l'arabo, il giapponese e il coreano. Sono lingue con particolarità molto diverse: l'arabo molto legato e con andamento da destra a sinistra; il giapponese che usa quattro sistemi di simboli diversi alcuni dei quali sono formati dagli ideogrammi cinesi; il coreano che ha un alfabeto con un numero limitato di simboli che però vanno raggruppati in modo creativo per formare dei segni sillabici. Naturalmente il problema per comporre queste lingue è costituito dalla tastiera usata dal compositore.

Gianluca Pignalberi presenta un tutorial destinato a coloro che si avvicinano al mondo del sistema TEX e che devono trovare l'editor testuale più adatto al loro modo di gestire la composizione dei loro testi. L'articolo *Yudit: un editor* UNICODE *che "parla" (anche) LATEX* ha un titolo che si spiega da solo. L'autore mostra i principi del programma di editing Yudit e ne mostra diverse caratteristiche; ne mette in luce la sua capacità di gestire in modo molto valido, ma conservando una semplicità di fondo, i file sorgente scritti in linguaggio LATEX.

L'articolo *Lo scriba A del Codex Sinaiticus e il font Simonides* è scritto a due mani da Claudio Vincoletto e Massimiliano Dominici, già noti ai frequentatori dei passati incontri annuali per avere trattato temi relativi ai font e alla loro realizzazione con i programmi disponibili con il sistema TEX. In questo articolo esaminano sia il font Simonides, sia i mezzi per dotare tale font di una scelta pseudocasuale di glifi leggermente variati rispetto al corrispondente prototipo, in modo da poter ricomporre il Codex Sinaiticus come se fosse scritto a mano, più o meno come se lo scriba, individuato con la lettera A dagli studiosi, lo stesse riscrivendo.

L'articolo *Requirements for a music engraving program: a composer's point of view* di Jean-Michel Hufflen mette a confronto i programmi per comporre la musica dal punto di vista tipografico con i programmi che consentono al compositore di scrivere la sua musica. La tematica è nuova per ArsTEXnica, anche se il primo aspetto, quello tipografico, è già stato trattato in tempi recenti. L'autore svolge le sue argomentazioni e finisce per concludere che i campi di applicazione dei programmi "musicali" sono complementari gli uni con gli altri e che il compositore, in definitiva, ha bisogno di entrambi.

L'articolo *Condizionali in LATEX* di Enrico Gregorio parla di cose inizialmente semplici, ma via via più complesse, quali sono i comandi condizionali. ArsTEXnica ha già trattato in passato queste tematiche, ma l'articolo le riprende in modo più generale ed esteso con le funzionalità del linguaggio L3 (LATEX 3, ormai estremamente diffuso senza che gli utenti normali se ne accorgano); queste permettono di fare cose impensabili con il nucleo primitivo di LATEX. C'è molto da imparare anche da parte di coloro che usano LATEX da molti anni.

Per la prima volta su ArsTEXnica Maurizio Molinaro affronta in modo molto chiaro l'*Introduzione alla composizione di testi scacchistici*. L'argomento sembra confinato all'ambito del gioco degli scacchi; in realtà, dopo le parti introduttive, l'articolo si concentra sulle descrizioni dell'andamento dei giochi e, in sostanza, discute come descrivere una strategia che sotto molti aspetti è applicabile anche ad altri ambiti, non esclusi i "giochi" che si svolgono in ambito economico e finanziario. Questa almeno è la mia interpretazione, anche se una lettura superficiale, per altro molto interessante, sembra destinata solo al pubblico degli scacchisti.

Roberto Giacomelli ci presenta l'articolo *Let's connect LATEX to the world*; ci mostra come LuaTEX possa interfacciarsi con altri software attraverso programmi e librerie che, adeguatamente usati, permettono di collegare diversi applicativi attraverso particolari messaggi che si scambiano per trasmettersi i dati da elaborare. L'argomento è molto utile sia per il lavoro cooperativo quando gli applicativi usano linguaggi diversi, sia perché, oggi più che mai, i dati che bisogna elaborare richiedono funzionalità specifiche purché i programmi che li elaborano possano farlo in modo autonomo, almeno sui dati di loro competenza, ma possano anche trasmetterli nei formati adeguati agli altri programmi. Questo genere di ricerche, di studi e di soluzioni efficaci diventa sempre più importante nel mondo del lavoro.

L'articolo di Renato Battistin ha un titolo forse un po' ermetico *Un DTD SGML per la generazione di codice NGSpice e CircuiTikZ*; l'autore descrive un nuovo metodo e lo applica per generare il codice per disegnare con Ti*k*Z un circuito e contemporaneamente per scrivere il codice per Spice perché ne faccia l'analisi al fine di comporne un rapporto con LaTeX. Sembra solo un esempio particolare, ma lo si può concettualmente usare in diversi altri ambiti. La metodologia che l'autore ci presenta è diversa da quella presentata in articoli simili pubblicati in passato, ma la natura delle esigenze stimola la creatività per trovare altre soluzioni valide per problemi specifici.

Paulo Cereda, l'autore del programma **arara**, nell'articolo *A template engine with TEX and friends* ci presenta un altro programma che facilita lo sviluppo di progetti di composizione piuttosto complessi. Si tratta di un linguaggio di scripting che organizza e sviluppa le varie fasi di un progetto complesso procedendo in modo selettivo per diverse vie. Come **arara**, così questo linguaggio è messo a disposizione degli utenti del sistema TeX. Il programma **arara** ha avuto un grande successo; questo nuovo programma, con scopi più ambiziosi, avrà sicuramente un impatto importante nella comunità degli utenti del sistema TeX.

Mi ripeterò, ma lo devo fare perché le persone che voglio ringraziare se lo meritano ampiamente. Sono i membri del comitato di redazione, il comitato scientifico che ha esaminato e valutato gli articoli qui pubblicati, i revisori editoriali e i correttori che sono intervenuti sugli articoli in italiano e in inglese per migliorarne il testo. Grazie agli autori e grazie allo staff di ArsTEXnica, la rivista si mantiene all'alto livello che è sotto gli occhi dei lettori.

▷ Claudio Beccari
Professore emerito
Politecnico di Torino
`claudio dot beccari at gmail dot com`

La composizione tipografica di alcune lingue orientali

Claudio Beccari

Sommario

Scrivere nelle lingue orientali, usate dal Medio Oriente fino alla Cina e al Giappone, è una cosa che si può fare anche con i programmi del sistema TEX. La difficoltà maggiore è la conoscenza di quelle lingue e dei loro sistemi di scrittura. Per brevità qui si parlerà dell'arabo, del giapponese e del coreano, con qualche accenno al cinese.

Abstract

Typesetting documents with the languages written and spoken in the Near East to the Far East is possible with the TEX system programs. The major difficulty, besides knowing those languages, consists in their writing systems. For conciseness here we shall deal with Arabic, Japanese and Korean with modest hints to Chinese.

1 Introduzione

Con i programmi del sistema TEX è relativamente semplice comporre documenti con alfabeti diversi dai font latini, con direzioni di scrittura diverse e usando font molto legati. Con le lingue che fanno uso dei caratteri arabi la direzione di scrittura è da destra a sinistra e i font sono strettamente legati.

Col cinese e il giapponese la direzione di scrittura è generalmente da sinistra a destra, ma per alcuni testi formali non è esclusa la direzione dall'alto al basso. I font sono costituiti da ideogrammi. Altri sistemi di caratteri sono "letterali", con segni diversi da quelli latini, ma disposti in sillabe di forma pressoché quadrata come il coreano.

Si rende necessario usare font di tipo OpenType adatti alle lingue in questione; con l'arabo è necessario scrivere una specie di codice, un po' come si scrive un documento con LATEX o come si programma un disegno con Ti*k*Z; ma per chi conosce l'arabo e i suoi problemi di vocalizzazione o, per il Corano, i problemi di ritmica per marcare lo scritto con i segni prosodici necessari alla lettura rituale, non dovrebbe essere particolarmente complicato. In confronto all'arabo, l'ebraico è molto più semplice, anche se questa lingua ha i soliti problemi di vocalizzazione delle lingue semitiche[1], i cui sistemi di caratteri rappresentano per lo più delle consonanti.

Un altro problema per scrivere in queste lingue orientali è la tastiera. Va da sé che le tastiere dei calcolatori personali, che dispongono di un centinaio di tasti, benché siano usate correntemente anche in quei paesi hanno un numero di tasti di solito abbondantemente insufficiente per le lingue che scrivono con ideogrammi. Sembra che il totale dei caratteri cinesi, usati col nome di *kanji* anche in giapponese, ammontino a circa 35 000. Abituati come siamo con l'alfabeto latino di 26 caratteri (più altri 26 per le maiuscole), i numeri degli ideogrammi sono impressionanti. La presenza di accenti o altri segni diacritici non rende il nostro alfabeto sostanzialmente più complesso.

I giapponesi e i cinesi devono usare la loro tastiera anche per scrivere in caratteri latini; quindi devono disporre di tasti per commutare dalla tastiera latina (generalmente contente solo i 95 caratteri ASCII senza accenti) alla tastiera kanji. Per i giapponesi la cosa è ancora più complicata, perché devono usare anche i sillabari *katakana* e *hiragana*, che contengono ciascuno una sessantina di segni.

2 Le tastiere

2.1 La tastiera araba

I caratteri arabi sono relativamente pochi: nella documentazione del pacchetto arabluatex la tabella 1 riporta 28 righe, in una delle quali sono indicate le tre vocali che di tanto in tanto è necessario introdurre; queste non rappresentano una forma di vocalizzazione, ma sono "vocali di sostegno" che si usano in certe parole; esse corrispondono pressappoco ai nostri suoni 'u', 'a', 'i'. Ma il punto non è tanto questo, quanto il fatto che questi segni e quelli di tutte le consonanti hanno generalmente quattro forme per la lettera isolata, per l'iniziale, per la mediale e per la finale; ecco quindi che i segni diventano circa 120. Se si aggiungono i segni per la vocalizzazione, la moltitudine di segni aumenta molto. Magra consolazione: non ci sono maiuscole e minuscole.

Per questo motivo si è sviluppata una forma di scrittura codificata mediante lettere latine minuscole e maiuscole intercalate con segni analfabetici, in modo da caratterizzare ogni aspetto della lingua araba scritta, ma ci vuole un preprocessore o un programma comunque invocato dal programma di

1. Il maltese è l'unica lingua semitica che si scrive con caratteri latini e le vocali sia pure con diversi diacritici, anche se le numerose dominazioni che l'isola ha subito hanno introdotto molti forestierismi; quelli italiani sono facilmente riconoscibili sia per iscritto sia quando si parla; oggi il maltese è lingua ufficiale della Repubblica di Malta a pari livello dell'inglese.

composizione che trasformi l'input codificato, come si è detto, con caratteri ASCII in una sequenza di caratteri UNICODE, adatti per combinarsi nelle giuste forme e per legarsi fra di loro come richiede la scrittura araba. Questa transcodifica dell'arabo viene consigliata anche a coloro che parlano e scrivono in quella lingua, anche se devono usare la modalità ASCII della tastiera.

Di fatto, quindi, per scrivere in arabo non occorre una tastiera araba o "bilingue", cosa che per noi europei torna particolarmente comoda.

2.2 La tastiera giapponese

Come ho detto, la tastiera che usano i giapponesi permette loro, mediante opportuni tasti, di cambiare la modalità di scrittura, dal *romaji* (caratteri latini), al katakana (sillabario giapponese usato specialmente per traslitterare i nomi stranieri, figura 1), al hiragana (usato specialmente per scrivere le pre- e post-posizioni della lingua giapponese, figura 2), e il *kanji* (per usare gli ideogrammi cinesi, troppo numerosi per poterli raffigurare in questo articolo). In teoria, quindi, con la tastiera giapponese si può scrivere anche in cinese.

Ma il problema è che gli ideogrammi cinesi sono delle singole parole di solito monosillabiche, pronunciate con i debiti toni. Gli stessi ideogrammi convogliano la stessa "idea" in giapponese, ma vengono pronunciati con parole generalmente polisillabiche, magari raggruppando in una sola parola giapponese l'idea complessiva del gruppo di ideogrammi. Il giapponese non ha toni, ma ha sillabe lunghe, medie e brevi. Quelle lunghe, scrivendo in romaji si rendono o raddoppiando la vocale della sillaba, o sovrapponendo un segno di lunga: *arigatoo*, oppure *arigatō*; talvolta in hiragana scrivono una sillaba 'u' alla fine della parola ありがとう, o appendono un segno di lunga dopo la sillaba 'to', ありがとー. Quelle brevi possono essere marcate con un segno di breve (ŭ) o vengono scritte in carattere di corpo minore.

Il sillabario hiragana è originale giapponese antico e risale al XI secolo; fu usato dalla dama di compagnia Murasāki Shikibu per scrivere *La leggenda di Genji* (Orsi, 2012), apparentemente il più antico romanzo della storia mondiale. Tutto il testo è scritto usando il sillabario hiragana; il testo è ancora usato oggi per rappresentare nel teatro tradizionale Kabuki l'interpretazione teatrale divisa in una sequenza interminabile di atti, che dura diversi mesi e che vengono rappresentati ciclicamente da alcuni secoli.

Il sillabario katakana, che comprende essenzialmente linee diritte, è stato "inventato" per scrivere telegrammi e permettere alle telescriventi civili e militari di trasmettere in giapponese. Gli ideogrammi cinesi sono entrati nell'uso diversi secoli fa quando per il Giappone l'Impero cinese rappresentava un modello di cultura e di stile.

Ovviamente il romaji è il più recente sistema di scrittura diffusosi anche con l'avvento dei calcolatori personali, in quanto prima i centri di calcolo erano sostanzialmente dedicati al trattamento di dati numerici; la gente scriveva con la penna, o meglio, esercitava la sua calligrafia con pennelli ed inchiostri speciali. Ancora oggi la calligrafia con il pennello è un'arte apprezzatissima.

Sul Mac si possono installare diversi driver di tastiera e sceglierli cliccando sull'icona apposita nella barra superiore. Come farlo dipende dal particolare sistema operativo, ma sostanzialmente bisogna cliccare l'icona "System Preferences"; nel dialogo che appare si clicca sull'icona della tastiera e si apre un altro dialogo; in questo si sceglie la voce di menu "Input Sources" dove bisogna marcare l'opzione "Show keyboard and emoji viewers in menu bar", in modo che quando si vuole cambiare driver lo si sceglie dalla lista dei driver installati, ma si ha anche la possibilità di mostrare la tastiera virtuale; in questo dialogo si sceglie quali driver aggiungere e quali togliere. Quando si mostra la tastiera virtuale vi si può cliccare col mouse; in ogni caso è utilissima per sapere quali tasti hardware (con i simboli serigrafati sulla faccia superiore di ogni tasto della tastiera reale) corrispondano a quali tasti della tastiera virtuale.

Con il driver per il giapponese scegliere "Hiragana" fra i metodi di input del driver; nella tastiera virtuale compaiono i tasti romaji (latini), battendo la parola giapponese da scrivere appaiono via via le sillabe hiragana in una finestrina a parte, ma quando la parola è completamente scritta compaiono anche le versioni hiragana e kanji nella finestrina fra le varie parole si sceglie quella giusta cliccandovi sopra e la parola si trasferisce al documento che si sta scrivendo, mentre si chiude la finestrina, pronta a riaprirsi per la parola successiva.

Conviene definire nel preambolo un paio di comandi per scegliere un font OpenType che contenga i caratteri giapponesi e il cui nome sia `\japanesefont`; si definisca anche un comando per scrivere con quel font, e magari anche un ambiente; per esempio:

```
\newjfontfamily{\japanesefont}{Hiragino Mincho Pro}
\newcommand*\textjapanese[1]{{\japanesefont #1}}
\newenvironment{japanese}[1][]{#1\japanesefont}{}
```

In questo modo il font scelto per scrivere in giapponese è l'Hiragino Mincho Pro[2]; il comando `\textjapanese` compone la stringa che gli si dà come argomento usando il font scelto. L'ambiente `japanese` accetta un argomento facoltativo che viene usato come prima cosa nell'apertura dell'ambiente; può servire per specificare comandi di corpo o qualunque altra cosa sia utile in un ambiente.

Per comporre quindi la parola 'arigatoo' in giapponese, basta inserire nel file sorgente il codice:

2. Questo font è uno dei vari disponibili sul Mac; altri sistemi operativi dispongono di font con altri nomi.

FIGURA 1: I 107 segni del sillabario giapponese katakana (カタカナ), dei quali 46 sono principali, e gli altri 61 sono repliche modificate mediante delle specie di diacritici.

FIGURA 2: I 97 segni del sillabario giapponese hiragana (ひらがな) dei quali 41 sono principali e gli altri 56 sono repliche modificate mediante delle specie di diacritici.

```
\textjapanese{ありがとう}
```

e il gioco è fatto. Ovviamente per tratti più corposi di testo conviene usare l'ambiente.

2.3 Tastiera cinese

Come ho detto la tastiera giapponese consente di scrivere anche in cinese. Sono sicuro che in Cina usano tastiere adeguatamente configurate.

Sono altrettanto sicuro che per il cinese sia disponibile un metodo per immettere gli ideogrammi, tenendo conto che in questa lingua non esistono sillabari come in giapponese.

Non avendo la benché minima nozione di cinese, non mi sono cimentato nell' usare qualche driver del mio Mac, quindi non sono in grado di dare ulteriori informazioni. Più avanti indicherò un esempio di input in cinese; ma il testo l'ho semplicemente copiato da internet, perché la mia conoscenza di quella lingua è nulla.

3 La tastiera coreana

La lingua coreana si scrive con un alfabeto vero e proprio che contiene 14 consonanti e 10 vocali; in realtà queste cifre sono origine di controversie; sta di fatto che i tasti alfabetici della tastiera coreana sono 26; evidentemente la classificazione minimale di 24 lettere non è quella seguita da chi ha predisposto quella tastiera. I segni che rappresentano i vari suoni e le loro varianti si chiamano *jamo* e hanno forme completamente diverse da quelle latine, quindi, senza conoscere quell'alfabeto, la lingua coreana sembra scritta con strani ideogrammi. In realtà le singole lettere sono aggiustate dentro un ideale contorno quadrato che contiene un'intera

FIGURA 3: La tastiera coreana sul Mac

sillaba; la sillaba può essere composta con uno, due o tre suoni elementari, quindi può contenere fino a tre lettere; se la sillaba è fatta di un solo suono, quindi un solo segno, gli viene aggiunto un segno speciale che non ha suono, di forma rotondeggiante con il significato di "aprire la bocca per parlare". Questo modo di scrivere di chiama *hangul*. Ecco, questa parola si scrive con due sillabe 'han' e 'gul'; essa dunque compare nel testo come due gruppi di singole lettere in questo modo 한글 È difficile per un occhio inesperto distinguere i tre segni in ciascuna delle due sillabe, ma guardando bene i segni ingranditi esse si riescono a distinguere (figura 4).

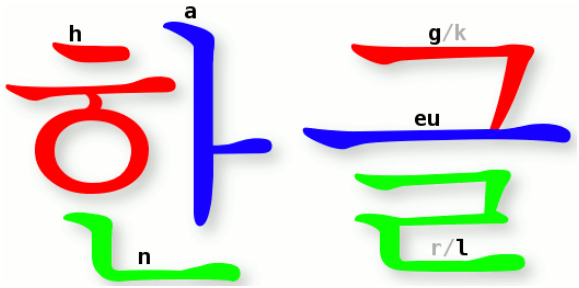


FIGURA 4: La parola coreana 한글 (hangŭl o hangeul) ingrandita e marcata con le lettere latine che corrispondono ad ogni lettera coreana (jamo).

Per comporre in coreano conviene di nuovo definire un comando `\textkorean` e un ambiente korean in modo del tutto simile a quanto detto per il giapponese. Se si specifica con polyglossia la lingua `korean`, questi comandi sono già definiti e, come per la altre lingue, il modulo di polyglossia per il coreano prevede le solite parole fisse, la data e qualche altro dettaglio di impostazione tipografica.

Per il Mac suggerisco di installare il driver 2-Set Korean; quando si imposta questo driver e si sceglie di mostrare la tastiera (figura 3), il Mac mostra il layout dei tasti con le lettere coreane disegnate al loro posto; digitando le lettere in sequenza, come si farebbe per scrivere con l'alfabeto latino, il driver forma i gruppi di lettere di ciascuna sillaba e nel testo sorgente esse appaiono già correttamente raggruppate negli schemi quadrati descritti sopra.

Per i font OpenType di solito quelli che contengono gli ideogrammi cinesi e giapponesi contengono anche le lettere coreane; volendo distinguere lo stile dei font coreani, si potrebbe definire nel preambolo il comando:

```
\newjfontfamily{\koreanfont}{UnBatang}
```

Un esempio di più vasto respiro potrebbe essere il seguente:

모든 인간은 태어날 때부터 자유로우며 그 존엄과 권리에 있어 동등하다. 인간은 천부적으로 이성과 양심을 부여받았으며 서로 형제애의 정신으로행동하여야 한다.

4 L'arabo nel file sorgente

Come già detto il modo preferibile per inserire il testo in arabo è quello di usarne una versione codificata con caratteri latini. Di traslitterazioni latine, che bisognerebbe chiamare *romanizzazione*, almeno secondo gli arabisti anglosassoni, ce ne sono diverse; le più comuni traslitterazioni/romanizzazioni sono le seguenti: **dmg**, **loc**, **arabica**; esse sono descritte in dettaglio poco più avanti. Il pacchetto arabi (JABRI, 2006) indica una traslitterazione; il pacchetto arabluatex (ALESSI, 2017) ne indica delle altre.

Benché apparentemente il pacchetto arabi abbia ricevuto recentemente un aggiornamento, personalmente ho incontrato delle difficoltà a farlo convivere con l'opzione per il greco. Questo, ovviamente non ha nulla a che vedere con la composizione tipografica della sola lingua araba. Infatti, per evitare conflitti ho composto in arabo con questo pacchetto quello che volevo comporre, poi ho ritagliato dalla pagina il testo composto e l'ho inserito come figura nell'altro documento che conteneva anche diffuse inserzioni in greco. Le indicazioni per rivolgersi al curatore del pacchetto arabi riportate nella prima pagina della documentazione sono errate o, comunque, non aggiornate nemmeno dopo quest'ultimo aggiornamento, quindi è impossibile segnalare il problema, perché il curatore non sembra più reperibile.

TABELLA 1: Le traslitterazioni principali in lettere latine dell'arabo come sono riportate nella documentazione del pacchetto arabluatex. Tratto integralmente dalla documentazione di arablualatex, compresi i richiami delle note, per le quali si rimanda direttamente a quella documentazione.

Letter	Transliteration[9]			ArabTEX notation
	dmg	loc	arabica	
إ أ أُ[10]	ʾu ʾa ʾi	’u, ’a, ’i	ʾu, ʾa, ʾi	'u or 'a or 'i
ب	b	b	b	b
ت	t	t	t	t
ث	t̲	th	t̲	_t
ج	ǧ	j	ǧ	^g or j
ح	ḥ	ḥ	ḥ	.h
خ	ḫ	kh	ḫ	_h or x
د	d	d	d	d
ذ	d̲	dh	d̲	_d
ر	r	r	r	r
ز	z	z	z	z
س	s	s	s	s
ش	š	sh	š	^s
ص	ṣ	ṣ	ṣ	.s
ض	ḍ	ḍ	ḍ	.d
ط	ṭ	ṭ	ṭ	.t
ظ	ẓ	ẓ	ẓ	.z
ع	ʿ	ʿ	ʿ	`
غ	ġ	gh	ġ	.g
ف	f	f	f	f
ق	q	q	q	q
ك	k	k	k	k
ل	l	l	l	l
م	m	m	m	m
ن	n	n	n	n
ه	h	h	h	h
و	w	w	w	w
ي	y	y	y	y[11]
ة	ah	ah	a	T

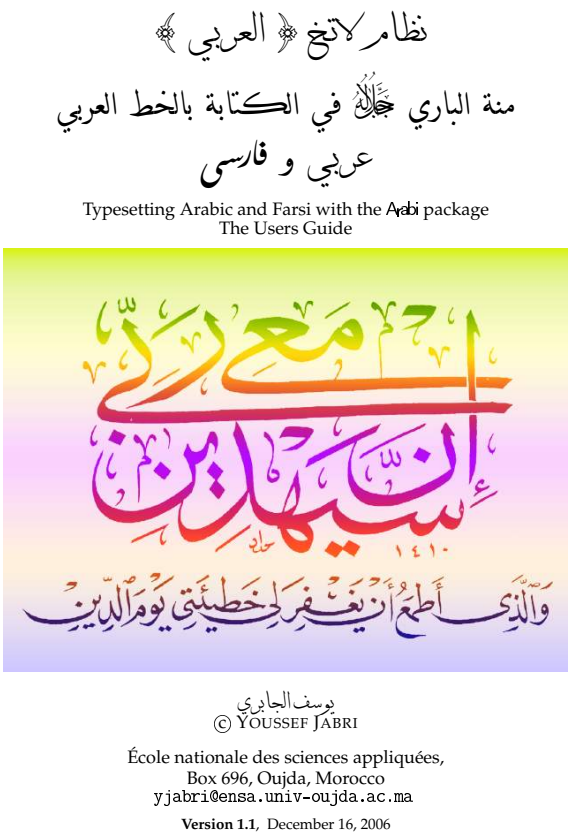


FIGURA 5: Frontespizio della documentazione del pacchetto arabi. Composizione di Youssef Jabri.

Il frontespizio della documentazione di arabi è particolarmente attraente (figura 5).

Devo segnalare anche che il pacchetto bidi (KHALIGHI, 2017), caricato da arabi, per comporre l'arabo in modo bidirezionale, da destra a sinistra, e da sinistra a destra la lingua italiana, dava problemi al contenuto di certi comandi; per esempio tre parole italiane composte con il comando `\text` in un contesto italiano e in ambiente matematico, a causa del pacchetto bidi risultavano scritte da destra a sinistra, o meglio, le singole parole erano composte correttamente da sinistra a destra, ma le parole erano allineate da destra a sinistra. Veramente strano, ma pur avendo segnalato questo bug, non ho ricevuto nessuna risposta.

C'era anche un conflitto con l'ebraico, che si scrive da destra a sinistra come l'arabo, ma i relativi pacchetti caricavano entrambi il pacchetto bidi senza eseguire nessun controllo se il pacchetto fosse già stato caricato. Questo era un piccolo difetto che si risolveva con un poco di astuzia TEXnica; tuttavia è un peccato che i curatori dei rispettivi pacchetti non facciano attenzione a questi dettagli.

Questo modo di procedere, comporre a parte, ritagliare e incollare, non è molto efficiente se si deve comporre, per esempio, una edizione critica che contenga il testo arabo inframmezzato con qualche lingua composta in caratteri latini.

Ho la sensazione che il pacchetto arabluatex, più recente e assistito dal linguaggio Lua integrato con il motore di composizione, sia più efficiente ed efficace. All'occorrenza esiste anche il pacchetto arabxetex per comporre con `xelatex`.

Il pacchetto arabluatex può eseguire anche alcune traslitterazioni. Ma non bisogna confondere quest'operazione con la codifica d'entrata del file sorgente da comporre in arabo. Nella tabella 1 sono riportate sia le traslitterazioni che arabluatex è capace di eseguire, sia la codifica di entrata (nella colonna *Arab TEX notation*); si vede che in qualche modo questa codifica assomiglia alle romanizzazioni delle prime colonne, ma si tratta di cose molto diverse. Le romanizzazioni che arabluatex è finora in grado di fare sono descritte qui di seguito[3].

dmg *Deutsche Morgenländische Gessellschaft* è la romanizzazione adottata dalla Convenzione Internazionale degli Orientalisti a Roma nel 1935; questa è la romanizzazione preimpostata dal pacchetto.

loc *Library of Congress* appartiene a un vasto insieme di standard per la romanizzazione delle scritture non romane adottato dall'American Library Association e dalla Library of Congress.

arabica *Journal of Arabic and Islamic Studies / Revue d'études arabes et islamiques*: è lo standard maggiormente usato dagli Arabisti

Ovviamente il pacchetto in esame consente anche la vocalizzazione parziale o totale del testo arabo. La cosa è troppo complessa da descrivere qui; per questo è necessario rivolgersi alla documentazione.

Naturalmente, per un testo completamente in arabo o per testi più lunghi di poche parole oltre al comando `\arb` si dispone anche dell'ambiente arab, come lo si vede usato nella figura 7.

Merita notare che tutti i pacchetti che consentono di inserire il testo arabo codificato mediante caratteri latini provvedono automaticamente a gestire i numeri, almeno gli interi che si trovano frequentemente nei testi, scrivendoli cominciando dalle unità; siccome la lettura avviene da destra a sinistra, le unità si trovano a destra nella stinga numerica, quindi per noi sembrano scritti nel verso normale; va da sé che se i numeri sono scritti con i numerali arabi, è necessario saperli leggere, visto che questi numerali possono avere forme diverse (ne esistono almeno due varietà in arabo) e comunque sono diversi da quelli che noi chiamiamo "cifre arabe".

Tutti i pacchetti che ho potuto esaminare provvedono all'inserimento della *kashida*, un allungamento verso sinistra delle lettere terminali delle parole, in modo che la giustificazione avviene riempiendo lo spazio interparola mediante la kashida,

3. Il pacchetto è ancora in fase di sviluppo ma, rimanendo retrocompatibile, amplierà questa lista.

```
From \textcite[i. 1 A]{Wright}:--- Arabic, like Hebrew and
Syriac, is written and read from right to left. The letters
of the alphabet (\arb{.hurUf-u 'l-hijA'-i}, \arb{.hurUf-u
'l-tahajjI}, \arb{al-.hurUf-u 'l-hijA'iyyaT-u}, or
\arb{.hurUf-u 'l-mu`jam-i}) are twenty-eight in number and
are all consonants, though three of them are also used as
vowels (see §~3).
```

From Wright (1896, i. 1 A):— Arabic, like Hebrew and Syriac, is written and read from right to left. The letters of the alphabet (حُرُوفُ, حُرُوفُ الهِجَآءِ, اَلحُرُوفُ الهِجَآئِيَّةُ, التَّهَجِّي, or حُرُوفُ المُعجَمِ) are twenty-eight in number and are all consonants, though three of them are also used as vowels (see § 3).

FIGURA 6: Testo inglese con brevi inserti in arabo. Tratto dalla documentazione di arabluatex.

```
\begin{arab}
  'at_A .sadIquN 'il_A ju.hA ya.tlubu min-hu .himAra-hu
  li-yarkaba-hu fI safraTiN qa.sIraTiN fa-qAla la-hu:
  \enquote{sawfa 'u`Idu-hu 'ilay-ka fI 'l-masA'-i
  wa-'adfa`u la-ka 'ujraTaN.} fa-qAla ju.hA:
  \enquote{'anA 'AsifuN jiddaN 'annI lA 'asta.tI`u 'an
  'u.haqqiqa la-ka ra.gbata-ka fa-'l-.himAr-u laysa hunA
  'l-yawm-a.}  wa-qabla 'an yutimma ju.hA kalAma-hu bada'a
  'l-.himAr-u yanhaqu fI 'i.s.tabli-hi. fa-qAla la-hu
  .sadIqu-hu: \enquote{'innI 'asma`u .himAra-ka yA ju.hA
  yanhaqu.} fa-qAla la-hu ju.hA: \enquote{.garIbuN
  'amru-ka yA .sadIqI 'a-tu.saddiqu 'l-.himAr-a
  wa-tuka_d_diba-nI?}
 \end{arab}
```

أَتَى صَدِيقٌ إِلَى جُحَا يَطلُبُ مِنهُ حِمَارَهُ لِيَركَبَهُ فِي سَفرَةٍ قَصِيرَةٍ فَقَالَ لَهُ: "سَوفَ أُعِيدُهُ إِلَيكَ فِي المَسَآءِ وَأَدفَعُ لَكَ أُجرَةً." فَقَالَ جُحَا: "أَنَا آسِفٌ جِدًّا أَنِّي لَا أَستَطِيعُ أَن أُحَقِّقَ لَكَ رَغبَتَكَ فَالحِمَارُ لَيسَ هُنَا اليَومَ." وَقَبلَ أَن يُتِمَّ جُحَا كَلَامَهُ بَدَأَ الحِمَارُ يَنهَقُ فِي إِصطَبلِهِ. فَقَالَ لَهُ صَدِيقُهُ: "إِنِّي أَسمَعُ حِمَارَكَ يَا جُحَا يَنهَقُ." فَقَالَ لَهُ جُحَا: "غَرِيبٌ أَمرُكَ يَا صَدِيقِي أَتُصَدِّقُ الحِمَارَ وَتُكَذِّبَنِي؟"

FIGURA 7: Esempio d'uso dell'ambiente arab. Tratto dalla documentazione di arabluatex.

invece che con spazio bianco. Invece non sono al corrente della possibilità di eseguire la cesura delle parole nemmeno nelle traslitterazioni in caratteri latini. Quindi la giustificazione dei testi in lettere arabe è affidata esclusivamente alla kashida.

4.1 Codifica del testo sorgente

Sempre prendendo integralmente dalla documentazione di arabluatex, si può vedere come si codifica un testo in arabo mescolato a un testo in lettere latine nella figura 6.

4.2 L'uso del pacchetto arabluatex

Il pacchetto si carica nel solito modo, solo che bisogna anche scegliere un font per comporre in arabo.

```
\usepackage{⟨opzioni⟩}{arabluatex}
```

Fra le opzioni ci sono `voc`, `fullvoc`, `novoc`, `trans`. L'ultima opzione serve per produrre la traslitterazione secondo una delle tre scelte attualmente disponibili, **dmg**, **loc**, oppure **arabica**; senza specificare nulla viene usata la traslitterazione **dmg**. Le altre tre opzioni riguardano la vocalizzazione: dovrebbero essere auto esplicative; ma la cosa comoda è che, indipendentemente dall'opzione specificata al pacchetto, ciascuna di esse può essere specificata come opzione sia al comando `\arb` sia all'ambiente arab.

4.3 I font OpenType che contengono anche l'arabo

Per quanto riguarda i font, questi devono essere OpenType. Non mi pare che attualmente il sistema TEX includa fra i suoi font OpenType anche font che contengano le pagine UNICODE relative all'arabo.

Invece fra i font disponibili con il sistema operativo ce ne sono molti che contengono l'arabo in diversi stili. Fra i primi che si incontrano scorrendo l'elenco dei font installati in un Mac, c'è la collezione di font Amiri; una loro particolarità sono i font specialmente adatti per la stampa del Corano, che prendono il nome di Amiri Quran e Amiri Quran colored, il secondo dei quali serve per inserire le lettere e le vocalizzazioni nel testo coranico in modo che i colori svolgano i compiti speciali loro riservati durante le preghiere collettive recitate nelle moschee. In ogni caso, se non si specifica nessun font, il pacchetto cerca di caricare il font Amiri, se lo trova nella particolare macchina su cui viene usato.

Se non ci si accontenta dei font installati nella propria macchina, se ne possono sempre installare altri; io ho trovato molto interessante il font Scheherazade completo anche dei simboli coranici.

Per terminare la descrizione della composizione in arabo, segnalo che prima dell'esistenza di arabluatex era già disponibile il pacchetto arabxetex per comporre con l'interprete `xetex` (quindi lavorando a livello utente con `xelatex`) e senza le funzionalità del linguaggio Lua. Forse il precedente pacchetto arabxetex offre maggiori funzionalità per l'utente, ma soffre del fatto che il formato di default di `xelatex` è una variante estesa del formato DVI, che normalmente viene trasformata al volo nel formato PDF. Questo passaggio per il formato DVI presenta numerosi svantaggi sotto molti aspetti. Per questo preferisco usare il pacchetto arabluatex anche se è ancora sotto sviluppo e se funziona solo con `lualatex`.

5 Comporre in giapponese

Si è già detto molto su come configurare il preambolo e su come usare il driver della tastiera giapponese. Non lo si ripete. Qui parliamo di come comporre davvero in questa lingua.

Normalmente oggi il giapponese si scrive da sinistra a destra in righe giustificate ma seguendo una specie di *lectio continua*. Non si pongono problemi di divisione in sillabe, perché ogni ideogramma è una sillaba; senza alcun segno di "a capo" con un trattino o con un segno equivalente, ogni parola, quasi sempre polisillabica, si può dividere in fin di riga in una posizione qualunque. Questo è ciò che avviene quando si usano gli ideogrammi. Scrivendo in romaji, sarebbe desiderabile che esistesse un file contenente un sia pur minimo insieme di pattern, almeno il romaji potrebbe essere composto con parole staccate e correttamente divise in fin di riga. Tuttavia, ad oggi i colleghi giapponesi non hanno ancora sentito questa necessità, forse perché considerano il romaji come un ripiego a cui ricorrere solo in caso di bisogno estremo; per esempio vengono usati nei tabelloni delle stazioni dove i binari e le destinazioni dei treni sono presentati alternativamente in romaji, in katakana (figura 1, più adatto per i display con pochi pixel luminosi) e in kanji (comunque poco adatto a quei display).

L'ambiente japanese definito in precedenza viene ripreso:

```
\newenvironment{japanese}[1][]
{#1\japanesefont}{}
```

Esso accetta un argomento facoltativo, che è predefinito come argomento vuoto; se lo si usa potrebbe servire per specificare la composizione in bandiera, così da non avere problemi di giustificazione. Per esempio, il codice

```
\begin{japanese}[\raggedright]
まだ何もない時、
神様は天と地をお造りになりました
地球はまだ形が定まらず
やみにおおわれた氷の上を
さらに神様の霊がおおっていました
「光よ、輝け」と神様が命じました。
すると光がさっと輝いたのです。
\end{japanese}
```

Testo in orizzontale
人人生而自由, 在尊嚴和權 利上一律平等。

Testo in verticale
人人生而自由, 在尊嚴和權 利上一律平等。

FIGURA 8: Testo composto dentro due minipage affiancate con le dichiarazioni \yoko e \tate

produce il testo

まだ何もない時神様は天と地をお造りになりました地球はまだ形が定まらずやみにおおわれた氷の上をさらに神様の霊がおおっていました「光よ輝け」と神様が命じました。すると光がさっと輝いたのです。

Di fatto, la giustificazione si può ottenere scrivendo un testo sorgente molto ordinato e inserendo fra una parola e l'altra qualche comando di leggera spaziatura che permette al programma di inserire un po' di gomma fra le parole in modo da agevolare la lettura.

Tuttavia, in giapponese formale qualche volta si scrive in verticale con le colonne che si leggono ciascuna dall'alto al basso e le colonne intere da destra a sinistra. Conviene fare riferimento al seguente esempio (scritto in cinese), dove lo stesso testo è scritto in righe composte da sinistra a destra che si susseguono dall'alto in basso, oltre che in colonne composte dall'alto in basso che si susseguono da destra a sinistra. Per comporre con lo stile formale con xelatex il testo viene composta normalmente in una minipage larga abbastanza per contenere senza andare a capo la riga più lunga. Poi bisogna mandare nell'output questa minipage ruotata di 90° in senso orario controruotando gli ideogrammi in essa contenuti di 90° in senso antiorario; con le opzioni di fontspec si può fare (figura 9).

Il risultato della figura 9 si ottiene con il codice seguente quando si usa xelatex:

```
\begin{minipage}[t]{45mm}
Testo in orizzontale:\\[1.5ex]
\chinesefont
人人生而自由，在尊嚴和權 利上一律平等。
\end{minipage}
\qquad
\begin{minipage}[t]{45mm}
\hspace*{-0.25em}Testo in verticale:\\
\rotatebox{-90}[origin=lt]{%
\parbox[t]{45mm}{%
\chinesefont
\addfontfeatures{Renderer=AAT,
    Vertical=RotatedGlyphs}%
人人生而自由，在尊嚴和權 利上一律平等。}}
\end{minipage}
```

Testo in orizzontale:
人人生而自由,在尊嚴和權 利上一律平等。

Testo in verticale:
人人生而自由,在尊嚴和權 利上一律平等。

FIGURA 9: Composizione con XeLaTeX dello stesso testo cinese in orizzontale e in verticale

Con lualatex la procedura in un certo senso è più semplice, ma da solo non ci sarei arrivato; ringrazio Luigi Scarso che mi ha indicato il pacchetto e i comandi da usare, ma dei quali non ho trovato traccia nella documentazione né di fontspec né di luatex; forse ho guardato male, ma... Infatti, invece di caricare il pacchetto fontspec, bisogna caricare il pacchetto luatexja-fontspec (THE LUATEX-JA PROJECT TEAM, 2017) scritto apposta per comporre in giapponese con lualatex. Va notato che con questo pacchetto per caricare i font alfabetici come il latino, il greco, il cirillico si usano i soliti comandi, ma per caricare i font del set completo CJK (Chinese, Japanese, Korean) bisogna usare i comandi che contengono una 'j' nel loro nome, come, per esempio, \newjfontfamily invece di \newfontfamily.

Tra l'altro questo pacchetto luatexja-fontspec rende disponibili i comandi \yoko e \tate; il primo dichiara di voler comporre in orizzontale, il secondo in verticale. Ecco allora il codice

```
\usepackage{luatexja} % Nel preambolo...

%   e nel corpo del documento
\begin{minipage}[t]{40mm}
Testo in orizzontale\par\medskip
\vbox{\hsize=\textwidth
\yoko\noindent
人人生而自由，在尊嚴和權 利上一律平等。}
\end{minipage}
%
\hfill
%
\begin{minipage}[t]{40mm}
Testo in verticale\par\medskip
\vbox{\hsize=\textwidth
\tate\noindent
人人生而自由，在尊嚴和權 利上一律平等。}
\end{minipage}
```

compone il suo testo nel modo mostrato nella figura 8. Si osservi che i comandi \yoko e \tate non possono essere preceduti da altro ed è per questo che sono stati introdotti dentro una scatola nativa dell'interprete, \vbox, con la sintassi nativa per

祇園絵や 不味くが原の 風かおる	Gion e ya Makazu ga hara no kaze kaoru	Visione di Gion. Della piana di Makazu profuma il vento.

FIGURA 10: Lo stesso *haiku* in giapponese kanji verticale, in romaji e la sua traduzione in italiano

impostare la larghezza della scatola pari a quella della minipage che la contiene: infatti, in questo caso `\textwidth` si riferisce alla giustezza del testo dentro la minipage.

La figura 10 contiene un *haiku*, tratto da (STARACE, 2005), composto in tre modi diversi esposti da sinistra a destra: la versione dell'haiku in giapponese verticale, poi lo stesso haiku scritto in romaji, infine la traduzione italiana.

6 Conclusioni

I programmi `lulatex` e `xelatex` sono ottimi per gestire i font OpenType con i quali bisogna confrontarsi per comporre testi che contengano brani di testo oppure l'intero testo con caratteri diversi da quelli latini. Volendo trattare di lingue orientali in un'università, è decisamente necessario saper maneggiare questi font completi dei segni di quelle lingue. Questo articolo è solo un tutorial ma dovrebbe essere sufficiente per iniziare la strada dei programmi basati sul mark up LaTeX; la bibliografia dovrebbe contenere sufficienti informazioni per poter andare oltre questi primi rudimenti.

Nello stesso tempo, le lingue orientali non sono pane per il dente di tutti; un arabista, oltre a conoscere e a saper leggere e scrivere l'arabo o il farsi, deve imparare a scrivere il testo sorgente in arabo o in farsi servendosi della codifica in caratteri latini del sottoinsieme ASCII. Uno studioso di cinese o di giapponese, oltre a conoscere le lingue specifiche, deve conoscere migliaia, se non decine di migliaia, di ideogrammi. Gli studiosi di giapponese devono conoscere anche i sillabari hiragana e katakana, cosa che non dovrebbe essere loro particolarmente difficile, perché le due collezioni di segni ammontano ad un totale che non supera il numero di 150, un'inezia rispetto alle migliaia di kanji necessari per quella lingua.

Le difficoltà tecniche sono minime, rispetto al numero enorme di sillabe e di ideogrammi che bisogna saper maneggiare con destrezza.

Il coreano, a confronto è semplicissimo con un numero limitato segni per le consonanti e per le vocali o suoni vocalici che noi considereremmo dittonghi.

Non ho nemmeno accennato a due altre lingue e ai rispettivi alfabeti molto in uso in Oriente: il devanagari e il thai. Il devanagari ha una importanza enorme, anche grazie a tutta la letteratura relativa all'induismo e al buddismo. Il thai è certamente meno importante del devanagari, ma si riferisce anch'esso non solo ai testi moderni, ma anche a quelli tradizionali del buddismo, che in Thailandia è religione di stato al punto che persino la data civile è calcolata dalla nascita del Budda.

Spero che la lettura di questo testo introduttivo possa richiamare l'attenzione degli orientalisti, non solo per la loro personale utilità, ma anche per contribuire a migliorare il software esistente; giustamente quando gli utenti orientali predispongono qualcosa per il sistema TeX, hanno l'uso quotidiano in mente, non le esigenze degli *scholars*, studiosi occidentali di queste discipline.

7 Ringraziamenti

Ringrazio di cuore Luigi Scarso; senza il suo aiuto non avrei saputo districarmi nelle finezze di LuaLaTeX con la sua particolare gestione dei font, specialmente quelli orientali. Per giunta in questo periodo di tempo l'interprete `luatex` ha subito gli ultimi ritocchi prima di essere rilasciato in versione stabile, per cui è stato necessario ricorrere ad un vero esperto. Quindi la mia esperienza precedentemente acquisita con le versioni "beta" dell'interprete ha dovuto essere completamente riveduta e, specialmente, aggiornata.

Riferimenti bibliografici

ALESSI, R. (2017). «The arabluatex package». PDF document. Leggibile con `texdoc arabluatex` con TeXLive.

BECCARI, C. e GORDINI, T. (2016). «L'Arte di scrivere in diverse lingue con (Xe)LaTeX». PDF document. Scaricabile dal sito del GUIT.

CHARETTE, F. (2015). «ArabXeTeX – An ArabTeX-like interface for typesetting languages in Arabic script with XeLaTeX». PDF document. Leggibile con `texdoc arabxetex` con la distribuzione TeXLive.

GOOSSENS, M. (2011). *The XeTeX Companion – TeX meets OpenType and Unicode*. LaTeX Team. In http://xml.web.cern.ch/XML/lgc2/xetexmain.pdf.

GOOSSENS, M., MITTELBACH, F. e BRAAMS, J. (2004). *The LaTeX Companion*. Addison-Wesley.

Gregorio, E. (2011). «Introduzione a XeLaTeX». PDF document. URL http://profs.sci.univr.it/~gregorio/introxelatex.pdf.

Jabri, Y. (2006). «Typesetting Arabic and Farsi with the Arabi package –The Users Guide». PDF document. Leggibile con texdoc arabi con la distribuzione TEXLive.

Khalighi, V. (2017). «The bidi Package». PDF document. Leggibile con texdoc bidi con la distribuzione TEXLive.

Knuth, D. E. (1996). *The TEXbook*. Addison Wesley, Reading, Mass., 16ª edizione.

Lamport, L. (1994). *LATEX. A Document Preparation System*. Addison-Wesley.

LuaTEX Development Team (21017). «LuaTeX Reference Manual». PDF document. Leggibile con texdoc luatex con TEXLive.

Mittelbach, F., Goosens, M. *et al.* (2004). *The LATEX companion*. Addison Wesley, Reading, Mass., 2ª edizione.

— (2007). *The LATEX graphics companion*. Addison Wesley, Reading, Mass., 2ª edizione.

Orsi, M. T. (a cura di) (2012). *La storia di Genji*. Collana *I Millenni*. Einaudi, Torino. Prima traduzione dal giapponese antico (secolo xi) comprendente tutti i 54 capitoli. Titolo originale 源氏物語 (Genji monogatari).

Robertson, W. (2011). «The XeTeX reference guide». PDF document. $TEXMF/doc/xetex/xetexref/XeTeX-reference.pdf.

Starace, I. (a cura di) (2005). *Il grande libro degli haiku*. Castelvecchi.

The LuaTeX-ja project team (2017). «The LuaTeX-ja package». PDF document. Leggibile con texdoc luatexja con TEXLive.

▷ Claudio Beccari
claudio dot beccari at gmail dot com

Tutorial
Yudit: un editor Unicode che “parla” (anche) LaTeX

Gianluca Pignalberi

Sommario

Gli utenti LaTeX hanno a disposizione un'ampia gamma di editor tra cui scegliere il preferito: dai più spartani editor di testo privi di funzioni specifiche ai più complessi IDE da cui lanciare le compilazioni, verificare la coerenza e correttezza del codice e ottenere aiuti alla scrittura grazie alle scorciatoie. Tra le varie possibili scelte c'è Yudit, un editor Unicode leggero scritto agli albori dello standard Unicode su Linux.

Abstract

LaTeX users can pick their favorite editor among a wide range of such programs: from the most basic text editors without specific functions to the most complex IDEs to compile documents, verify the code coherence and correctness and obtain typing shortcuts from. Among the several choices we have Yudit, a lightweight Unicode editor written during the very first days of the Unicode standard on Linux.

1 Introduzione

Da diversi anni, ormai, gli utenti LaTeX possono beneficiare del supporto Unicode durante la composizione dei proprî documenti. La parte “difficile”, eventualmente, è inserire determinati caratteri in maniera efficiente.

Tutti i moderni sistemi operativi offrono un ottimo supporto multilingua e mettono a disposizione degli utenti diverse configurazioni di tastiera. Purtroppo anche questo può essere limitante perché non tutti gli utenti hanno diverse tastiere da collegare al computer[1] e, in ogni caso, nessuna tastiera normale avrà mai tutti i possibili caratteri.[2]

Esistono diversi programmi di utilità (*utility*) che mostrano una mappa dei caratteri selezionando i quali si compone una stringa che può poi essere copiata e incollata nei proprî documenti. Questa soluzione è molto versatile ma può essere scomoda se la quantità di stringhe da scrivere supera le poche unità.

Nei casi in cui le precedenti soluzioni non siano praticabili è sempre possibile ricorrere a editor di testo[3] specializzati, verosimilmente “residui” di quando l'immissione di testo multilingue era ancor più complicata. Emacs, per esempio, ha un ottimo supporto multilingue ma è tutto tranne che un editor leggero: è più un coltellino svizzero, delle dimensioni di un *machete*, capace di moltissime funzioni. Un editor leggero ma molto versatile dal punto di vista dell'immissione di testo multilingue è Yudit, tra le cui capacità c'è quella di far inserire direttamente dei caratteri Unicode anche se non mappati da alcuna tastiera e che offre un moderato supporto a LaTeX permettendo all'utente di scrivere dei glifi ricorrendo proprio ai comandi del nostro compositore preferito.

Questo tutorial costituisce un breve avvio all'uso di Yudit (nella versione per Linux) con l'auspicio che soprattutto i linguisti possano trarre beneficio dalle sue funzionalità.

2 Installazione

Sul sito ufficiale del programma troviamo gli eseguibili per Windows (32 e 64 bit) e per Linux, quest'ultimo in un comodo file RPM (comodo solo per gli utenti di distribuzioni derivate da Red Hat). Oltre a questi, abbiamo a disposizione i sorgenti (distribuiti sotto forma di *tarball* e RPM), le istruzioni per compilare l'eseguibile per Linux e Windows e per realizzare un pacchetto `.deb` per Debian e derivate. Sui repository, ufficiali e non, delle varie distribuzioni troviamo le versioni pacchettizzate.

L'installazione non comporta difficoltà e pure la compilazione non è molto difficoltosa perché Yudit è interamente basato sulla libreria standard del C e su un toolkit a finestre generico, non dipende da altro; questo è il motivo del suo aspetto “peculiare”. La versione stabile corrente è la 2.9.6 e la beta della 2.9.7 è già disponibile.

3 Uso e configurazione

Una volta avviato, Yudit appare come uno spartano editor di testo, con una barra di informazioni

1. Non è necessario per un computer avere la tastiera russa per permettere all'utente di digitare il russo perché il sistema riconfigura i tasti mappando, per esempio, la Q in Й. L'utilità di avere la perfetta corrispondenza mappa-tastiera è per l'utente che avrà più facilità a digitare il testo.

2. In realtà non credo neanche che esista un un font con tutti i glifi a disposizione.

3. La locuzione *editor di testo* ha un'accezione piuttosto limitata per gli informatici formati negli anni '70-'80: è un programma che permette di scrivere un semplice testo, modificarlo, salvarlo in formato `.txt` e poco altro. Sono editor di testo `edit` di MS DOS e Blocco note di Windows il quale permette pure la ricerca all'interno del testo; Word *non* lo è: è un *elaboratore* di testo.

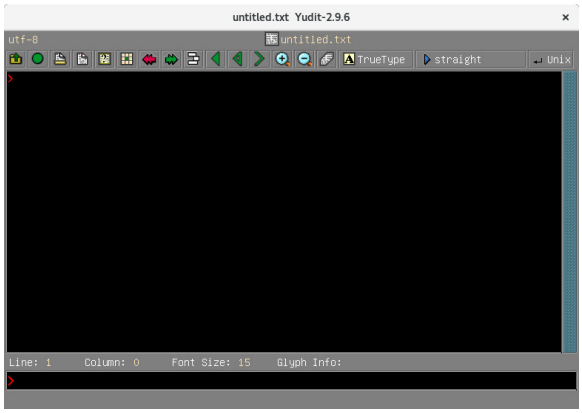


FIGURA 1: L'interfaccia di Yudit.

superiore, una barra dei pulsanti, lo spazio di inserimento del testo (che può avvenire da sinistra a destra — SD da ora in poi — o da destra a sinistra — DS da ora in poi —), una barra di informazioni inferiore e una riga di comando. La figura 1 mostra l'editor appena avviato.

Nella barra superiore delle informazioni troviamo l'indicazione sulla codifica del file (utf-8) e il nome del file aperto. Per default viene tentata l'apertura di `untitled.txt`. La sottostante barra dei pulsanti mette a disposizione dell'utente le seguenti funzionalità: "apri file", "salva", "stampa", "stampa anteprima", "cerca", "vai a", "annulla", "riesegui", "allinea testo", "ignora direzione", "ignora inclusione", "forza inclusione", "ingrandisci" e "rimpicciolisci" (il font del testo), "modalità di colorazione contestuale del testo" o "evidenziazione" (*highlighting*), "font" (il pulsante mostra quello in uso correntemente), "modalità di inserimento" (il pulsante mostra quella correntemente in uso) e "fine riga" (Dos, Mac, Unix e, dalle versioni più recenti, Ps/Ls, cioè *Unicode Paragraph/Line Separator*).

Sullo spazio di inserimento del testo non c'è nulla da dire se non che il testo può essere inserito nelle varie modalità previste dallo standard Unicode. La sottostante barra delle informazioni contiene il numero di riga e colonna in cui si trova il cursore, la dimensione in punti del font usato nell'editor e le informazioni sul glifo, segnatamente il suo codice ASCII/Unicode. A seconda della modalità di inserimento e della direzione troviamo informazioni aggiuntive sul tasto o sulla sequenza di tasti usati per digitare il glifo e la direzione.

Da bravo editor spartano, Yudit non ci fornisce alcun menù tramite cui configurare il suo aspetto o comportamento. Possiamo però agire sul file delle proprietà per configurare molto dettagliatamente il programma. Nella cartella (che per retaggio culturale mi ostinerò a chiamare directory) `~/.yudit` troviamo le sottodirectory `data`, `font`, `syntax` e il file `yudit.properties`. Quest'ultimo è un semplice file di testo che possiamo modificare in base alle nostre esigenze. Vediamo come.

Il file inizia con una serie di righe di spiegazione iniziate dal carattere `#`. Questo indica che tali righe sono dei commenti utili all'utente ma inutilizzate da Yudit. La prima proprietà, che però non possiamo personalizzare, è il numero di versione di Yudit. Pur provandoci, Yudit sovrascriverà comunque l'informazione con quella propria.

Le successive proprietà configurabili sono: colore[4] dello sfondo della finestra, colore di sfondo della riga di comando e del suo cursore SD e DS, font usato per i comandi e sua dimensione, colore dei caratteri per l'immissione SD e viceversa. Seguono due informazioni su dove si trova il file locale di configurazione (i commenti iniziali ci ricordano che, in mancanza di esso, Yudit si rifarà alla configurazione generale sita in `/usr/share/yudit/config/yudit.properties`) e dei file `.my` (cioè i file delle mappe della tastiera contenuti nella directory `data`).

Le due seguenti proprietà indicano quale codifica avrà la *clipboard* (cioè lo spazio di memoria in cui viene salvato ciò che viene tagliato o copiato) e il file di default (`untitled` in formato UTF-8).

Immediatamente di seguito troviamo la configurazione del formato del file di testo. Il formato predefinito è, di nuovo, UTF-8 ma abbiamo a disposizione un nutrito numero di formati alternativi, peraltro selezionabili nella finestra di salvataggio (come mostrato nella figura 2).

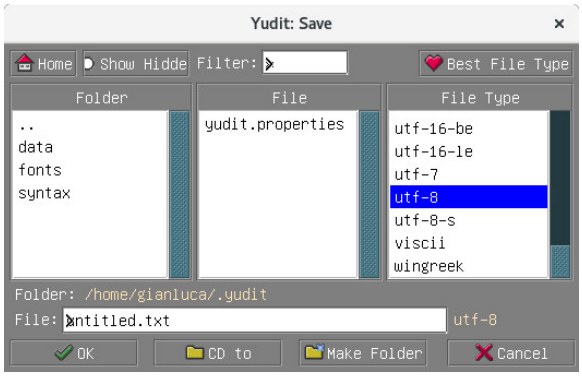


FIGURA 2: Nella colonna di destra della finestra di salvataggio possiamo selezionare il formato del file di testo. Il numero quasi rivaleggia con quello messo a disposizione da `iconv`.

Quindi sono indicate le configurazioni relative al font usato da Yudit per tutte le indicazioni del programma all'utente. Sconsiglio di modificare tali impostazioni perché a ogni modifica del font (che è indicato come "default" ed è di sicuro uno di quelli indicati a pagina 4) ho sempre ottenuto delle scritte illeggibili perché composte da quadrati col codice Unicode del glifo anziché dal glifo stesso.

Seguono l'indicazione della geometria della finestra, della lingua, del comando da usare per l'anteprima (`gv`), il colore di sfondo dell'area di

4. I (nomi dei) colori sono quelli definiti per X11. Si possono trovare anche nel manuale del pacchetto xcolor.

scrittura e del cursore (nei casi SD e DS), i tipi di file (in aggiunta a quelli di `uniconv`), il font corrente nell'editor, quindi segue una lunga lista di associazioni di etichette e font possibili da usare nell'area di editing e di colorazioni dell'evidenziazione.

Per avere le informazioni relative alla configurazione possiamo digitare, nella riga di comando di Yudit (si passa dallo spazio di editing alla riga di comando e viceversa premendo Esc, quasi in stile `vi`, o col mouse),

```
howto configure
```

mentre per avere informazioni generali digitiamo

```
help
```

nello stesso spazio adibito ai comandi all'editor.

Iniziamo la digitazione in modalità straight, cioè quella identicamente uguale alla tastiera di sistema (o quasi: io ho difficoltà a inserire le accentate in questa modalità da quando ho Linux Fedora 26). Per cambiare modalità di inserimento (cioè cambiare mappa della tastiera e quella dei caratteri attivi[5]), che da ora in poi chiameremo pur impropriamente *mappa tastiera*, abbiamo due modi: normale e rapido. Per cambiare in modo normale premiamo il pulsante inserimento e si aprirà una finestra (figura 3). Nella colonna di sinistra troviamo tutte le mappe disponibili, nella colonna centrale ne troviamo 12 associate ai tasti F1...F12 (in mezzo a queste due colonne c'è un pulsante che useremo per associare una delle mappe a uno dei tasti funzione dopo aver selezionato entrambi) e nella colonna di destra che mostra quale glifo è associato a quale tasto o sequenza di tasti.

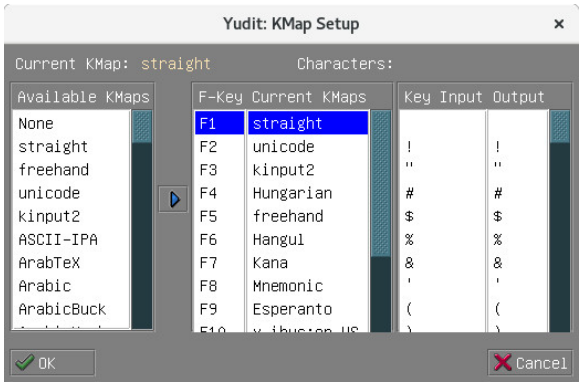


FIGURA 3: Associazione mappa tastiera-tasti funzione al primo avvio di Yudit.

Proviamo a cambiare associazioni per un caso reale: il mio. Voglio associate al tasto F3 il russo in configurazione яверты (più comoda perché mappa i caratteri traslitterandoli quando possibile: a → a, b → б e così via) e non йцукен e al tasto F4 TEX. Basta selezionare la mappa corretta e il tasto a

5. Definiamo *caratteri attivi* quei caratteri che in combinazione con determinati altri caratteri successivi inseriscono nel testo un glifo specifico diverso da quelli inseriti.

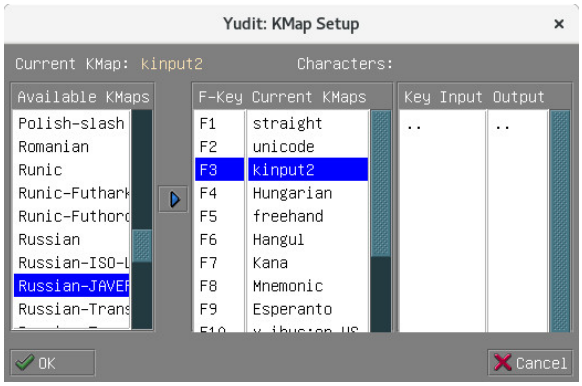


FIGURA 4: Quale mappa tastiera vogliamo associare a quale tasto funzione?

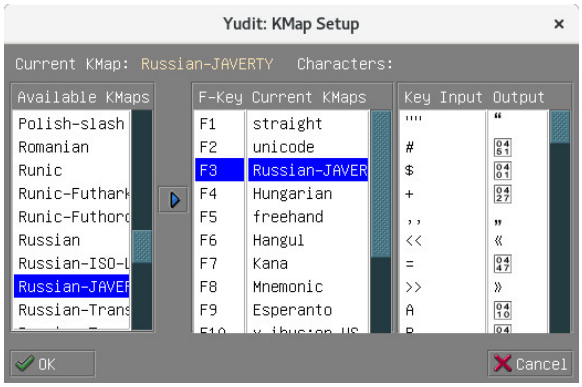


FIGURA 5: La nuova associazione mappa tastiera-tasto funzione è ora attiva.

cui associarla (figura 4), quindi premere il pulsante con la freccia per aggiornare le associazioni e il relativo file di configurazione (figura 5).

Una volta che abbiamo personalizzato o meno le associazioni mappa tastiera-tasto funzione in base ai nostri bisogni possiamo cambiare rapidamente la modalità di inserimento premendo il tasto funzione associato a tale mappa.

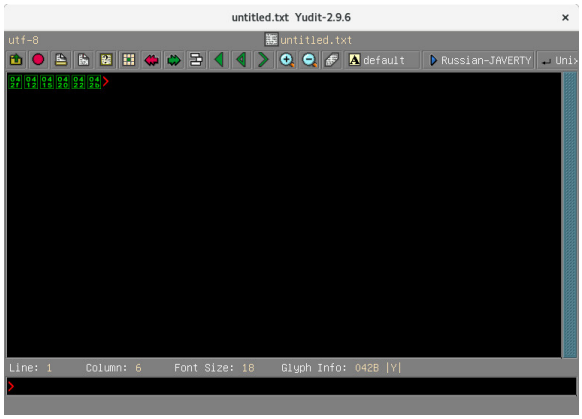


FIGURA 6: Il testo russo risulta illeggibile perché il font in uso non dispone dei glifi adatti.

Nelle figure 5 e 6 notiamo che i glifi cirillici sono invisibili. Al loro posto compaiono i relativi codici Unicode. Questo significa che i font

di default di Yudit (molti dei quali non possono essere distribuiti col programma e vanno quindi cercati e copiati nella directory `~/.yudit/fonts`) non sono presenti nel sistema usato o non contengono quei glifi. Dunque rimettiamo mano al file di configurazione per fare in modo di usare un font Unicode diverso da quelli impostati di default. Ci affideremo a GNU Free Serif, Free SansSerif e Free Mono, gli equivalenti liberi di Times, Helvetica e Courier. Per prima cosa troviamo i relativi file nel nostro sistema (in Linux Fedora sono sotto `/usr/share/fonts/gnu-free`) e copiamoli in `~/.yudit/fonts`.[6] Cerchiamo le stringhe `yudit.font.SansSerif` e `yudit.font.Serif` e dopo il segno `=` scriviamo, rispettivamente, `FreeSans.ttf` e `FreeSerif.ttf` per istruire Yudit a usare gli onnipresenti GNU Free SansSerif e Free Serif. Per usare Free Mono creiamo una nuova categoria di font in `yudit.editor.fonts`: `Mono`. Dopodiché andiamo, sempre nel file di configurazione, dopo la riga in cui abbiamo configurato Free Serif e scriviamo:

```
yudit.font.Mono=FreeMono.ttf
```

Ora basta salvare le modifiche e riavviare Yudit per aver modo di scrivere anche in Courier (figura 7).[7]

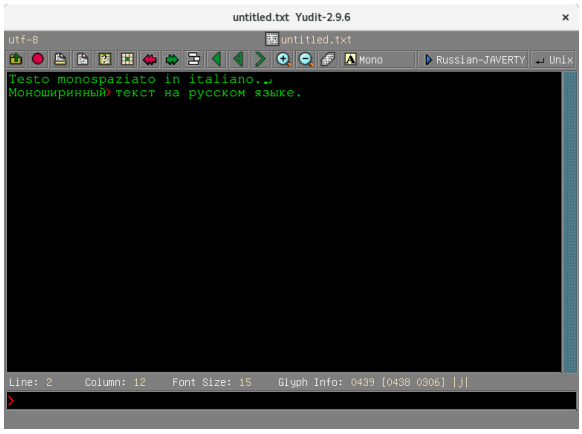


FIGURA 7: Yudit con un font monospaziato.

L'obiettivo, però, non è tanto usare un font monospaziato nell'editor quanto usarlo nella finestra di configurazione delle mappe di tastiera per fare in modo che i glifi non ancora visibili lo diventino. Purtroppo la cosa non è possibile a livello di configurazione. Come confermatomi dall'autore, i font usati in questo caso sono predeterminati nel codice sorgente, precisamente nel file `yudit-2.9.6/swindow/SFont.cpp`:

```
66  SStringVector m(
67      /* yudit.hex has ohungarian.hex and part of unifont.hex */
68      // Indic range is not necessary free. But they are the best
69      "yudit.hex,arabforms.hex,syriacforms.hex,unifont.hex,"
70      "markus9x18.bdf,markus18x18ja.bdf,"
71      "-*-*-medium-r-normal--16-*-*-*-c-*-iso8859-1,"
72      "-*-*-*-*-*--16-*-*-*-c-*-iso8859-1,"
73      "THOOLIUC.TTF:mlym,"
74      "MuktiNarrow.ttf:beng,"
75      "ani.ttf:beng,"
76      "pothana2000.ttf:telu,"
77      "TCRCYoutsoUnicode.ttf:tibt,"
78      "raghu.ttf:deva,"
79      "mangal.ttf:deva,tunga.ttf:knda,code2000.ttf:taml,"
80      "raavi.ttf:guru,shruti.ttf:gujr,"
81      "arialuni.ttf,cyberbit.ttf,"
82      "code2000.ttf,code2001.ttf:unicode:RL,arial.ttf,"
83      //"rovasSMP.ttf," // Hosszu Gabor
84      //"oldhunSMP.ttf," // Michael Everson
85      "yudit.ttf"
86  );
```

A meno di non modificare i sorgenti e ricompilare tutto,[8] la soluzione è installare nel proprio sistema uno dei font elencati. Dopo aver copiato `unifont.hex` nella directory locale dei font il risultato è quello mostrato nella figura 8.

Vediamo ora come l'uso di caratteri attivi ci permetta di inserire una serie di simboli non presenti nella tastiera. Selezioniamo la mappa tastiera Unicode e scriviamo la seguente stringa:

u2014␣u263a␣u2469␣u2127␣u2055

Man mano che scriviamo un codice Unicode ben formato, cioè formato da una 'u' e un numero di 4 cifre esadecimali (la u, mostrata sottolineata, indica l'immissione di un carattere attivo; i successivi

6. Naturalmente potremmo sempre modificare la proprietà `yudit.fontpath` del file di configurazione per far sì che Yudit possa riconoscere uno o più font installati nel sistema senza doverli copiare localmente, come invece abbiamo fatto qui. La copia locale dei font è utile solo nel caso in cui tali font non siano installati nel sistema, non possano esserlo o, pur essendolo, debbano essere rimossi.

7. Ricordiamo che Yudit è un editor di testo e non un word processor, pertanto il font usato sarà usato per tutto il testo; ogni cambiamento di font si ripercuoterà su tutto il testo e nessuna informazione sui font verrà salvata.

8. Ho suggerito all'autore di includere nel sorgente ufficiale anche i font GNU Free, Linux Libertine/Biolinum e DejaVu.

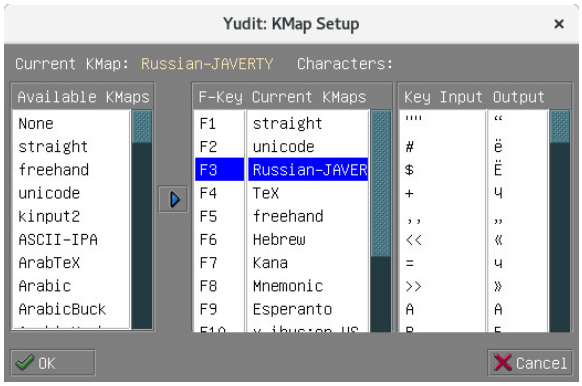


FIGURA 8: Il testo russo risulta leggibile anche nelle finestre dopo aver installato un font che Yudit riconosce perché incluso nel suo codice sorgente.

numeri o lettere a–f sottolineati indicano l'immissione di una sequenza attiva; si veda la figura 9), questo viene sostituito dal corrispondente carattere. La stringa risultante è mostrata nella figura 10.

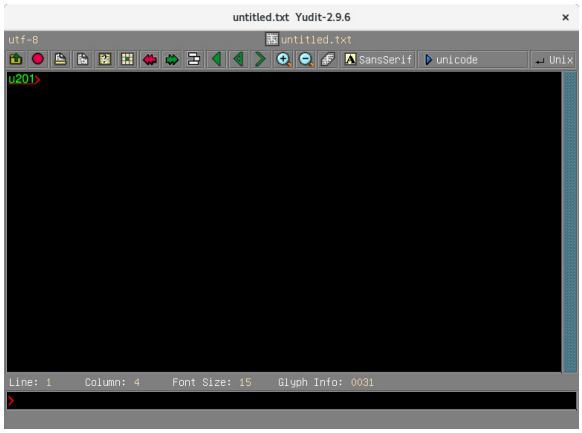


FIGURA 9: Una sequenza attiva inserita con la modalità di inserimento Unicode.

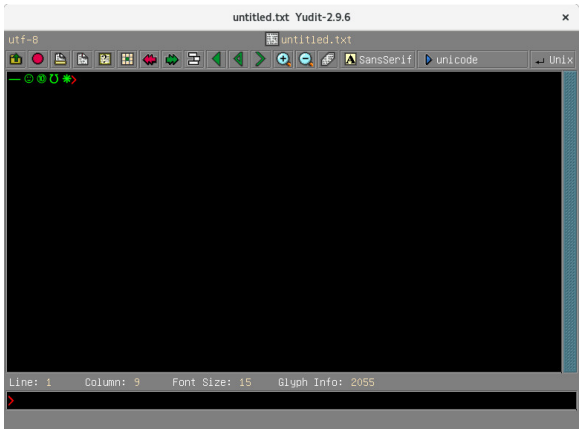


FIGURA 10: Yudit rimpiazza una sequenza attiva col carattere corrispondente se questo è presente nel font usato.

La scrittura DS è un compito molto facile per Yudit. Se scegliamo una lingua scritta in tale verso, ad esempio l'ebraico, Yudit imbandiera a destra tutto il testo scritto in questa lingua (a meno di non aver agito sul pulsante "allinea testo" per forzare l'allineamento sempre a sinistra o sempre a destra) e tale testo è riconoscibile per il diverso colore (a meno di non aver modificato tale proprietà) rispetto al testo SD.

Può capitare che all'interno di un testo SD occorra inserire delle parole o frasi scritte in una lingua DS. In questi casi bisogna fare attenzione a che la punteggiatura ricada in maniera corretta nel testo. A questo servono i pulsanti "ignora direzione" e "ignora inclusione". La sezione `bidi` degli *howto* di Yudit fornisce dettagli tecnici e alcuni esempi utili a chiarire la questione. Basta digitare

```
howto bidi
```

nella riga di comando di Yudit per accedervi. Qui viene spiegato anche l'uso del pulsante "forza inclusione" che serve a portare il testo evidenziato alla direzione principale del testo.

Un'ultima interessante caratteristica di Yudit è quella offerta dalla mappa `freehand`. Questa selezione fa assumere a Yudit l'aspetto mostrato nella figura 11. Nell'area sottostante allo spazio di editing troviamo al centro l'area in cui disegnare un carattere col mouse, a sinistra una serie di "alfabeti" orientali in cui cercare una corrispondenza col carattere disegnato e a destra la lista dei caratteri possibili. La figura 12 mostra la lista dei risultati relativi alla lista "roman" trovati per lo scarabocchio a forma di F. Ovviamente fornisce altri risultati per gli altri "alfabeti" elencati. Basta selezionare il carattere cercato per vederlo comparire nello spazio di editing.

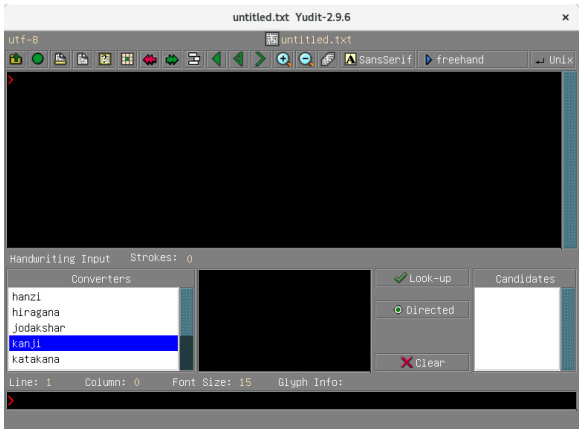


FIGURA 11: Yudit con la mappa `freehand` ci permette di disegnare un carattere per trovarlo in una lista di possibili candidati (cosa utile con lingue come il giapponese).

4 Yudit e LaTeX

Ora che abbiamo tutte le basi d'uso di Yudit, sarà facilissimo scrivere qualunque documento LaTeX, tenendo presente che Yudit è un editor *general purpose* e quindi non offre scorciatoie per inserire comandi quali `\emph` o simili, però possiamo

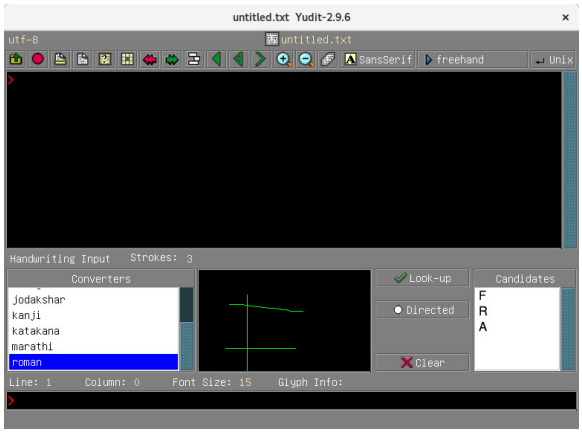


Figura 12: Uno scarabocchio a forma di F viene "riconosciuto" come F, R o A secondo l'alfabeto roman.

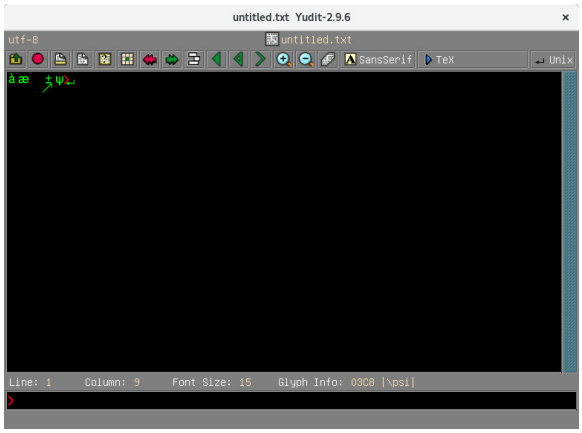


Figura 13: Sequenza di caratteri ottenuti tramite la mappatura TEX.

scrivere molti caratteri assenti sulla tastiera con la sintassi uguale o molto simile a quella di LATEX. Per esempio, la sequenza mostrata nella figura 13, è stata ottenuta selezionando la mappa tastiera TEX e digitando:

```
\‘a␣\ae␣\swarrow␣\pm␣\psi
```

Per un TEXnico sarà immediato scrivere qualunque documento sfruttando la propria conoscenza dei comandi di LATEX.

5 Conclusioni

I linguisti potrebbero incontrare alcune difficoltà nella stesura dei proprî documenti nonostante la sempre maggior potenza e versatilità dei più comuni *word processor* e il sempre miglior supporto multilingua dei moderni sistemi operativi. In certi casi può essere utile avere a disposizione un editor specializzato nell'inserimento di testi multilingua.

Yudit è senz'altro uno dei prodotti di riferimento del campo: permette di rimappare con molta facilità la propria tastiera per supportarne in diverse lingue; permette di inserire molti caratteri mediante comandi di noti linguaggi di markup (SGML) o composizione tipografica (`troff`, TEX); permette la scrittura da destra a sinistra; consente di salvare il testo col fine riga di diversi sistemi operativi; contiene anche un modulo per la scrittura manuale per trovare agevolmente "caratteri" che si sanno disegnare ma non digitare.

Pur non essendo il mio editor preferito né quello di immediato ripiego (nell'ordine, `vi` e Emacs), di sicuro Yudit mi ha tolto spesso da grandi impacci in ambito lavorativo.

Riferimenti bibliografici

«The Unicode Consortium». URL `http://www.unicode.org`.

«Yudit». URL `http://www.yudit.org`.

Korpela, J. K. (2006). *Unicode explained*. O'Reilly, Sebastopol (CA).

▷ Gianluca Pignalberi
`g dot pignalberi at gmail dot com`

Lo scriba A del *Codex Sinaiticus* e il font *Simonides*

Claudio Vincoletto, Massimiliano Dominici

Sommario

L'automazione del tratto calligrafico è uno dei sogni ricorrenti della tipografia digitale. Il percorso qui intrapreso ne rappresenta una delle più recenti incarnazioni, sebbene si configuri anche come uno specchio del passato: il respiro dello scriba ritorna a vivere, palpitando sulla pagina stampata.

Abstract

The calligraphic stroke automation is one of the recurring dreams of digital typography. The path taken here represents one of the most recent incarnations, though it also functions as a mirror of the past: the scribe's breath returns to life, vibrating on the printed page.

1 *Envoi*

Tutto ciò che rimanda al termine *traduzione*, nel suo significato etimologico di "portare oltre, tramandare", può rappresentare lo sfondo necessario sul quale si dipana l'insieme dei propositi che hanno guidato l'indagine di cui si vuole esporre brevemente.

L'essere postumo della scrittura è connaturato alla trasmissione della memoria umana alle nuove generazioni, l'estremo tentativo di fissare la precarietà del presente in una sopravvivenza artificiale che sfida i limiti insuperabili e irreversibili della morte. Una sottile traccia della vita dello scriba permane fisicamente nei segni grafici, e soprattutto nella loro riproducibilità, in grado di porre rimedio anche alla dissoluzione dei supporti materiali più durevoli.

La configurazione semiologica del testo si staglia irrimediabilmente sulla pagina, virtuale o immanente che sia, in cui viene a instaurare un insieme di relazioni cronotopiche. In tale campo di tensione si possono riconoscere alcuni elementi di *persistenza* e altri di *mutamento*.

Ogni tratto, ogni glifo, ogni riga ben composta, manifestano ordine e ripetizione. Senza un codice di riferimento condiviso non è possibile alcuna forma di comunicazione, e per essere assimilato il codice deve perdurare. Diversamente, l'impiego di tali elementi in una situazione più vasta e contingente richiede una sorta di adattabilità, di riconfigurazione, in cui si possono notare lievi variazioni di accomodamento delle regole. Questo è ciò che rimane della scrittura, un fluire costante di oscillazioni tra la competenza del saper scrivere e le trasformazioni in cui si risolve la sua stessa esecuzione.

La nostra ricerca si muove in tale ambito.

2 Il manoscritto

Un anno prima della prematura morte, nel lontano 1761, il naturalista Vitaliano Donati annotava nei suoi diari di aver ammirato, in seguito a una visita presso il monastero di Santa Caterina sul monte Sinai, «una Bibbia in membrane bellissime, assai grandi, sottili, e quadre, scritta in carattere rotondo e bellissimo». Già si rendeva conto di avere fra le mani un codice molto antico, almeno anteriore al settimo secolo.

Fu solo però nel 1844 che il filologo Constantine Tischendorf, assolutamente convinto dell'inestimabile valore del testo, riuscì a convincere i monaci del monastero affinché gli donassero alcune pagine del codice, che furono decifrate e successivamente pubblicate a Lipsia Tischendorf (1862). Qualche anno dopo, grazie all'intercessione dello zar Alessandro II di Russia, il volume fu trasportato alla Biblioteca Imperale di San Pietroburgo. Solo nel 1933 trovò la sistemazione attuale presso la British Library.

Grazie a un intenso lavoro di digitalizzazione, il manoscritto è oggi consultabile online, arricchito di numerosi apparati la cui analisi accurata è stata fondamentale per la realizzazione del presente lavoro Codex Sinaiticus Project (2015).

Il *Codex Sinaiticus* risale probabilmente al quarto secolo e rappresenta uno dei più antichi codici in pergamena, si presume stilato ad Alessandria o Cesarea. Il testo presenta uno dei repertori più autorevoli e completi del Vecchio e del Nuovo Testamento, ma può anche considerarsi tra gli onciali greci biblici quello di più notevole pregio. È possibile distinguere la mano di tre scribi, malgrado la scrittura a calamo risulti essere piuttosto omogenea Knight (2009); Parker (2010).

Il *Simonides* prende spunto dallo scriba A, riproducendo nel dettaglio un carattere onciale biblico, unanimemente riconosciuto come elegante e prestigioso.

3 La lezione del *Punk Nova*

Con l'introduzione della tipografia digitale si pensò da subito che il computer fosse in grado di automatizzare una serie di processi in grado di simulare la complessità della calligrafia. Tuttavia le strategie messe in opera furono abbastanza limitate e ancora oggi molti strumenti vengono sottovalutati se non ignorati dalle grandi industrie del settore.

Tra i primi tentativi di realizzare un font i cui parametri sono progettati per dare vita a una compilazione variegata e dalle infinite possibilità di tra-

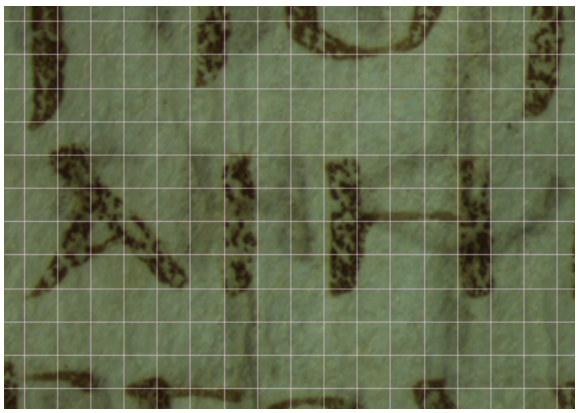


Figura 1: Particolare del *Codex Sinaiticus*: una griglia rivela il *modulo*.

sformazione figura in primo luogo il *Punk Font* di Donald Knuth Knuth (1988). Il disegno è originale (1985) e si ispira alle lezioni di Gerard and Marjan Unger. La natura stessa di METAFONT si rivela congeniale al progetto, proprio per la possibilità di organizzare i dati in modo flessibile.

Nel 2008 Taco Hoekwater e Hans Hagen, con l'idea di impiegare la tecnologia dei font `.otf` nel sistema ConTEXt, si cimentano con il tentativo di portare il *Punk Font* a tale formato, operazione tutt'altro che banale. Nasce l'esperienza del *Punk Nova* Hoekwater e Hagen (2008), un font corredato di una serie di glifi richiamati casualmente durante la compilazione del documento. In questo caso METAPOST viene usato in sostituzione di METAFONT, per la possibilità di generare i contorni di ogni glifo secondo le specifiche PostScript.

Uno script Python invoca FontForge che riassembla tutte le varianti generate per ogni glifo e predispone le funzioni che rendono casuale il principio di scelta nel corso della permutazione.

4 Dalla copia al modello

Al fine di ottenere delle varianti compatibili con l'originale è necessario immaginare un punto di partenza, una sorta di disegno ideale da cui derivano tutti gli altri. L'esame calligrafico del *Sinaiticus* non è semplice, i materiali fotografici utili allo scopo e forniti in rete non sono molti, e in questa fase non si è pensato di accedere direttamente al manoscritto. Con le immagini disponibili è comunque possibile individuare quel che si può definire un *modulo* di riferimento (figura 1).

Il tratteggio, per quanto irregolare, presenta dei punti focali facilmente individuabili nella sovrapposizione fotografica di un reticolato. Tale stratagemma è impiegato più frequentemente nell'analisi dei caratteri tipografici piuttosto che in paleografia. In ogni caso l'espediente si è rivelato efficace e sufficiente all'isolamento di un modulo, debitamente riportato su carta millimetrata.

Questa base, convertita in un sorgente per METAFONT, ha rappresentato lo stadio primordiale del progetto. Successivamente si è calcolato l'ambito in cui poteva variare ogni punto del tratto, circoscrivendo un campo di aleatorietà a sua volta ricondotto a coordinate spaziali.

La variabilità dei glifi è uno degli aspetti salienti della scrittura ed è stata implementata sfruttando alcune caratteristiche del programma di compilazione. L'introduzione di un'indeterminazione controllata è possibile con METAFONT in almeno due modalità: con `uniformdeviate` t viene fornito un numero u che è distribuito casualmente tra 0 e t; con `normaldeviate` invece ne risulta un numero casuale x tra zero e uno.

La strategia impiegata per il *Simonides* è la stessa usata da Knuth per il font *Punk* e qui se ne farà breve cenno. Si immagini uno spazio piano in cui ogni punto sia determinato dalle sue coordinate (x, y). Con il parametro `dev`, derivante dalla dimensione del glifo, si delimita un ambito di variazione che andrà a incidere sulla coordinata stessa.

```
hair_space# = 1/9 size#;
dev#:=1/12hair_space#;
```

Ridefinendo il comando `beginchar`, che stabilisce la larghezza, l'altezza e la profondità della scatola contenente il glifo, vengono inseriti due ulteriori valori numerici, uno per la componente orizzontale (h) l'altro per la verticale (v), i quali vengono ricalcolati in funzione della deviazione predisposta da `dev`.

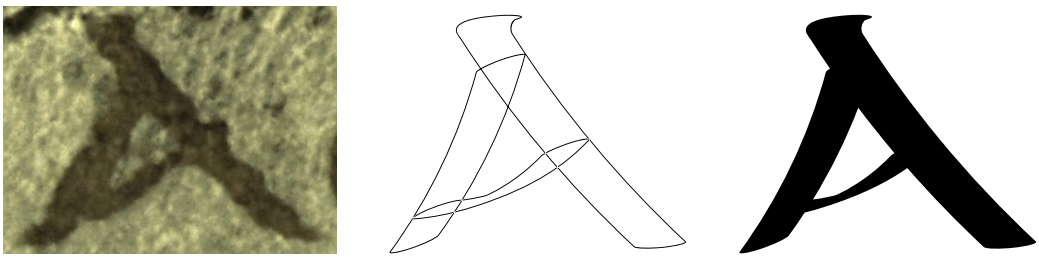
```
hdev := h * dev ;
vdev := v * dev ;
```

Infine una *macro* modifica il valore delle coordinate di un punto `z`, inserendo l'indice di casualità desiderata.

```
vardef pp expr z =
 z + (hdev * normaldeviate, vdev * normaldeviate)
 enddef;
```

Per quel che riguarda il disegno dell'alfabeto non ci si discosta molto da quello del greco classico, sebbene dotato di pochissimi accenti e allestito in almeno due corpi: uno più minuto per le abbreviazioni. È stata aggiunta una batteria di varianti ulteriori, per ampliare il criterio di scelta del compositore. Quest'ultima si è rivelata necessaria, in quanto lo scarto introdotto dalla compilazione casuale non riesce a colmare tutti i casi presentati dal manoscritto. Un timido tentativo che non esaurisce certo la flessibilità della scrittura manuale.

Alcuni accorgimenti sono stati adottati per superare i limiti della penna fissa di cui è dotato METAFONT: penne di diversa angolatura, effetti di curvatura dei tratti, *macro* in grado di simulare le grazie appena accennate che contraddistinguono lo scritto del *Codex Sinaiticus*.



(a) Immagine fotografica. (b) Contorno dei tratti. (c) Riempimento.

Figura 2: Composizione del glifo *alpha*, dal *Codex Sinaiticus* al *Simonides*.

5 Dal modello alla copia

Benché METAFONT sia uno strumento straordinario per il disegno dei contorni, esso mostra tutti i segni del tempo quando arriva il momento di assemblare in un unico font i singoli glifi così prodotti. METAFONT precede di molti anni l'introduzione delle diverse tecnologie che costituiscono l'attuale stato dell'arte in materia di caratteri digitali. Laddove METAFONT poteva gestire al massimo 256 glifi in un font, Unicode permette oggi di catalogare virtualmente ogni glifo, appartenente a un qualsiasi alfabeto, all'interno di un unico spazio di codifica, eliminando di fatto il ricorso a una pletora di codifiche, spesso *ad hoc*, che rendevano difficoltoso l'uso di un font su diverse piattaforme e in programmi diversi. Type1 e TrueType rappresentano i due standard alternativi per il disegno dei contorni. Il formato OpenType, combinando insieme Unicode, Type1/TrueType e una serie di strumenti che semplificano l'accesso a operazioni complesse quali sostituzioni di varianti, ecc., è ormai diventato la tecnologia di riferimento per i caratteri digitali.

Il problema di ottenere, a partire da sorgenti METAFONT, dei font in un formato corrispondente allo standard *de facto* del momento (ieri Type1, oggi OpenType) è stato già più volte affrontato e risolto in passato.[1] Il metodo più comune, oggi, consiste nel compilare i singoli glifi tramite METAPOST e poi usare FontForge come libreria per riassemblare i glifi in un unico font. Questa operazione è automatizzata tramite uno script Python che si incarica anche di aggiungere tutte le eventuali *feature* OpenType (nel nostro caso l'accesso a varianti casuali).

Un esempio di questo approccio è rappresentato da `mf2outline`,[2] un sistema messo a punto da Linus Romer inizialmente per la realizzazione di FetaMont. Il sistema consta essenzialmente di due parti: un insieme di macro che estendono METAFONT/METAPOST e permettono, per esempio, di usare codici Unicode per identificare i singoli glifi, di definire classi di crenatura o di specificare *lookup* per varianti randomizzate;[3] e uno script Python che gestisce l'intera compilazione.

Lo script è ben fatto, ma pensato per un font molto particolare (è un'estensione del font usato per il logo di METAFONT/METAPOST) e con un numero limitato di varianti. Nel caso del *Simonides* (che pure è un font molto particolare) si possono avere fino a 38 varianti di un singolo carattere. Si pongono quindi due problemi, uno di ordine concettuale e uno di ordine pratico.

Il primo problema concerne il modo di codificare queste varianti. La regola vorrebbe che alle varianti di un glifo (maiuscoletto, varianti stilistiche, varianti casuali) non venga assegnato un codice Unicode e che esse siano selezionabili solo attraverso la rispettiva *feature* OpenType. Questo non è possibile, nel nostro caso, perché lo script sfrutta esplicitamente il valore del codepoint Unicode durante la compilazione dei singoli glifi. Si è deciso, quindi, di ripiegare sulla soluzione più vicina in termini di accettabilità: alle varianti viene assegnato un codepoint che si trova all'interno della *Private User Area*. Ciò garantisce che non ci siano collisioni con caratteri che hanno una codifica Unicode, anche se non si possono escludere ambiguità in caso di sostituzione del font (font diversi possono usare la *PUA* per codificare glifi con diverso significato). A parte situazioni particolari, tuttavia, questo modo di procedere non dovrebbe creare grattacapi.

Il secondo problema consiste nell'evitare di dover riscrivere 38 volte, una per ogni variante, il codice di ciascun glifo. Una semplice macro, riportata nella figura 3, è sufficiente a questo proposito. La macro, `char_with_variants`, genera un glifo (ad esempio `alpha`) e tutte le sue varianti ordinarie. La macro richiede cinque argomenti, di cui l'ultimo (`body`) è il codice che disegna il glifo. I primi quattro argomenti rappresentano rispettivamente il codice Unicode del carattere, dato in forma di numero esadecimale; il codice, nella *PUA*, in cui

1. Alcuni tra i programmi sviluppati a questo scopo: `m2ps` (Yanai e Berry, 1990); `mftrace` (Nienhuys, 2017); MetaFog (Kinch, 1995); METATYPE1 (Jackowski *et al.*, 2003); MFLua (Scarso, 2011, 2012). Utili informazioni si trovano anche in Píška (2005).

2. `https://github.com/linusromer/mf2outline`; se ne trova una breve descrizione in Romer (2014).

3. Il metodo usato per specificare il *lookup* `randomize` può essere facilmente esteso anche ad altri *lookup* come è stato verificato durante i nostri esperimenti con il *Simonides*, anche se alla fine `randomize` è stata l'unica *feature* OpenType che abbiamo implementato.

```
% Variants
string codepoint; % Full unicode codepoint.
string variant;   % Encodes the symbolic name of each single variant in a block ("alpha.1", "alpha.2", ecc.).

def char_with_variants(expr code, start, num_var, name) (text body) =
  % draws a single character and its variants
  % code = unicode codepoint
  % start = position of the first variant; should be in the PUA
  % num_var = number of variants
  % name = conventional name
  % body = glyph code
  addrandvariant(code,code); % It is convenient to add here infos for the ‘randomize’ lookup
  variant:= "Unicode " & name;
  codepoint:= code;
  body;
  for i=1 step 1 until num_var: % Usually we have 32 variants
    codepoint:= hexadecimal(start+i-1);
    variant:= name & "." & decimal(i);
    body;
    addrandvariant(code,codepoint); % Add variant to ‘randomize’ lookup
  endfor;
enddef;
```

Figura 3: Il codice per generare automaticamente l'intero insieme delle varianti di un glifo.

cominciare a codificare le varianti casuali relative al glifo, sotto forma di numero decimale; il numero delle varianti; il nome del glifo. Purtroppo non è possibile esprimere il "punto di partenza" delle varianti casuali sotto forma di numero esadecimale, come è consueto per Unicode, a causa di una limitazione del comando `hexadecimal`, usato nella conversione da valore esadecimale a valore decimale, il quale, anche nel caso sia stata passata all'eseguibile `mpost` un'opzione per l'uso del calcolo in virgola mobile, restituisce un errore per valori che eccedono i limiti classici del programma METAFONT. La macro esegue una prima volta il codice contenuto in `body`, gli assegna il relativo codepoint e lo aggiunge a una tabella di sostituzioni (`addrandvariant(code,code)`). Poi ripete l'operazione, all'interno di un ciclo, tante volte quante sono le varianti volute, posizionando i glifi così ottenuti nella *PUA*, in slot adiacenti a partire dal valore `start` indicato, e aggiunge ciascuna variante alla tabella di sostituzioni di cui sopra. Per il carattere `alpha` il codice assume l'aspetto seguente:

```
char_with_variants("03B1",57344,32,"alpha",
beginsimonideschar(codepoint,6,cap_height#,0,1,2);
  variant;
  z1=pp(2hair_space,bot h);
  z2=pp(lft w-1/6hair_space,0);
  z0.5=z1+2serif_a;
  stroke1= z0.5...z1{x2-x1,1.3(y2-y1)}..z2;
  z3=point 1.1 of stroke1;
  z4=pp(rt 1/6hair_space,0);
  stroke2= z3{x4-x3,1.8(y4-y3)}..z4;
  z5=point .8 of stroke2;
  z6=point 1.5 of stroke1;
  stroke3= z5{dir pa}..z6;
  draw stroke1;
  pickup simonides_pen_tilted_b;
  draw subpath(0.02,length stroke2) of stroke2;
  draw subpath(0,.98) of stroke3;
  penlabels(0.5,1,2,3,4,5,6); endchar;
);
```

6 Sfogliare le pagine del *Codex Sinaiticus*

Non è facile testare un font come il *Simonides* senza collocarlo in un contesto adeguato, che consenta al tempo stesso di reggere il confronto con il manoscritto originale da cui è estrapolato. Si è scelta a questo scopo una pagina del Vangelo di Giovanni, che presenta delle caratteristiche alquanto peculiari.

Innanzitutto si deve tener presente la natura del *Simonides*, che è un font OpenType dotato di molte varianti. Per gestire questa complessità si è fatto ricorso a LuaLaTeX come motore di composizione, il quale si è dimostrato perfettamente in grado di gestire tale compito.

Cruciale, in questo senso, si rivela l'impiego del pacchetto fontspec, poiché permette un controllo accurato delle specifiche relative ai font OpenType. Consente altresì di impostare il parametro `rand` (Random), essenziale per la distribuzione automatica delle varianti di ogni glifo.

```
\usepackage{fontspec}
\setmainfont[Renderer=Full,
             Letters=Random,
             WordSpace=0,
             Opacity=0.9,
             Script=Greek]{simonides.otf}
```

Per quel che concerne la configurazione della pagina, non si riscontrano grandi difficoltà, pertanto la classe article risulta più che sufficiente, se corredata dei pacchetti geometry (per le dimensioni del foglio e del blocco del testo), fancyhdr (per le testatine) e luacolor (per gli effetti di colore).

Le pagine del *Sinaiticus* sono abbastanza ampie (380 mm×345 mm) e il testo viene distribuito prevalentemente su quattro colonne e di rado su due, ad esempio per i libri poetici del Vecchio Testamento.

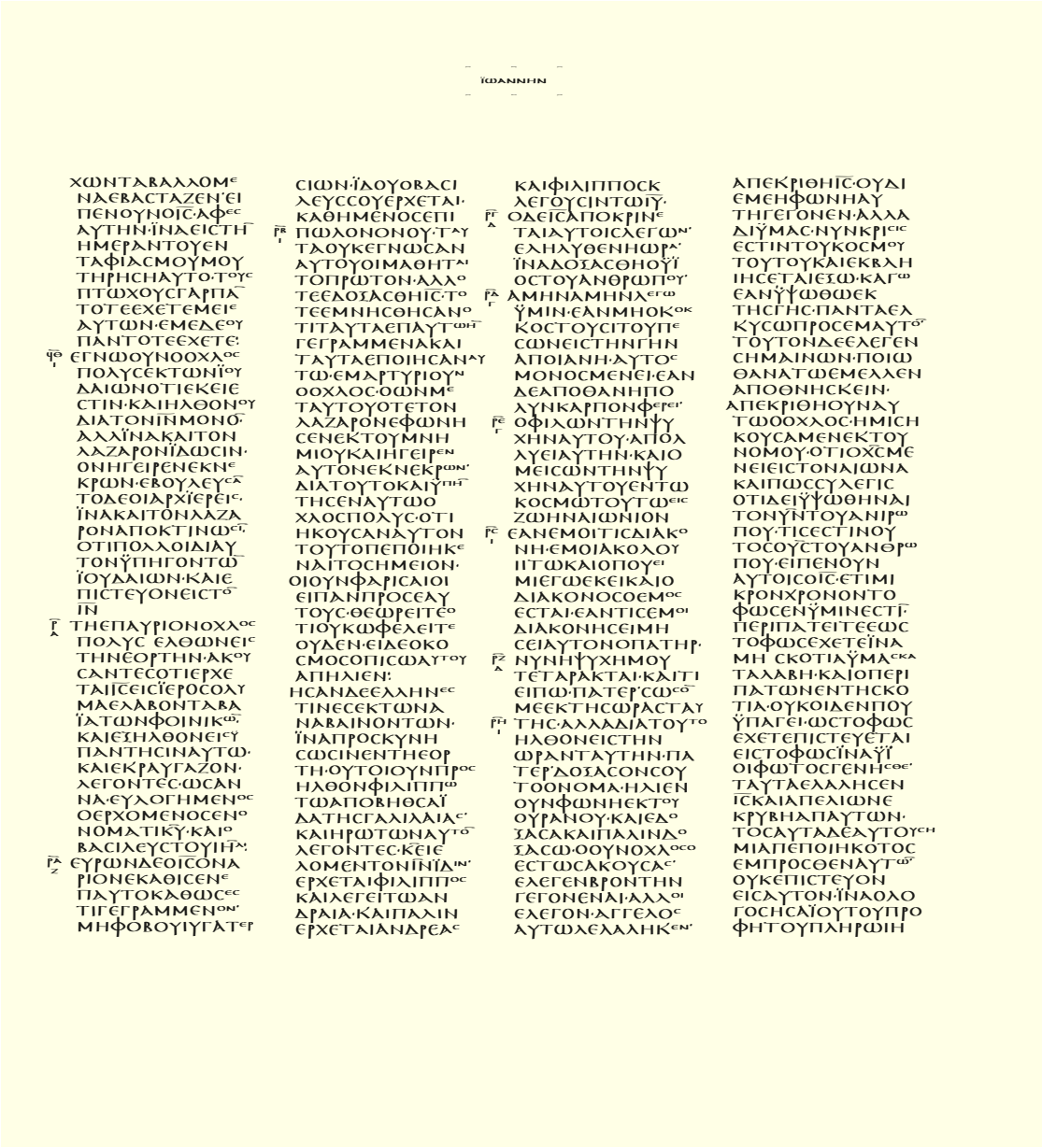


Figura 4: Riproduzione di *Codex Sinaiticus*, BL 255: Giovanni 12, 6 sgg.

Per semplificare la costruzione di questa struttura si è ricorso a multicol, che risolve la maggior parte dei problemi. L'uso di questo pacchetto però ha rivelato almeno due limiti per i nostri scopi, la cui soluzione ha richiesto opportuni accorgimenti.

Ad esempio, al fine di ottenere un'armonica distribuzione delle righe che devono presentarsi equidistanti, si è pensato di aggiungere i seguenti comandi.

```
\baselineskip=1\baselineskip plus 1fill
\lineskip=0pt plus 1fill
\lineskiplimit=-\maxdimen
```

Un'ulteriore complicazione deriva dall'impossibilità del pacchetto di gestire adeguatamente le note a margine, che nel *Sinaiticus* si presentano al fianco sinistro di ogni colonna. Si tratta propriamente della notazione dei versetti biblici, risolta con l'indicazione di un comando in grado di creare una scatola di testo nello spazio voluto.

```
\newcommand{\versiculus}[1]{%
  \makebox[0pt][r]{%
    \parbox[t][][t]{10mm}{%
      \centering\baselineskip=2.6mm #1
    }
  }
}
```

Per concludere, è necessario considerare altri aspetti del testo. La versione riprodotta è quella fornita in formato XML dal sito ufficiale del progetto, nato per valorizzare lo studio e la divulgazione del manoscritto. Come già accennato, nella composizione digitale si è dovuto ricorrere a caratteri più minuti, che figurano abbastanza spesso al termine di ogni riga, probabilmente usati dallo scriba per giustificare quanto più possibile

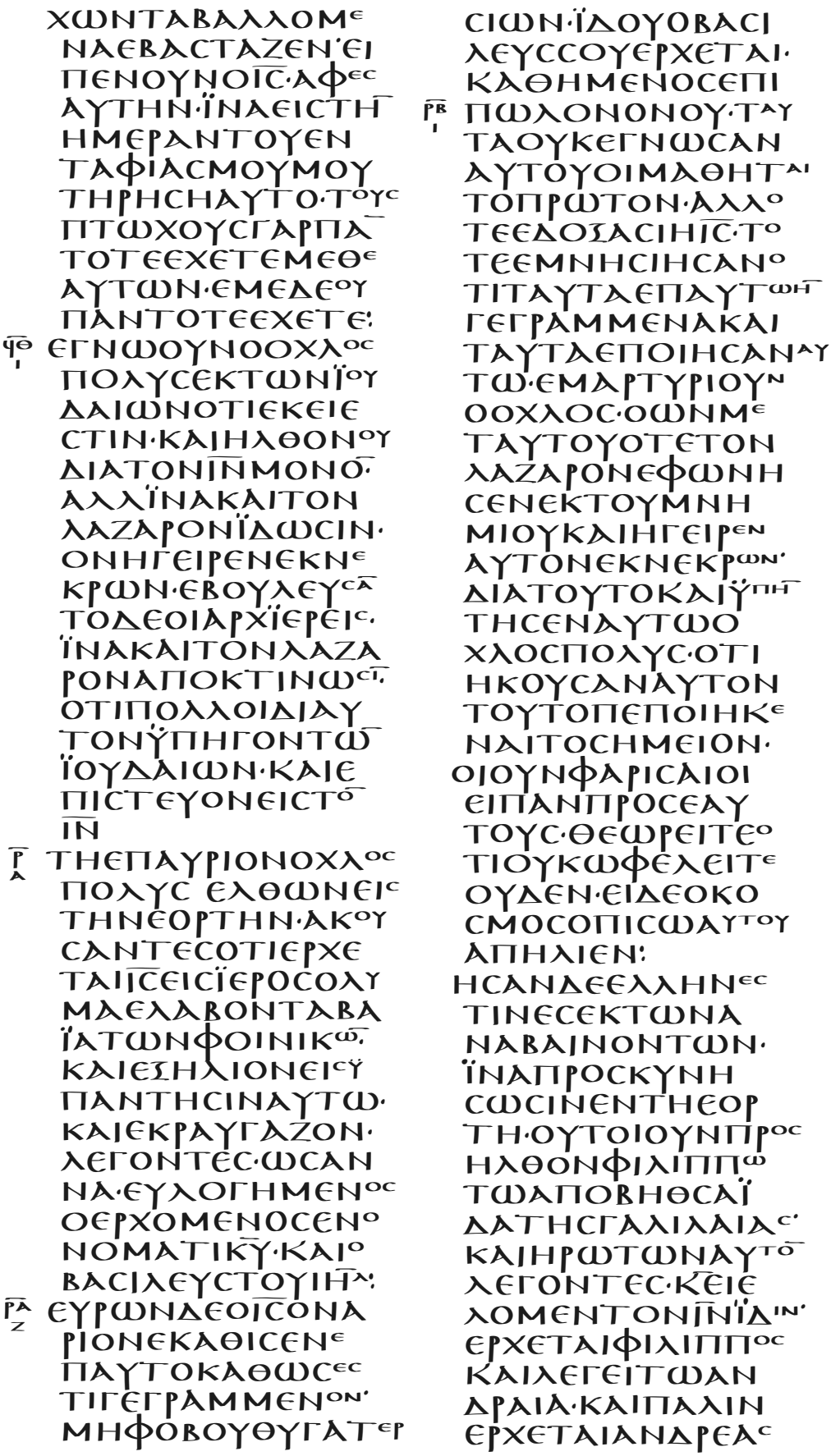


Figura 5: Riproduzione di *Codex Sinaiticus*, BL 255: Giovanni 12, 6 sgg. Dettaglio delle prime due colonne.

il margine della colonna. Il font *Simonides* è dotato di questi particolari caratteri, così come dei segni diacritici e di interpunzione che si possono incontrare nell'originale.

In definitiva, esistono sul mercato diversi font che riproducono l'alfabeto di un onciale greco, e altri comunque validi vengono forniti gratuitamente. Il confronto con il *Simonides* può essere solo parziale, perché diverse sono le modalità e le finalità di utilizzo. L'unico paragone pertinente di cui si ha conoscenza, relativo quindi a un carattere tipografico ispirato alla calligrafia del *Sinaiticus*, è quello in metallo utilizzato per l'edizione in facsimile del 1862 e curata dallo stesso Tischendorf. Questo si presenta molto bello, sufficientemente leggibile e al tempo stesso fedele: un contributo ancora essenziale per lo studioso d'oggi. Il *Simonides* ne vuole essere un ideale perfezionamento, come alternativa al facsimile fotografico Tischendorf (1862).

7 Interpretazioni

L'idea di un *copista digitale* non è nuova, ma qui trova una delle sue più effettive reincarnazioni.

Alcuni aspetti vanno però segnalati, che ripropongono il dilemma dell'interazione uomo-macchina. L'aleatorietà con cui vengono distribuiti i glifi sulla riga richiama al lettore il moto vivace della scrittura calligrafica. Ma il calcolatore obbedisce alle regole progettate, privo quindi della facoltà di effettuare scelte contestuali.

L'amanuense cerca di ritrovare, nel disordine del mutamento repentino, una regola condivisa. Nel tentativo di riprodurre il modello mentale della scrittura, introduce suo malgrado dei sottili e spesso involontari cambiamenti: l'esecuzione tradisce il riferimento ideale. La macchina, inevitabilmente rigida, viene forzata a inserire nello schema elementi di rottura della rigidità, ma è guidata dal caso e non da principi di adattamento: la variazione è irrazionale e non guidata dal gusto artistico.

Tuttavia la simulazione inganna anche l'occhio più esperto, e rende possibile la meraviglia del gioco di prestigio.

8 Ouverture

Constantinos Simonides (1820–1890) fu un paleografo colto e profondo conoscitore di antichi manoscritti, abilissimo e impareggiabile calligrafo, nonché uno dei più grandi falsari di tutti i tempi Schaper (2013). Sebbene smentito da eminenti studiosi, Tischendorf e Bradshaw soprattutto, egli si attribuì la paternità del *Codex Sinaiticus*, come frutto di puro apprendistato giovanile.

A lui si è pensato di dedicare il nostro lavoro.

Riferimenti bibliografici

Codex Sinaiticus Project (2015). «Codex Sinaiticus». URL http://www.codexsinaiticus.org/en/.

Hoekwater, T. e Hagen, H. (2008). «Punk from Metafont to MetaPost». *MAPS*, (37), pp. 55–58. URL http://www.ntg.nl/maps/37/10.pdf.

Jackowski, B., Nowacki, J. M. e Strzelczyk, P. (2003). «Programming PostScript Type 1 Fonts Using METATYPE1: Auditing, Enhancing, Creating». *TUGboat*, **24** (3), pp. 575–581. URL https://www.tug.org/TUGboat/tb24-3/jackowski.pdf.

Kinch, R. J. (1995). «MetaFog: Converting METAFONT Shapes to Contours». *TUGboat*, **16** (3), pp. 233–243. URL https://www.tug.org/TUGboat/tb16-3/tb48kinc.pdf.

Knight, S. (1998,2009). *Historical Scripts: From Classical Times to the Renaissance*. Oak Knoll Press.

Knuth, D. (1988). «A Punk Meta-Font». *TUGboat*, **9** (2), pp. 152–168. URL http://tug.org/TUGboat/Articles/tb09-2/tb21knut.pdf.

Nienhuys, H.-W. (2017). «mftrace». URL http://lilypond.org/mftrace/.

Parker, D. (2010). *Codex Sinaiticus: The Story of the World's Oldest Bible*. British Library.

Píška, K. (2005). «Converting METAFONT sources to outline fonts using METAPOST». *TUGboat*, **26** (2), pp. 158–164. URL https://www.tug.org/TUGboat/tb26-2/piska.pdf.

Romer, L. (2014). «Fetamont: An extended logo typeface». *TUGboat*, **35** (1), pp. 17–21. URL https://www.tug.org/TUGboat/tb35-1/tb109romer.pdf.

Scarso, L. (2011). «Mflua». *TUGboat*, **32** (2), pp. 177–184. URL https://www.tug.org/TUGboat/tb32-2/tb101scarso.pdf.

— (2012). «MFlua: Instrumentation of METAFONT with lua». *ArsTEXnica*, (14), pp. 72–81. URL https://www.guitex.org/home/images/ArsTeXnica/AT014/mflua.pdf.

Schaper, R. (2013). *L'odissea del falsario. Storia avventurosa di Costantino Simonidis*. Bononia University Press.

Tischendorf, K. (1862). *Bibliorum Codex Sinaiticus Petropolitanus*. Giesecke & Devrient. URL http://www.sbible.ru/sinaj2.htm.

Yanai, S. e Berry, D. M. (1990). «Environment for Translating METAFONT to POSTSCRIPT». *TUGboat*, **11** (4), pp. 525–541. URL https://www.tug.org/TUGboat/tb11-4/tb30yanai.pdf.

▷ Claudio Vincoletto
Torino
claudio dot vincoletto at gmail dot com

▷ Massimiliano Dominici
Pisa
mlgdominici at gmail dot com

Requirements for a Music Engraving Program: a Composer's Point of View

Jean-Michel Hufflen

Abstract

It is well-known that some typesetting systems are *interactive*, that is, WYSIWYG, whereas some—WYSIWYM—work like *compilers* and process input files written using an *input language*. Interactive systems provide interesting features, whereas other qualities are implemented by WYSIWYM systems. We can observe the same points about music engraving programs. In this article, we summarise the properties of interest during the music composition process and review some music engraving programs from this point of view.

Keywords music engraving, scores, music typography, music composition, versioning, managing musical instruments, building MIDI files, Finale, LilyPond, MusiXTEX, NoteEdit, MuseScore.

Sommario

È ben noto che alcuni programmi di elaborazione di testo sono *interattivi*, cioè WYSIWYG, mentre alcuni — WYSIWYM — funzionano come *compilatori* ed elaborano i file di ingresso usando un particolare *linguaggio*. I programmi interattivi presentano funzionalità interessanti, mentre altre funzionalità sono fornite dai sistemi WYSIWYM. Possiamo osservare le stesse caratteristiche nei programmi per scrivere musica. In questo articolo presentiamo le funzionalità che interessano nel caso della composizione tipografica della musica e commentiamo alcuni programmi da questo punto di vista.

Parole chiave Incisione della musica, spartiti, tipografia musicale, composizione musicale, controllo delle versioni, gestione degli strumenti musicali, creazione di file MIDI, Finale, LilyPond, MusiXTEX, NoteEdit MuseScore.

1 Introduction

Among typesetting systems, the clear distinction between WYSIWYG[1] and WYSIWYM[2] systems is well-known. In the first case, such programs—an example being Adobe InDesign—are *interactive*, that is, the formatted text is directly displayed on screen, and updated as soon as end-users enter new characters or activate some menu operations. In the second case, a source file written using an *input language* is processed—this step may be viewed as a *compilation*—and result in the complete formatted text. Of course, LaTeX and other programs built out of TeX belong to this second category.

The same distinction exists for music engraving software, that is, programs drawing music scores. Some programs—e.g., Finale, MuseScore, NoteEdit—are interactive: a score is built step by step by means of graphical interface, even though notes and other musical signs are progressively written by hand on a sheet of paper. Some—e.g., LilyPond, MusiXTEX (Taupin *et al.*, 2002)—are clearly related to a WYSIWYM approach.

The advantages and drawbacks of these two approaches have already been described in many articles about word processing. However, writing music is a different task than writing texts, and some arguments relevant about written documents do not apply to music scores. Symmetrically, a music composer does not have the same requirements than a book writer. Since I personally compose music during my spare time, I judge interesting to give my point of view. I experienced some music software, not all of them, so my study is not exhaustive. However, I tried to express what I like or dislike in some programs, as precisely as possible.

In the first section I explain what a composer expects from a music program. I begin this section by summarising the requirements for a typesetting system in order to show how different is musical composition. Then I briefly describe the programs I practised in Section 3. I report my experience about music engraving programs in Section 4. Readers of this article are only required basic knowledge of music. Readers interested in precise definitions of musical terms can consult Jacobs (1988).

2 Requirements

In the following, I consider word processor basic tasks, and do not deal with specialised features such as tables, mathematical formulas, pictures of chemical molecules, etc. A word processor should allow its users to get high-quality print outputs. It should be able to implement basic graphical effects, that is, the use of boldface types, italicised characters, etc. It should reflect a document *structure*, from a graphical point of view, by means of hierarchical headings using different character

1. What You See Is What You Get.
2. What You See Is What You Mean.

sizes. It should also implement *cross references* among some subparts of such a structure. Last but not least, a good word processor should allow different parts of a large document to be written by several end-users, and the integration of these parts should be easy[3].

As mentioned above, many articles about the advantages and drawbacks of WYSIWYG and WYSIWYM systems already exist, so I just sum up some important points. Current interactive systems have been improved in comparison with programs used 40 years ago, but they are still limited by their interactivity: when an end-user types new characters, such a program must respond quickly and display a reformatted version of the current paragraph. On the contrary, a WYSIWYM system can examine many solutions before choosing the best way to split a paragraph into successive lines. So does LaTeX: it explores some solutions with respect to criteria summarised into a *badness* measurement, the chosen solution being minimal badness. An analogous *modus operandi* is applied for splitting a chapter into successive pages. An interactive system cannot explore many possible solutions and reach such efficiency.

More generally, WYSIWYM systems allow *style* and *content* to be clearly separated, so users can mainly focus on content when they are writing a document; considerations about style can be examined separately. Interactive systems may be preferred for short documents, e.g., administrative letters, but for large documents, consisting of many parts, such a separation makes it easier to merge these parts or to produce a new version according to another style. A very simple example: building a two-column version of a document previously typeset using a one-column format is easier with a WYSIWYM system than a WYSIWYG one.

Now let us go to music composition (first importanto point). Roughly speaking, there are two ways to get a score for a new musical piece: putting down all the notes and musical signs composing it, or deriving it from piano keyboard or from synthesised music like a MIDI[4] file. The second way outputs scores not ready to use. Such scores must be reworked in order to simplify them: they actually reflect *one* performance of a piece, *too much* exactly. In order words, it does not give the canonical way to express how to play it. A simple example: all of the notes of a *glissando* are explicitly put down onto the resulting score, whereas the standard specification of this effect consists of giving only the starting and ending notes, joined by a line. Like WYSIWYG word processors, music programs deriving scores from synthesised music have been improved compared to the first versions, but some limitations remain. As a consequence, a composer must always deal with scores, either because these scores are written from scratch, or because they are derived but should be reworked.

As a second important point, I think that getting a satisfactory version after improving or changing intermediate versions is a process longer for music scores than written documents. That is true for our personal production, and other composers I asked for this question confirmed that. The goal of such a process is to get the best version but, concerning music, there are some additional points. First, some notes may be misplaced, as if they are mistakes within a musical dictation. If you are able to compose music, you do only a few of such mistakes, but 'a few' does not mean 'nothing'; in fact, they are comparable to typing mistakes within written documents. In addition, as I explained in Hufflen (2017), the use of *accidentals* (♭, ♯, ♮, ...) is error-prone[5]. A synthesised version of a score can allow a composer to detect such mistakes which will have to be fixed. A second point is related to musical instruments: even though a composer knows how to use them, it is difficult—if not impossible—to master *all* of the technical features of *all* of the instruments[6]. In other words, some extracts may be impossible to play by the instrument planned for that[7]. Either such an extract may be suitably arranged[8], or given to another instrument. Similarly, if some instruments are unavailable when a musical piece is to be played, a solution can be to replace them by other instruments; this can yield changes in scores.

3 Some music software

First, let us consider the music programs built out of TeX, briefly described in Goossens *et al.* (2009, Ch. 9), with some examples. In the late 1980s, an early attempt for typesetting musical scores by TeX extensions was MuTeX (Schofer e Steinbach, 1987), which influenced MusicTeX (Taupin, 1992), definitively replaced by MusiXTeX in 1995. This program (Taupin *et al.*, 2002) has been maintained since the accidental death of its main creator, Daniel Taupin, in August 2003, but it seems that only minor development has been done. This robust program, still in use, is usable with *Plain TeX* or as a LaTeX 2ε

3. This point is important for industrially produced documents, but marginal about music pieces.

4. Musical Instrument Digital Interface.

5. Besides, musicologists often have doubts about their interpretation.

6. Many features are described in good orchestration manuals, e.g. Forsyth (1982) or Rimsky-Korsakov (1964). But they cannot be exhaustive. Besides, old manuals did not incorporate recent progresses and new features.

7. For example, the trombone can perform *glissandi* between many pairs of notes, but this effect is not always possible for *every* pair of notes.

8. ... or playable with a special trick: as an example, Stravinsky (1921, 2 bars after Mark 92, p. 79) uses a note too low for a clarinet, but makes it precise that a piece is to be inserted into the bell.

```
\version "2.18.0"

\score {
 \new Staff {
  \clef "treble" \time 4/4
  \accidentalStyle Score.dodecaphonic
  r8 d'8 bes'4~ bes'8[ d'8 bes'8 g'8] |
  c''4. a'8 a'8[ ees'8] ees'4~ |
  ees'16 ges'16 e'16 f'16 b'2 r4 |
 }
 \layout {}
}
```

Figure 1: Example using LilyPond.

package (musixtex). It has been designed to get high-quality print outputs concerning the layout of musical scores, as TEX does for texts. It does not aim to play music. It does not aim to be the input of a program playing music, either. Moreover, it has been reported as difficult to handle. Besides, MusiXTEX's manual says (Taupin *et al.*, 2002, § 2.2.1):

> *[. . .] If this sounds complicated, remember that TEX was designed to typeset text and not music.*

MusiXTEX's input language is very low-level; much placement is up to end-users, e.g., for the notes and corresponding accidentals of a chord. That is why some *preprocessors* have been developed in order to generate source files for MusiXTEX. They are text-based applications, using higher-level input languages and describing the contents of a score without reference to its layout. Historically, the first preprocessor is MPP[9] (Goossens *et al.*, 1997, § 7.4); this project being abandoned for several years. The second is PMX[10]. Scores are specified in a concise way, close to the horizontal and vertical arrangement of an 'actual' score, without reference to formatting; see Goossens *et al.* (2009, § 9.5) for an introduction and Simons (2004) for a complete description. The third is *M-Tx*[11]; see Goossens *et al.* (2009, § 9.6) or Laurie (2005). *M-Tx* language adds a layer of convenience to the PMX language: for example, instrument parts are input as they are printed—that is, from top to bottom—whereas they are entered last line first—that is, from bottom to top—with PMX. As another music program built out of TEX, let us mention TEX*muse* (Garcia, 2012).

Another WYSIWYM program is given by the GNU[12] LilyPond[13]. This project was started by

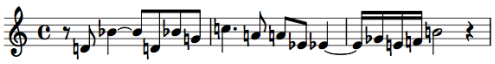


Figure 2: Output generated for Fig. 1.

MPP's authors. LilyPond's syntax evokes TEX since its commands are prefixed by the '\' character. As an example, Fig. 1 gives a specification of a tema of *Lulu-Suite* (Berg, 1935, *Rondo*, Mark 243, p. 3). The result is pictured in Fig. 2. After some global definitions, each note is defined by its pitch—or '`r`' for a rest—followed by a number specifying its rhythm[14]. This software can output music scores as PDF[15] files and generate MIDI files. Other formats are possible, too[16]. An introduction in Italian to this software is Gordini e Liessi (2014).

Interactive music engraving programs allow users to define *systems*, that is, set of staves, and place notes on staves interactively. To do that, a note's pitch and its rhythm have to be selected. Listening to the result is allowed. When a piece is saved, it is analysed and a warning message is emitted if some bars are rhythmically incorrect. Within this category, NoteEdit may also be viewed as a preprocessor since it allows to build input files for MusiXTEX or LilyPond. However, let us mention that files generated for MusiXTEX have to be reworked manually in order to get nice outputs (Hufflen, 2011). I personally experienced Finale and MuseScore. The latter is free, not the former. The services provided are comparable, Finale's ergonomy being better. Another difference is related to *importing* scores written using other conventions, including MIDI files. In both cases, there is no way to *share* common parts. For example, if the first and second violins play the same score, the only way to do that is to use the copy/paste buttons of your software menus.

4 Reporting our experience

Roughly speaking, two reasons may motivate the use of a music engraving program: giving a digital version of existing music pieces, or creating new pieces. In the first case, software built out of TEX can be used, because of the high quality of outputs. As a very good example, a rich collection of Polish traditional songs has been carried out by Odyniec (2016). He used *M-Tx* to ease the specification of lyrics and to get MIDI files to check results. He got very nice scores, but this is not a composers way of working: these songs were composed during a long lapse of time. I have the same

9. MusiXTEX PreProcessor.
10. Preprocessor for MusiXTEX.
11. Music From Text.
12. Recursive acronym: Gnu's Not Unix.
13. This name is a joke related to Rosegarden, an interactive music software.
14. Let us remark that LilyPond's `dodecaphonic` accidental style allows us to specify an accidental before each note, as did by A. Berg in his manuscript.
15. Portable Document Format.
16. LilyPond's older versions generated .tex files, but this backend is no longer supported in recent versions.

feeling by looking into examples accompanying MusiXTEX's documentation: they are either classical examples or very simple pieces or arrangements that have not been reworked and reworked.

As mentioned in § 2, it is *essential* to be able to listen to the result of specifying notes. Of course, listening to synthesised music—e.g., MIDI files—does not replace a performance by real instrumentists but provides some feedback and allows some possible mistakes to be fixed. I personally regret that NoteEdit is no longer developed: it allowed a score to be entered nicely, checked, and reworked if needed and, when the score reached some maturity, NoteEdit output files for MusiXTEX or LilyPond allowed high-quality print, even in case the score had to be reworked.

Putting a new score into action with LilyPond—or a MusiXTEX preprocessor—is possible. Moreover, LilyPond allows variables to be used, in particular, they can share some common parts without information redundancy. Advanced features such as *transposition*[17] are directly available. However, such a way is difficult in practice since input languages are quite far from the standard visual representation of music. This drawback exists about mathematical formulas written in LaTeX: it may be difficult to intuit the look of a complicated formula just by looking at the source text used to produce it. On the contrary, we can often guess the look of fragments in text mode even though they are marked up by commands. Let us go back to input texts for LilyPond: they are difficultly practicable for a musician who would not be also a computer scientist. For that reason, I personally use LilyPond for musical texts that reached an (almost) stable state, but prefer MuseScore for scores developed from scratch.

If we consider the specification styles for output scores, LilyPond and Finale are the best, but the other programs seem to me to be satisfactory. Let us remark those parameters related to *spacing*: horizontal distance between adjacent notes, vertical distance about staves and systems.

5 Conclusion

Musical typography is a fascinating domain and we can admire the results produced by music engraving programs in most cases, concerning classical and popular music, or modern music using 'classical' effects[18]. But that raises difficult problems: reading a score is not a linear process, contrarily to reading a book, as I show in Hufflen (2015). Some historical background can influence the look of a score (Hufflen, 2013). The most part of musical symbols have been included, incompletely (Hufflen, 2014, 2017), into Unicode.

Let us go back to music composition. I tried to give a synthetic point of view of this activity. More technical details related to particular cases can be found in Hufflen (2011, 2012). I think that WYSIWYM systems are interesting, as shown by LilyPond success, but with a WYSIWYG interface. In other words, the *process* of getting scores as PDF files or synthesised music as MIDI files does not need to be interactive, but entering data should, unless a nicer input language is proposed.

17. The specification of the same musical fragment, but at a different pitch. See Jacobs (1988) for more details.

18. I recommend the reading of Vogt *et al.* (1982) to people interested in graphical effects used within contemporary music's scores. There is still a lot to do... unless considering only graphical formats.

Acknowledgements

I thank Claudio Beccari, who has proof-read this article and for his Italian translations of the abstract and keywords.

References

Berg, A. (1935). *Lulu-Suite*, volume 12 674. Universal Edition.

Finale (2017). *Music Notation Software*. http://www.finalemusic.com.

Forsyth, C. (1982). *Orchestration*. Dover Publications, Inc., New-York.

Garcia, F. (2012). «TEX and music: an update on TEX*muse*». *TUGboat*, **33** (2), pp. 158–164.

Goossens, M., Rahtz, S. e Mittelbach, F. (1997). *The LaTeX Graphics Companion. Illustrating Documents with TEX and PostScript*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Goossens, M., Mittelbach, F., Rahtz, S., Roegel, D. B. e Voss, H. (2009). *The LaTeX Graphics Companion*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2ª edizione.

Gordini, T. e Liessi, D. (2014). «LilyPond: un *music engraver* integrabile con LaTeX». ArsTEXnica, **18**, pp. 97–118.

Hufflen, J.-M. (2011). «Some experience with MusiXTEX». In *Proc. EuroBachoTEX 2011 Conference*, a cura di K. Berry, J. B. Ludwichowski e T. Przechlewski. Bachotek, Poland, pp. 19–30.

— (2012). «A comparison of MusiXTEX and lilypond». In *Twenty Years After. Proc. BachoTEX 2012 Conference*, a cura di T. Przechlewski, K. Berry e J. B. Ludwichowski. Bachotek, Poland, pp. 103–108.

— (2013). «Why typesetting music is so difficult». In *Typography and Communication. Proc. EuroBachoTEX 2013 Conference*, a cura di T. Przechlewski, K. Berry e J. B. Ludwichowski. Bachotek, Poland, pp. 79–84.

— (2014). «Musical symbols at the digital age». In *What Can Typography Gain from Electronic Media? Proc. BachoTEX 2014 conference*, a cura di T. Przechlewski, K. Berry, B. Jackowski e J. B. Ludwichowski. Bachotek, Poland, pp. 17–24.

— (2015). «Musical typography's dimensions». In *Various Faces of Typography. Proc. BachoTEX 2015 conference*, a cura di T. Przechlewski, K. Berry, B. Jackowski e J. B. Ludwichowski. Bachotek, Poland, pp. 61–65.

— (2017). «History of accidentals in music». TUG*boat*, **38** (2), pp. 147–156.

Jacobs, A. (1988). *The New Penguin Dictionary of Music*. Penguin Books, 4ª edizione.

Laurie, D. (2005). *M-Tx: Music from Text. Version 0.60. User's Guide*. http://icking-music-archive.org/software/mtx/mtx060.pdf.

LilyPond (2017). *Music Notation for Everyone*. http://www.lilypond.org.

MuseScore (2017). *Créer, écouter et imprimer de magnifiques partitions*. http://www.musescore.org.

NoteEdit (2014). *A Score Editor*. http://sourceforge.net/projects/noteedit.berlios.

Odyniec, A. (2016). «Skladaj nuty! czyli śpiewnik w LaTeXu». In *Convergence: TEXu wyjdź z szafy! Proc. BachoTEX 2016*, a cura di T. Przechlewski, K. Berry, B. Jackowski e J. B. Ludwichowski. pp. 7–16.

Rimsky-Korsakov, N. A. (1964). *Principles of Orchestration*. Dover Publications, Inc., New-York. Completed posthumously by Maximilian Osseyevich Steinberg. English translation of the Russian text by Edward Agate.

Rosegarden (2013). *What is Rosegarden?* http://www.rosegardenmusic.com.

Schofer, A. e Steinbach, A. (1987). «Automatisierter notensatz mit TEX». Technical report, Rheinische Friedrich-Wilhelms-Universität, Bonn.

Simons, D. (2004). *PMX—a Preprocessor for MusiXTEX. Version 2.5*. http://icking-music-archive.org/software/pmx/pmx250.pdf.

Stravinsky, I. (1921). *Chant du rossignol. Poème symphonique pour orchestre*, volume 633. Boosey & Hawkes.

Taupin, D. (1992). «MusicTEX: Using TEX to write polyphonic or instrumental music». In *EuroTEX 1992*. pp. 257–271.

Taupin, D., Mitchell, R. e Egler, A. (2002). *MusiXTEX©. Using TEX to Write Polyphonic or Instrumental Music. Version T.104*. Part of LaTeX distribution.

Unicode (2016). *Unicode 9.0.0*. http://www.unicode.org/versions/Unicode9.0.0/.

Vogt, H., Bard, M., Bielitz, M., Haller, H.-P., Raiss, H.-P. con Seipt, A. (1982). *Neue Musik seit 1945*. Philipp Reclam, Stuttgart, 3ª edizione.

▷ Jean-Michel Hufflen
FEMTO-ST (UMR CNRS 6174) & University of Bourgogne Franche-Comté,
16, route de Gray,
25030 BESANÇON CEDEX
FRANCE
jmhuffle at femto-st dot fr

Condizionali in LaTeX

Enrico Gregorio

Sommario

Dopo un breve cenno ai condizionali in TeX e LaTeX e alle loro limitazioni, si introducono le strutture condizionali disponibili con expl3, l'interfaccia di programmazione del futuro LaTeX3.

Abstract

After briefly touching on conditionals in TeX and LaTeX, with their limitations, we introduce the conditional structures available with expl3, the programming interface of the future LaTeX3.

1 Condizionali in TeX

Non ci possono essere dubbi sul fatto che le strutture condizionali siano fondamentali in qualsiasi linguaggio di programmazione e, ovviamente, TeX non fa eccezione.

Tuttavia, i condizionali in TeX sono, come al solito, piuttosto peculiari. Un condizionale prevede la sintassi

⟨*condizionale*⟩⟨*test*⟩
  ⟨*codice per vero*⟩
`\else`
  ⟨*codice per falso*⟩
`\fi`

dove la parte `\else`⟨*codice per falso*⟩ è facoltativa.

Il ⟨*test*⟩ dipende dal ⟨*condizionale*⟩, secondo la seguente tabella:

- `\ifx`, il cui ⟨*test*⟩ consiste dei due token che seguono (senza espansione);
- `\if`, il cui ⟨*test*⟩ consiste dei due token che seguono (dopo espansione);
- `\ifcat`, il cui ⟨*test*⟩ consiste dei due token che seguono (dopo espansione);
- `\ifnum`, il cui ⟨*test*⟩ deve avere la forma ⟨*intero*⟩⟨*relazione*⟩⟨*intero*⟩;
- `\ifodd`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifdim`, il cui ⟨*test*⟩ deve avere la forma ⟨*lunghezza*⟩⟨*relazione*⟩⟨*lunghezza*⟩;
- `\ifvoid`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifhbox`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifvbox`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifeof`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifhmode`;
- `\ifvmode`;
- `\ifmmode`;
- `\ifinner`;
- `\iftrue`;
- `\iffalse`.

Per gli ultimi cinque il ⟨*test*⟩ è vuoto; per `\ifvoid`, `\ifhbox` e `\ifvbox` il numero ⟨*intero*⟩ è di solito definito con `\newsavebox` o `\newbox` (se non è il numero di uno dei registri provvisori); per `\ifeof` il numero ⟨*intero*⟩ è di solito definito con `\newread`.

La ⟨*relazione*⟩ è uno fra $<_{12}$, $=_{12}$ e $>_{12}$, dove il codice di categoria *deve* essere quello indicato. Siccome TeX esegue l'espansione per trovare la ⟨*relazione*⟩, in caso di dubbio si può scrivere `\string<` e simili.

La descrizione di questi condizionali può essere trovata nelle solite fonti, per esempio Gregorio (2009) o, ovviamente, Knuth (1986) e Eijkhout (1992). Esiste anche `\ifcase`, con una sintassi ancora più peculiare; ε-TeX e i vari motori ne definiscono altri ancora. Nel seguito, `\ifZ` rappresenterà uno dei condizionali primitivi.

I più importanti, oltre agli ovvi `\ifnum`, `\ifodd` e `\ifdim`, sono `\ifx` e `\if`. Il primo confronta il *significato* dei due token che seguono, il secondo il loro codice di carattere (un token simbolico è considerato come un carattere oltre il dominio possibile). Il condizionale seguirà il ramo 'vero' se i due token sono uguali, il ramo 'falso' altrimenti. Due token sono uguali per `\ifx` quando `\show` stampa la stessa cosa (con qualche eccezione in casi esoterici). Due token carattere daranno lo stesso risultato con `\if` e `\ifx`, ma occorre ricordare che il primo espande, il secondo no.

Va anche notato che i condizionali sono del tutto indipendenti dai *gruppi*, quindi `\ifx aa{\else}\fi` equivale a inserire una graffa aperta. Un trucco del genere è usato in LaTeX nella forma

```
{\ifnum0=‘}\fi
\ifnum0=‘{\fi}
```

per 'ingannare' il meccanismo delle tabelle; si noti che i due ⟨*test*⟩ sono entrambi falsi. L'analisi di questa costruzione e dei suoi usi è interessante, ma ci porterebbe troppo distante.

Quando si definisce un nuovo condizionale con `\newif`, in realtà si stabilisce l'equivalenza del nuovo condizionale con `\iftrue` o `\iffalse`. Qualsiasi istruzione del tipo

```
\let\cs=\ifZ
```

rende `\cs` del tutto equivalente al condizionale primitivo, in particolare per quanto riguarda l'accoppiamento con `\else` e `\fi`.

2 Espansione dei condizionali

I condizionali `\ifZ`, al pari di `\else` e `\fi`, sono *espandibili*. L'espansione di `\ifZ` funziona così:

1. viene deciso se il ⟨*test*⟩ è vero o falso;
2. nel caso sia vero, l'espansione di `\ifZ` è semplicemente ciò che rimane rimuovendo `\ifZ` e il ⟨*test*⟩;
3. nel caso sia falso, l'espansione di `\ifZ` è tutto ciò che va da `\else` (escluso) fino a `\fi`;
4. se non c'è alcun `\else`, l'espansione è il solo `\fi`.

Un contatore interno viene aumentato di uno per il `\ifZ` che ha avviato la procedura e anche per ogni `\ifZ` incontrato fino a che si trova un `\else` o `\fi` allo stesso livello. Ogni `\fi` diminuisce di uno il contatore interno, così che TEX sa quando finire. Il testo ignorato quando il ⟨*test*⟩ è falso non viene interpretato in alcun modo, ma si tiene solo conto dei condizionali trovati durante la *traversata*.

Per vedere questo tipo di espansione all'opera, si provi

```
\def\fdef#1{%
  \expandafter\def
  \expandafter#1\expandafter
}
\fdef\testa{\iftrue a\else b\fi}
\show\test
\fdef\testb{\iffalse a\else b\fi}
\show\test
```

L'idea è che `\fdef` definisce l'argomento come l'espansione di primo livello del testo seguente. Si otterrà

```
> \testa=macro:
->a\else b\fi .

> \testb=macro:
->b\fi .
```

come previsto. In questi casi il ⟨*test*⟩ è vuoto; si può verificare che il ⟨*test*⟩ viene rimosso con `\if TT` o `\if TF` invece di `\iftrue` o `\iffalse`

L'espansione di `\else` funziona allo stesso modo dell'espansione di `\ifZ`, eccetto che il contatore interno non viene modificato da `\else`, ma solo dai condizionali che si trovano fino al `\fi` corrispondente che viene rimosso anch'esso. Per finire, l'espansione di `\fi` è vuota, dopo che il contatore interno è diminuito di uno.

La rimanenza di `\else` o `\fi` è spesso causa di grattacapi, che si risolvono in vari modi. Uno dei più comuni è quello di adoperare `\@firstoftwo` e `\@secondoftwo`, definendo macro di livello più alto che prendono argomenti. Per esempio

```
\newcommand{\eqtest}[4]{%
  \ifx#1#2%
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {#3}%
  {#4}%
}
```

produce il terzo argomento se i primi due argomenti sono token uguali (per `\ifx`), il quarto altrimenti. I due `\expandafter` servono a togliere di mezzo `\else` fino a `\fi` oppure `\fi` *prima* che `\@firstoftwo` e `\@secondoftwo` siano espansi a loro volta. Perciò nel caso vero e falso si avrà, rispettivamente

```
\@firstoftwo{#3}{#4}
\@secondoftwo{#3}{#4}
```

(i parametri `#3` e `#4` sono già stati sostituiti con gli argomenti effettivi).

Confusi? Be', ci vuole un po' di studio e di pazienza per entrare nei meandri di `\ifZ`, `\else`, `\fi` e `\expandafter` e venirne fuori.

3 Condizionali complessi

Non c'è alcun supporto primitivo per espressioni booleane. Se volessimo implementare un condizionale che valuta un "e", potremmo fare così: ci interessa sapere se un certo numero è maggiore di 0 e minore di 42; nel caso vogliamo eseguire `A`, altrimenti `B`. Un modo è facile:

```
\ifnum#1>0
  \ifnum#1<42
    A%
  \else
    B%
\else
  B%
\fi
```

La duplicazione di codice è però indesiderabile. Un altro modo è

```
\ifnum
  \ifnum#1>0 0\else 1\fi
  \ifnum#1<42 0\else 1\fi
=0
  A%
```

```
\else
  B%
\fi
```

che funziona perché i due condizionali interni producono 0 solo se entrambi falsi. Si potrebbero avere condizioni aggiuntive, con la stessa idea; similmente si può implementare il connettivo "o".

È chiaro però che espressioni booleane più complesse diventano difficili da gestire. Esistono altri modi, più o meno misteriosi.

Un'altra applicazione, con una strategia diversa per i tempi di espansione è

```
\newif\ifxetexorluatex
\begingroup
  \catcode94=7 % ASCII 94 is ^
  \catcode0=9 % ASCII 0 is ignored
  \catcode30=12 % ^^^ is ASCII 30
\expandafter\endgroup
\if\relax^^^^0000\relax
  \xetexorluatextrue
\else
  \xetexorluatexfalse
\fi
```

Qui l'espansione di `\if` avviene prima che termini il gruppo con le impostazioni dei codici di categoria date per precauzione. Se il motore è XeTeX o LuaTeX, `^^^^0000` diventa il carattere ASCII 0, quindi ignorato perché di categoria 9, e `\if` confronta `\relax` con un altro `\relax`. Altrimenti, `^^^` rappresenta il carattere ASCII 30 che non è uguale a `\relax`; i token `^000\relax\xetexorluatextrue` diventano il ⟨*codice per vero*⟩ e sono quindi ignorati. Più *code golfing* che altro, ma interessante lo stesso.

4 Condizionali in LaTeX

In LaTeX, per gli scopi della programmazione, esistono vari condizionali piuttosto utili:

```
\@ifnextchar
\@ifstar
\@ifpackageloaded
\@ifclassloaded
\@ifpackagelater
\@ifclasslater
\@ifpackagewith
\@ifclasswith
```

che seguono il paradigma

`\@ifZ`⟨*test*⟩`{`⟨*true*⟩`}{`⟨*false*⟩`}`

Per esempio, il ⟨*test*⟩ per `\@ifnextchar` è un singolo token; per `\@ifstar` è vuoto; per `\@ifpackagewith` è una coppia di argomenti tra graffe che rappresentano rispettivamente il nome di un pacchetto e un'opzione. Esiste anche `\@ifdefinable` che però accetta solo l'argomento per ⟨*true*⟩; nel caso il test risulti falso, cioè il token passato come primo argomento sia già definito, il codice ⟨*false*⟩ è implicito, cioè un messaggio di errore.

Un'importante estensione è fornita dal pacchetto etoolbox che definisce alcuni condizionali nello stile LaTeX e permette anche, con una sintassi nemmeno troppo complessa, di valutare espressioni booleane basate su altri condizionali. Anche ifthen ha la possibilità di valutare espressioni booleane, ma è meno flessibile e, a differenza di etoolbox, non prevede l'espandibilità.

Per esempio, la condizione `\ifxetexorluatex` può essere impostata con i pacchetti ifxetex e ifluatex scrivendo [1]

```
\newif\ifxetexorluatex
\ifboolexpr{
  bool{xetex} or bool{luatex}
}{%
  \xetexorluatextrue
}{%
  \xetexorluatexfalse
}
```

Per il problema precedente, la soluzione sarebbe

```
\newcommand{\ininterval}[1]{%
  \ifboolexpe{
    test{\ifnumcomp{#1}{>}{0}}
    and
    test{\ifnumcomp{#1}{<}{42}}
  }{#1: Yes}{#1: No}
}
```

La macro `\ifboolexpe` è espandibile, ma le espressioni ammesse sono limitate.

Definire macro ricorsive con etoolbox è più facile; per esempio, se volessimo inscatolare i termini di una stringa di caratteri, possiamo scrivere

```
\documentclass{article}
\usepackage{etoolbox}

\makeatletter
\newcommand{\boxit}[1]{%
  \boxit@aux#1\@nil
}
\def\boxit@aux#1#2\@nil{%
  \fbox{\strut#1}%
  \ifblank{#2}
    {}
    {\kern-\fboxrule\boxit@aux#2\@nil}%
}
\makeatother

\begin{document}

X\boxit{abc}X

\end{document}
```

1. https://tex.stackexchange.com/a/47711/

e il risultato sarebbe

X|a|b|c|X

In questo caso si potrebbe adoperare un altro trucco

```
\def\boxit@aux#1#2\@nil{%
  \fbox{\strut#1}%
  \if\relax\detokenize{#2}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\kern-\fboxrule\boxit@aux#2\@nil}%
}
```

ma `\ifblank` è più versatile e risulta vero anche se l'argomento contiene solo spazi.

Il pacchetto etoolbox fornisce anche `whileexpr` che prende due argomenti: il primo è un'espressione booleana, il secondo il codice da eseguire finché l'espressione è vera. Molto più flessibile di `\@whilenum`, `\@whiledim` e `\@whilesw`.

Ci sarebbero molte altre soluzioni al problema, ma è tempo di guardare avanti.

5 Ora c'è expl3

L'ambiente di programmazione expl3 (The LaTeX Project, 2015) rinomina i condizionali primitivi secondo le sue convenzioni:

- `\ifx` diventa `\if_meaning:w`;
- `\if` diventa `\if_charcode:w` (ma anche `\if:w`);
- `\ifcat` diventa `\if_catcode:w`;
- `\ifnum` diventa `\if_int_compare:w`;
- `\ifodd` diventa `\if_int_odd:w`;
- `\ifdim` diventa `\if_dim:w`;
- `\ifvoid` diventa `\if_box_empty:N`;
- `\ifhbox` diventa `\if_hbox:N`;
- `\ifvbox` diventa `\if_vbox:N`;
- `\ifeof` diventa `\if_eof:w`;
- `\ifhmode` diventa `\if_mode_horizontal:`;
- `\ifvmode` diventa `\if_mode_vertical:`;
- `\ifmmode` diventa `\if_mode_math:`;
- `\ifinner` diventa `\if_mode_inner:`;
- `\iftrue` diventa `\if_true:`;
- `\iffalse` diventa `\if_false:`.

Vanno aggiunti `\if_case:w`, `\else:` e `\fi:`.

Meglio chiarire subito che l'uso diretto dei condizionali che hanno `w` come *signature* è sconsigliato per la programmazione ad alto livello. Tuttavia, anche gli altri condizionali che richiedono `\fi:` (con il facoltativo `\else:`) hanno una controparte con argomenti che sono la versione preferita.

Per esempio, se non ci serve davvero la capacità di `\if_charcode:w` di espandere i token che seguono, è meglio adoperare `\token_if_eq_meaning:NNTF` secondo la sintassi

```
\token_if_eq_meaning:NNTF
  ⟨token1⟩
  ⟨token2⟩
  {⟨codice per vero⟩}
  {⟨codice per falso⟩}
```

Esistono anche

```
\token_if_eq_charcode:NNTF
\token_if_eq_catcode:NNTF
\token_if_group_begin:NTF
\token_if_group_end:NTF
\token_if_math_toggle:NTF
```

e molti altri per esaminare la natura del token passato come primo argomento.

In tutti i casi di condizionali predefiniti, gli argomenti di tipo `TF` sono facoltativi, nel senso che esistono anche

```
\token_if_eq_meaning:NNT
\token_if_eq_meaning:NNF
```

per avere codice più compatto. Gli argomenti di tipo `T` e `F` non differiscono da quelli di tipo `n` per quanto riguarda la sintassi, ma vengono distinti proprio per applicare queste abbreviazioni e per maggiore chiarezza sul loro impiego.

Per comparazioni di interi, ci sono due possibilità:

```
\int_compare:nNnTF
\int_compare:nTF
```

La prima è più efficiente perché è un'interfaccia diretta con `\if_int_compare:w`

```
\int_compare:nNnTF { 2 } < { 3 }
 { A }
 { B }
```

che però nel primo e terzo argomento accetta *espressioni intere* senza bisogno di `\int_eval:n`.

Il secondo, seppure un po' meno efficiente, è di gran lunga più flessibile: il problema dell'intervallo analizzato prima si risolve con

```
\int_compare:nTF { 0 < #1 < 42 }
 { A }
 { B }
```

Il numero di relazioni ammesse è arbitrario; non solo, anche le relazioni sono più flessibili. Se, per esempio, gli estremi dell'intervallo sono accettabili, nella programmazione classica si dovrebbero adoperare -1 e 43. Qui, invece, possiamo più facilmente scrivere

```
\int_compare:nTF { 0 <= #1 <= 42 }
 { A }
 { B }
```

Le relazioni ammesse sono `=`, `<`, `<=`, `>`, `>=` e `!=` (che sta per $\neq$). Anche in questo caso è possibile usare *espressioni intere*:

```
\int_compare:nTF
 { \int_mod:nn {6}{3} <= #1 <= 6*7 }
 { A }
 { B }
```

Il prezzo da pagare in termini di efficienza è elevato,[2] ma la flessibilità è enormemente maggiore.

Non è il caso di elencare tutti i condizionali disponibili, perché sono moltissimi. Vanno citati

```
\dim_compare:nNnTF
\dim_compare:nTF
\fp_compare:nNnTF
\fp_compare:nTF
```

che funzionano in modo analogo ai precedenti.

Naturalmente expl3 fornisce anche *variabili booleane*:

```
\bool_new:N \l_manual_foo_bool
\bool_new:N \g_manual_foo_bool
```

definiscono nuove variabili booleane, una *locale* e una *globale* (valgono le solite convenzioni), inizialmente con valore 'falso'. Queste possono essere modificate con

```
\bool_set_true:N \l_manual_foo_bool
\bool_set_false:N \l_manual_foo_bool
\bool_gset_true:N \g_manual_foo_bool
\bool_gset_false:N \g_manual_foo_bool
```

e adoperate come primo argomento di `\bool_if:NTF`. Per esempio

```
\bool_if:NTF \l_manual_foo_bool
 { A }
 { B }
```

è la controparte del classico

```
\iffoo A\else B\fi
```

Dove però expl3 sbaraglia la concorrenza è nella possibilità di sfruttare *espressioni booleane*. Un'espressione booleana *atomica* è una variabile booleana oppure un *predicato*. Ogni condizionale espandibile predefinito fornisce anche la forma predicativa, per esempio

```
\int_compare_p:n { #1 > 0 }
```

è un predicato che darà 'vero' se l'argomento della macro è un intero positivo, 'falso' altrimenti. Il nome della forma predicativa è facilmente prevedibile, perché si ottiene togliendo `TF` e aggiungendo `_p` prima della *signature*.

Un'espressione booleana si ottiene applicando nel modo tradizionale connettivi e parentesi. I connettivi ammessi sono `!`, `&&` e `||`, per 'non', 'e', 'o' rispettivamente, in ordine di precedenza. L'esempio nella documentazione è

```
\int_compare_p:n { 1 = 1 } &&
  (
   \int_compare_p:n { 2 = 3 } ||
   \int_compare_p:n { 4 <= 4 } ||
   \str_if_eq_p:nn { abc } { def }
  ) &&
  ! \int_compare_p:n { 2 = 4 }
```

La funzione `\bool_if:nTF` prende come primo argomento un'espressione booleana.

Va notato che *tutte* le espressioni atomiche sono valutate, per poi stabilire la verità o falsità dell'intera espressione. Fino a qualche tempo fa, si era tentato di permettere valutazioni *lazy*, cioè solo delle espressioni atomiche necessarie, ma questo si è rivelato troppo complicato. Un esempio è un baco di fontspec venuto alla luce quando il cambio è diventato operativo;[3] il codice adoperava

```
\bool_if:nTF
 {
  \tl_if_single_p:n {##1}
  &&
  \token_if_cs_p:N ##1
 }
 {true}{false}
```

che fallisce quando `##1` è `'A` (e in molti altri casi): infatti la prima espressione atomica dà falso, ma la seconda è illecita, perché lascia `A` nella lista di token da esaminare. Questo si risolve con `\bool_lazy_and:nnTF` che appunto fornisce una valutazione *lazy*:

```
\bool_lazy_and:nnTF
 { \tl_if_single_p:n {##1} }
 { \token_if_cs_p:N ##1 }
 {true}{false}
```

(i codici per vero e falso non sono rilevanti). Con questa funzione, se la prima espressione risulta falsa la seconda non viene nemmeno esaminata, perché l'espressione globale con il connettivo 'e' è certamente falsa. Il risultato dà vero solo se la prima espressione è vera, cioè `##1` è un singolo token, e anche la seconda, cioè il token è un token simbolico. Quando la prima espressione risulta falsa, la seconda viene semplicemente ignorata (che sia valida sintatticamente o no).

2. Un rapido test fatto eseguendo dieci milioni di istruzioni dice che la seconda forma è cinque volte più lenta della prima: 50 secondi contro 10.

3. https://tex.stackexchange.com/questions/382880/

Le valutazioni *lazy* sono disponibili solo per espressioni che coinvolgono i connettivi 'e' e 'o':

```
\bool_lazy_and:nnTF
\bool_lazy_or:nnTF
\bool_lazy_all:nTF
\bool_lazy_any:nTF
```

Il condizionale precedente potrebbe essere scritto come

```
\bool_lazy_and:nTF
 {
  { \tl_if_single_p:n {##1} }
  { \token_if_cs_p:N ##1 }
 }
 {true}{false}
```

ma quando le espressioni atomiche sono solo due è più efficiente la versione a due argomenti.

Possiamo ovviamente definire nuovi condizionali. Prendiamo come esempio proprio una funzione che produca vero se l'argomento è una lista di token che consista di un singolo token simbolico. Ecco una possibilità:

```
\prg_new_conditional:Nnn
 \manual_tl_if_singlecs:n
  { TF, T, F, p }
  {
   \bool_lazy_and:nnTF
    { \tl_if_single_p:n {#1} }
    { \token_if_cs_p:N #1 }
   { \prg_return_true: }
   { \prg_return_false: }
  }
```

che definirà

```
\manual_tl_if_singlecs:nTF
\manual_tl_if_singlecs:nT
\manual_tl_if_singlecs:nF
\manual_tl_if_singlecs_p:n
```

Il primo argomento è il nome del nuovo condizionale (meglio se contiene `if`), con una *signature* provvisoria che indichi quanti argomenti verranno passati per costruire il condizionale; il secondo è l'elenco delle funzioni effettivamente definite (dove `p` sta per la forma predicativa).

La versione expl3 del condizionale per 'motori Unicode' sarebbe

```
\prg_new_conditional:Nnn
 \manual_if_unicode_engine:
  { TF, T, F, p }
  {
   \bool_if:nTF
    {
     \sys_if_engine_xetex_p:
     ||
     \sys_if_engine_luatex_p:
    }
    { \prg_return_true: }
    { \prg_return_false: }
  }
```

Non è possibile specificare la forma predicativa nella definizione se il codice nel terzo argomento contiene parti non espandibili (per esempio `\tl_if_in:nnTF`). Nel manuale di expl3 le funzioni espandibili sono marcate con una stellina rossa. Se il codice non è espandibile, si userà

```
\prg_new_protected_conditional:Nnn
```

6 *Case switch*

Un *case switch* è un particolare tipo di condizionale 'multiplo'. Alcuni linguaggi forniscono *else if* e direttamente funzioni per *case switch* più complessi che però non esistono in TEX. Ovviamente si possono programmare e si trovano parecchi esempi su TUGboat.

In TEX si trova una funzione primitiva in tutti i sensi, cioè `\ifcase`:

```
\ifcase⟨intero⟩
  ⟨caso 0⟩\or
  ⟨caso 1⟩\or
  ...
  ⟨caso n⟩\else
  ⟨altri casi⟩\fi
```

L'espansione è analoga a quella dei condizionali: TEX esamina l'intero e scarta tutto ciò che trova fino all'`\or` (compreso) che precede il caso corrispondente, oppure a `\else` o a `\fi`. L'espansione di `\or` è vuota ed elimina tutto quanto si trova fino a `\fi`.

Un *case switch* elementare si può ottenere assegnando valori interi a macro opportune. Per esempio

```
\def\cs@apple{0}
\def\cs@peach{1}
\def\cs@apricot{2}
\def\frutto#1{%
  \ifcase\csname cs@#1\endcsname
    pomo\or
    persego\or
    armelin\else
    fruto\fi
}
```

Le difficoltà di questo approccio sono evidenti, perché occorre far corrispondere manualmente il codice della macro con la lista dei numeri assegnati.

La stessa cosa in expl3:

```
\cs_new:Nn \manual_fruit:n
 {
  \str_case:nnF { #1 }
   {
    {apple}{pomo}
    {peach}{persego}
```

```
    {apricot}{armelin}
   }
   {fruto}
 }
```

Esistono varie altre funzioni predefinite:

```
\str_case_x:nnTF
\tl_case:NnTF
\int_case:nnTF
\dim_case:nnTF
```

In tutte l'argomento corrispondente a `T` può essere omesso come abbiamo fatto nell'esempio precedente; se adoperato, il testo viene inserito dopo quello corrispondente al caso *matched*. Normalmente non lo si adopera.

La versione `\str_case_x:nnTF` espande sia il primo argomento sia la prima parte dei casi. Un esempio d'uso[4] è nella tavola 1. Si noti che

```
\str_case:nnF
 {\languagename}
 {
  {english}{EN}
  {spanish}{SP}
 }
 {??}
}
```

produrrebbe `??` in ogni caso perché la stringa data come primo argomento non viene interpretata in alcun modo; con `\str_case_x:nnF` invece il primo argomento viene espanso prima del confronto.

Un'interessante macro che implementa i vari *case switch* a livello utente può essere[5]

```
\NewExpandableDocumentCommand
 {\switchcondition}
 {O{string}mmO{}}
 {
  \use:c { manual_#1_switch:nnn }
   { #2 }
   { #3 }
   { #4 }
 }

\cs_new:Nn \manual_string_switch:nnn
 {
  \str_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_stringx_switch:nnn
 {
  \str_case_x:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_token_switch:nnn
 {
  \tl_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_integer_switch:nnn
```

4. https://tex.stackexchange.com/a/348677/4427
5. https://tex.stackexchange.com/a/386307/4427

```
 {
  \int_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_dimen_switch:nnn
 {
  \dim_case:nnF { #1 } { #2 } { #3 }
 }
\ExplSyntaxOff
```

Un esempio d'uso è

```
\newcommand{\placement}[1]{%
  \switchcondition{#1}{
    {ul}{\let\position\AtPageUpperLeft}
    {ll}{\let\position\AtPageLowerLeft}
    {ur}{\let\position\AtPageUpperRight}
    {lr}{\let\position\AtPageLowerRight}
  }[\let\position\ERROR]%
}
```

L'argomento facoltativo di `\switchcondition` decide il tipo di *case switch*.

Riferimenti bibliografici

Eijkhout, V. (1992). *TEX by Topic, A TEXnician's Reference*. Addison-Wesley, Reading, MA, USA. http://eijkhout.net/texbytopic/.

Gregorio, E. (2009). «Appunti di programmazione in TEX e LATEX». http://profs.scienze.univr.it/~gregorio/introtex.pdf.

Knuth, D. E. (1986). *The TEXbook*, volume A di *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA.

The LATEX Project (2015). «The expl3 package and LATEX3 programming». http://ctan.org/pkg/l3kernel.

▷ Enrico Gregorio
Dipartimento di Informatica
Università di Verona
enrico dot gregorio at univr dot it

```
\documentclass{article}
\usepackage[english,ngerman]{babel}
\usepackage{amsmath}
\usepackage{xparse}

\ExplSyntaxOn
\NewDocumentCommand\DeclareBabelMathOperator{mmO{???}}
 {
  \DeclareMathOperator{#1}
   {
    \str_case_x:nnF { \use:c { bbl@main@language } } { #2 } { #3 }
   }
 }
\NewDocumentCommand\DeclareVariableBabelMathOperator{mmO{???}}
 {
  \DeclareMathOperator{#1}
   {
    \str_case_x:nnF { \languagename }{ #2 } { #3 }
   }
 }
\ExplSyntaxOff


\DeclareBabelMathOperator{\range}{
  {english}{ran}
  {ngerman}{Bild}
}
\DeclareBabelMathOperator{\kernel}{
  {english}{ker}
  {ngerman}{Kern}
}[ker]
\DeclareVariableBabelMathOperator{\vrange}{
  {english}{ran}
  {ngerman}{Bild}
}
\DeclareVariableBabelMathOperator{\vkernel}{
  {english}{ker}
  {ngerman}{Kern}
}[ker]

\begin{document}

\section{Fixed names}
\[
  \dim(V) = \dim(\range(T)) + \dim(\kernel(T))
\]
\begin{otherlanguage*}{english}
\[
  \dim(V) = \dim(\range(T)) + \dim(\kernel(T))
\]
\end{otherlanguage*}

\section{Variable names}
\[
  \dim(V) = \dim(\vrange(T)) + \dim(\vkernel(T))
\]
\begin{otherlanguage*}{english}
\[
  \dim(V) = \dim(\vrange(T)) + \dim(\vkernel(T))
\]
\end{otherlanguage*}

\end{document}
```

1 Fixed names

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

2 Variable names

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

$$\dim(V) = \dim(\mathrm{ran}(T)) + \dim(\ker(T))$$

Tavola 1: Esempio d'uso di `\str _case_x:nnF`

Introduzione alla composizione di testi scacchistici

Maurizio Molinaro

Sommario

In questa introduzione alla composizione di testi scacchistici, dopo aver definito alcuni obiettivi e alcuni requisiti necessari, si esaminano le caratteristiche generali e, più approfonditamente, gli ingredienti specifici dei testi scacchistici (linea principale, analisi delle varianti, diagrammi), evidenziando gli errori e i difetti più comuni. Segue una rassegna dei principali strumenti attualmente messi a disposizione da LaTeX, in particolare i pacchetti dedicati e i caratteri scacchistici, e si fa cenno ad alcuni programmi esterni che possono essere di interesse. Si presentano alcuni esempi completi che mostrano l'applicazione pratica dei concetti introdotti in precedenza. Nelle conclusioni si riassumono gli argomenti che meritano di essere ripresi e ampliati in futuro.

Abstract

In this introduction to typesetting techniques for chess texts, after stating some goals and some necessary requirements, the general characteristics of chess texts are examined, with emphasis on the specific ingredients (main line, variation analysis, diagrams), and highlighting the most common errors and defects. It follows a review of the main tools presently offered by LaTeX, in particular dedicated packages and chess characters, and reference is made to external programs that can be of interest. Some complete examples are presented, showing the practical application of the concepts previously introduced. The last section summarizes the topics worthy to be resumed and widened in the future.

1 Introduzione

Questo studio introduttivo ha lo scopo primario di suscitare interesse per la composizione dei testi scacchistici e si propone di fornire al lettore le informazioni basilari sugli elementi che li costituiscono e sui principali strumenti attualmente a disposizione con LaTeX.

Il fascino del gioco degli scacchi è dovuto a varie ragioni, tra cui probabilmente le principali sono la sua collocazione al confine tra il gioco, l'arte e la scienza (Tarrasch) e la sua capacità di appassionare persone di tutte le età e di tutte le culture, offrendo possibilità illimitate all'agonismo e allo studio individuale.

Per seguire la trattazione è sufficiente una conoscenza generale del gioco, derivabile da un manuale quale Messa e Mearini (2017), ma naturalmente una conoscenza più approfondita può essere utile.

La letteratura scacchistica è immensamente vasta (sono stati pubblicati più libri e articoli sugli scacchi che su tutti gli altri giochi e sport messi insieme) e ha alcune peculiarità:

- molti libri continuano a essere scritti nelle lingue nazionali (la supremazia dell'inglese non è ancora così schiacciante come in altre discipline): sono molto frequenti le traduzioni (talvolta eseguite non sull'edizione originale, ma su di una traduzione intermedia) ed è forte il peso delle tradizioni tipografiche generali dei vari paesi;
- alcuni libri, considerati ormai classici, continuano a essere ristampati anche a grande distanza di tempo; altri invece, più legati all'attualità, hanno una vita molto più breve;
- negli ultimi 40 anni si è sempre più accentuato l'uso di simboli in sostituzione di parti del testo; in questo modo si riesce a comprendere, almeno nelle linee essenziali, anche un libro scritto in una lingua poco conosciuta, sia pur a costo di un notevole impoverimento del testo.[1]

Per questo articolo sono stati esaminati libri e articoli pubblicati in un arco di tempo che va dal 1965 al 2015 e in varie lingue, con particolare attenzione a testi pubblicati recentemente in Italia, quali Barletta e Messa (2014) e Grooten (2015). Il campione è significativo, ma necessariamente molto ristretto rispetto all'universo della letteratura scacchistica; inoltre, per motivi di spazio, la trattazione è limitata ai libri.

Si possono individuare varie categorie di testi scacchistici:

- testi generali, utili per l'apprendimento del gioco;
- testi specializzati in una delle tre fasi della partita (apertura, centro di partita, finale);
- testi diversi, quali:
  - raccolte di partite (per esempio di un torneo, del Campionato del Mondo, delle Olimpiadi);
  - monografie dedicate a un giocatore;
  - esercizi, quiz;
  - problemi, studi.

Gli "ingredienti" fondamentali dei testi scacchistici sono: *testo corrente*; *linea principale*; *analisi delle varianti*; *diagrammi*.

Nelle categorie sopra elencate gli ingredienti sono all'incirca gli stessi, ma il peso relativo di ciascuno

1. A questo proposito meritano di essere citate le pubblicazioni dell'Informatore (Šahovski Informator, Chess Informant), dove si fa largo uso di simboli e sono disponibili tavole esplicative in varie lingue.

è, di solito, piuttosto diverso: un testo didattico contiene una quantità maggiore di testo corrente (ossia, assomiglia di più a un libro generico) rispetto a una monografia dedicata a un'apertura, in cui quasi tutto lo spazio è occupato dall'analisi delle varianti; le raccolte di partite sono costituite in gran parte da linea principale, diagrammi e analisi, con conseguente riduzione del testo corrente.

2 Obiettivi e requisiti

L'obiettivo generale che ci si prefigge è la produzione di testi scacchistici di qualità elevata. Si può affermare con buona sicurezza che i margini di miglioramento sono piuttosto ampi, anche se nei testi più recenti si riscontra, per lo meno, una maggiore uniformità rispetto al passato.[2]

I requisiti più importanti in vista dell'obiettivo anzi detto possono essere così riassunti:

- *correttezza linguistica e tecnica*; vale tanto per il testo corrente, quanto per l'analisi delle varianti e i diagrammi; qui si intende per correttezza tecnica la correttezza formale dei contenuti tecnici, indipendentemente dalla loro validità concettuale;
- *facilità di comprensione del testo*; la lettura e lo studio richiedono certamente la massima concentrazione, ma non si devono creare ostacoli aggiuntivi;
- *eleganza tipografica*; questo requisito può avere varie articolazioni, quali: coerenza nelle scelte, uniformità di presentazione, capacità di evitare compressioni o dilatazioni eccessive degli spazi;
- *efficienza nella composizione*; in particolare è consigliabile, dovunque sia possibile, far ricorso a strumenti che consentano di automatizzare almeno una parte del processo.

Eccettuato il primo, l'ordine di priorità dei requisiti può mutare sensibilmente da una situazione all'altra; alcuni requisiti possono entrare in conflitto con altri. Nel proporre una soluzione occorre perciò trovare una "media pesata" tra le varie esigenze. Quando sono state delineate più soluzioni, la scelta può essere dettata da considerazioni diverse: una soluzione può essere preferibile a un'altra perché soddisfa meglio l'esigenza di risparmiare spazio; una soluzione non eccezionale può essere accettabile se è applicata con coerenza, e così via.

Non è escluso che, se si riesce a soddisfare un requisito, si dia un contributo anche a soddisfare uno degli altri. Per esempio, se è vero che tutti i testi si possono costruire manualmente, è altrettanto vero che la costruzione manuale fa crescere a dismisura le probabilità di errore, soprattutto quando si passa da frammenti a testi completi. Quindi, se si riesce ad automatizzare, almeno in parte, il processo di costruzione delle porzioni specificamente scacchistiche del testo, è probabile che si abbia anche un aumento della correttezza complessiva.

Le considerazioni su errori e difetti saranno riprese nel §9, dopo aver passato in rassegna le caratteristiche generali e gli elementi più importanti dei testi scacchistici.

3 Caratteristiche generali dei testi scacchistici

Un testo scacchistico può sicuramente essere annoverato tra i testi di natura tecnico-scientifica (Beccari *et al.*, 2016, capitolo 2, p.15).

Come anticipato nel §1, in questo articolo si prendono in considerazione soltanto i libri.

Le caratteristiche generali di un libro di scacchi possono essere così sintetizzate.

- Formato della pagina: variabile (formati comuni: $11,5 \times 19$ cm; 15×21 cm; 17×24 cm); raramente si incontrano dimensioni eccezionali.
- Geometria della pagina: regolare; raramente si incontrano forti asimmetrie.
- Intestazioni e piè di pagina: molto sobri, e di solito più simili a un testo letterario.
- Margini laterali: estremamente ridotti e non utilizzati.
- Composizione sia su colonna unica, sia su due colonne. Il ricorso alla doppia colonna è utile soprattutto per risparmiare spazio in presenza di diagrammi (§7). Non è raro il passaggio da colonna unica a due colonne e viceversa all'interno dello stesso testo.

Per quanto riguarda i caratteri, per il testo si usa quasi senza eccezioni un tipo con grazie. Il testo corrente è spesso composto in corpo 10/12 oppure 11/13, con un numero di caratteri per riga attorno a 60 nel caso di colonna unica e attorno a 40 nel caso di due colonne.

Il testo può essere suddiviso in parti, capitoli, paragrafi. I rimandi da un punto all'altro del testo non sono particolarmente frequenti e consistono spesso del solo numero di capitolo. Per questo motivo talvolta i paragrafi non sono numerati.

Appendici, glossari, indici analitici sono spesso presenti, ma potrebbero essere impiegati in modo più sistematico (in tutti i testi, per esempio, potrebbe essere opportuno disporre di un indice dei giocatori e un indice delle aperture).

I riferimenti bibliografici si trovano nelle note o addirittura all'interno del testo, più raramente in una raccolta strutturata.[3] Lo stile bibliografico è estremamente variabile da un testo all'altro.

2. Come in altri generi letterari, anche nei testi scacchistici la tolleranza dei lettori alle imperfezioni tipografiche (anche le più grossolane) sembra essere eccezionalmente alta, forse perché si tratta di una lettura avvincente e intellettualmente impegnativa. Pertanto, se si invitassero i lettori a esprimere un giudizio sulla qualità dei testi, è probabile che non ne scaturirebbero troppe osservazioni negative.

3. Le raccolte bibliografiche di testi scacchistici potrebbero essere un tema di studio molto interessante per la presenza frequente di opere tradotte, anche a distanza di decenni dall'uscita dell'opera originale, e di edizioni "incerte" come data e luogo di pubblicazione.

I rimandi ad altri testi non sono particolarmente frequenti, anche se è presente una raccolta di riferimenti bibliografici.

Le note a piè di pagina non sono molto frequenti; spesso sono note del traduttore.

Come in tutti i testi tecnico-scientifici, il testo corrente è spesso inframmezzato da *inserti*. Di questi, alcuni sono generici, ossia comuni a quelli che si trovano in testi dedicati ad altri argomenti; alcuni sono specificamente legati agli scacchi, e naturalmente sono quelli che riceveranno maggiore attenzione nel seguito.

Inserti generici

- Elenchi contrassegnati e numerati: abbastanza frequenti.
- Definizioni ed esempi: non molto frequenti (o meglio, possono essere presenti, ma non sono evidenziati come tali).
- Figure: non molto frequenti, tranne qualche fotografia (quasi sempre fuori testo) e qualche schema a blocchi.
- Tabelle: non molto frequenti; per lo più contengono i risultati di un torneo o di un incontro.
- Citazioni: non molto frequenti; per lo più sono tratte da articoli di giornali o riviste, interviste radio-televisive, ecc.

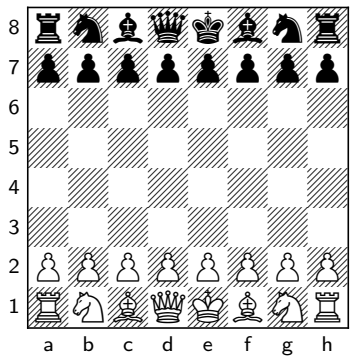
Inserti specifici

- Linea principale: vedi §5; può essere anche in linea con il testo.
- Analisi delle varianti: vedi §6; può essere anche in linea con il testo.
- Diagrammi: vedi §7.

4 Notazione scacchistica

Come si vede nel diagramma 1, che riporta la posizione iniziale dei pezzi, ognuna delle 64 case della scacchiera è individuata da una coppia costituita da una lettera per le colonne e da un numero per le traverse (*notazione algebrica*), quindi da a1 a h8.

Diagramma 1



I pezzi sono individuati da lettere (di norma le iniziali, variabili da una lingua all'altra; in italiano: R, D, T, A, C, P) o da simboli (§11): ♔ ♕ ♖ ♗ ♘ ♙. I simboli hanno incominciato a diffondersi all'incirca dal 1990 e ora sono diventati di uso pressoché universale. Se la lingua in cui è scritto il testo è poco nota, è chiaro che i simboli si riconoscono più facilmente delle lettere (si tenga presente anche quanto detto nel §1 circa la migrazione da testo a simboli), ma l'appesantimento grafico della pagina che ne consegue è sensibile. L'uso delle lettere, per contro, non incide seriamente sulla comprensione: l'associazione lettera – pezzo è quasi immediata, soprattutto all'inizio della partita, ed è sufficiente una tabellina di corrispondenze collocata opportunamente per togliere ogni dubbio. Nel seguito lettere e simboli saranno alternati liberamente.

Anche quando si usano i simboli invece delle lettere, sembra meno elegante (e qualche volta anche meno comprensibile) usare il simbolo anche nel testo corrente, ossia scrivere "la ♖h1" anziché "la Th1", oppure "il ♙" anziché "il Pedone".

Nella *notazione completa* si riportano, dopo la lettera o il simbolo, la casa di partenza e quella di arrivo (es.: Th4-e4); nella *notazione abbreviata* si riporta solo la casa di arrivo (es.: Te4), con qualche complicazione quando occorre eliminare le possibili ambiguità (es.: The4 opp. T5e4). La notazione abbreviata è ovviamente più frequente perché permette di risparmiare spazio; la notazione completa si usa solo nella linea principale, soprattutto se è fuori linea rispetto al testo e in verticale, come nell'esempio 2 del §5.

L'arrocco è rappresentato da simboli speciali (O-O e O-O-O).

Le strutture fondamentali sono tre:

1) 1. Cf3, Cf6; (mossa completa)
2) 1. Cf3, (semi-mossa del Bianco)
3) 1. ..., Cf6; (semi-mossa del Nero)

Per la numerazione delle mosse la forma "1." non ha alternative valide. Lo spazio dopo "1." dovrebbe essere insecabile per evitare lo spezzamento immediato della struttura tra due righe, ma in presenza di colonne molto strette lo spezzamento è praticamente inevitabile.

La virgola e il punto e virgola sono facoltativi. Sono sicuramente superflui per la linea principale in verticale, come nell'esempio 2 del §5. La virgola sembra sempre superflua; il punto e virgola può essere utile se segue un'altra mossa.

Per la semi-mossa del Nero la forma riportata è la più comune; 1. — Cf6 è senza dubbio preferibile.

È evidente l'importanza di proporzionare correttamente gli spazi tra le semi-mosse e tra una mossa e l'altra.

In aggiunta alle informazioni basilari (pezzo e casa di arrivo), si trovano poi:

- *segni* (x + # ! ?), che possono essere variamente combinati;
- *scritte* (=D e simili per la promozione del Pedone; e.p. per la presa *en passant*);

- *simboli dell'Informatore* (§1), che rappresentano un'estensione della logica sottostante a segni come ! e ?, ma per lo più sono comprensibili solo con l'ausilio di una tavola esplicativa.

I segni e le scritte si possono trovare all'interno della mossa o come suffissi; i simboli dell'Informatore prima o dopo la mossa.

Esempi di segni. 2. exd5 (cattura); 2. Ab5+ (scacco); 2. g4?? (grave errore); 2. — Dh4# (scacco matto).

Esempi di simboli dell'Informatore. 12. — Ce3; 13. Ae3 ± ⟳ ↑≪ con il significato: il Bianco è superiore; è in vantaggio di sviluppo; ha l'iniziativa sull'ala di Donna.
Altri esempi: ⌓ (è migliore...); △ (con l'idea di...); ⧉ (Alfieri di colore opposto).

Le alternative (più o meno frequenti) rispetto a exd5 (come e:d5, ed5:, e:d, ed) o rispetto a Txe8 (come TxT) non sono corrette, ma non sono significative ai fini della composizione.

Tutto sommato, i componenti delle strutture sono pochi, ma facendo intervenire anche le varianti di carattere (grassetto, corsivo, colore) è possibile generare numerosissime combinazioni. La disponibilità di soluzioni diverse è preziosa perché consente di definire *stili di composizione* costituiti da più livelli (§5 – §6).

5 Linea principale

La linea principale (*main line*) può contenere:
- lo svolgimento di una partita, dall'inizio o da un punto qualsiasi;
- la variante principale nell'esposizione della teoria di un'apertura;
- la variante principale nella soluzione di uno studio, di un problema, ecc.

La linea principale è sempre presente e, pur essendo spesso un *inserto nel testo* (§3), di fatto assume un ruolo dominante.

Rispetto al testo corrente, la linea principale può essere collocata: *fuori linea*, con sviluppo orizzontale o verticale, oppure *in linea*, sempre con sviluppo orizzontale.

Entrambe le forme sono frequenti e possono tranquillamente coesistere all'interno di un libro, ma all'interno della stessa partita non devono essere mescolate: la partita è tutta fuori linea o tutta in linea.

Nella forma fuori linea, lo sviluppo orizzontale è adatto a colonne di larghezza qualsiasi; lo sviluppo verticale è più adatto a colonne strette. All'interno della stessa partita lo sviluppo verticale e lo sviluppo orizzontale non devono essere mescolati.

Nella gerarchia di uno *stile di composizione* (§4) la linea principale è il **livello 1**.

La linea principale dev'essere riconoscibile senza possibilità di equivoci rispetto al testo corrente e agli altri inserti, ma anche rispetto a brevi commenti assimilabili al testo corrente. Quando è in linea, ciò richiede di far ricorso a qualche accorgimento tipografico, mentre quando è fuori linea se ne potrebbe far a meno.

Nella linea principale si trovano impiegati: grassetto, corsivo, corpo maggiore, colore diverso, tipo di carattere diverso. È consigliabile non sovrapporre troppi accorgimenti e mantenere la massima coerenza (spesso assente, invece, soprattutto nei testi meno recenti, dove si trovano brutti esempi di mescolanza di normale, grassetto, corsivo, corpo maggiore, tipo diverso).

La scelta che sembra più appropriata per l'impatto visivo è: grassetto + tipo di carattere diverso. Per uniformità la stessa scelta dovrebbe essere adottata anche quando la linea principale è fuori linea.

Il tipo di carattere diverso dev'essere armonizzato con tutti gli elementi, come titoli e didascalie, per i quali non si utilizza il tipo del testo corrente. Dal momento che quest'ultimo è quasi senza eccezioni con grazie, quello della linea principale può essere senza grazie, come negli esempi che seguono.

Esempio 1 (fuori linea orizzontale)
1. e4 e6; 2. d4 d5; 3. ♘d2

Il Sistema Tarrasch nella Difesa Francese.

3. — ♘f6; 4. e5 ♘fd7; 5. ♗d3 c5; 6. c3 ♘c6; 7. ♘e2 ♕b6

Esempio 2 (fuori linea verticale, con notazione completa)

1.	**e2–e4**	**e7–e6**
2.	**d2–d4**	**d7–d5**
3.	**Cb1–c3**	**Cg8–f6**

Il Sistema Classico nella Difesa Francese.

4.	**Ac1–g5**	**Af8–e7**
5.	**e4–e5**	**Cf6–d7**
6.	**Ag5×e7**	**Dd8×e7**

Esempio 3 (in linea)
1. e4 e6; 2. d4 d5; 3. e5 Il Sistema Nimzowitsch nella Difesa Francese. **3. — c5; 4. c3 Cc6; 5. Cf3 Db6; 6. Ae2 c×d4; 7. c×d4 Cge7**

Gli esempi mostrano i rientri e i distanziamenti verticali corretti. Quando la linea principale è fuori linea orizzontale (esempio 1), non vi è motivo che sia rientrata come un normale capoverso, ma dev'essere adeguatamente distanziata dal testo intercalato. Quest'ultimo, invece, può essere o no rientrato, in accordo con la regola generale seguita per i rientri dei capoversi. Quando la linea principale è fuori linea verticale (esempio 2), è bene che sia fortemente rientrata, oltre che adeguatamente distanziata dal testo intercalato; per quest'ultimo vale quanto detto nell'esempio 1. Infine quando

la linea principale è in linea (esempio 3), è bene seguire in tutto e per tutto la regola generale valida per i rientri dei capoversi.

La linea principale può essere preceduta da un *titolo esteso*, allineato a sinistra o centrato. Vi sono numerosissime combinazioni, basate sull'uso di filetti, grassetto, colore, sfondi, ecc.; la scelta non è facile.

Il titolo esteso è senza dubbio preferibile alle scarne indicazioni date direttamente nel testo, che talvolta rendono difficile rintracciare la partita anche in presenza di un indice; tuttavia sembra ragionevole scartare soluzioni graficamente ingombranti.

Oltre ai nomi dei giocatori, le informazioni tipiche del titolo esteso sono: anno, evento, turno o numero della partita, punteggio Elo dei giocatori, apertura.

Esempio 1

Kasparov – Karpov
Campionato del Mondo – Mosca 1984
32ª partita – Difesa Ovest Indiana

Esempio 2

Torneo dei Candidati 2016	Primo turno
□ S. Karjakin	2760
■ V. Anand	2762
Apertura Inglese (A13)	

Il titolo esteso dev'essere indivisibile rispetto all'inizio della partita, quindi non deve risultare tipograficamente orfano.

L'inizio della linea principale può non coincidere con l'inizio della partita. In questo caso la linea principale è preceduta da un diagramma che fornisce la posizione di partenza o da un'indicazione testuale quale: "fino a 11. O-O come nella partita precedente". In questo caso di solito manca il titolo esteso: la partita è introdotta direttamente nel testo (ma devono essere ben evidenziati i dati tipici elencati sopra), oppure si fa ricorso alle scritte associate al diagramma (§8).

6 Analisi delle varianti

A questo punto è opportuno dare una definizione più precisa del termine "variante" che in modo intuitivo compare già nei paragrafi precedenti.

Si definisce *variante* una successione di una, due o più mosse, collegate tra loro al fine di raggiungere una certa posizione. Spesso le varianti si diramano una dall'altra, dando origine a una struttura ad albero.[4]

4. La rappresentazione grafica dell'albero delle varianti è un argomento interessante a cui riservare un possibile futuro articolo.

Va sotto il nome di *analisi delle varianti* il complesso di tutto ciò che non rientra né nel testo corrente, né nella linea principale.

L'analisi delle varianti è probabilmente l'ingrediente su cui occorre lavorare di più per accrescere la qualità della composizione. Qui i requisiti primari sono la chiarezza e la leggibilità. L'uniformità esasperata potrebbe essere nociva; è invece consigliabile mantenere la massima flessibilità e adattabilità alle diverse situazioni.

Nell'analisi delle varianti esiste anche una componente tecnica rilevante, in quanto si devono riportare solo le varianti più significative e nell'ordine più efficace. Per la natura di questo articolo la componente tecnica non può essere approfondita, ma è importante non dimenticarla. In altre parole, la buona composizione può essere di grande aiuto nell'analisi delle varianti, ma non è sufficiente per garantire il risultato più valido per il lettore.

L'analisi delle varianti può essere:
1) incorporata nel testo corrente;
2) intercalata o immersa nella linea principale;
3) costruita come un elenco fuori testo.

Il **caso 1)** è ristretto a varianti molto brevi e si può considerare un caso particolare, estremamente semplificato, dell'analisi intercalata o immersa nella linea principale (caso 2)).

Il **caso 2)** è preferibile se si hanno poche diramazioni. L'analisi è intercalata tra una porzione e l'altra della linea principale quando questa è fuori linea; è immersa nella linea principale quando questa è in linea.

Il **caso 3)** si applica soltanto se la linea principale è fuori linea ed è preferibile se si hanno molte diramazioni. L'elenco dev'essere studiato in modo che, se si prolunga, non abbia un impatto negativo sull'identificazione della linea principale.

Naturalmente nello stesso testo si possono avere, successivamente, sia il caso 2), sia il caso 3).

In entrambi i casi la sezione di analisi può essere aperta o chiusa da una breve porzione di testo; si usa il tondo, salvo che nelle parentesi lunghe (vedi più avanti), e di norma il corpo è uguale a quello del testo corrente e della linea principale. Un corpo minore, a meno che non sia utilizzato sistematicamente, può trasmettere l'idea errata che quell'analisi abbia un'importanza minore rispetto alle altre.

I problemi comuni ai casi 2) e 3) sono:
- *identificazione delle varianti*, in particolare delle alternative a una data mossa;
- *disposizione delle varianti*, con la scelta tra parallelismo (non richiede parentesi) e annidamento (richiede parentesi).

Per quanto riguarda l'*identificazione delle varianti*, è evidente che lo scrittore può ricorrere alle tecniche da lui preferite (come in un libro qualsiasi), tuttavia è sicuramente un buon risultato se si riesce, grazie a qualche accorgimento tipografico, a

facilitare la separazione delle varianti, riflettendo la struttura dell'albero ed evitando ambiguità. Gli accorgimenti a cui si fa spesso ricorso (grassetto, corsivo, colore diverso, sottolineatura) non sono privi di inconvenienti, oltre a essere impiegati non sempre in modo coerente.

Per quanto riguarda la *disposizione delle varianti*, possono sorgere esigenze contrastanti. Qualche volta la chiarezza (che suggerirebbe il parallelismo, ossia l'uso di un identificatore della variante) deve cedere il passo al contenuto tecnico. Infatti vi sono varianti secondarie che non ha senso collocare in parallelo ad altre: a differenza del parallelismo, le parentesi stabiliscono automaticamente una gerarchia.

Come indicazione generale per l'uso delle parentesi (di regola tonde, molto più raramente quadre), sono accettabili le *parentesi brevi*; sono da limitare le *parentesi lunghe*; sono da evitare nel modo più assoluto le *parentesi annidate*.

Le *parentesi brevi* sono appropriate soprattutto nelle circostanze che seguono.

1) Si vuole indicare un'alternativa alla mossa corrente: 1. Cd5 (migliore di 1. Cb5) | (da considerare 1. Cb5) | (o anche 1. Cb5).
2) Si vuole indicare una continuazione immediata della mossa corrente: 1. Cd5 (seguito da 2. Cc7) | (minacciando 2. Cc7) | (preparando 2. Cc7); la numerazione della mossa tra parentesi può essere omessa.
3) Si vuole riportare la confutazione immediata di un'alternativa alla mossa corrente: 1. — Rf8 (se 1. — Rh8, allora 2. Dh7 matto) | 1. — Rf8 (non è possibile 1. — Rh8 per 2. Dh7 matto); qui la numerazione delle mosse è obbligatoria perché si tratta di una variante vera e propria, sia pur brevissima.

Lo stesso criterio vale nel caso di varianti secondarie più lunghe, che può essere conveniente "incapsulare", ossia non mettere in parallelo alla variante che si sta sviluppando. È inevitabile far ricorso a *parentesi lunghe* (ma non troppo, al più 4-5 mosse, preferibilmente senza diramazioni).

All'interno delle parentesi, siano esse brevi o lunghe, lo stile è forzatamente sintetico, ma la sintassi non dev'essere troppo maltrattata. Per esempio: 1. — Rf8 (se 1. — Rh8, allora 2. Dh7 matto) è migliore di: 1. — Rf8 (se 1. — Rh8; 2. Dh7 matto).

Le *parentesi annidate* devono essere evitate perché pregiudicano gravemente la comprensione, anche se in qualche modo si riesce a differenziare le parentesi più interne.

Nell'analisi delle varianti si utilizzano i livelli dal secondo in poi nella gerarchia di uno *stile di composizione* (§4). I livelli devono essere differenziati in base a qualche caratteristica tipografica. Tra le innumerevoli soluzioni la più semplice – valida sia per il caso 2), sia per il caso 3) – è questa:

- **livello 2** (in tondo): le varianti ordinarie, dentro o fuori elenco, e tutto quanto contenuto nelle parentesi brevi;
- **livello 3** (in corsivo): le varianti contenute nelle parentesi lunghe.

Vi sono casi in cui non è facile decidere se una parentesi è breve o lunga; l'uniformità può allora prevalere su altre considerazioni, come si vede negli esempi che seguono.

Per il **caso 2)** (analisi intercalata o immersa) si propone l'impiego sistematico di un *elenco numerato in linea* che serve soprattutto per rendere più ordinata la successione delle varianti, come mostra quest'esempio di struttura: (a) variante 1; (a.1) diramazione 1; (a.2) diramazione 2; (a.3) diramazione 3; (b) variante 2; (c) variante 3.

L'inizio della struttura a elenco non coincide necessariamente con l'inizio della sezione di analisi: può avvenire che fino a un certo punto si abbia una variante unica, che incomincia a diramarsi più avanti. La comparsa dell'elenco indica, appunto, l'inizio delle diramazioni. Se vi sono parentesi (più o meno brevi) vanno inserite dove occorre.

L'uso delle lettere (a) è più conveniente dei numeri (1) per evitare equivoci con il numero della mossa che segue immediatamente. La forma (b.1) è preferibile a (b1). È inutile evidenziare le lettere e i numeri dell'elenco con il grassetto o il corsivo. È anche inutile evidenziare la prima mossa di una variante con il grassetto o il corsivo, perché le mosse, "appoggiate" alle parentesi dell'elenco, si distinguono senza bisogno di artifici.

Per il **caso 3)** (analisi fuori testo) si propone l'impiego sistematico di un *elenco numerato fuori linea*, con più livelli gerarchici[5], alternando numeri e lettere, come mostra quest'esempio di struttura.

(1) primo livello 1;
  (1.a) secondo livello 1;
  (1.b) secondo livello 2;
    (1.b.1) terzo livello 1;
      • ultimo livello
    (1.b.2) terzo livello 2;
  (1.c) secondo livello 3;
(2) primo livello 2;
  (2.a) secondo livello 1;
  (2.b) secondo livello 2;
(3) primo livello 3.

Questa soluzione è senza dubbio preferibile alle parentesi annidate o al corsivo variamente mescolato al tondo, che si incontrano spesso e generano solo confusione.

L'elenco può anche occupare una pagina intera ed essere corredato di diagrammi (§7). A ogni livello della struttura si possono avere parentesi più o meno brevi, come nel caso 2).

5. Per evitare ambiguità rispetto ai livelli dello stile di composizione, nel seguito del paragrafo i livelli gerarchici sono denominati "livelli della struttura".

Al livello più alto della struttura si possono avere lettere invece di numeri, come nel caso 2). La forma (1.b.1) è preferibile a (1b1). Come nel caso 2), è inutile evidenziare con il grassetto o il corsivo i numeri e le lettere dell'elenco, come pure la prima mossa di ogni variante.

Se la struttura comprende più di tre livelli, si continua con la stessa tecnica di alternare numeri e lettere: (1.a.1.a). Per alleggerire la struttura, si può stabilire che nell'ultimo livello la numerazione sia sostituita da un contrassegno, come nel caso del circoletto nero dell'esempio alla pagina 6.

L'elenco può prevedere rientri successivi, a partire dal secondo livello della struttura. In presenza di colonne molto strette i rientri possono essere ridotti (come nel succitato esempio per il caso 3 alla pagina 6) o addirittura aboliti.

Vi sono diverse soluzioni alternative per la numerazione, che sembrano però meno chiare di quella proposta. In realtà il difetto principale riscontrabile in moltissimi testi sono le variazioni ingiustificate tra un elenco e l'altro, tanto nella numerazione, quanto nei rientri ai diversi livelli della struttura.

Nei testi che trattano la teoria delle aperture, dove l'analisi delle varianti è al centro dell'attenzione, sembra ragionevole ammettere, a scopo didattico, qualche forma più estrosa (linea principale colorata, cinque o più livelli nella struttura gerarchica, prima mossa in grassetto, ecc.), purché sia mantenuta la massima coerenza.

Esempio per il caso 2)
(Pirc – Stolz, Saltsjoebaden 1948, adattato da Euwe (1980), p. 102)

18. — Rg8

Anche le varianti che seguono avrebbero presentato i loro inconvenienti: (a) 18. — Df6; 19. D×f6+; (a.1) 19. — R×f6; 20. Ce4+ Rg7; 21. Cc5, con guadagno di Pedone per il Bianco (...); (a.2) 19. — e×f6, e il Bianco ha praticamente un Pedone in più (...); (b) 18. — f6, indebolendo considerevolmente il Pedone e7 e rendendo impossibile la spinta e6 (...).

19. Dh4 Tfe8?

Questa mossa equivale all'abbandono. Era corretto: 19. — Cc8, sebbene il Bianco dopo 20. Ce4 De5 (se 20. — Db8, allora 21. Cg5 h5; 22. A×h5) 21. d6 C×d6 (se 21. — e×d6??, allora 22. Cf6+) ottenga con 22. C×d6 D×d6; 23. Tad1 una posizione migliore. (...)

20. Ce4 De5; 21. d6

Esempio per il caso 3)
(Smyslov – Reshevsky, Campionato del Mondo 1948, adattato da Euwe (1980), p. 97)

24. ♗b6

(...) Vi è ora la minaccia di 25. ♖d2 e 26. ♕d1, seguita dalla conquista del Pedone d6. Contro tutto ciò il Nero non ha grandi cose, come dimostreranno le seguenti varianti.

(1) 24. — ♘cd8; 25. ♖d2
  (1.a) 25. — f6; 26. ♗×e6 ♘×e6; 27. ♕d1;
  (1.b) 25. — ♖c8; 26. ♕d1 ♖c6; 27. ♗a5
    (1.b.1) 27. — f6; 28. ♗×e6 ♘×e6; 29. ♗b4 (...);
    (1.b.2) 27. — b6; 28. ♗b4 ♘b7; 29. ♗d5;
    (1.b.3) 27. — ♘c5; 28. ♗d5 ♖c8; 29. b4
      • 29. — ♘d7; 30. ♗×d8, con cattura del Pedone b;
      • 29. — ♘ce6; 30. ♗×e6, con cattura del Pedone d;
(2) 24. — ♖c8; 25. ♖d2
  (2.a) 25. — ♘b8; 26. ♗×e6 f×e6; 27. ♕d1 (...);
  (2.b) 25. — ♘cd8, cfr. (1.b).
(3) 24. — ♖e8; 25. ♖d2, e il Pedone d6 deve cadere.

24. — ♘b8

7 Diagrammi

I diagrammi sono di gran lunga la forma di grafica che si incontra più frequentemente nei testi scacchistici.

La distinzione più importante è tra:

- diagrammi che contengono una posizione (di una partita, di uno studio, di un problema): riportano sempre per intero la scacchiera;
- diagrammi didattici: sono estremamente variabili come contenuto; talvolta riportano soltanto una parte della scacchiera.

I diagrammi possono avere vari formati, da scegliere anche in base alla giustezza disponibile. Il formato *grande* è più adatto a testi didattici; il formato *medio* al testo corrente e alla linea principale; il formato *piccolo* alle analisi e alle raccolte di quiz ed esercizi (per motivi di ingombro è quello scelto per gli esempi contenuti in questo articolo).

Se il testo è composto su colonna unica, il diagramma può essere: (a) centrato; (b) circondato dal testo (a sinistra o a destra); (c) affiancato a un altro diagramma. Nei casi (b) e (c) è evidente il proposito di risparmiare spazio. Se il testo è composto su due colonne, il diagramma di regola è centrato, con qualche eccezione nelle sezioni di analisi delle varianti.

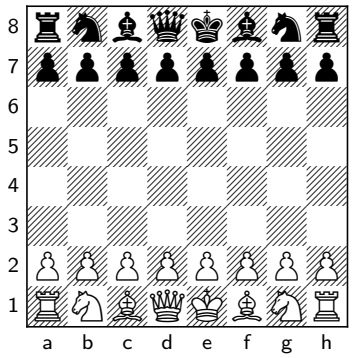
I diagrammi sono quasi sempre accompagnati da scritte, collocate sopra o sotto o di lato. Si può andare da scritte molto semplici (anche il solo numero del diagramma) a scritte molto complesse (§8).

L'identificazione delle colonne (a-h) e delle traverse (1-8) è certamente più utile nei testi didattici; di solito le colonne vanno al di sotto del diagramma e le traverse a sinistra, come nel diagramma 1 del §4 e in altri esempi di questo paragrafo.

Una possibile variabilità è nelle forme dei pezzi: per esempio nel diagramma 1 (§4) è utilizzato il tipo Skaknew; nel diagramma 2 un altro tipo

particolarmente diffuso (Alpha). L'argomento sarà ripreso nel §11.

Diagramma 2



Altre possibili variabilità riguardano il contorno della scacchiera, il colore dei pezzi, il colore e lo sfondo delle case e altri effetti grafici (trasparenza, opacità); sono tutte caratteristiche che non aggiungono granché alla comprensione, anzi sono spesso dannose, tranne nei diagrammi didattici. Per semplicità tutti gli esempi contenuti in questo articolo sono privi di effetti.

Come già accennato, i diagrammi possono essere dotati di un riferimento numerato, e quindi, in teoria, sono *oggetti mobili*. La mobilità potrebbe essere utile perché il diagramma, come qualsiasi figura, non può essere spezzato; può darsi il caso, perciò, che sia impossibile includerlo interamente nella pagina corrente. La difficoltà consiste nel collocarlo nel punto migliore, senza lasciare aree bianche e senza stravolgere il testo. In pratica, la mobilità di un diagramma scacchistico è fortemente limitata dal fatto che il lettore si aspetta di trovarlo *immediatamente dopo* (o al più *immediatamente prima*) rispetto al testo che lo introduce. Un diagramma nella colonna a fianco non è particolarmente gradito al lettore; ancor meno uno nella pagina seguente. È quindi impensabile spostare i diagrammi all'inizio della pagina o raccoglierli in una *page of floats*.

I requisiti ragionevoli per la loro mobilità potrebbero essere i seguenti:

1) il diagramma sia collocato *preferibilmente dopo* il testo che lo introduce, o al più *immediatamente prima*;
2) il diagramma sia collocato tra l'inizio e la fine della partita a cui si riferisce, ovvero all'interno della sezione di analisi delle varianti;
3) il diagramma sia collocato, salvo casi eccezionali, nella stessa pagina del testo che lo introduce.

Per uniformità si potrebbe stabilire che tutti i diagrammi siano numerati e richiamati esplicitamente nel testo. L'appesantimento del testo sarebbe minimo (per esempio, sarebbe sufficiente scrivere "(diagramma 125)" o anche "(125)"), e si avrebbe il vantaggio di poter far riferimento a qualsiasi diagramma in qualsiasi punto del testo.

In pratica però, come è abbastanza raro che *nessun* diagramma sia numerato, così è abbastanza raro che *tutti* i diagrammi siano numerati.

La ragione principale è che all'interno del testo raramente è necessario richiamare un diagramma lontano dal punto in cui compare, per esempio in un altro capitolo. Quando il diagramma è vicino e non vi è possibilità di equivoco, non è indispensabile richiamarlo esplicitamente nel testo, oppure è sufficiente richiamarlo in modo generico (per convenzione (D) o (Diagramma)). Inoltre è possibile utilizzare le scritte associate (§8) per supplire alla mancanza di numerazione del diagramma. In particolare, all'interno delle sezioni di analisi la numerazione e il richiamo sono ininfluenti ai fini della comprensione; a nessuno verrà mai in mente di richiamare quel diagramma al di fuori di quella sezione. Si potrebbero pertanto assimilare i diagrammi non numerati a figure prive di didascalia.

La numerazione dei diagrammi, se presente, è collocata in alto o a lato del diagramma. Si utilizzano sistematicamente i numeri arabi; di regola la numerazione è unica, senza distinzioni di parti o capitoli, ovvero ricomincia da 1 a ogni parte o capitolo senza indicazione esplicita (es. 4 e non 2.4). L'assenza di univocità si spiega con la scarsa frequenza di riferimenti incrociati, già segnalata. Talvolta può essere utile distinguere 1.A rispetto a 1.B, per esempio se si intende mostrare due posizioni aventi un'origine comune. Le forme più frequenti sono: 25. | Diagramma 25 | N. 25; più rara: 25. – <testo>. Quando occorre, il numero può essere accompagnato da "Esercizio", "Problema" e simili.

Anche i *diagrammi didattici* possono essere costruiti su una scacchiera completa; tuttavia, per maggiore evidenza e per risparmiare spazio, può essere conveniente rappresentare solo una parte della scacchiera. In questo caso è consuetudine contornare il diagramma soltanto in corrispondenza dei confini reali della scacchiera. In considerazione degli intenti didattici, ci si può sbizzarrire con colori diversi, con frecce, con segni (di solito cerchi) che evidenziano i pezzi critici.

Si riportano alcuni esempi di diagrammi didattici.

Diagramma 3
Rosa del Cavallo

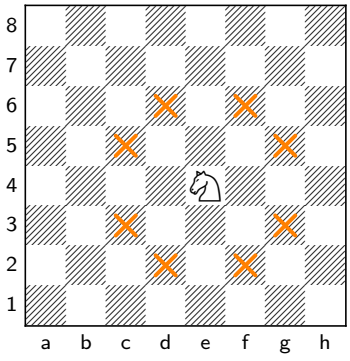


Diagramma 4
Campo di azione della Donna

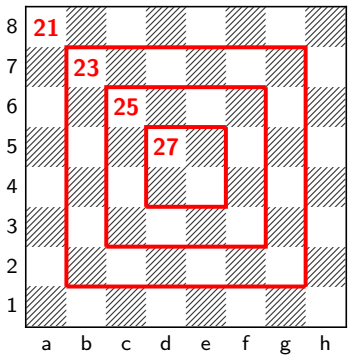


Diagramma 5
Infilata Re-Donna

Diagramma 6
Posizione di matto

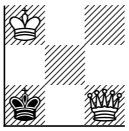
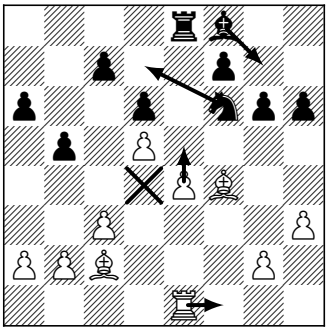


Diagramma 7
Linee strategiche



8 Scritte associate ai diagrammi

Si è già accennato nel §7 al fatto che i diagrammi sono quasi sempre accompagnati da scritte, fortemente variabili come complessità; i diagrammi didattici del §7 ne mostrano alcuni esempi elementari.

Le scritte devono essere agganciate rigidamente al diagramma a cui si riferiscono; pertanto, anche se sono efficaci nell'attirare l'attenzione del lettore, conviene non caricarle troppo per non aumentare le difficoltà di composizione della pagina. In altre parole: talvolta è già complicato collocare il diagramma; la presenza di una scritta non può che peggiorare le cose.

Rispetto al diagramma le scritte, quando presenti, si possono trovare sopra, sotto o a lato; troppo eterogenee per essere assimilate a didascalie. Inoltre, diversi diagrammi possono avere scritte uguali o molto simili, il che rende inutile la composizione di un *indice dei diagrammi*. In linea di principio, le scritte laterali dovrebbero essere alternative rispetto alle scritte superiori o inferiori. Infatti, se si decide di sfruttare i lati del diagramma per risparmiare spazio (e anche per semplificare la composizione), sembra corretto collocare lì *tutte* le informazioni, anche se non obbligatoriamente nello stesso lato.

Le principali caratteristiche tipografiche delle scritte sono le seguenti:

- il tipo di carattere è quello del testo corrente, più raramente quello dei titoli o della linea principale;
- il corpo di solito è minore rispetto al testo corrente;
- sono abbastanza frequenti varianti quali grassetto, maiuscoletto e corsivo; il grassetto sembra appropriato (così in molti degli esempi di questo articolo); le altre varianti decisamente meno.

La scritta dev'essere centrata rispetto al diagramma, con una possibile eccezione per la scritta in basso se si estende su più righe: in questo caso tutte le righe, tranne l'ultima che è centrata, sono giustificate.

Oltre al numero del diagramma, le scritte possono contenere:

- i nomi dei giocatori (entrambi in alto; uno in alto, uno in basso; entrambi in basso);
- informazioni "di contesto": torneo o incontro / turno o partita / luogo e anno (in alto o in basso);
- altre informazioni tecniche concise (in alto o in basso), quali, a titolo puramente esemplificativo:
  - Muove il B | il N
  - Il B | Il N muove e vince | patta
  - Matto in 3 mosse
  - Posizione dopo la 33ª mossa del Bianco
  - Posizione finale
  - Conclusione dello studio
  - Premio di bellezza

Altri commenti o quesiti riguardanti il diagramma si possono trovare nelle scritte sottostanti; la collocazione in basso è la più razionale perché presumibilmente vi è una ripresa immediata nel testo che segue. È impossibile produrre una casistica completa, ma quel che importa osservare è che queste scritte possono essere costituite da una o più righe brevi e indipendenti, ovvero da un testo unico; in quest'ultimo caso si può applicare la soluzione indicata in precedenza, con le prime $n-1$ righe giustificate e l'ultima centrata, dal momento che le scritte centrate di due o più righe consecutive non sono particolarmente gradevoli.

Le informazioni che possono andare a lato del diagramma sono molto ridotte, in quanto lo spazio disponibile in orizzontale è piuttosto scarso, anche nell'ipotesi più favorevole (diagramma unico su tutta la larghezza della pagina). È vero che lo spazio disponibile in verticale è sicuramente sufficiente per diverse righe, ma la lettura potrebbe risultare scomoda. In pratica è raro trovare qualcosa più del numero del diagramma e dell'indicazione "Muove

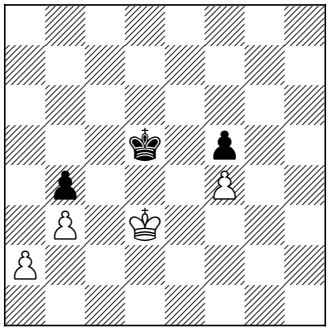
il B | il N". I simboli □ | ■ (equivalenti a: Muove il B | il N) sono logicamente a fianco del campo del giocatore corrispondente, quindi è buona norma collocarli nel lato opposto al numero del diagramma, a meno di non ritornare alla collocazione del numero in alto (è una soluzione ibrida in contraddizione con quanto detto sopra).

Le scritte collocate di lato pongono anche qualche problema di centratura. La centratura dev'essere applicata al solo diagramma, indipendentemente dalle scritte. Questa soluzione è la più semplice ed evita qualsiasi oscillazione in senso orizzontale tra un diagramma e l'altro.

Gli esempi che seguono mostrano alcune possibilità.

Esempio 1 (Euwe, 1980, p. 24)

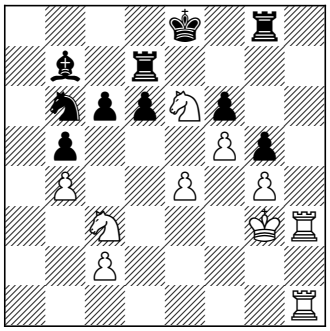
Diagramma 8



Un Pedone arretrato annulla il valore di un vantaggio numerico.

Esempio 2

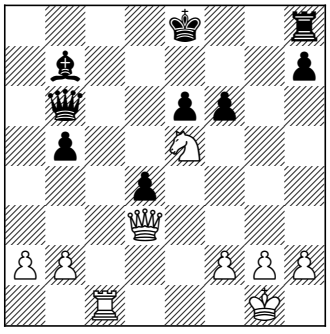
9.
Capablanca



Lasker
Muove il Bianco

Esempio 3 (Botvinnik – Euwe, 1948)

N.10



□

9 Errori e difetti

Dopo aver passato in rassegna le caratteristiche generali e gli elementi più importanti dei testi scacchistici, si riprendono le considerazioni su errori e difetti iniziate nel §2.

Per quanto riguarda gli *errori*, si possono distinguere:

- errori nella numerazione delle mosse;
- errori nella scrittura delle mosse (sul pezzo; sulle case; sui segni o le scritte o i simboli);
- errori nella costruzione dei diagrammi (il diagramma appartiene a un'altra partita; il diagramma non corrisponde al punto del testo o dell'analisi a cui si riferisce; le scritte associate sono errate, ecc.).

Per quanto riguarda i *difetti*, si richiamano brevemente quelli evidenziati nel corso della trattazione, ricordando che il peso di ciascuno di essi è variabile in funzione dei diversi requisiti:

- distanze orizzontali (rientro della linea principale e del testo intercalato) irregolari;
- distanze verticali irregolari tra testo corrente, linea principale, sezione di analisi, diagramma, tali da rendere meno immediata l'associazione corretta delle varie sezioni del testo;
- segni superflui nella notazione (virgole e punti e virgola);
- stili di composizione con troppi livelli e con varianti di carattere (grassetto, corsivo, ecc.) arbitrarie o superflue;
- variazioni ingiustificate nel formato della linea principale (normale, grassetto, corsivo, ecc.);
- variazioni ingiustificate nel formato delle analisi (identificazione dei livelli, rientri per gli elenchi fuori linea, uso delle parentesi);
- variazioni ingiustificate nella numerazione e nelle scritte associate ai diagrammi.

A ben guardare, una parte consistente di questi difetti nasce dall'esigenza di risparmiare spazio, quindi sono soluzioni valide quelle che eliminano i difetti senza provocare un aumento eccessivo dello spazio occupato.

Tutti questi difetti sono legati alle porzioni specificamente scacchistiche del testo. Accanto a questi vi possono essere altri difetti di composizione, più generali, che non si considerano in questo articolo, ma che sarebbe sbagliato dimenticare (si può anzi supporre che i difetti generali siano ancor più numerosi di quelli specifici).

10 Strumenti a disposizione con LaTeX

10.1 Un po' di storia

Prima di esaminare gli strumenti che attualmente LaTeX mette a disposizione di chi vuole comporre un testo scacchistico, può avere qualche interesse ricordare brevemente i precursori.

Le strade di TEX/LATEX e degli scacchi hanno incominciato a incrociarsi abbastanza presto. La prima testimonianza risale al 1988 ed è l'articolo di Wolfgang Appelt, *Typesetting Chess*, pubblicato su TUGboat (Appelt, 1988).

In questo articolo Appelt descrive alcune macro, tra cui le due più importanti sono: `\move` per inserire e per stampare le mosse; `\showboard` per produrre il diagramma con la posizione raggiunta. L'autore conclude che TEX può essere usato per la composizione integrata di testo e diagrammi e che il risultato ottenuto, anche se non ancora di livello professionale (nei diagrammi mancano veri simboli per i pezzi!), è un buon punto di partenza.

Negli anni successivi (1989-1990) appaiono su TUGboat alcuni contributi, dedicati prevalentemente alla creazione e al perfezionamento dei simboli per i diagrammi.

Si arriva così ad una pietra miliare: la nascita del pacchetto chess, opera di Piet Tutelaers e poi integrato con i contributi di altri autori (Frank Hassel, Sebastian Rahtz, Michel Goossens). La rev. 1.0 è datata gennaio 1990; il pacchetto è stato aggiornato fino alla rev. 1.3 (agosto 1995), che lo ha adattato a LATEX 2_ε. Prima dell'uscita del pacchetto skak (§10.2) chess era l'unico strumento valido per comporre testi scacchistici con LATEX, ed è tuttora disponibile in CTAN!

La prima descrizione delle prestazioni di chess si trova nell'articolo *A Font and a Style for Typesetting Chess* (Tutelaers, 1992). L'autore si sofferma dapprima sui font da lui sviluppati con METAFONT, quindi sull'insieme di macro che permettono: di aprire una partita; di definire una posizione qualsiasi; di inserire le mosse sia nella notazione completa, sia nell'abbreviata. È possibile distinguere la linea principale dalle varianti e anche inserire alcuni simboli dell'Informatore. L'esempio riportato nell'articolo è di buon livello qualitativo.

10.2 Pacchetti di LATEX

I pacchetti di LATEX sviluppati espressamente per la composizione di testi scacchistici possono essere suddivisi, per comodità, in tre gruppi.

Il **Gruppo I** comprende i pacchetti generali per la redazione del testo e la costruzione dei diagrammi. Sono cinque:

- skak (versione 1.0, giugno 2002; versione attuale: 1.5.2, gennaio 2014)
- xskak (versione 1.0, dicembre 2007; versione attuale: 1.4, gennaio 2015)
- chessboard (versione 1.0, anteriore a giugno 2006; versione attuale: 1.7, maggio 2014)
- chessfss (versione 1.0, marzo 2005; versione attuale: 1.2a, gennaio 2008)
- texmate (versione 1, marzo 2005; versione attuale: 2, luglio 2006)

Non essendo possibile, per motivi di spazio, un'analisi accurata delle caratteristiche di ciascuno di questi pacchetti, ci si accontenta di una rapida sintesi, attenta soprattutto al processo di evoluzione. Per gli eventuali approfondimenti è opportuno consultare la documentazione a corredo.

Per primo è comparso skak, creato da Torben Hoffmann, che ha esteso in modo sostanziale le prestazioni di chess (§10.1), specialmente nei riguardi della composizione del testo scacchistico (linea principale e analisi delle varianti): prevede una sintassi più leggibile per inserire la posizione iniziale e le mosse; rende possibile salvare e recuperare la posizione corrente; offre più stili di composizione. È ancora un pacchetto relativamente semplice, che si impara a usare in poco tempo: il manuale (Hoffmann, 2013) è di sole 13 pagine ed è accompagnato da una guida di riferimento (purtroppo non aggiornata). Consente di comporre testi abbastanza complessi, accompagnati da diagrammi convenzionali.

I pacchetti xskak e chessboard, entrambi opera di Ulrike Fischer, hanno ripreso e ampliato le funzioni di skak: xskak (Fischer, 2015) ha esteso la capacità di composizione del testo scacchistico; chessboard (Fischer, 2014) la capacità di produzione dei diagrammi.

Più precisamente, xskak è in grado di leggere un formato "sorgente" più complesso rispetto a skak; di salvare una grande mole di informazioni sull'avanzamento della partita che possono poi essere sfruttate per vari scopi, per esempio gli effetti di animazione; di definire un numero praticamente illimitato di stili di composizione.

Quanto a chessboard, consente di avere un pieno controllo del contenuto, delle dimensioni e della forma del diagramma. È possibile costruire manualmente il diagramma pezzo per pezzo; definire margini, contorni, "etichette" esterne; colorare o evidenziare una o più case; stampare anche parzialmente la scacchiera o nascondendo una o più case; applicare frecce, segni, scritte di molte forme diverse. Inoltre è possibile definire una serie di caratteristiche comuni a uno o più diagrammi e applicare queste definizioni a tutti i diagrammi che seguono. Quel che invece chessboard non fa è stampare le scritte associate al diagramma o agire sulla sua collocazione nella pagina.

Mentre xskak continua ad appoggiarsi a skak per alcune funzioni fondamentali (in particolare la scansione delle mosse), chessboard lo ha sostituito in tutto e per tutto. Quindi è corretto parlare di skak/xskak (insieme) e di chessboard.

Entrambi i pacchetti sono molto ricchi e molto complessi; i manuali non sempre sono di facile comprensione, e neppure di facile consultazione. Inoltre molte opzioni, soprattutto nel caso di chessboard, sono di utilità modesta o addirittura, se applicate scriteriatamente, possono portare a risultati disastrosi.

Si deve poi considerare chessfss (Fischer, 2006), anch'esso opera di Ulrike Fischer, che fornisce un

meccanismo generale per selezionare i font da utilizzare nel testo e nei diagrammi (il nome è evidentemente ispirato al LaTeX NFSS) e, in più, permette di passare con disinvoltura dalla notazione con lettere alla notazione con simboli. Questo pacchetto sarà ripreso nel §11.4.

Invece texmate (García, 2006), che pure ha delle caratteristiche interessanti, si può considerare superato. Non può essere utilizzato se non in unione a skak e chessfss; il risultato è all'incirca allo stesso livello di complessità ottenibile con skak. La coesistenza con xskak e chessboard è abbastanza problematica, e alla fine le sue caratteristiche originali possono essere emulate da skak/xskak e chessboard.

I pacchetti che si possono considerare vivi, e che non a caso sono quelli utilizzati negli esempi dei paragrafi precedenti, sono dunque in massima parte dovuti a Ulrike Fischer, che per di più ha dato contributi anche a skak. Le date delle ultime versioni riportate sopra non devono però trarre in inganno: se si leggono con attenzione i manuali si vede che dopo il 2008 quasi tutta l'attività è consistita nella correzione di errori. Non sembrano quindi probabili evoluzioni significative, almeno a breve termine.

Nel **Gruppo II** sono inclusi classi e pacchetti specializzati per alcuni compiti particolari. Ecco due esempi, per altro collegati tra loro.[6]

- schwalbe-chess → schwalbe (classe + pacchetto): versione attuale 2.1, gennaio 2016; serve per la composizione della rivista tedesca *Die Schwalbe*, dedicata ai problemi.
- chess-problem-diagrams → diagram: versione attuale 1.12, gennaio 2016; serve per la generazione di diagrammi contenuti nei problemi.

Nel **Gruppo III** sono inclusi i "contenitori" di font, quali: skaknew, enpassant, bartel-chess-fonts, per i quali si rimanda al §11.

Un rapido cenno, infine, ai pacchetti generici che possono intervenire nella composizione dei testi scacchistici. Sono da segnalare almeno enumitem, di capitale importanza per la costruzione di elenchi complessi, e pgf/tikz. Quest'ultimo è già richiamato all'interno di chessboard, ma per alcuni compiti particolari potrebbe essere conveniente utilizzarlo direttamente, ossia in aggiunta a chessboard.

10.3 LaTeX Graphics Companion

Sin dalla prima edizione (Goossens *et al.*, 1997), il *LaTeX Graphics Companion* dedica un intero capitolo ai giochi (Capitolo 8 – *Playing Games*), e naturalmente al primo posto si trovano gli scacchi (pp. 277-291). L'argomento principale è chess e le sue estensioni. Un paragrafo riguarda l'interfacciamento con database scacchistici e uno il font Cheq di Adobe.

6. Per evitare ambiguità si usa la scrittura a → b, intendendo che "a" sia il nome del "contenitore" in MiKTeX e TeX Live, "b" il nome del pacchetto o della classe.

Nella seconda edizione (Goossens *et al.*, 2007) gli scacchi occupano nuovamente il primo paragrafo del capitolo 10, dedicato ai giochi (pp. 668-687). Sono trattati con una certa ampiezza tutti i pacchetti citati nel §10.2, tranne xskak che è nato più tardi.

10.4 Fonti di informazione

In questa panoramica di strumenti disponibili non possono mancare le fonti di informazione *on-line*. Sono tutte molto estemporanee e informali, ma consentono di capire meglio i problemi incontrati da chi si cimenta con la composizione di testi scacchistici, e talvolta forniscono soluzioni immediatamente applicabili (tra l'altro, quando i quesiti riguardano xskak, chessboard, chessfss, molte risposte provengono dall'autrice stessa, Ulrike Fischer).

Le principali sono:

- TeX – LaTeX Stack Exchange (`https://tex.stackexchange.com`);
- LaTeX Community (`http://latex.org/forum/index.php`);
- comp.text.tex (`https://groups.google.com/forum/#!forum/comp.text.tex`); ed eventualmente anche de.comp.text.tex e fr.comp.text.tex.

Inoltre si può tener presente il sito *Chess Stack Exchange* (`https://chess.stackexchange.com`), in verità rivolto a tutti gli aspetti del gioco, non solo alla composizione di testi.

Per i lettori di ArsTEXnica è anche interessante citare un contributo recente (dicembre 2016) nel Forum GUIT ("Testo particolareggiato di analisi scacchistica").

11 Caratteri scacchistici

Nei paragrafi precedenti sono state fatte alcune anticipazioni riguardanti i "caratteri scacchistici" (ossia, i caratteri specifici per i testi scacchistici); in questo paragrafo si espongono in modo più ordinato le questioni principali, anche se per forza di cose non si può andare al di là di una semplice presentazione delle varie questioni.

11.1 Serie di caratteri di interesse

Le serie di caratteri di interesse per i testi scacchistici sono complessivamente cinque:

1) simboli dei pezzi utilizzati nel testo;
2) simboli dei pezzi utilizzati nei diagrammi;
3) simboli dell'Informatore;
4) caratteri utilizzati per indicare le mosse;
5) caratteri utilizzati per identificare colonne e traverse.

Ci si concentra sulle prime tre serie, rimandando per la serie 4 ai paragrafi 4-5-6 e per la serie 5 al paragrafo 7.

I simboli dei pezzi utilizzati per la scrittura delle mosse (*figurine characters*) sono stati introdotti nel §4 con la notazione scacchistica. Sono sufficienti i 6 simboli bianchi (o meglio, corrispondenti

ai pezzi bianchi), che talvolta possono avere le varianti medio e grassetto. I simboli neri sono raramente utilizzati e in realtà sono superflui; anzi, se combinati malamente, possono produrre risultati sgradevoli.

I simboli dei pezzi utilizzati nei diagrammi (*board characters*) sono stati introdotti nel §7 con i diagrammi. In totale vi sono 24 simboli (6 pezzi × 2 colori × 2 sfondi), a cui bisogna aggiungere le 2 case vuote. Talvolta sono presenti i simboli ruotati di 90° o di 180°. La presenza di simboli ruotati sembra interessante, ma può portare a risultati ridicoli se si mescolano inavvertitamente le varie forme nello stesso diagramma.

È chiaro che le due serie di simboli devono essere armonizzate nel disegno. Alla fine si assomigliano a tal punto che si può correre il rischio di confondere i simboli dei pezzi bianchi per il testo con i simboli dei pezzi bianchi su sfondo bianco per i diagrammi. Questi ultimi hanno però tutti la stessa larghezza e sono leggermente sollevati rispetto alla linea di base, e quindi non sono adatti per la composizione del testo.

I simboli dell'Informatore, di cui sono stati mostrati alcuni esempi nel §4, sono sostanzialmente indipendenti dalle altre due serie. Molti simboli sono esclusivamente grafici; altri invece sono legati al tipo in uso per il testo. L'elenco completo dei simboli (circa 50 in tutto) si trova nella pagina *System of Signs* dell'Informatore (`http://www.chessinformant.org/pages.php?pageid=15`) e nel manuale del pacchetto chessfss (Fischer, 2006, p. 25).

11.2 Ne servono tanti o ne bastano pochi?

Se si sfogliano libri o riviste pubblicati in tempi e in paesi diversi, non si può non rimanere colpiti dalla grande varietà delle forme che assumono i simboli dei pezzi nei diagrammi. Questa variabilità nel disegno dei pezzi risalta maggiormente nei testi pubblicati prima del 1990; nei testi più recenti è evidente la tendenza a una maggiore uniformità.

Si trovano così decine di forme diverse, opera di disegnatori molto abili e molto fantasiosi. Per averne un'idea più precisa si possono esaminare la raccolta *The chess fonts gallery* allegata alla documentazione del pacchetto chessfss (Fischer, 2006), che comprende 25 esempi, e il documento *Chess Fonts* (`http://www.chessdiagrammer.com/download/zip/chessfonts_samples.pdf`), allegato al programma **Chessdiagrammer** (§12.4), che comprende 30 esempi.

A prima vista, quindi, la scelta della soluzione più appropriata sembra piuttosto difficile. Tuttavia i margini di libertà si restringono alquanto, sia perché le differenze tra alcuni tipi sono veramente lievissime (per esempio Mérida rispetto a Skaknew o 1echecs o ChessOle! o Smart Regular), sia perché molti tipi sono improponibili in un testo di natura tecnica, in quanto renderebbero la lettura faticosa, o addirittura impossibile. In definitiva, al di là di preferenze dovute al gusto individuale, le soluzioni applicabili si riducono a Mérida (e simili), Alpha, Good Companion, Kingdom, Condal.

11.3 Formati e codici

Le serie di caratteri scacchistici compaiono in diversi formati, ovvero:

- TrueType: senza dubbio il più comune;
- Adobe Type1: limitato a qualche esempio;
- Metafont: da considerare soprattutto per ragioni "storiche";
- OpenType: tuttora rappresentato da pochissimi esempi.

Nel seguito si approfondisce il formato TrueType, mentre qualche cenno a Type1 e a Metafont è contenuto nel §11.4.

Prima di tutto, dove si trovano i file `.ttf`? Il punto di riferimento indiscutibile (e quasi unico) è il sito *EnPassant* del Nørresundby Chess Club (Danimarca) (vedi anche §12). Purtroppo non tutti i font elencati nella pagina *Chess Fonts* (`http://www.enpassant.dk/chess/fonteng.htm`) sono effettivamente accessibili!

Inoltre alcuni dei programmi esterni a LaTeX che saranno discussi nel §12 citano o rendono disponibili altri font in formato `.ttf`:

- **epd2diag**: in aggiunta a molti font presenti nel sito *EnPassant*, cita alcuni font commerciali (Hastings, Linares, Zurich) della Alpine Electronics (`https://www.partae.com/fonts`);
- **Chessdiagrammer**: mette a disposizione dell'utente 6 font non accessibili dal sito *EnPassant*;
- **ChessTask**: fornisce il font DBchess, che è anche disponibile nel formato Type1.

In conclusione, unendo le varie fonti accessibili si arriva a 30-35 tipi diversi. Questi file `.ttf` mostrano notevoli differenze uno rispetto all'altro, quali:

- presenza di gruppi di caratteri diversi: sono sempre presenti i simboli dei pezzi per i diagrammi e per il testo bianchi e gli identificatori di colonne e traverse con i contorni della scacchiera; *i simboli dell'Informatore possono essere presenti o no*, così come (ma sono assai meno utili) i simboli dei diagrammi ruotati e i simboli per il testo neri;
- collocazione dei simboli nel campo di valori 00-FF; si incontrano diversi schemi, non uno solo comune a tutti come ci si aspetterebbe;
- presenza di caratteri Unicode da U+2000 in avanti (solo per alcuni font).

La mancanza dei simboli dell'Informatore è grave perché obbliga a ricorrere al tipo del testo, ma, come si vede nel seguito, neppure questo ne è sempre provvisto.

Oltre ai font TrueType "scacchistici" (cioè contenenti esclusivamente caratteri scacchistici) bisogna ricordare che font generici possono contenere caratteri Unicode utili per la composizione dei testi scacchistici. In particolare, Unicode definisce:

- Simboli dei pezzi per il testo (bianchi e neri): U+2654 – U+265F
- Simboli dell'Informatore: variamente compresi in U+20xx, 21xx, 22xx, 23xx, 25xx, 27xx, 2Axx

Una tabella con simboli e valori si trova nella pagina *Numeric Annotation Glyphs* di Wikipedia.

Tra i font TrueType dotati di simboli dei pezzi (U+2654 e seguenti) si possono citare Arial Unicode MS o Symbola, mentre mancano, per esempio, in Lucida Sans Unicode. È difficile, pertanto, fare previsioni sulla loro presenza, così come sulla presenza dei simboli dell'Informatore.

11.4 Quali caratteri usare con LaTeX?

Per esplorare il mondo TeX/LaTeX conviene partire dai pacchetti chess (§10.1) e skak (§10.2). Entrambi avevano al loro interno ("hardwired") un font in formato Metafont, avente lo stesso nome del pacchetto. (Incidentalmente si può notare che skak consentiva già di introdurre con la massima facilità i simboli dell'Informatore.) Tutto bene, dunque, ma – nel caso l'utente avesse desiderato impiegare altri font in formato Type1 già disponibili – praticamente non ne avrebbe avuto la possibilità.

La situazione è cambiata con il pacchetto chessfss (§10.2), il quale consente di selezionare qualsiasi font in formato Type1 presente nel sistema. Prima di tutto, chessfss definisce come default non più Skak, ma Skaknew, che rappresenta una conversione di Skak (con alcune variazioni e correzioni) dal formato Metafont al formato Type1. Skaknew è disponibile in un "contenitore" indipendente con lo stesso nome.

Inoltre Ulrike Fischer si è accollata il compito di convertire un certo numero di font da TrueType a Type1 per renderli selezionabili mediante chessfss. Il risultato di questo lavoro è confluito in un altro "contenitore", chiamato enpassant (il nome è evidentemente collegato al sito citato nel §11.3). Purtroppo non tutti i font reperibili sono stati inclusi, e per alcuni la conversione non è completa, dal momento che manca il file `.pfb`. Oltre a Skaknew, attualmente sono utilizzabili soltanto Alpha, Berlin, Cheq, Pirat, Utrecht.

Per allargare la platea l'utente deve perciò provvedere per conto proprio, seguendo il procedimento descritto nel capitolo 10 del manuale di chessfss (Fischer, 2006). Sarebbe interessante capire se non vi sia un modo più semplice per arrivare allo stesso risultato.

Come soluzione alternativa, si potrebbe pensare di evitare la conversione di formato ricorrendo a un'estensione di TeX come XeTeX, in grado di accedere direttamente ai font residenti nel sistema. Anche se l'irregolarità della struttura dei file `.ttf` scacchistici, già evidenziata, non è un elemento a favore, sembra ragionevole ritenere che con XeTeX si possa utilizzare qualsiasi file `.ttf`, ma in questo caso si renderebbe necessario un adattamento di chessfss.

L'interesse di queste operazioni non è esclusivamente concettuale, in quanto consentirebbero di recuperare un certo numero di tipi adatti anche alla composizione di testi complessi.

12 Applicazioni esterne

Lo scopo di questo paragrafo è di offrire una rassegna, sia pur schematica, di programmi esterni rispetto a LaTeX, in grado di contribuire a migliorare la qualità dei testi scacchistici.

Rispetto alla vastità del SW scacchistico, si considera pertanto un sotto-insieme ristretto, escludendo, per esempio, i motori di analisi e i server per il gioco *on-line*, che non interessano per la composizione dei testi.

Può riuscire utile, almeno per un primo orientamento, una classificazione del SW in gruppi (più o meno omogenei):

- Gruppo I – Programmi per la manipolazione di file descrittivi della posizione e della partita
- Gruppo II – Programmi per la costruzione di diagrammi
- Gruppo III – Programmi di supporto alla composizione dei testi
- Gruppo IV – Programmi per la costruzione di database
- Gruppo V – Database operativi

Le distinzioni tra i gruppi non sono sempre così nette; spesso vi sono funzioni comuni a più di un gruppo. Inoltre nella classificazione sono da tenere in conto anche altre caratteristiche come la piattaforma HW, il sistema operativo, il linguaggio di programmazione utilizzato, e le consuete alternative: SW gratuito o commerciale; pronto all'uso o da compilare; da installare o fruibile via web; dotato o no di interfaccia grafica.

Per quanto riguarda la possibilità di reperire materiale in rete, si segnalano i siti *EnPassant* (`http://www.enpassant.dk/chess/homeeng.htm`), già citato nel §11.3, e *Chess Poster* (`http://www.chess-poster.com`). In genere le applicazioni hanno una pagina dedicata e spesso anche una pagina in SourceForge (`https://sourceforge.net`).

Questo paragrafo è così articolato. Inizialmente (12.1-12.2) si descrivono i formati standard FEN e PGN, utili per rappresentare rispettivamente una posizione o l'intera partita, e le loro interazioni con i pacchetti di LaTeX esaminati nel §10.2; in particolare si vuole mettere in rilievo che cosa serve per alimentare un documento LaTeX. Successivamente (12.3-12.5) si forniscono alcuni esempi di programmi appartenenti ai Gruppi I, II, III, cioè a quelli di maggior rilevanza per la composizione dei testi.

12.1 FEN

FEN (Forsyth-Edwards Notation) è una notazione che permette di descrivere in modo codificato la posizione di una partita. Per semplicità si chiama FEN anche il file di testo che contiene la posizione codificata: di solito è costituito da una sola linea e ha estensione `.fen`. Dal FEN si può risalire al numero di mossa e a quale dei due giocatori ha il tratto, oltre che ad altre informazioni più specializzate, ma non al nome dei giocatori, al luogo e alla data della partita, ecc.

La diffusione del formato FEN è universale; una possibile evoluzione è il formato EPD (Extended Position Description).[7]

Come esempi di FEN si riportano quello della posizione iniziale e quello corrispondente all'esempio 3 del §8:

```
[rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR
w KQkq - 0 1]
[4k2r/1b5p/1q2pp2/1p2N3/3p4/3Q4/PP3PPP/2R3K1
w - - 0 0]
```

Tranne gli ultimi campi, questi file si riescono a decifrare quasi senza spiegazioni.

Il FEN è utile per avviare la linea principale da un determinato punto o per produrre un diagramma, ma non consente di ricostruire l'andamento della partita.

Il FEN può essere generato:

- costruendolo manualmente, per esempio a partire da un diagramma cartaceo contenuto in un libro o in una rivista; le probabilità di errore sono altissime, ma con la pratica tendono a ridursi;
- acquisendolo da una delle innumerevoli fonti disponibili (un database *on-line*, ma anche un sito web generico o un supporto informatico qualsiasi);
- producendolo per mezzo di applicazioni specifiche.

È importante notare che skak ha un generatore di FEN "integrato": in qualsiasi punto si può dare il comando `\savegame{alpha}` e generare immediatamente il file `alpha.fen`.

Per quanto riguarda la lettura, skak ha i comandi `\fenboard...` (diretto) e `\loadgame{alpha}` (da file), e analogamente xskak e chessboard, sia per avviare la linea principale, sia per produrre un diagramma.

Un lettore di FEN *stand-alone* è importante se serve per produrre un diagramma (§12.4); è meno importante se serve semplicemente per visualizzare la posizione, in quanto si tratta di un'informazione statica.

Un correttore di FEN potrebbe essere utile, ma ovviamente sono rilevabili pochi errori non puramente formali, e pochissimi sono correggibili con sicurezza.

7. L'EPD ha i primi quattro campi in comune con il FEN, seguiti da un numero variabile di "operazioni", due delle quali coincidono con i campi 5 e 6 del FEN. Nei paragrafi 12.3-12.4 si citano diversi programmi in grado di accettare anche questo formato, che però in pratica non è molto diffuso.

12.2 PGN

PGN (Portable Game Notation) è un formato utilizzato per registrare una partita completa (mosse e informazioni accessorie). Anche per questo formato si suole chiamare con lo stesso nome il file di testo corrispondente (di solito con estensione `.pgn`). Un file `.pgn` può contenere più partite, addirittura migliaia o decine di migliaia.

Il formato PGN ha una diffusione universale, anche se si scontra con la concorrenza di formati proprietari, fortunatamente di minore importanza ai fini di questo articolo.

Come esempio di PGN si riporta quello corrispondente alla partita dell'esempio 1 del §13.

```
[Event "Portoroz Interzonal"]
[Site "Portoroz SLO"]
[Date "1958.08.16"]
[EventDate "1958.08.05"]
[Round "8"]
[Result "1-0"]
[White "Robert James Fischer"]
[Black "Bent Larsen"]
[ECO "B77"]
[WhiteElo "?"]
[BlackElo "?"]
[PlyCount "61"]
1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6
5. Nc3 g6 6. Be3 Bg7 7. f3 O-O 8. Qd2 Nc6
9. Bc4 Nxd4 10. Bxd4 Be6 11. Bb3 Qa5
12. O-O-O b5 13. Kb1 b4 14. Nd5 Bxd5
15. Bxd5 Rac8 16. Bb3 Rc7 17. h4 Qb5 18. h5 Rfc8
19. hxg6 hxg6 20. g4 a5 21. g5 Nh5 22. Rxh5 gxh5
23. g6 e5 24. gxf7+ Kf8 25. Be3 d5 26. exd5 Rxf7
27. d6 Rf6 28. Bg5 Qb7 29. Bxf6 Bxf6 30. d7 Rd8
31. Qd6+ 1-0
```

Anche in questo caso non occorrono quasi spiegazioni. Tuttavia, se si tiene conto del gran numero di possibili informazioni accessorie, soprattutto quelle intercalate alle mosse, lo standard PGN risulta assai complesso.

Il PGN può essere generato:

- costruendolo manualmente, per esempio a partire da un formulario di torneo o da un libro o da una rivista; anche in questo caso le probabilità di errore sono altissime, ma con la pratica tendono a ridursi;
- acquisendolo da una delle innumerevoli fonti disponibili (un database *on-line*, ma anche un sito web generico o un supporto informatico qualsiasi);
- producendolo per mezzo di applicazioni specifiche; i due casi tipici sono molto simili:
  - estrazione di una porzione di un file `.pgn` contenente più partite (l'estrazione potrebbe anche essere fatta manualmente, ricadendo così nel caso precedente, ma è un'operazione più complicata);
  - interrogazione (*query*) a un database: si può estrarre dal database un file `db1.pgn` con tutte

le partite di un certo giocatore in un certo anno, poi un file `db2.pgn` contenente una singola partita, e così via.

Il PGN è una fotografia della partita statica e dinamica al tempo stesso: perciò è utile sia per visualizzare una determinata posizione, sia per rivedere tutta la partita in forma animata.

Un visualizzatore di PGN si trova frequentemente in un database o in un generico sito web. Più precisamente, si tratta di un visualizzatore dell'intera partita che si appoggia al file `.pgn` della partita.

Data la grande eterogeneità delle fonti, ogni applicazione che importi un PGN dev'essere "permissiva", ossia tollerare fin dove possibile gli errori formali; ogni applicazione che esporti un PGN dev'essere "restrittiva", ossia conformarsi pienamente alle regole.

Un correttore di PGN può essere di grande utilità, anche se, come per il FEN, sono rilevabili pochi errori non puramente formali, e pochissimi sono correggibili con sicurezza. Forse, più che per eliminare errori, il correttore serve per fare pulizia, togliendo i campi superflui, ma senza incidere sulla successione delle mosse.

Dal PGN si può anche pensare di estrarre altre informazioni. Per esempio, il *titolo esteso* di cui si è parlato nel §5 si può costruire, almeno in parte, direttamente dal PGN.

Per skak/xskak il formato PGN è importante nella scrittura della linea principale e delle varianti. È molto semplice fare "copia e incolla" della parte di `.pgn` che interessa e darla in pasto a skak/xskak, con qualche piccolo adattamento quando si ha un'interruzione dopo la semi-mossa del Bianco e tenendo presente che skak/xskak coprono soltanto un sotto-insieme dello standard PGN. Non vi sono invece interazioni tra il PGN e chessboard.

12.3 Programmi per la manipolazione dei file descrittivi (Gruppo I)

Elencate le principali funzioni richieste ai programmi che operano sui file descrittivi della posizione e della partita, si riportano alcuni esempi rappresentativi.

- **FEN/EPD Viewer**: visualizzatore (*on-line*) di FEN/EPD;
- **LT PGN File Viewer**: visualizzatore (*on-line*) di PGN;
- **40H-EPD Utility**: collezione di 24 programmi di utilità (correzione, riordinamento, estrazione di informazioni, ecc.) per file EPD;
- **pgnutils**: collezione di 20 programmi di utilità per file PGN;
- **40H-PGN Utility**: collezione di 53 programmi di utilità per file PGN;
- **pgn-extract**: programma per la pulizia, la ricerca di errori e soprattutto l'estrazione – secondo i criteri più vari – di porzioni di un file `.pgn`.

Per quanto riguarda la conversione tra formati standard, quella che interessa è da PGN a FEN/EPD. A questo proposito si possono citare **pgn2fen** e **LT PGN to FEN Converter** (quest'ultimo *on-line*). Entrambi producono un unico file `.fen` con n righe distinte; **pgn2fen** può anche produrre un file `.epd` con la stessa struttura.

12.4 Programmi per la costruzione di diagrammi (Gruppo II)

I programmi di questo gruppo consentono di produrre un diagramma:

- da istruzioni testuali o click del mouse: caso abbastanza frequente;
- dal FEN: caso abbastanza frequente; infatti questa è una scelta logica, sia che si abbia come origine: (a) un libro o una rivista in formato cartaceo (dalla fonte cartacea si genera il FEN e si trasferisce al costruttore di diagrammi); (b) un PGN (dal PGN si genera il FEN che interessa e poi si procede come nel caso (a));
- direttamente dal PGN: caso meno frequente.

Di solito i costruttori di diagrammi mostrano il diagramma sullo schermo prima di salvarlo. È quindi possibile che questi costruttori siano anche visualizzatori di FEN/EPD o PGN (Gruppo I).

Il diagramma è prodotto in un formato grafico da incorporare nel documento LaTeX con `\includegraphics`. I formati più comuni sono `.bmp`, `.gif`, `.jpg`, `.png`, `.eps`.

Alcuni esempi rappresentativi:

- **fen2eps** (è l'unico a produrre il diagramma in formato `.eps`)
- **epd2diag**
- **Chessdiagrammer**
- **Chess Diagram Editor**

L'utente può scegliere tra molte opzioni, assai diverse da un programma all'altro; in particolare può selezionare il font desiderato tra quelli disponibili, o anche aggiungerne di nuovi.

Questi costruttori sono un'alternativa al generatore di diagrammi "integrato", disponibile con skak e chessboard, e anche con texmate.

Vantaggi e svantaggi sono gli stessi delle figure costruite nativamente in LaTeX o per mezzo di applicazioni esterne, come spiegato in Pantieri e Gordini (2017, paragrafo 6.4, p. 103 e sgg.) e Beccari *et al.* (2017, capitoli 9-10).

12.5 Programmi di supporto alla composizione di testi (Gruppo III)

Ricordando quanto detto nel §2 circa l'esigenza di accelerare la composizione dei testi, riducendo al tempo stesso il più possibile gli errori imputabili alla composizione manuale, questo gruppo dovrebbe essere il più importante.[8]

8. Alcune interessanti considerazioni sull'argomento si trovano anche nel manuale di xskak (Fischer, 2015, pp. 48-51).

In realtà sono stati trovati solo due esempi significativi, e nemmeno recenti. Visti i magri risultati di questa ricerca, eventuali segnalazioni dei lettori sarebbero graditissime.

Il primo programma, opera di Dirk Baechle, è **Pgn2ltx** (`http://pgn2ltx.sourceforge.net`, rev. 1.1, 2003), che trasforma un file `.pgn` in un file LaTeX, o meglio in una porzione di file LaTeX. Sembra che sia in grado di coprire tutto lo standard PGN, ma il suo limite fondamentale (per altro inevitabile, non essendo più stato aggiornato) è che si appoggia solo sul pacchetto skak, e quindi non può sfruttare tutte le possibilità aggiuntive offerte da xskak.

Il secondo programma è **ChessTask** (`http://chesstask.sourceforge.net`, rev. 2.0, 2004), anch'esso opera di Dirk Baechle. È adatto alla preparazione di testi didattici, in particolare esercizi con molti diagrammi. Agisce come *front-end* per LaTeX, permettendo all'utente di assemblare, per mezzo di mouse e tastiera, un file intermedio (`.tsk`) che contiene la posizione di partenza e le successive, il titolo e il testo dell'esercizio, la soluzione (che può essere inclusa o no, eventualmente al termine della serie di esercizi) e di esportare il risultato in formato TeX/LaTeX (o in alternativa HTML). Anche **ChessTask** ha come limite fondamentale il fatto di appoggiarsi solo su skak. Alcuni esempi di applicazione di **ChessTask** sono riportati nell'articolo Baechle (2007).

13 Esempi di partite complete

In questo paragrafo si propongono due esempi di partite complete, che riprendono molti degli argomenti trattati e mirano a rafforzare quanto è stato discusso a proposito dei risultati ottenibili con LaTeX.

Rispetto ai paragrafi precedenti, questi esempi sono certamente un banco di prova ben più serio per le soluzioni applicate; infatti molte soluzioni si rivelano poco adatte non appena il testo diventa più lungo e complesso.

La prima partita è tratta dal celeberrimo *My Sixty Memorable Games* (Fischer, 1972, pp. 12-17); la seconda dal *Mammoth Book* (Burgess *et al.*, 2004, pp. 133-138).

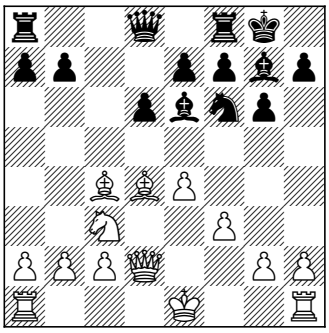
Rispetto agli originali, entrambe le partite sono state semplificate soprattutto nelle parti testuali e modificate, là dove necessario, per mostrare meglio le soluzioni applicate. Sarebbe interessante (ma purtroppo impossibile) affiancare originale e copia, in modo da apprezzare le differenze, e in qualche caso anche i miglioramenti introdotti.

Esempio 1

Fischer – Larsen
Torneo Interzonale – Portorose 1958
Ottavo turno – Difesa Siciliana

1. e4 c5; 2. Cf3 d6; 3. d4 c×d4; 4. C×d4 Cf6; 5. Cc3 g6; 6. Ae3 Ag7; 7. f3 O-O; 8. Dd2 Cc6; 9. Ac4 C×d4; 10. A×d4 Ae6 (diagramma 1)

1.
Larsen



Fischer
Posizione dopo la 10ª mossa del Nero

11. Ab3 Da5; 12. O-O-O b5; 13. Rb1 b4; 14. Cd5

Più debole risulta 14. Ce2, a causa di 14. — A×b3; 15. c×b3 Tfd8.

14. — A×d5

Non va bene 14. — C×d5 per 15. A×g7 R×g7; 16. e×d5 Ad7; 17. Tde1, con netto vantaggio per il Bianco (Suetin – Korčnoj, Semifinale del Campionato dell'URSS, 1953).

15. A×d5

Più forte è 15. e×d5! Db5; 16. The1 a5; 17. De2! (Tal' – Larsen, Zurigo, 1959): il Bianco tralascia l'attacco per giocare sulla debolezza della colonna *e*.

15. — Tac8?

La mossa che perde. Al termine della partita Larsen spiegò che voleva giocare per vincere e che aveva quindi scartato l'idea della inevitabile patta, che si sarebbe verificata dopo 15. — C×d5; 16. A×g7 Cc3+; (a) 17. A×c3 b×c3; 18. D×c3 D×c3; 19. b×c3 Tfc8, e il Pedone in più del Bianco è reso inutile; (b) 17. b×c3 Tab8!; 18. c×b4 D×b4+!; 19. D×b4 T×b4+; 20. Ab2 Tfb8 ecc. Comunque dopo 15. — C×d5 io avevo intenzione di giocare semplicemente 16. e×d5 D×d5; 17. D×b4, mantenendo le possibilità di complicazione.

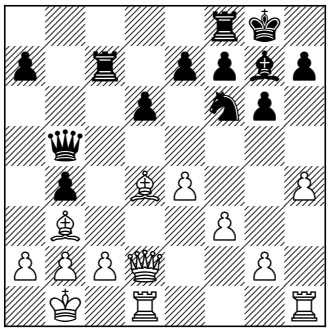
16. Ab3! Tc7; 17. h4 Db5 (diagramma 2)

Non vi è alcun modo soddisfacente per impedire l'attacco del Bianco. Se 17. — h5, allora 18. g4!, e quindi:

(1) 18. — h×g4; 19. h5! g×h5 (se *19. — C×h5*, allora *20. A×g7 R×g7; 21. f×g4 Cf6; 22. Dh6+* e matto alla seguente); 20. f×g4
  (1.a) 20. — C×e4; 21. De3 Cf6 (se *21. — A×d4*, allora *22. D×e4 Ag7; 23. T×h5*); 22. g×h5 e5; 23. h6, e vince.
  (1.b) 20. — h×g4; 21. Tdg1 e5; 22. Ae3 Td8; 23. Ah6 ecc.
  (1.c) 20. — C×g4; 21. Tdg1 A×d4; 22. T×g4+! h×g4; 23. Dh6, con matto in poche mosse.

(2) 18. — Tfc8; 19. Tdg1 h×g4; 20. h5! g×h5; 21. f×g4 C×e4; 22. Df4 e5; 23. D×e4 e×d4; 24. g×h5 Rh8; 25. h6 Af6; 26. Tg7!, e vince.

2.
Larsen



Fischer
Posizione dopo la 17ª mossa del Nero

Con la mossa del testo il Nero cerca di organizzare un certo controgioco con la spinta a7-a5-a4.

18. h5 Tfc8

A 18. — g×h5 segue: 19. g4! h×g4; 20. f×g4! C×e4; 21. Dh2 Cg5; 22. A×g7 R×g7; 23. Td5 Tc5; 24. Dh6+ Rg8; 25. T×g5+ T×g5; 26. D×h7 matto.

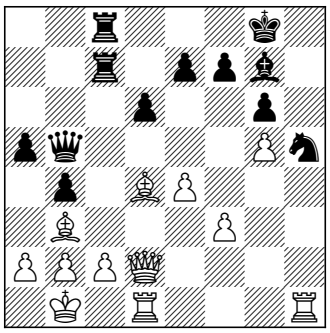
19. h×g6 h×g6; 20. g4

Sbagliata e precipitosa 20. A×f6? perché con 20. — A×f6; 21. Dh6 e6! (minacciando 22. — De5) il Nero difende tutto.

20. — a5; 21. g5 Ch5 (diagramma 3)

Vasjukov suggerisce come possibile difesa 21. — Ce8 (non *21. — a4?* per *22. g×f6 a×b3; 23. f×g7 b×c2+; 24. D×c2! e5; 25. Dh2*, e vince); ma il Bianco sfonda rovinosamente con 22. A×g7 C×g7 (a *22. — R×g7* segue *23. Dh2*); 23. Th6! e6 (a *23. — a4* segue *24. Dh2 Ch5; 25. T×g6+*); 24. Dh2 Ch5; 25. A×e6! f×e6 (a *25. — D×g5* segue *26. T×g6+! D×g6; 27. A×c8*, minacciando *28. Tg1*); 26. T×g6+ Cg7; 27. Th1 ecc.

3.
Larsen



Fischer
Posizione dopo la 21ª mossa del Nero

22. T×h5! g×h5

Non migliora le cose 22. — A×d4 per 23. D×d4 g×h5; 24. g6 De5 (se *24. — e6*, allora *25. D×d6*); 25. g×f7+ Rh7 (se *25. — Rf8*, allora *26. D×e5 d×e5; 27. Tg1 e6; 28. A×e6 Re7; 29. A×c8 T×c8; 30. Tg5*, e vince); 26. Dd3!, e la minaccia di spinta in f4 risulta decisiva.

23. g6 e5

Se 23. — e6, allora 24. g×f7+ R×f7 (24. — T×f7; 25. A×e6); 25. A×g7 R×g7; 26. Tg1+ Rh7; 27. Dg2 De5; 28. Dg6+ Rh8; 29. Tg5 Tg7; 30. T×h5+ Rg8; 31. A×e6+ Rf8; 32. Tf5+ Re7; 33. Tf7+, con facile vittoria.

24. g×f7+ Rf8; 25. Ae3 d5!

Un disperato sacrificio nel vano tentativo di salvarsi. Dopo 25. — a4 (25. — Td8; 26. Ah6) si ha: 26. D×d6+ Te7; 27. Dd8+! T×d8; 28. T×d8+ Te8; 29. Ac5+ D×c5; 30. T×e8 matto.

26. e×d5

E non 26. A×d5 per 26. — T×c2!.

26. — T×f7

A 26. — a4 segue 27. d6! a×b3; 28. d×c7, e vince.

27. d6 Tf6

Dopo 27. — Td7 il Bianco può sia riprendere la qualità con 28. Ae6, sia persistere nell'attacco con 28. Ah6. A 27. — T×f3 seguirebbe invece 28. d7, minacciando 29. Dd6 matto.

28. Ag5 Db7

Se 28. — Dd7, allora 29. Dd5!, e quindi: (a) 29. — Df7; 30. A×f6, guadagnando un pezzo; (b) 29. — Tf7; 30. Ae7+!.

29. A×f6 A×f6; 30. d7 Td8; 31. Dd6+

Un errore! Con 31. Dh6+ si forza il matto in tre mosse. Il Nero abbandona.

Esempio 2

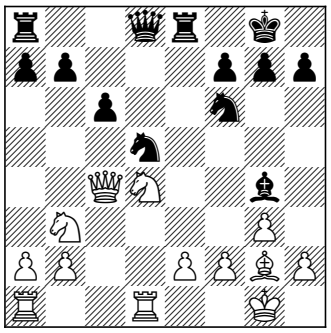
Richard Réti – Alexander Alekhine
Baden Baden 1925
Alekhine Reversed

1. g3 e5; 2. ♘f3 e4; 3. ♘d4 d5; 4. d3 e×d3; 5. ♕×d3 ♘f6; 6. ♗g2 ♗b4+; 7. ♗d2 ♗×d2+; 8. ♘×d2 O-O; 9. c4 ♘a6; 10. c×d5 ♘b4; 11. ♕c4 ♘b×d5; 12. ♘2b3 c6

Aiming to retain Black's main asset, the strongly posted knight on d5. Of course, White can drive the knight away by playing e4, but this would block the action of his bishop on the long diagonal. Réti soon decides to undermine Black's queenside pawn chain by b4-b5, thereby also destabilizing the d5-knight. This is a strong but rather slow plan, and Alekhine is forced to search for counterplay on the opposite side of the board.

13. O-O ♖e8; 14. ♖fd1 ♗g4

Aiming for counterplay against e2.



15. ♖d2

Relatively best, since after 15. h3 ♗h5 Black will gain control of e4 by — ♗g6. The alternative 15. ♘c5 ♕e7 is even worse, as 16. ♖d2? fails to 16. — ♘e3.

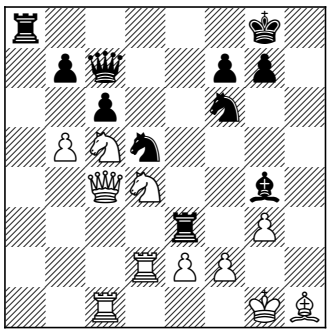
15. — ♕c8

A typical manoeuvre: Black aims for the exchange of light-squared bishops by — ♗h3. The alternative 15. — ♕e7; 16. ♕c5 would favour White, as without queens White would be free to pursue his queenside attack.

16. ♘c5 ♗h3; 17. ♗f3

White cannot grab a pawn: 17. ♗×h3 ♕×h3; 18. ♘×b7 ♘g4; 19. ♘f3 ♘de3; 20. f×e3 ♘×e3; 21. ♕×f7+ ♔h8; 22. ♘h4 ♖f8 would cost him his queen.

17. — ♗g4; 18. ♗g2 ♗h3; 19. ♗f3 ♗g4; 20. ♗h1 h5; 21. b4 a6; 22. ♖c1 h4; 23. a4 h×g3; 24. h×g3 ♕c7; 25. b5! a×b5; 26. a×b5 ♖e3!



Just as White's queenside attack arrives, this spectacular rook sacrifice energizes Black's counterplay.

27. ♘f3?

This is the critical moment of the game. White has several plausible moves to meet the threat of 27. — ♖×g3+ and it is certainly not easy to decide which is the most appropriate. (...)

Here are the alternatives in (roughly) ascending order of merit.

(1) 27. f×e3?? ♕×g3+; 28. ♗g2 ♘×e3 mates.

(2) 27. b×c6? ♖×g3+; 28. ♗g2 (*28. f×g3 ♕×g3+; 29. ♗g2 ♘e3* mates) 28. — ♘e3!; 29. f×e3 ♗h3, and wins.

(3) 27. ♗g2?! ♖×g3; 28. e3 (*28. e4 ♖×g2+; 29. ♔×g2 ♘f4+*, with a very strong attack, e.g. *30. ♔h1 ♕e5; 31. b×c6 ♕h5+; 32. ♔g1 ♗f3*, and wins) 28. — ♘×e3; 29. f×e3 ♕e5, and Black has very good compensation for the piece.

(4) 27. ♗×d5 c×d5; 28. ♕b4, and now Black can force a draw by 28. — ♖×g3+ or play on by 28. — ♖ee8, with an unclear position.

(5) 27. ♗f3 ♗×f3; 28. e×f3 c×b5; 29. ♘×b5 ♕a5; 30. ♖×d5 ♖e1+; 31. ♖×e1 ♕×e1+; 32. ♔g2 ♘×d5 (*32. — ♖a1?; 33. ♖d8+ ♔h7; 34. ♕h4+ ♔g6; 35. f4*, and White wins) 33. ♕×d5 ♖a1; 34. ♕d8+ ♔h7; 35. ♕h4+ ♔g8, with perpetual check.

(6) 27. ♔h2! ♖aa3!, and now:

(6.a) 28. ♘db3 ♖e5, and Black's pieces are very active, whereas 28. — ♕e5; 29. f×e3 ♕h5+; 30. ♔g1 ♕h3; 31. ♗×d5 ♕×g3+; 32. ♔h1 ♕h3+ is a draw.

(6.b) 28. ♘cb3 ♕e5, with a further branch:

(6.b.1) 29. f×e3 ♕h5+; 30. ♔g1 ♕h3; 31. ♗×d5 ♘×d5! (better than forcing an immediate draw) 32. ♘f3 ♕×g3+; 33. ♔h1 ♗×f3+ (*33. — ♘×e3; 34. ♖d8+ ♔h7; 35. ♘g5+ ♔h6; 36. ♘×f7+ ♔g6; 37. ♘h8+ ♔h6* is a draw); 34. e×f3 ♕×f3+; 35. ♔h2 ♕×e3!, and White is in difficulties, e.g.

- 36. ♖d3 ♕f2+; 37. ♔h1 ♖a2; 38. ♖d2 ♖×d2; 39. ♘×d2 ♕×d2;
- 36. ♖e2 ♕h6+; 37. ♔g1 ♖×b3;
- 36. ♖×d5 c×d5; 37. ♕c8+ ♔h7; 38. ♕c2+ g6;

with an advantage for Black in every case.

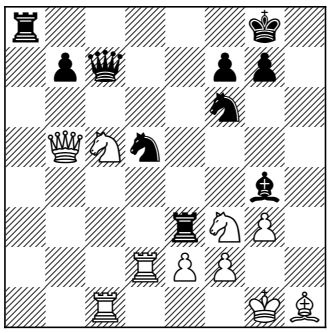
(6.b.2) 29. b×c6 b×c6; 30. f×e3 ♕h5+; 31. ♔g1 ♕h3 (Alekhine stopped his analysis here, implying that Black is better; however, it seems to be a draw); 32. ♗×d5 (*32. ♗f3 ♕×g3+; 33. ♔h1* is also a draw) 32. — ♘×d5; 33. ♘f3 ♕×g3+ (*33. — ♘×e3?; 34. ♖d8+* wins); 34. ♔h1 ♗×f3+; 35. e×f3 ♕×f3+; 36. ♔h2 ♕×e3; 37. ♕×c6 (possible thanks to the preliminary exchange on c6) 37. — ♖×b3; 38. ♕c8+ ♔h7; 39. ♕f5+ ♔h6; 40. ♖c6+ g6; 41. ♖×g6+ f×g6; 42. ♕f8+, with perpetual check.

(6.c) 28. ♘d3! (nobody seems to have considered this move, which again blocks the third rank, but also keeps the black queen out of e5) 28. — ♘h5 (*28. — ♖×g3; 29. f×g3 ♘h5; 30. ♖g1 ♘e3; 31. ♕c1 ♖c3; 32. ♕e1* wins for White, whereas *28. — ♘e4; 29. ♗×e4 ♖×e4; 30. ♕×d5 c×d5; 31. ♖×c7 ♖×d4; 32. ♖×b7 ♗×e2; 33. ♖×e2 ♖a×d3; 34. ♖e8+ ♔h7; 35. ♖ee7* gives White a very favourable ending) (...).

After Réti's choice Black decides the game with a series of hammer blows. White's moves are virtually forced until the end of the game.

27. — c×b5; 28. ♕×b5

The alternative 28. ♕d4 is strongly met by 28. — ♖e4.



28. — ♘c3!; 29. ♕×b7

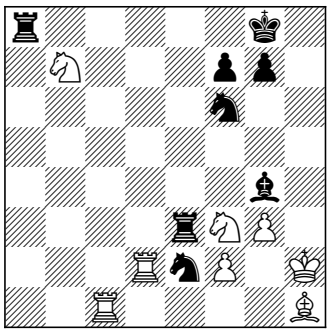
After 29. ♕c4 b5 the queen cannot continue to defend e2.

29. — ♕×b7

A much stronger continuation than 29. — ♘×e2+; 30. ♖×e2 ♕×b7; 31. ♖×e3, when the resulting position offers few winning prospects.

30. ♘×b7 ♘×e2+; 31. ♔h2

Or 31. ♔f1 ♘×g3+; 32. f×g3 ♗×f3; 33. ♗×f3 ♖×f3+; 34. ♔g2 (a) 34. — ♖aa3; 35. ♖d8+ ♔h7; 36. ♖h1+ ♔g6; 37. ♖h3 ♖fb3, with a decisive attack; (b) 34. — ♖×g3+; 35. ♔×g3 ♘e4+ is also effective.



There are several pieces hanging in this remarkable position, but the winning move does not involve taking any of them!

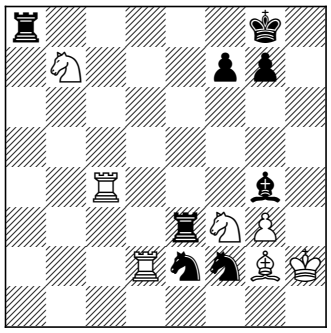
31. — ♘e4!

Instead, 31. — ♖×f3; 32. ♖×e2 ♖×g3; 33. f×g3 ♗×e2 should be a draw.

32. ♖c4

The best defence is 32. ♖d8+ (32. f×e3 ♘×d2 loses at once) 32. — ♖×d8; 33. f×e3, when Black's only clear-cut win is by means of the beautiful continuation 33. — ♖d5!; 34. ♖c4 (White must skewer Black's minor pieces, otherwise 34. — ♖h5+ wins out of hand) 34. — ♘2×g3 (if 34. — ♖h5+, then 35. ♘h4); 35. ♗g2 ♘f1+!!; 36. ♔g1 (after 36. ♗×f1 ♗×f3, White cannot meet the threat of 37. — ♖h5+) 36. — ♖d1; 37. ♗×f1 ♗×f3, with the deadly threat of 38. — ♘d2. Curiously, Alekhine made no mention of 32. ♖d8+.

32. — ♘×f2; 33. ♗g2



33. — ♗e6

Black could have won more simply by 33. — ♘e4; 34. ♖dc2 ♖a6, with the lethal threat of 35. — ♖h6+, but Alekhine's move is also decisive. The remaining moves are forced.

34. ♖cc2 ♘g4+; 35. ♔h3 ♘e5+; 36. ♔h2 ♖×f3; 37. ♖×e2 ♘g4+; 38. ♔h3 ♘e3+; 39. ♔h2 ♘×c2; 40. ♗×f3 ♘d4 (0-1)

White loses a piece after 41. ♖f2 ♘×f3+; 42. ♖×f3 ♗d5.

14 Conclusioni

Questo studio introduttivo dovrebbe aver consentito al lettore di acquisire una conoscenza di massima degli elementi che costituiscono i testi scacchistici e dei principali strumenti attualmente a disposizione con LaTeX.

Negli esempi proposti sono state sfruttate (qualche volta anche in maniera abbastanza ingegnosa) le prestazioni offerte da LaTeX in generale, e in particolare dai pacchetti dedicati (§10.2), ma senza contributi originali in termini di programmazione TeX/LaTeX.

È stato largamente confermato che già con gli strumenti che LaTeX attualmente mette a disposizione è possibile produrre testi di buona qualità.

Per andare oltre occorre:

- trovare nuove soluzioni, più efficaci, per alcuni casi critici;
- perfezionare metodi che consentano di rilevare e, se possibile, correggere un numero maggiore di errori e difetti;
- sviluppare strumenti che contribuiscano a rendere più efficiente il processo di composizione.

Questi tre punti si traducono nella necessità di superare alcune limitazioni degli strumenti attuali; non è da escludere, a tale scopo, qualche modificazione ai pacchetti dedicati.

Per esempio, skak dispone già di alcune funzioni di diagnostica, ma non sono abbastanza complete, e soprattutto sono poco prevedibili, nel senso che skak è in grado di intercettare errori che non ci si aspetterebbe, e viceversa lascia passare errori apparentemente rilevabili con facilità.

Se la composizione di testi scacchistici susciterà interesse potrebbe veder la luce una *guida tematica* sull'argomento, nella quale potrebbero essere ripresi e approfonditi gli argomenti introdotti in questo articolo e potrebbe essere ampliato il campione dei testi analizzati, estendendo la trattazione anche agli articoli pubblicati su riviste e ai prodotti di editoria elettronica. In breve, essa potrebbe diventare, come suggerito dal nome, una vera e propria guida completa alla composizione dei testi scacchistici. Prima di procedere, tuttavia, sembra sensato tener conto del giudizio dei lettori.

L'autore ringrazia sin d'ora chi si farà carico di segnalare lacune e inesattezze, pressoché inevitabili come conseguenza dell'aver condensato la materia entro limiti ragionevoli per un articolo della rivista.

Riferimenti bibliografici

Appelt, W. (1988). «Typesetting Chess». *TUGboat*, **9** (3), pp. 284–287.

Baechle, D. (2007). «Square concepts». *The PracTEX Journal*, (1), pp. 1–10 + 1–14.

Barletta, M. e Messa, R. (2014). *Fabulous Fabiano*. Messaggerie Scacchistiche, Brescia.

Beccari, C. *et al.* (2016). *Saper Comunicare – Cenni di scrittura tecnico-scientifica*. Politecnico di Torino. Versione 2.4.

— (2017). *Introduzione all'arte della composizione tipografica con LATEX*. GuIT. Versione 0.99.16.

Burgess, G., Nunn, J. e Emms, J. (2004). *The World's Greatest Chess Games*. Carroll & Graf, New York, 2ª edizione.

Euwe, M. (1980). *Trattato di scacchi*. Mursia, Milano, 2ª edizione. Trad. P. Bagnoli.

Fischer, B. (1972). *60 partite da ricordare*. Mursia, Milano, 3ª edizione. Trad. A. Capece.

Fischer, U. (2006). *chessfss: A Chess Font Selection Scheme*. Versione 1.2; `texdoc chessfss`.

— (2014). *chessboard: A package to print chessboards*. Versione 1.7; `texdoc chessboard`.

— (2015). *xskak: An extension to the package skak*. Versione 1.4; `texdoc xskak`.

García, F. (2006). *TEXmate2: User's manual*. Versione 2; `texdoc texmate`.

Goossens, M., Rahtz, S. e Mittelbach, F. (1997). *The LATEX Graphics Companion*. Addison-Wesley, Reading, MA, 1ª edizione.

Goossens, M., Mittelbach, F., Rahtz, S., Roegel, D. e Voss, H. (2007). *The LATEX Graphics Companion*. Addison-Wesley, Reading, MA, 2ª edizione.

Grooten, H. (2015). *Il manuale della Strategia Scacchistica*. Le Due Torri, Bologna. Trad. C. A. Veronesi.

Hoffmann, T. (2013). *Typesetting Chess in LATEX with the skak Package*. Versione 1.5.1; `texdoc skak`.

Messa, R. e Mearini, M. T. (2017). *Il gioco degli scacchi*. Messaggerie Scacchistiche, Brescia, 7ª edizione.

Pantieri, L. e Gordini, T. (2017). *L'arte di scrivere con LATEX*.

Tutelaers, P. (1992). «A Font and a Style for Typesetting Chess using LaTeX or TeX». *TUGboat*, **13** (1), pp. 85–90.

▷ Maurizio Molinaro
`maurizio dot molinaro3 at gmail dot com`

Let's Connect LuaTEX to the World

Roberto Giacomelli

Sommario

Una delle migliori novità del mondo TEX è LuaTEX non solamente per la tipografia in sé. Questo potente motore di composizione è capace di recuperare dati da molte sorgenti diverse, una caratteristica molto importante nei contesti di produzione di imprese e studi professionali, dove le informazioni condivise sono una risorsa vitale.

In questo articolo discuterò le soluzioni che ho trovato fino a ora durante la mia attività professionale, per estendere il sistema TEX con funzioni esterne come quelle dei driver SQL. Illustrerò poi l'ultima mia inattesa scoperta, sorprendentemente semplice e innovativa: *connettere il codice tramite messaggi*.

Niente più codice di basso livello di cui prendersi cura, niente più vincoli dovuti alle incompatibilità tra programmi, ma uno strato software fornito dalla libreria ZeroMQ con cui connettere LuaTEX a servizi esterni.

Illustrerò in dettaglio l'installazione di ZeroMQ, e con l'aiuto di esempi funzionanti, mostrerò come usufruire in LuaTEX di servizi costruiti con il linguaggio Rust.

Abstract

One of the best pieces of news in the TEX world is LuaTEX not only for typography itself. In fact, this very powerful typesetting engine is capable to retrieve data from a variety of sources, a very important feature especially in the production context such as business and professional teams, where shared data are a vital resource.

In this paper I will describe several solutions achieved so far during my job, to execute external code like SQL drivers and pass the datasets to TEX. Then, I will introduce the latest point of view that I suddenly discovered a few weeks ago, surprisingly simple and innovative as well: *connect code by messaging*.

No more low level artifacts to care about, no more constraints due to incompatibility between programs, but a communication layer provided by the ZeroMQ library connecting LuaTEX with distributed services.

I will detail how to install ZeroMQ and, with the help of several working examples, I will show how to benefit from services built with Rust within LuaTEX.

1 Introduzione

Per illustrare le tecnologie dei moderni strumenti del sistema TEX con cui è possibile realizzare documenti che fanno parte di processi produttivi, è utile definirne le principali caratteristiche e delineare alcuni punti critici che ne limitano la produzione.

Parliamo di relazioni tecniche, lettere, elenchi di oggetti, istruzioni, frutto del lavoro quotidiano di moltissime persone all'interno di aziende o studi professionali, la cui caratteristica essenziale è la correttezza dei dati che contengono.

In generale, le informazioni dell'organizzazione sono archiviate in qualche forma per poi essere elaborate e incluse all'interno di documenti. I modi in cui i dati vengono resi persistenti e poi utilizzati, ne determinano l'*affidabilità* e l'*efficienza*.

L'affidabilità è la misura della sicurezza di accesso ai dati e il mantenimento del loro stato di correttezza nel tempo, mentre l'efficienza è la misura della rapidità con cui vengono create, reperite e modificate le informazioni. Si tratta di fattori chiave poiché un buon flusso informativo abbassa i costi e incrementa evoluzione e miglioramento dei prodotti.

L'utilizzo di TEX non è quindi quello classico della scrittura di un singolo documento, come una tesi di laurea o un libro, ma la redazione di documenti di carattere aziendale dallo schema predefinito che includono dati condivisi nell'organizzazione.

Nella sezione 2 ripercorrerò i diversi metodi che ho sperimentato con lo scopo di migliorare l'affidabilità e l'efficienza dei dati, elaborando complesse collezioni di dati per generarne documenti con la qualità tipografica del sistema TEX.

Dalla sezione 3 svilupperò una nuova ulteriore idea basata sullo scambio di *messaggi* tra codici in esecuzione in processi indipendenti, introducendo ZeroMQ. Un'ampia unità applicativa incentrata sull'uso di servizi esterni in LuaTEX comprenderà la creazione di simboli QRCode e l'accesso a database attraverso la connessione a *socket* ZeroMQ secondo lo schema client/server.

La seconda parte dell'articolo tratterà delle questioni più tecniche: la sezione 8 riporterà i dettagli d'installazione e le principali risorse di documentazione sui software utilizzati. La sezione 9 sarà dedicata alle funzionalità FFI in LuaTEX e alla creazione di librerie dinamiche. Le tre sezioni successive tratteranno l'implementazione in Rust dei servizi server impiegati in precedenza.

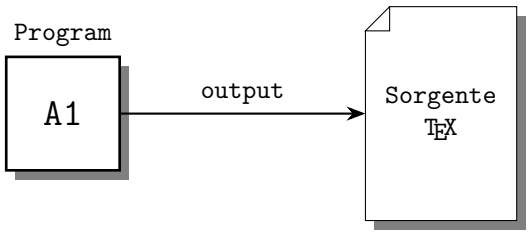


FIGURA 1: Schema di funzionamento della tecnologia di costruzione del sorgente TEX che permette l'indiretta e indipendente elaborazione esterna dei dati.

2 Far colloquiare il codice

Parte del problema più generale descritto nell'introduzione è dare una risposta alla seguente domanda: come implementare il codice che ricava i dati memorizzati all'interno di un archivio informatico e con essi comporre un documento con TEX?

Per riuscirci è essenziale estendere le funzioni di base del motore di composizione con moduli software specifici e il successo di queste tecnologie dipende da quanto sia sicuro far colloquiare questi moduli con TEX.

Le soluzioni che illustrerò con maggior dettaglio nelle prossime sezioni, sono:

- costruzione automatica del sorgente TEX;
- esecuzione di codice nativo Lua;
- esecuzione tramite Lua di codice compatibile C utilizzando l'interfaccia FFI (Foreign Function Interface);
- esecuzione tramite Lua di codice compatibile C utilizzando il binding diretto a librerie a caricamento dinamico;
- esecuzione tramite Lua di codice compatibile C utilizzando librerie a caricamento dinamico conformi all'API di Lua.

2.1 Costruzione del sorgente TEX

Il più semplice metodo d'elaborare i dati di un archivio è quello di scrivere un programma la cui uscita sia il file di testo contenente il codice TEX che una volta compilato dà origine al documento desiderato.

Questa tipo di tecnica è rappresentata nello schema di figura 1. Il programma `A1` si compone essenzialmente di due parti: la prima si occupa di predisporre i dati per la pubblicazione estraendoli dagli archivi, e la seconda di costruire con essi il file sorgente per uno dei formati della famiglia TEX.

Non vi è alcuna necessità di coinvolgere TEX in nessuna di queste due fasi perciò l'utente è completamente libero sia di scegliere il linguaggio di programmazione con cui implementarle, sia di scegliere il motore tipografico con cui sarà compilato il sorgente di output.

Il ruolo del sistema di composizione è secondario poiché l'attenzione è quasi sempre rivolta al codice che genera il sorgente, mentre dal punto di vista del flusso dati il file sorgente generato è in sostanza il modo con cui il programma `A1` *comunica* con TEX, ovvero con una forma statica e unilaterale memorizzata su disco.

Il vantaggio di questo metodo è la completa separazione tra il programma e il sistema TEX, tuttavia l'ulteriore indirezione aggiunta a quella sempre presente tra sorgente `.tex` e documento finale limita l'utente nell'apportare modifiche dovute a cambiamenti nei contenuti del documento o alla volontà di migliorarne l'aspetto.

Un'analisi della tecnica di generazione del sorgente TEX che comprende numerosi esempi, si può trovare nell'articolo (GIACOMELLI e PIGNALBERI, 2015) pubblicato su questa stessa rivista.

2.2 Esecuzione di codice esterno

Fino all'uscita del compositore LuaTEX non vi erano altre strade praticabili per raggiungere lo scopo che non rifarsi alla tecnica precedente di *costruzione del sorgente*.

LuaTEX apre numerosi e ampi scenari poiché, includendo Lua — un linguaggio di tipo dinamico implementato in C —, è in grado di eseguire codice in molti modi diversi. Diviene possibile, durante la compilazione di un sorgente `.tex`, eseguire un numero molto grande di librerie esistenti oppure codice ad alte prestazioni scritto appositamente, secondo lo schema di figura 2.

Tuttavia è fondamentale che l'esecuzione del codice esterno sia affidabile e sicura, caratteristica che però non è affatto scontata.

2.3 Foreign Function Interface

In generale si può definire la Foreign Function Interface — in breve FFI — una tecnologia che facilita a un programma scritto in un dato linguaggio di programmazione l'esecuzione di codice scritto in un altro linguaggio.

LuaTEX offre, o meglio offrirà, funzionalità stabili FFI nel prossimo futuro (si veda l'articolo

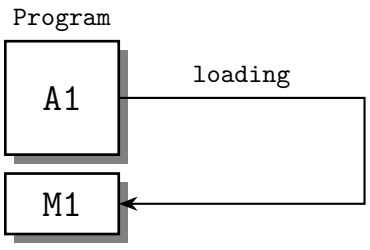


FIGURA 2: Schema generale di funzionamento della tecnologia di esecuzione di codice esterno. Nel programma principale `A1`, Lua carica il modulo `M1` che estende le funzionalità di base con quelle necessarie per l'elaborazione dei dati esterni. Una volta acquisite le informazioni, esse vengono composte nel documento da LuaTEX.

sull'argomento di Hans Hagen e Luigi Scarso (Hagen e Scarso, 2017)) mentre LuajitTEX già ne dispone perché FFI è incluso come estensione in LuaJIT, il compilatore Just In Time per Lua che sostituisce il classico interprete in questo motore di composizione relativamente poco noto agli utenti.

Grazie alla tecnologia FFI è possibile, per esempio, connettersi a database relazionali e comporre i risultati delle query direttamente in un sorgente TEX. L'idea può essere approfondita leggendo Giacomelli (2017), che spiega nel dettaglio l'installazione e la configurazione dei moduli necessari e fornisce esempi applicativi.

Un esempio d'uso di FFI è riportato alla sezione 9, con la compilazione di una libreria dinamica elementare scritta in Rust e l'esecuzione dell'unica funzione contenuta in essa all'interno di LuaTEX.

2.4 L'API di Lua

La seconda parte del libro di Roberto Ierusalimschy su Lua (Ierusalimschy, 2016) descrive l'API del linguaggio, ovvero come sia possibile aggiungere a esso nuove e potenti funzionalità attraverso lo stack virtuale e le relative funzioni in C.

Questo fa di Lua un linguaggio estensibile attraverso il caricamento dinamico di librerie binarie scritte appositamente. Alla sezione 9.2 vedremo un esempio di codice che ne chiarisce le modalità.

Questa terza via all'esecuzione di codice esterno è quella più intimamente legata al funzionamento interno di Lua, tutto incentrato sullo *stack*; questa struttura dati fu ritenuta la migliore soluzione dai progettisti del linguaggio per risolvere i conflitti tecnologici tra Lua e C, il linguaggio d'implementazione.

Grazie allo stack possiamo scambiare dati da e verso Lua dall'ambiente di esecuzione in C, per esempio scrivendo una funzione in C (o in Rust come nell'esempio concreto della sezione 9.2) che crea una tabella Lua. Chiaramente con questa tecnica potremmo scrivere in C un modulo per accedere a un database e renderne disponibili i dati all'interno di LuaTEX.

Nel farlo, dobbiamo occuparci di tutti i dettagli, definire i nomi delle funzioni dell'API di Lua che verranno risolti a runtime, lavorare con lo stack trovando la corrispondenza tra i tipi, quelli del C e quelli di Lua, scrivere la funzione di inizializzazione della libreria, e provare la validità del codice.

Possiamo definire l'operazione *un compito non banale dalle prestazioni eccellenti* sia in termini di tempi di esecuzione sia in termini di occupazione di memoria.

3 Canali di comunicazione

Mentre ragionavo sulle difficoltà tecniche di eseguire codice esterno in LuaTEX mi venne in mente una soluzione che avrebbe aggirato tutte le difficoltà senza rinunciare alla comunicazione bidirezionale tra i due ambienti di esecuzione: si trattava di sfruttare i canali dello standard input e output del sistema operativo come ponte tra due diversi programmi indipendenti.

In altre parole, il codice Lua eseguito in LuaTEX avrebbe avviato un comando esterno tramite la funzione `io.popen()` passando i dati da elaborare come argomenti. Il programma esterno avrebbe stampato i dati di uscita sul terminale così che, al termine della sua esecuzione, Lua avrebbe potuto acquisirli.

A dimostrazione pratica dell'idea, costruiamo il comando `double` con Rust[1] come un programma per il terminale che accetta un numero intero come primo e unico argomento e ne stampa il corrispondente valore doppio[2]:

```
// a simple Rust program to print the double
// of an integer
use std::env;
fn main() {
    let args: Vec<String> = env::args()
        .collect();
    let n: i32 = args[1]
        .parse()
        .unwrap();       // input
    println!("{}", 2*n); // output
}
```

Compiliamo il codice con il comando seguente lanciato in una sessione di terminale la cui directory di lavoro corrisponde a quella del file sorgente:

```
$ rustc double.rs
```

e verifichiamone il funzionamento con l'ulteriore comando:

```
$ ./double 34
68
```

Siamo ora pronti per passare a LuaTEX salvando il seguente sorgente nella stessa directory dell'eseguibile `double`:

```
% !TeX program = LuaTeX--shell-escape
\directlua{
local f = io.popen("./double 123", "r")
local res = f:read("*n")
f:close()
tex.print(res)
}
\bye
```

Aggiungendo l'opzione `--shell-escape` al comando di compilazione per permettere l'esecuzione della funzione `io.popen()`[3], LuaTEX dovrebbe produrre un file PDF contenente il numero 246.

1. La sezione 8.2 è dedicata alla presentazione di questo linguaggio.
2. Per semplicità il codice dell'esempio non gestisce gli eventuali errori dovuti alla non validità dell'argomento.
3. La funzione della libreria interna di Lua `io.popen()` accetta un comando esterno, apre un file virtuale come suo canale di output e ne restituisce un riferimento.

L'idea è molto semplice ma consente la comunicazione bidirezionale tra Lua, che richiede un servizio, e il componente esterno, che esegue quanto richiesto restituendo al mittente il risultato.

3.1 L'entrata in scena di ZeroMQ

Diverso tempo fa io e Luigi Scarso avemmo in chat una conversazione il cui argomento era l'affidabilità in scrittura dei dati su un database SQLite3. Luigi mi propose l'idea di costruire un server attraverso ZeroMQ — una libreria open source che non conoscevo — e far colloquiare diversi processi d'esecuzione attraverso messaggi usando protocolli riconosciuti.

Il server avrebbe reso più sicure le operazioni di scrittura e quindi aumentato l'affidabilità del database. In quel momento utilizzavo per l'operazione degli script in LuaJIT per popolare un database con i dati di centinaia di impianti elettrici condominiali inseriti in un programma di manutenzione; tali dati comprendevano voci di opere, computi metrici, stati d'avanzamento lavori, ecc.

Grazie a quella conversazione, il passaggio dall'idea di usare i canali di input/output del sistema operativo a quella di usare ZeroMQ è stato quasi immediato: una volta ottenuto l'accesso a ZeroMQ, LuaTEX avrebbe potuto richiedere dati ed elaborazioni a programmi esterni in modo sicuro e affidabile, semplice e veloce.

4 Lo schema Client/Server

I due programmi della sezione precedente, LuaTEX da un lato e `double` dall'altro, comunicano con il *message pattern* chiamato Client/Server.

Descrivendo a parole lo schema riportato nella figura 3, durante la compilazione del sorgente LuaTEX, il codice Lua `A1` invia un messaggio che rappresenta l'informazione da elaborare a un preciso *indirizzo* ed entra poi nello stato di attesa sullo stesso canale.

Il programma *server* `A2`, in ascolto a quell'indirizzo, riceve il messaggio, compie le elaborazioni necessarie e inoltra al processo client la risposta. Di conseguenza, il programma client si riattiva per elaborare i dati contenuti nel messaggio di risposta.

Da questa semplice descrizione, possiamo subito ricavare le seguenti osservazioni:

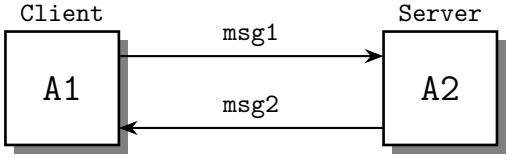


Figura 3: Schema di comunicazione Client/Server tra il programma `A1` che invia un messaggio al programma `A2` e quest'ultimo che risponde reinviando verso il mittente un secondo messaggio.

- il server deve essere attivo e disponibile;
- sia il client che il server devono conoscere l'indirizzo esatto di scambio dei messaggi;
- sia il client che il server dovranno conoscere le specifiche di formato dei messaggi perché sia possibile la loro interpretazione;
- un messaggio è una sequenza di byte;
- di per sé lo scambio dei messaggi non identifica il tipo di nodo, in particolare se client o server; è invece l'ordine con cui essi sono spediti a farlo. Più in generale, è il *comportamento* dei nodi a definirne la dinamica delle relazioni. Il sistema di comunicazione è unicamente tale, non influisce cioé sul comportamento dei nodi.

4.1 Il doppio di 28

Riprendendo l'esempio illustrato all'inizio della sezione precedente con cui un numero veniva raddoppiato, implementiamo la tipologia a due nodi costruendo un server in Rust attravero il progetto `rust-zmq` ospitato alla pagina `https://github.com/erickt/rust-zmq` e un client in Lua con la libreria `lzmq` in esecuzione all'interno di un sorgente LuaTEX[4].

Il server, ricevuto un testo che rappresenta un intero, ne calcola il doppio e reinvia questo valore al mittente. Il client invia il testo che rappresenta un numero intero al server e prosegue l'esecuzione una volta acquisita la risposta.

La libreria ZeroMQ è progettata per essere intuitiva da usare: nel codice, il reinvio della risposta da parte del server non necessita dell'individuazione del client ma semplicemente dell'invio di un messaggio al *socket*. Il server inoltre risponde a qualsiasi client evadendo le richieste in ordine di arrivo e, ancora una volta, ci pensa ZeroMQ a gestire questa coda.

4.1.1 Preparazione del progetto Rust

Come primo passo creiamo con il package manager `cargo` un progetto Rust aprendo un terminale sulla directory scelta per ospitare i file:

```
$ cd ~
$ mkdir project
$ cd project
$ cargo new basic_server --bin
$ cd basic_server
$ tree .
.
|-- Cargo.toml
+-- src
    +-- main.rs

1 directory, 2 files
```

4. L'installazione e le configurazioni dei moduli necessari per compilare ed eseguire il codice proposto, sono illustrate nella sezione 8 dell'articolo.

Editiamo poi il *manifest* `Cargo.toml` appena creato in automatico modificando il nome del pacchetto in `startserver`, e dichiarando la dipendenza *zmq* alla versione 0.8[5]:

```
[package]
name = "startserver"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
zmq = "0.8"
```

4.1.2 Il server

Editiamo adesso il file `main.rs` nella cartella `src` del progetto, in modo che contenga il seguente codice:

```
extern crate zmq;
fn main() {
    let ctx = zmq::Context::new();
    let responder = ctx.socket(zmq::REP)
                       .unwrap();
    assert!(
        responder.bind("tcp://*:5555")
                 .is_ok()
    );
    let mut msg = zmq::Message::new()
                               .unwrap();
    println!("Server ready...");
    loop {
        responder.recv(&mut msg, 0);
        // message decoding
        let n: i32 = msg.as_str()
                        .unwrap()
                        .parse()
                        .unwrap();
        println!("Received [{}]", n);
        // reply
        responder.send_str(
            &(2 * n).to_string()[..], 0
        ).unwrap();
    }
}
```

Anche se sono scritti in Rust i passaggi hanno validità generale:

1. per prima cosa viene creato l'oggetto di tipo `Context` che rappresenta il processo nel quale è in esecuzione ZeroMQ, affidandone il riferimento alla variabile `ctx`;

2. subito dopo otteniamo un socket predisposto per il REPLAY dei messaggi e infatti al metodo `socket()` del contesto si passa la costante predefinita `zmq::REP`;

3. colleghiamo ora al socket un indirizzo di comunicazione per il protocollo `TCP` situato alla porta numero 5555 dell'host locale, che altri non è che lo stesso computer su cui stiamo lavorando, ma potremmo indicare un qualsiasi IP sulla rete locale;

4. creiamo adesso un oggetto di tipo `Message` in cui depositare il contenuto delle richieste. Nella libreria `rust-zmq` esistono funzioni di più alto livello per ottenere dal socket i messaggi ma quell'oggetto può essere riutilizzato più volte essendo in effetti un contenitore e ciò rende il server ancor più veloce.

A questo punto la connessione è stabilita e il server è pronto per rispondere ai messaggi. Potremmo considerare di rispondere a un'unica richiesta ma conviene prevedere un ciclo in modo da lasciare il server sempre attivo, e poter compilare tutte le volte che si vuole il sorgente LuaTEX che ogni volta richiederà i servizi. Sarà tuttavia necessario interromperne l'esecuzione del server con la combinazione di tasti CTRL+C:

1. all'interno del ciclo infinito mettiamo in ricezione il server chiamando il metodo `recv()` del socket con il puntatore all'oggetto di tipo `Message` e un flag di valore 0, che richiede che l'operazione di ricezione sia in *blocking mode* così da attendere l'arrivo delle richieste;

2. alla linea di codice successiva il messaggio sarà contenuto nell'oggetto `msg`. Una catena di chiamate, molto comuni in Rust, ne esegue il metodo `as_str()` che restituisce il messaggio in forma di stringa. A sua volta su questo dato si esegue il metodo `parse()`[6] da cui si ricava l'intero che ho previsto sia a 32 bit con segno (tipo `i32`);

3. non rimane che inviare al socket un messaggio che giungerà nel formato testo al richiedente, chiamandone il metodo `send_str()` a cui si affida la stringa che rappresenta il valore doppio dell'intero e il flag 0 che regola il modo blocking o non-blocking dell'invio, qui ininfluente.

Si noti che non è necessario chiamare i metodi `close()` dell'oggetto messaggio, `disconnect()` del socket e `destroy()` del contesto, perché ciò avviene automaticamente grazie all'implementazione interna della libreria `rust-zmq` — quegli oggetti infatti implementano tutti il trait `Drop` — e alla gestione della memoria di Rust.

5. Potremo anche non esplicitare la versione della dipendenza — che segue le specifiche *semver* — indicando un asterisco al posto del numero di release. Così facendo sarà l'ultima versione disponibile di quella libreria a essere scaricata e utilizzata ma potremmo incorrere nella mancata risoluzione della versione nei casi con schemi complessi di dipendenze ramificate e circolari. Per questo motivo specificare un numero di versione è la norma.

6. Il metodo `parse()` sfrutta due altre funzionalità molto potenti di Rust: l'*inferenza*, quella capacità del compilatore di determinare il tipo degli oggetti senza che lo sviluppatore debba farlo esplicitamente, e il meccanismo dei *trait*, un modo per far sì che i tipi aderiscano a un'interfaccia„ cosa che tra l'altro rende generiche le funzioni.

Per scrivere il codice Rust che ho appena illustrato è necessario tenere a portata di mano la documentazione di `rust-zmq` e conoscere il tipo generico `Result<T, E>` definito nella libreria standard di Rust e brevemente introdotto alla sezione 8.2.1. È infatti molto importante essere consapevole dell'effetto dell'inserimento nel codice dei metodi `unwrap()`.

4.1.3 Compilazione del server

Per scaricare le dipendenze, compilare ed effettuare i necessari link alle librerie dinamiche è sufficiente dare il seguente comando in una finestra di terminale che punti alla directory principale del progetto, quella cioé che contiene il *manifest* `Cargo.toml`:

```
$ cargo build
```

Il package manager fa un gran lavoro ma la procedura non andrà a buon fine se non avremo installato i file di sviluppo di ZeroMQ come illustrato alla sezione 8.3.

4.1.4 Il client

Il client è un sorgente LuaTEX — ed è quindi scritto per il formato plain — che salveremo con il nome di `client.tex` in una directory per esempio `~/Scrivania/test_client`:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\directlua{
local zmq = require "lzmq"

print("Connecting to the server...")
local context = zmq.init(1)
local socket = context:socket(zmq.REQ)
socket:connect("tcp://localhost:5555")
print("Connection ready...")

local n = 28
print("Sending [" .. n .. "] to the server")
socket:send(tostring(n))

local reply = socket:recv()
socket:close()
context:term()
tex.print("Received... [" .. reply .. "]")
}
\bye
```

Il sorgente si apre caricando il pacchetto luapackageloader che modifica i percorsi di ricerca di script e librerie LuaTEX allargandone il campo anche al di fuori di TEX Live. Infatti, il binding a ZeroMQ `lzmq`, installato via Luarocks secondo le istruzioni della sezione 8.3, si trova nella directory inclusa nel percorso di ricerca dei normali script Lua, ovvero in `/usr/local/lib/lua/5.2/`. Questo significa che potremmo raggiungere il server anche scrivendo un normale script in Lua.

Il codice è analogo a quello del server con la differenza che basta l'invio e la ricezione di un solo messaggio. In sequenza si tratta di:

1. ottenere un riferimento al contesto con il metodo `init()` e da questo un riferimento a un oggetto socket attraverso il metodo `socket`, specificando la costante `REQUEST` che ne caratterizza il tipo, scelto in base al message pattern corrispondente;
2. stabilire la connessione con il metodo `connect` del socket all'indirizzo di protocollo `TCP` identico a quello del server. La comunicazione avverrà tra processi in esecuzione sullo stesso computer;
3. inviare al socket il messaggio di testo che rappresenta il numero 28, tramite il metodo `send()`;
4. ricevere il messaggio di testo di ritorno del server, che rappresenta il numero 56, tramite il metodo `recv()` del socket;
5. chiudere sia il socket con il metodo `close()`, sia il contesto con il metodo `term()`.

Terminata la comunicazione tra client e server, il codice prosegue con un'ultima istruzione eseguendo tramite la funzione `tex.print()` l'invio del messaggio di ritorno verso TEX che comporrà sulla pagina il testo seguente:

```
Received... [56]
```

4.1.5 Esecuzione

È il momento di provare se tutto funziona avviando il server in una finestra di terminale con il comando[7]:

```
$ cd ~/project/basic_server/target/debug
$ ./startserver
```

e compilando il nostro sorgente LuaTEX in una seconda finestra di terminale:

```
$ cd ~/Scrivania/test_client
$ luatex --shell-escape client.tex
```

5 Multipart messaging

In ZeroMQ un messaggio è una sequenza di uno o più *frame*, ciascuno dei quali composto dallla lunghezza in byte del contenuto seguito dal contenuto stesso, come illustrato nella figura 4. ZeroMQ garantisce che i messaggi multiparte siano spediti effettivamente in un unico blocco solo quando viene inviato l'ultimo frammento.

7. L'eseguibile del server, dopo la compilazione per mezzo del package manager `cargo`, si trova nella sotto-cartella `target/debug` della directory del progetto in Rust. La versione ottimizzata e priva delle informazioni di debug si ottiene invece aggiungendo al comando `cargo build` l'opzione `--release` e l'eseguibile si troverà nella sotto-cartella `target/release`. Si può anche compilare e lanciare il programma con il seguente unico comando `$ cargo run`, con o senza il flag di ottimizzazione.

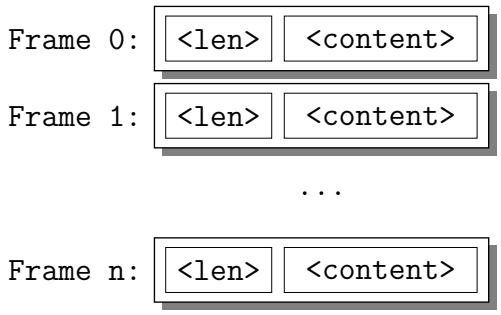


FIGURA 4: Costruzione del messaggio in ZeroMQ: l'insieme di uno o più frame ciscuno dei quali composto dalla lunghezza della sequenza di byte del contenuto e la sequenza stessa.

La composizione di un messaggio in più frame è il modo più semplice per strutturare l'informazione da trasmettere. Per esempio, il contenuto del primo frame potrebbe determinare se il messaggio di richiesta è stato elaborato con successo oppure se si sono verificati degli errori.

In questa sezione impiegheremo i messaggi multiparte affinché si possa includere il QRCode di un testo in un sorgente LuaTEX: il testo dell'indirizzo web della pagina del forum del sito GUIT dei post più recenti `http://www.guitex.org/home/it/forum/recent-topics`.

Per far sì che l'attenzione si focalizzi sul codice LuaTEX, la spiegazione dell'implementazione del server è rimandata più avanti alla sezione 10. Dal lato TEX dividerò il problema in quattro parti:

1. definizione del formato dei messaggi multiparte;

2. messa a punto del codice Lua per l'invio e la ricezione dei messaggi;

3. messa a punto del codice Lua per il disegno di una griglia generica;

4. verifica di funzionamento, con la scrittura del sorgente LuaTEX che conterrà il simbolo QRCode con il link previsto.

FIGURA 5: Il simbolo QRCode della pagina del sito GUIT dei post più recenti del forum, ottenuto in LuaTEX interrogando un server esterno, come illustrato nel testo.

5.1 QRCode

Il QRCode *Quick Response Code* (`https://it.wikipedia.org/wiki/Codice_QR`) si presenta come una griglia regolare $n \times n$ di piccoli elementi, a loro volta quadrati, che possono essere di colore nero oppure bianco. La figura 5 mostra l'esempio di simbolo QRCode che ci proponiamo di realizzare, verificabile scandendo il simbolo con uno smartphone.

Per prima cosa definiamo il formato del messaggio di richiesta: useremo molto semplicemente tre frame[8] dalla struttura seguente:

```
Frame 0: "QRCODE"
Frame 1: <encoding message>
Frame 2: "END"
```

Il servizio esterno potrà rispondere con due diversi messaggi identificati dal contenuto del frame iniziale `"QRCODE"` o `"ERR"`, per rappresentare il successo o meno dell'operazione di codifica del QRCode. Sulla base del frame 0 il successivo frame dovrà essere interpretato rispettivamente come la codifica del simbolo oppure come la descrizione dell'errore:

```
Frame 0 : "QRCODE"
Frame 1 : <qrcode 0/1 sequence>
Frame 2 : "END"
```

oppure:

```
Frame 0 : "ERR"
Frame 1 : <error description>
Frame 2 : "END"
```

5.2 Connessione al servizio

La connessione a una porta di un socket in ZeroMQ può dar luogo a errori. Un modo semplice ed efficace per gestire eventuali problemi è scrivere le funzioni Lua in modo che restituiscano due valori: in caso di successo, in prima posizione ci sarà il valore atteso e in seconda il valore `nil`, mentre all'opposto in caso di errore, in prima posizione ci sarà il valore `nil` e in seconda la stringa contenente la descrizione dell'errore stesso.

Nel file `libmsg.lua` cominciamo quindi con il creare una tabella in cui memorizzeremo le funzioni con il paradigma degli oggetti, e con lo scrivere la funzione `service()` che dalla stringa che descrive la porta del servizio, tenterà di crearne il socket:

```
-- libmsg.lua file
local libmsg = {}
libmsg.__index = libmsg

local zmq = require "lzmq"

function libmsg:service(port)--> table, err
    -- Connecting to the server
    local ctx, err = zmq.context{io_thread=1}
    if err then
        return
```

8. Come richiesto dal server esterno il contenuto testuale dovrà essere conforme alle specifiche Unicode UTF-8.

```
        nil, "[Context Error] "..err:msg()
    end
    local skt, err = ctx:socket(zmq.REQ)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end
    local _, err = skt:connect(port)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end

    local o = {
        context = ctx,
        socket  = skt,
    }
    setmetatable(o, self)
    return o, nil
end
```

La connessione al socket si potrà poi dismettere con la seguente funzione `disconnect()` che in caso di errore ne restituisce la descrizione:

```
function libmsg:disconnect() --> nil or error
    local _, err = self.socket:close()
    if err then
        return "[Socket error] "..err:msg()
    end
    local _, err = self.context:term()
    if err then
        return "[Context error] "..err:msg()
    end
end
```

Le ultime linee contengono la definizione della funzione centrale che ho chiamato `send_and_recv()`, quella che invia un messaggio multiparte e restituisce la risposta, e l'istruzione `return` che ritorna la tabella di libreria quando il file sarà richiamato da codice Lua con la funzione `require()`.

Il binding per Lua di ZeroMQ `lzmq` semplifica molto il lavoro con i messaggi multiparte grazie alla funzione `Socket:send_all()` a cui si passa l'array contenente la collezione ordinata dei frame:

```
function libmsg:send_and_recv(tmsg)--> t, err
    local skt = self.socket
    -- multipart message
    local _, err = skt:send_all(tmsg)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end

    local dataset, err = skt:recv_all()
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end
    return dataset
end

return libmsg
-- end of libmsg.lua file
```

5.3 Disegno del simbolo

Dal servizio remoto riceveremo in risposta alla stringa di testo da codificare in un simbolo QR-Code, la sequenza di zero e uno raggruppabili in righe dall'alto verso il basso grazie al carattere di separazione `\n`. Associando al numero 1 il significato di quadratino pieno e al numero 0 quello di quadratino vuoto, la sequenza può essere disegnata in modo vettoriale sfruttando i nodi *pdfliteral* di LuaTEX, come mostrato nel dettaglio nell'articolo (Giacomelli, 2016) apparso su Ars 22.

Allo scopo, creiamo un nuovo file `libgrid.lua` con le stesse modalità del precedente, che come prima cosa definisca la funzione costruttore di un oggetto griglia a partire dalla stringa di zeri e di uno. Nella funzione `new()` ho considerato anche il carattere di separazione spazio perché nelle prove con sorgenti LuaTEX il segno di ritorno a capo è convertito automaticamente in un carattere spazio da TEX durante l'espansione dei token.

Nel dettaglio, la funzione elabora la sequenza in ingresso trasformandola in un array di valori booleani — `true` corrisponderà al quadratino da disegnare e `false` a nessuna azione — e negli interi contenenti il numero di righe e di colonne, perciò la griglia può anche essere rettangolare. Per gestire casi di errore, i dati di uscita saranno ancora la coppia di ⟨*valore*⟩/⟨*errore*⟩:

```
-- libgrid.lua file
local libgrid = {}
libgrid.__index = libgrid

-- constructor
function libgrid:new(qrseq) --> obj, err
    local seq = {}
    local row = 1
    for c in string.gmatch(qrseq, '.') do
        if c == '1' then
            seq[#seq + 1] = true
        elseif c == '0' then
            seq[#seq + 1] = false
        elseif c == '\n' or c == ' ' then
            row = row + 1
        else
            return
            nil, "Unexpected char '"..c.."'"
        end
    end
    local col = math.floor(#seq/row)
    if not (col*row == #seq) then
     return
     nil,
     "The sequence doesn't fit the rectangle"
    end
    local o = {
        qrseq = seq,
        row = row,
        col = col,
    }
    setmetatable(o, self)
    return o
end
```

Senza entrare troppo nei dettagli che si trovano nell'articolo citato, i codici in notazione postfissa previsti dal formato PDF per disegnare i rettangoli sono i seguenti:

```
q
<x> <y> <width> <height> re
<x> <y> <width> <height> re
...
f
S
Q
```

che genereremo con la funzione `_pdfliteral()`, da intendersi privata. La funzione crea questo testo calcolando le coordinate x e y dell'angolo inferiore sinistro di ciascun quadratino a partire dall'ultima riga in basso e in direzione da destra a sinistra, e inserendo la dimensione del *modulo* in punti grandi (*bp*), per la misura dei lati:

```
function libgrid:_pdfliteral(mod_bp)
    mod_bp = mod_bp or 4 -- module size
    local seq = self.qrseq
    local row, col = self.row, self.col

    local pdflit = {"q"}
    local fmtpair = "%0.6f %0.6f"
    local fmtdim = string.format(fmtpair,
        mod_bp, mod_bp
    )
    local fmt = string.format("%s %s re",
        fmtpair, fmtdim
    )
    -- fmt => <x> <y> <width> <height> re
    local x, y = 0.0, 0.0
    for r = row-1, 0, -1 do
        local start = r*col
        for c = 1, col do
            if seq[start + c] then
                pdflit[#pdflit + 1] =
                    string.format(fmt, x, y)
            end
            x = x + mod_bp
        end
        x = 0.0
        y = y + mod_bp
    end
    pdflit[#pdflit + 1] = "f"
    pdflit[#pdflit + 1] = "S"
    pdflit[#pdflit + 1] = "Q"
    return table.concat(pdflit, "\n"), mod_bp
end
```

L'ultima funzione necessaria è quella che costruisce una scatola orizzontale contenente il disegno della griglia. Non sarebbe difficile scriverla se non fosse che i nodi di tipo *pdfliteral* hanno dimensione nulla perciò, se non adeguassimo il contenuto della scatola, il disegno si sovrapporebbe al testo o supererebbe i margini della pagina.

Non solo. Dobbiamo anche prevedere una cornice bianca attorno al simbolo spessa almeno quattro volte la dimensione del modulo, detta *quiet zone*, per consentire la scansione del simbolo del codice a barre da parte dei dispositivi.

La funzione `boxpack()` si occuperà della creazione della scatola orizzontale con le corrette dimensioni. Essa inizia con il controllare l'esistenza del registro box il cui nome è contenuto nella variabile `boxreg`, corrispondente a quello che avremo avuto l'accortezza di definire in precedenza nel sorgente con il classico comando primitivo TEX `\newbox`. Una volta ottenuto il codice *pdfliteral* con l'omonima funzione, la costruzione della scatola avviene con la concatenazione di nodi di tipo diverso, seguendo attentamente i seguenti passi[9]:

passo a) costruzione del nodo *pdfliteral* `n0`;

passo b) costruzione del nodo spazio elastico `vqz0` di altezza pari alla quiet zone;

passo c) concatenazione dei nodi `n0` e `vqz0`. Il nodo `n1` restituito dalla funzione di concatenazione è un puntatore allo stesso nodo `n0`;

passo d) costruzione della scatola verticale `vbox1` con il nodo di testa `n1`;

passo e) costruzione della distanza elastica orizzontale `hqz1` di larghezza pari alla quiet zone;

passo f) creazione lista `n2` con nodo distanza orizzontale e nodo scatola verticale;

passo g) inserimento finale della lista nella scatola orizzontale `hbox2` a cui poi si impongono le dimensioni in larghezza e altezza della griglia compreso la cornice di quiet zone.

Traducendo le istruzioni in codice Lua otteniamo la funzione:

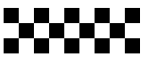
```
function libgrid:boxpack(boxreg, mod_bp)
    assert(
        tex.isbox(boxreg),
        string.format(
           "Box register [%s] doesn't exist",
           boxreg
        )
    )
    local pdf, mod = self:_pdfliteral(mod_bp)
    -- symbol dimensions (scaled point)
    mod = mod * tex.sp "1bp"
    local width  = self.col * mod -- sp
    local height = self.row * mod -- sp
    local quietzone = 4 * mod     -- sp

    -- step a: pdfliteral node
    local n0 = node.new(
        "whatsit", "pdf_literal"
```

9. Per una maggiore comprensione, ho numerato con lo stesso indice le variabili dei nodi che sono allo stesso livello nella lista e assegnato un passo diverso ogni volta che viene creato un nuovo nodo.

```
    )
    n0.data = pdf

    -- step b: vertical quietzone dim
    -- (respect to the baseline)
    local vqz0 = node.new "glue"
    vqz0.width = quietzone

    -- step c: vlist
    local n1 = node.insert_after(
        n0, n0, vqz0
    )
    -- step d: vertical box
    local vbox1 = node.vpack(n1)

    -- step e: horizontal dim
    local hqz1 = node.new "glue"
    hqz1.width = quietzone
    -- step f: left horizontal quietzone
    local n2 = node.insert_after(
        hqz1, hqz1, vbox1
    )
    -- step g: final horizontal box
    local hbox2 = node.hpack(n2)
    hbox2.width  = width + 2 * quietzone
    hbox2.height = height + 2 * quietzone
    tex.box[boxreg] = hbox2
end

return libgrid
-- end of libgrid.lua file
```

5.3.1 *Verifica di* `libgrid`

La griglia rettangolare di prova:



è correttamente disegnata dal codice:

```
% !TeX program = LuaTeX
\nopagenumbers
\newbox\mybox
\directlua{
local libgrid = require "libgrid"
local grid, err = libgrid:new
    [[101010101
      010101010
      101010101]]
assert(not err, err)
grid:boxpack "mybox"
}
\leavevmode\box\mybox
\bye
```

5.4 Il sorgente LuaTEX conclusivo

Il passo finale è la realizzazione del QRCode con il link che punta alla pagina dei post recenti sul forum del GUIT, simbolo riportato nella figura 5. Nel sorgente LuaTEX riportato qui sotto una buona parte del codice Lua è impegnato nelle operazioni di parsing del messaggio di risposta:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\nopagenumbers
\newbox\mybox

\directlua{
local libmsg = require "libmsg"
local libgrid = require "libgrid"
local QRCodeSrv, err = libmsg
      :service "tcp://localhost:5556"
assert(not err, err)

local dataset, err = QRCodeSrv:send_and_recv{
   "QRCODE",
   "http://www.guitex.org/home/"
   .."it/forum/recent-topics",
   "END"
}
assert(not err, err)

local err = QRCodeSrv:disconnect()
assert(not err, err)

local cmd    = dataset[1]
local msg    = dataset[2]
local endmsg = dataset[3]
assert(endmsg == "END")

if cmd == "ERR" then
    error(msg)
end

local grid, err = libgrid:new(msg)
assert(not err, err)

local width = grid.row
assert(grid.row == grid.col)

grid:boxpack "mybox"
}
\leavevmode\box\mybox
\bye
```

6 Un server, più servizi

I messaggi non sono vincolati a una struttura fissa. Ne deriva che un servizio remoto può incorporare più di un tipo di elaborazione. Per esempio, il client potrebbe inviare al server un messaggio multiparte il cui primo frame è il codice che corrisponde a una particolare elaborazione.

In questa sezione utilizzeremo questo dato per realizzare un servizio remoto multifunzione, con cui richiedere tre differenti elaborazioni sui numeri primi:

`PRIME-CHECK:` test di primalità;

`PRIME-COUNT:` conteggio dei primi in un intervallo;

`PRIME-LIST:` lista dei primi in un intervallo.

Per poterci concentrare sul client in LuaTEX, rimando alla sezione 11 la costruzione del server in Rust.

6.1 Messaggi e funzioni

Il formato generale dei messaggi di richiesta prevede più frame e, in sostanza, è equivalente alla chiamata di una funzione che ne specifica il nome e gli argomenti di ingresso con questa struttura:

```
Frame 0: <function ID>
Frame 1: <arg1>
Frame 2: <arg2>
...
Frame n: "END"
```

Altrettanto generale può essere il formato dei messaggi di risposta:

```
Frame 0: <function ID>
Frame 1: <result1>
Frame 2: <result2>
...
Frame n: "END"
```

6.2 Funzione di alto livello

L'idea è di scrivere la funzione `libmsg:call()` che accetta il nome della funzione remota disponibile sul server, e un numero variabile di argomenti che dipende dal servizio stesso. Questa funzione di alto livello dovrà eseguire i seguenti passi:

1. codifica del messaggio di richiesta;
2. invio del messaggio di richiesta e ricezione della risposta;
3. decodifica del messaggio di risposta.

Aggiungiamo alla libreria in Lua `libmsg`, costruita nella sezione precedente, due funzioni per la codifica e la decodifica dei messaggi che restituiscono una tabella Lua contenente i frame del messaggi:

```
-- generic encoding of the request message
function libmsg:encode(fname, ...)
    local tmsg = {fname}
    for _, elem in ipairs {...} do
        tmsg[#tmsg + 1] = tostring(elem)
    end
    tmsg[#tmsg + 1] = "END"
    return tmsg
end

-- generic decoding of the replay message
function libmsg:decode(tmsg)
    if tmsg[1] == "ERR" then
        assert(tmsg[3] == "END")
        return nil, tmsg[2]
    end
    assert(tmsg[#tmsg] == "END")
    return tmsg
end
```

La funzione `call()` è la diretta implementazione dei tre passi dell'elenco di apertura di sezione. Essa restituisce i valori alternativamente validi della tabella Lua con i risultati in prima posizione oppure della descrizione dell'errore in seconda posizione:

```
function libmsg:call(fname, ...)
    local tmsg = self:encode(fname, ...)
    local res, err = self:send_and_recv(tmsg)
    if err then
        return nil, err
    end
    return self:decode(res)
end
```

6.3 Verifica di primalità

Per come è implementato il server, i risultati di tutte le funzioni remote sono sequenze di interi, perciò il servizio `PRIME-CHECK` ritorna la lista contenente un solo valore, 0 se il numero trasmesso nel messaggio di richiesta è primo, altrimenti 1. Qui di seguito il listato di un sorgente LuaTEX in cui la verifica di primalità è ottenuta semplicemente chiamando la funzione Lua `Service:call()`:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty

\directlua{
local libmsg = require "libmsg"
PrimeSrv = libmsg
           :service "tcp://localhost:5557"
}
\def\checkprime#1{\directlua{
local n = tonumber(#1)
local ans, err = PrimeSrv
                 :call("PRIME-CHECK", n)
assert(not err, err)
local isprime = tonumber(ans[2])
assert(isprime == 0 or isprime == 1)

if isprime == 1 then
    tex.print("prime")
else
    tex.print("not prime")
end
}}

The number 62643217 is \checkprime{62643217}
while 920419813 is \checkprime{920419813}.

\directlua{
local err = PrimeSrv:disconnect()
if err then error(err) end
}
\bye
```

6.4 Tabella dei primi

Il risultato del secondo esempio, riferito al servizio remoto che restituisce la lista dei primi in un dato intervallo, è riportato in tabella 1. Il contenuto del messaggio di risposta può quindi essere piuttosto grande: esso infatti dedica un frame per ciascuno dei primi della lista. Tuttavia nelle prove non ho notato alcun problema né alcun evidente rallentamento di ZeroMQ.

Il sorgente da compilare con LuaLATEX è il seguente. Una volta ottenuta dal servizio remoto la lista dei primi, la creazione della tabella avviene stampando

Tabella 1: Tabella dei primi inferiori a 1000 ottenuta in LuaLaTeX scambiando messaggi con un server per mezzo della libreria ZeroMQ, come spiegato in dettaglio nel testo. Questa tabella è ispirata a quella identica che compare nelle prime pagine del libro "L'Ossessione dei numeri primi" di John Derbyshire.

2	3	5	7	11	13	17	19	23	29	31	37	41	43
47	53	59	61	67	71	73	79	83	89	97	101	103	107
109	113	127	131	137	139	149	151	157	163	167	173	179	181
191	193	197	199	211	223	227	229	233	239	241	251	257	263
269	271	277	281	283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479	487	491	499	503	509	521
523	541	547	557	563	569	571	577	587	593	599	601	607	613
617	619	631	641	643	647	653	659	661	673	677	683	691	701
709	719	727	733	739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863	877	881	883	887
907	911	919	929	937	941	947	953	967	971	977	983	991	997

il corpo dell'ambiente tabular con la funzione interna di LuaTEX `tex.print()` come se fosse stato digitato manualmente, ma con il numero di colonne specificate nella definizione della macro `\column`:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{booktabs}

\directlua{
    local libmsg = require "libmsg"
    PrimeSrv = libmsg
        :service "tcp://localhost:5557"
}
\def\column{14}

\begin{document}
\catcode`\#=12
\begin{tabular}{*{\column}{c}}
\toprule
\directlua{
local list, err = PrimeSrv:call(
    "PRIME-LIST", 1, 1000
)
assert(not err, err)

local len = #list
local col = \column
local row = math.floor(len/col)
if col * row < len then
    row = row + 1
end
local dbs = string.char(92, 92)
for r = 1, row do
    local i = (r-1) * col + 1
    local j; if r == row then
        j = len
    else
        j = i + col - 1
    end
    tex.print(
        table.concat(list, "&", i, j)..dbs
    )
end
}\bottomrule
\end{tabular}
\directlua{
local err = PrimeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

6.5 Conteggio dei primi

Il grafico di figura 6 è stato ottenuto grazie alla *funzione* `PRIME-COUNT` disponibile sul server. Nel sorgente che lo produce, riportato di seguito, le coordinate dei punti sono inserite nel flusso dei token nel momento in cui viene espansa la primitiva `\directlua` che si trova all'interno dell'argomento della macro `\addplot`.

Il server viene interrogato ogni volta che occorre conoscere il numero dei primi nell'intervallo di estremi $[a, a+s)$ con s costante e a che si incrementa ogni volta di s. La costruzione del grafico di figura 6 richiede in particolare 281 richieste al server, e anche qui non si apprezza nessun rallentamento durante la compilazione.

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{pgfplots}
\pgfplotsset{compat=1.15}

\footnotesize
\directlua{
    local libmsg = require "libmsg"
    PrimeSrv = libmsg
        :service "tcp://localhost:5557"
}
\def\nstep{100}   % interval size
\def\npoints{281} % number of points

\begin{document}
\begin{tikzpicture}
```

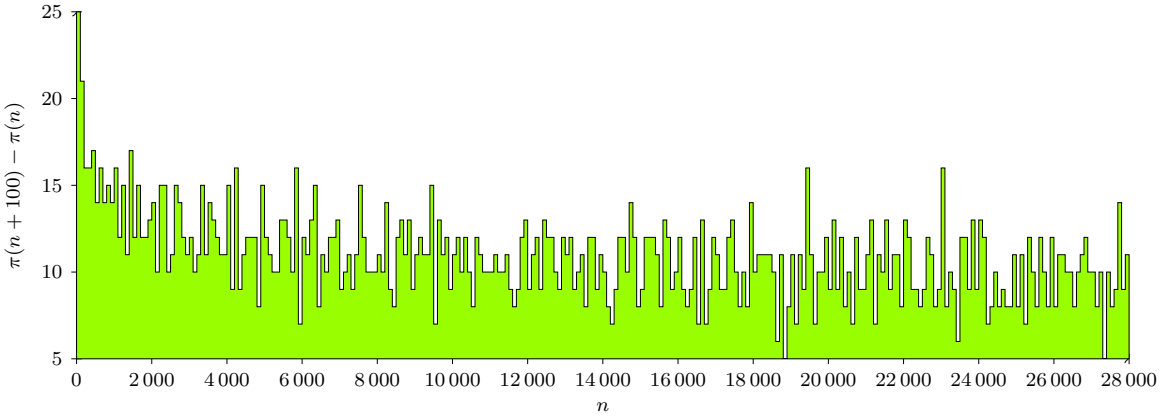


FIGURA 6: Grafico in pgfplots risultato della compilazione del sorgente LuaLaTeX riportato nel testo in cui si ricava il conteggio dei numeri primi per ogni intervallo consecutivo di ampiezza 100, da un servizio esterno raggiunto tramite la libreria ZeroMQ.

```
\begin{axis}[
const plot,
width=16.5cm, height=6.5cm,
scaled ticks=false,
ylabel = $\pi(n+\nstep) - \pi(n)$,
xlabel = $n$,
axis x line=bottom,
axis y line=left,
axis line style={->},
tick style={color=black},
tick label style={%
    /pgf/number format/1000 sep={\,}},
    font=\footnotesize
]
\addplot[
    black,
    fill=green!40!yellow
] coordinates {
    \directlua{
    local step, points = \nstep, \npoints
    local a = 0
    local perc = string.char(37)
    local fmt = "("..perc.."d,"..perc.."s)"
    for _ = 1, points do
        local tprime, err = PrimeSrv:call(
            "PRIME-COUNT", a, a + step - 1
        )
        assert(not err, err)
        local p = tprime[1]
        tex.print(
            string.format(fmt, a, p)
        )
        a = a + step
    end
    }
}
\closedcycle;
\end{axis}
\end{tikzpicture}
\directlua{
local err = PrimeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

7 Database server

Il tema di questa quarta e ultima sezione applicativa dell'articolo è l'accesso a un database — una delle più importanti funzionalità richieste in ambito aziendale — tramite la tecnologia dello scambio di messaggi.

Dal punto di vista del sorgente LuaTeX — ovvero il nodo che assume il ruolo di client — non solo non si conosce come è stato implementato il server, ma nemmeno quale sia il reale motore SQL del database. Le uniche informazioni necessarie sono solamente:

1. l'indirizzo del server,
2. il formato dei messaggi.

7.1 Messaggi e query SQL

Il server accetta messaggi di richiesta che contengono al secondo frame il testo della query SQL e in quelli successivi gli eventuali parametri corrispondenti. Possiamo quindi effettuare ogni sorta di interrogazione nei confronti del database connesso con il server a patto di conoscerne la struttura.

Poiché ogni database implementa il linguaggio SQL con alcune varianti rispetto alla versione standard, dovremmo in realtà conoscere il motore effettivo dell'archivio SQL per poter interagire con più efficacia con i dati. Il server impiegherà in effetti SQLite3 http://www.sqlite.org/, una delle soluzioni più semplici e affidabili per la gestione dati oggi disponibili.

Per incrementare la sicurezza dei dati, il server non accetterà le query in scrittura che modificano il database. Questa condizione corrisponde allo scopo del client, che è quello di reperire le informazioni per comporre documenti, senza alcuna necessità di apportare modifiche.

In conclusione, poiché il messaggio di richiesta contiene il testo della query SQL, dobbiamo anche conoscere:

3. la struttura del database,

4. le specifiche esatte del linguaggio SQL.

7.2 Formato dei messaggi

La sequenza dei frame del messaggio di richiesta è la seguente:

```
Frame 0: "QUERY"
Frame 1: <prepared statement SQL>
Frame 2: <param1>
Frame 3: <param2>
...
Frame n: "END"
```

I parametri dal frame 2 in poi sono richiesti solo nel caso in cui il *prepared statement* li preveda. Eseguire le interrogazioni tramite questa funzione rende più sicuri i dati, impedendo azioni malevole di *SQL injection* quando i parametri sono definiti all'esterno. Nel caso del server di questa applicazione, il database rimane comunque in sola lettura e la definizione dei parametri è fatta dall'utente TEX.

Se ci sono stati errori o azioni non ammesse il server risponde con il seguente messaggio multiparte:

```
Frame 0: "ERR"
Frame 1: <error description>
Frame 2: "END"
```

altrimenti invia un messaggio piuttosto complesso, articolato in tre sezioni: la prima contiene il numero c delle colonne della tabella risultato della query, la seconda è la lista dei nomi delle colonne e la terza è la lista dei dati della tabella di r righe, una dopo l'altra, ed è quindi lunga $c \cdot r$:

```
Frame 0    : "TABLE"
Frame 1    : <column counter c>
Frame 2    : <column name 1>
Frame 3    : <column name 2>
...
Frame 2 + c: <column name c>
Frame 3 + c: <data row 1 col 1>
Frame 4 + c: <data row 1 col 2>
...
Frame 2 + (r+1)*c: <data row r col c>
Frame 3 + (r+1)*c: "END"
```

Come verificheremo nel prossimo esempio, il server aggiunge delle chiavi a ciascun elemento della tabella per caratterizzarne il tipo. SQLite infatti ammette che i dati possano essere di un tipo diverso rispetto a quello dichiarato per il campo della tabella, una scelta simile a quella dei linguaggi dinamici come Lua e Python nei confronti dei tipi dei dati. Il controllo dei tipi, invece, permetterebbe di non appesantire i dati associando il tipo alla colonna corrispondente.

L'elenco dei prefissi è il seguente:

`NULL:` tipo nullo in SQL;

`INTE:` intero con segno a 64 bit;

`REAL:` numero in virgola mobile a 64 bit;

`TEXT:` stringa di testo;

`BLOB:` stream di byte[10].

7.3 Verifica di base

Una volta avviato il servizio, eseguiamo una semplice query per ottenere la data impostata in SQLite. La libreria Lua `libmsg` è la stessa dei progetti precedenti e il relativo file deve essere presente nella cartella del seguente sorgente LuaTEX:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\directlua{
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)
}

\directlua{
local qmsg, err = DBSrv:call(
    "QUERY", [[SELECT DATE() AS date;]]
)
assert(not err, err)
tex.print(
    table.concat(
        qmsg, string.char(92)..[[par ]]
    )
)
}

% closing the service's connection
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
}
\bye
```

Il PDF, risultato della compilazione con l'opzione `--shell-escape` di attivazione dei comandi di sistema, dovrebbe contenere un testo simile al seguente, con l'intera struttura dei frame del messaggio di risposta:

```
TABLE
1
date
TEXT:2017-08-11
END
```

7.4 Iteratori di tabelle

Creiamo un nuovo file `libiter.lua` con le stesse modalità dei precedenti: conterrà codice che semplifichi l'iterazione delle righe di una tabella contenuta nel messaggio di risposta del server come risultato di una query.

Per esempio, con questo iteratore potremo scrivere il codice di richiesta (query) dei dati contenuti nella tabella `departments` del database di prova

10. `BLOB` sta per Binary large object.

Tabella 2: Semplice elenco dei dipartimenti contenuti nel database di esempio ottenuto con LuaTEX attraverso un servizio remoto e un iteratore scritto appositamente e spiegato nel testo.

d001 — Marketing
d002 — Finance
d003 — Human Resources
d004 — Production
d005 — Development
d006 — Quality Management
d007 — Sales
d008 — Research
d009 — Customer Service

(illustrato in dettaglio nella prossima sezione) e formanti l'elenco riportato nella tabella 2:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\nopagenumbers
\directlua{
libiter = require "libiter"
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)
}

\directlua{
local qmsg, err = DBSrv:call(
    "QUERY",
    [[SELECT dept_no, dept_name
      FROM departments;]]
); assert(not err, err)

for n, row in libiter.rows(qmsg) do
    tex.print(
        row.dept_no, "---", row.dept_name,
        string.char(92)..[[par ]]
    )
end
}

% closing the service's connection
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
}
\bye
```

Nella parte centrale del sorgente è presente l'iteratore (funzione `libiter.rows()`) in azione all'interno di un normale ciclo `for` di Lua. L'iteratore accetta direttamente la tabella contenente il messaggio multiparte con i dati restituiti dalla query. A ogni iterazione abbiamo a disposizione una prima variabile contenente il numero progressivo del record — non utilizzata nell'esempio — e un secondo oggetto indicizzabile con i nomi delle colonne.

L'iteratore è tecnicamente uno *stateless iterator* che impiega una metatabella per consentire l'indicizzazione riga per riga dei valori corrispondenti alle colonne della query. È il genere di cose che rende Lua un linguaggio elegante.

Il seguente è il codice contenuto nel file di libreria `libiter.lua`:

```
local libiter = {}
local transform = {
    ["NULL:"] = function (_) return nil end,
    ["INTE:"] = function (x)
        return tonumber(x)
    end,
    ["REAL:"] = function (x)
        return tonumber(x)
    end,
    ["TEXT:"] = function (x) return x end,
    ["BLOB:"] = function (_)
        return "unimplemented!"
    end,
}

local function convert(s)
    local key = s:sub(1,5)
    assert(transform[key])
    return (transform[key])(s:sub(6))
end

-- stateless iterator
function libiter.rows(tmsg)
    local cols = tonumber(tmsg[2])
    local colpos = {}
    local k = 1
    for i = 3, 2 + cols do
        colpos[tmsg[i]] = k
        k = k + 1
    end

    local obj = {}
    local mt = {}
    setmetatable(obj, mt)

    function mt.__index(t, key)
        if not colpos[key] then
            error(string.format(
                "The column name '%s' "..
                "doesn't exist",
                key
            ))
        end
        local pos = colpos[key]
        local nrow = t[0]
        local idx = 2 + nrow * cols + pos
        local res = tmsg[idx]
        return convert(res)
    end
    local function iter(o, nrow)
        nrow = nrow + 1
        o[0] = nrow
        local idx = 3 + nrow * cols
        if tmsg[idx] ~= "END" then
            return nrow, o
        end
    end
    return iter, obj, 0
end
return libiter
```

7.5 Il database di prova

Per effettuare le prove ho adattato il database del progetto https://github.com/vrajmohan/pgsql-sample-data a SQLite riducendolo a sole tre tabelle riguardanti la definizione dei dipartimenti, che abbiamo già incontrato alla sezione precedente, e l'anagrafica degli impiegati di un'ipotetica azienda.

Il codice SQL che genera le tabelle su cui il server eseguirà le query è il seguente. Esso rappresenta l'intera struttura del database di prova.

```
CREATE TABLE departments (
    dept_no     CHAR(4) PRIMARY KEY,
    dept_name   VARCHAR(40) NOT NULL,
    UNIQUE (dept_name)
);

CREATE TABLE employees (
    emp_no      INTEGER PRIMARY KEY,
    birth_date  CHAR(10)    NOT NULL,
    first_name  VARCHAR(14) NOT NULL,
    last_name   VARCHAR(16) NOT NULL,
    gender      CHAR(1)     NULL,
    hire_date   CHAR(10)    NOT NULL
);

CREATE TABLE dept_emp (
    emp_no      INT      NOT NULL,
    dept_no     CHAR(4)  NOT NULL,
    from_date   DATE     NOT NULL,
    to_date     DATE     NOT NULL,
    FOREIGN KEY (emp_no)
    REFERENCES employees (emp_no)
    ON DELETE CASCADE,
    FOREIGN KEY (dept_no)
    REFERENCES departments (dept_no)
    ON DELETE CASCADE,
    PRIMARY KEY (emp_no, dept_no)
);
```

7.6 Badge degli impiegati

L'esempio riportato in figura 7 contiene due *badge* dimostrativi di altrettanti impiegati memorizzati nel database. A sinistra compare il simbolo QRCode corrispondente all'identificativo personale mentre a destra una colonna dati con, tra le altre informazioni, il nome e il dipartimento di appartenenza del dipendente. In una situazione reale non sarebbero comprese le date personali per il rispetto della privacy.

A realizzare i badge è il seguente sorgente LuaLaTeX:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{booktabs}

% build the services
\directlua{
libgrid = require "libgrid"
```

ID: 10001
Name: Georgi Facello
Birth Date: 1953-09-02
Dept.: Development
Hire date: 1986-06-26
From: 1986-06-26

ID: 10002
Name: Bezalel Simmel
Birth Date: 1964-06-02
Dept.: Sales
Hire date: 1985-11-21
From: 1996-08-03

FIGURA 7: Due badge dimostrativi realizzati con LuaLaTeX come illustrato nel testo, impiegando la libreria ZeroMQ per accedere a due servizi esterni, il primo per ricavare il QRCode dell'identificativo dell'impiegato e il secondo per interrogare il database contenente l'anagrafica.

```
libiter = require "libiter"
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)

QRCodeSrv, err = libmsg
    :service "tcp://localhost:5556"
assert(not err, err)
}

\begingroup
\catcode`\#=12
% execute the query against the employees db
\directlua{
local employees, err = DBSrv:call(
"QUERY", [[SELECT
employees.emp_no      AS empid,
employees.first_name  AS firstname,
employees.last_name   AS lastname,
employees.birth_date  AS birthdate,
employees.hire_date   AS hiredate,
departments.dept_name AS dept,
dept_emp.from_date    AS fromdate
FROM departments, dept_emp, employees
WHERE
departments.dept_no = dept_emp.dept_no AND
employees.emp_no = dept_emp.emp_no AND
dept_emp.to_date = '9999-01-01'
ORDER BY empid
LIMIT ?;]], 2
); assert(not err, err)
Employees = employees

function printmacro(macro, ...)
    local s = {string.char(92)}
    s[#s+1] = macro
    for _, a in ipairs({...}) do
```

```
        s[#s+1] = "{"..a.."}"
    end
    tex.print(table.concat(s))
end
function printqrcode(empid)
    local tmsg, err = QRCodeSrv
        :call("QRCODE", empid)
    assert(not err, err)
    local grid = libgrid:new(tmsg[2])
    tex.box["mybox"] = grid:boxnode()
    printmacro("box")
    printmacro("mybox")
end
}
\endgroup

\newbox\mybox

% macro to typeset each row
\def\printqrcode#1{%
\begin{minipage}[c]{40mm}
\directlua{printqrcode(#1)}
\end{minipage}}
%
\def\minicell#1#2#3#4#5#6{%
\begin{tabular}{@{}l@{}}
#1\\#2\\#3\\#4\\#5\\#6
\end{tabular}}

% typesetting the table
\begin{document}
\begin{tabular}{cl}
\toprule
\directlua{
local isfirst = true
for _, emp in libiter.rows(Employees) do
    if isfirst then
        isfirst = false
    else
        printmacro("midrule")
    end
    printmacro("printqrcode", emp.empid)
    tex.print("&")
    printmacro("minicell",
    "ID: "        ..emp.empid,
    "Name: "      ..emp.firstname
                  .." "..emp.lastname,
    "Birth Date: "..emp.birthdate,
    "Dept.: "     ..emp.dept,
    "Hire date: " ..emp.hiredate,
    "From: "      ..emp.fromdate
    )
    tex.print(string.char(92, 92))
end
}
\bottomrule
\end{tabular}
% closing services' connections
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
local err = QRCodeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

Dopo aver effettuato le connessioni ai due servizi per il QRCode e per l'accesso al database degli impiegati, come primo passo si esegue la query su tutte e tre le tabelle dell'archivio per reperire i dati.

Nella colonna `to_date` della tabella `dept_emp`, la data `'9999-01-01'` indica che l'impiegato in questione è in servizio presso il dipartimento, perciò nella query imporremo tale valore.

La query contiene come parametro il numero delle righe che si desidera vengano al massimo incluse nella tabella risultato. Questo valore — pari a 2 nell'esempio — verrà utilizzato dal database al momento dell'esecuzione della query come prepared statement. Ciò dimostra la possibilità di creare una macro TEX che tra gli argomenti accetti parametri per le interrogazioni verso il database.

Se non vi sono errori, il messaggio di risposta dal server in lettura sul database è contenuto nella variabile globale `Employees` perché possa essere iterata in seguito con la funzione `libiter.rows()`.

All'interno dell'ambiente `document` compare un ambiente `tabular` a due colonne: nella prima verrà inserito il QRCode e nella seconda i dati anagrafici per mezzo di due corrispondenti macro: `\printqrcode` e `\minicell`. Esse *stampano* i caratteri con la funzione `tex.print()` che poi TEX elaborerà.

Questa tecnica è molto semplice; tuttavia, nello scrivere il codice, occorre tener presente che tutti i caratteri stampati cominceranno a essere elaborati solo alla conclusione del codice Lua della macro `\directlua` che costruisce la tabella. Non funzionerebbe quindi generare la scatola orizzontale con il QRCode e stampare i caratteri `\box\mybox` per riempire la prima cella perché TEX giunto alla fase di espansione troverà l'ultimo QRCode nel primo `hbox` della prima riga della tabella ed espanderà una scatola ormai vuota nelle successive.

Una soluzione è quella di stampare nella prima cella una macro seguita dal codice da tradurre in QRCOde, per esempio `\printqrcode{10001}`. Questa macro conterrà il codice Lua per reperire dal servizio illustrato alla sezione 5 la stringa di zeri e uno che poi la libreria `libgrid` tradurrà in una scatola orizzontale con il contenuto grafico del simbolo.

L'espansione della `\directlua` sarà vuota ma TEX si ritroverà i seguenti caratteri al termine dell'iterazione sui record reperiti dal database, a meno degli spazi e delle interruzioni di riga inseriti per maggiore chiarezza:

```
\printqrcode{10001} & \minicell{10001}
{Georgi Facello}{1953-09-02}{Development}
{1986-06-26}{1986-06-26}
\midrule
\printqrcode{10002} & \minicell{10002}
{Bezalel Simmel}{1964-06-02}{Sales}
{1985-11-21}{1996-08-03}
```

La funzione Lua `printmacro()` accetta infatti un primo argomento come nome della macro che stampa nello streaming del compositore dopo avervi premesso il carattere di backslash, e un numero variabile di argomenti che stampa uno dopo l'altro in sequenza dopo averli inseriti tra graffe.

Da notare che non funzionerebbe la soluzione di usare la funzione `node.write()` di LuaTEX per riempire la prima cella di ciascuna riga della tabella dei badge perché avrebbe l'effetto di inserire immediatamente la scatola orizzontale contenente il QRCode nella lista principale del compositore, e quindi inserirebbe i codici a barre tutti nella prima cella della prima riga.

Usare la tecnologia LuaTEX dei nodi per realizzare interamente la tabella darebbe ovviamente il risultato corretto ma esula dagli obiettivi dell'articolo che non sono incentrati sul problema di come far fluire i dati da Lua verso il compositore.

8 Risorse

In questa sezione citerò le risorse, sia software sia di documentazione, per la comprensione e l'esecuzione del codice riportato in questo articolo. Sono richieste una preparazione di base sulla programmazione e la familiarità con la riga di comando della shell.

Nonostante i software impiegati, tutti rigorosamente open source, siano disponibili per diversi sistemi operativi, tutte le analisi, le prove e le compilazioni sono state effettuate in ambiente Linux, in particolare per la distribuzione Xubuntu basata su Debian. I dettagli d'installazione potranno quindi essere differenti per i sistemi operativi diversi da quello del pinguino.

Prepariamo gli ingredienti per questa nostra particolare ricetta che sono, nell'ordine, il motore di composizione di nuova generazione LuaTEX, il linguaggio Rust e la libreria ZeroMQ.

8.1 LuaTEX

Il riferimento principale per LuaTEX, in particolare per la versione 1.0.4 distribuita con la TEX Live 2017, è il suo manuale curato direttamente dal team di sviluppo (The LuaTEX development team, 2017) e accessibile da terminale digitando il comando usuale:

```
$ texdoc luatex
```

Nel manuale non si trovano ancora informazioni sulla FFI — Foreign Function Interface, si veda in proposito l'articolo (Hagen e Scarso, 2017) — poiché, a differenza di LuajitTEX, in questo motore di composizione esse sono state aggiunte di recente a livello sperimentale. Un utile riferimento si trova sul sito del progetto LuaJIT `http://luajit.org/ext_ffi_tutorial.html`, poiché l'estensione dovrebbe esporre la stessa API sia per LuaTEX che per LuajitTEX.

Per l'installazione di TEX Live 2017, e quindi di LuaTEX, conviene scegliere una delle modalità previste e specificate alla pagina `https://www.tug.org/texlive/`. Se si dispone già della distribuzione 2017 o successiva, raccomando di aggiornare TEXLive con il comando da adattare in base al percorso dell'eseguibile `tlmgr`:

```
$ sudo /path/to/tlmgr update --all
```

8.2 Rust

Il C sarebbe dovuto essere uno dei protagonisti di questo lavoro, tuttavia non sono in confidenza con questo linguaggio. D'altra parte Rust, il relativamente nuovo linguaggio open source patrocinato da Mozilla Foundation, ha come obiettivo principale la sicurezza e la stabilità dei programmi senza che i controlli ne appesantiscano l'esecuzione. Inoltre Rust è in grado di interoperare da e verso il C facilmente e può produrre file binari per librerie statiche o dinamiche, o anche file oggetto.

Rust nasce con un sistema di gestione della memoria progettato per eliminare i problemi di consistenza e sicurezza dei dati tipici del C come i casi di puntatori a oggetti non più esistenti — i *dangling pointer* —, di memoria allocata e non più utile — il *memory leak* — o l'uso di variabili non inizializzate. Tutti questi casi sono intercettati in fase di compilazione e non comportano un aggravio dei tempi di elaborazione durante l'esecuzione del codice, mentre il superamento degli indici di un array provoca l'uscita del programma.

8.2.1 Uno sguardo a Rust

In Rust ci sono molti nuovi concetti che derivano dal complesso di studi ed esperienze fatti sia nell'area dei linguaggi in stile C, sia nell'area dei linguaggi funzionali come Haskell e OCalm. In questa sezione cercherò di illustrare in modo semplificato la caratteristica principale di questo linguaggio progettato per la programmazione di sistema e del suo compilatore chiamato `rustc` che produce codice macchina nativo e che ha tra i suoi componenti fondamentali LLVM al pari del linguaggio Clang utilizzato per gli stessi scopi sulle piattaforme OSX.

Come è naturale attendersi, le differenze tra Rust e Lua, il linguaggio incorporato in LuaTEX, sono profonde: in Rust occorre preoccuparsi di *come* i dati vengono memorizzati e di *quali* diritti si dispone per elaborarli. Il beneficio che si ottiene è un codice molto più veloce e sicuro, adatto per progetti come componenti di sistemi operativi, driver e moduli di basso livello. Uno dei principali progetti sviluppati in Rust è Servo, il browser web di Mozilla che dovrebbe nel prossimo futuro sostituire Firefox. Tuttavia la sintassi moderna e altre caratteristiche come l'inferenza dei tipi avvicinano Rust alla produttività dei linguaggi dinamici come Lua e Python.

In Rust la chiave del controllo statico della memoria è la semantica dell'*ownership*. Il linguaggio

chiarisce e rende espliciti i concetti della rappresentazione in memoria dei dati al prezzo di una curva di apprendimento più ripida e un maggiore impegno nella progettazione. Nel modello di memoria di Rust vi sono due modi di memorizzare un valore:

unboxed value dato nello *stack*; la memoria è liberata automaticamente non appena cessa la validità dello stack, ovvero al termine dello *scopo* per cui è stato definito;

owned boxed dato nell'*heap*, la memoria dinamica indipendente dai blocchi di scopo delle variabili, e creazione di un riferimento ad esso nello stack *unico proprietario* del dato; la memoria è liberata automaticamente quando ha termine lo scopo in cui è definito il riferimento, tenendo conto dei *passaggi di proprietà*.

I dati unboxed sono oggetti la cui vita è regolata dal blocco in cui vengono creati. Nel momento in cui questo contesto termina di esistere, anche gli oggetti che contiene vengono eliminati. Un *box* invece fa riferimento a un'area nell'*heap*: i valori in questa memoria meno efficiente dello stack, assumono una vita indipendente dal contesto. Nasce il problema di come rendere sicuro l'uso di questi dati e in particolare di come distruggerli.

In C l'operazione è manuale! Nel codice occorre fare estrema attenzione alla validità dei puntatori e all'eliminazione volontaria dei dati. All'opposto in Lua l'operazione è automatica. Un componente software, chiamato *Garbage Collector*, durante l'esecuzione del programma controlla quali oggetti non sono più raggiungibili ovvero non esiste più alcun riferimento che li indirizzi. Per scelta di progetto, gli errori di memoria in Rust sono intercettati a tempo di compilazione e ciò rende in pratica inutile un Garbage Collector.

La soluzione adottata in Rust è far sì che il boxed value, di tipo `T`, sia di *proprietà* del riferimento, di tipo `Box<T>`: quando il riferimento esce di scopo viene eseguito il corrispondente distruttore che libera correttamente la memoria indirizzata. In gioco ci sono quindi due oggetti: il primo è il boxed value che occupa un'area della memoria dinamica dell'heap e il secondo è il suo riferimento "*owner*" che si trova memorizzato nella memoria di lavoro dello stack e di cui il compilatore è in grado di conoscerne la validità in fase di compilazione.

La proprietà di un boxed value può cambiare grazie a quella che viene chiamata *move semantic*. Questo consente di prolungare la vita di un dato dell'heap oltre a un singolo constesto di scopo.

Il prossimo esempio evidenzia come non sia possibile violare la regola di unicità del riferimento proprietario di un boxed value. Se copiamo il riferimento in uno nuovo — la variabile `a` nella `b` nel codice — quello precedente non può più essere utilizzato. La proprietà passa al nuovo riferimento:

```
let mut a: Box<i32> = Box::new(100);
let b = a;
*a += 100; // <- compiler error:
// use of moved value: `*a`
```

così come non è possibile istanziare più di un puntatore mutevole, ovvero un riferimento in grado di modificare l'oggetto indirizzato:

```
let mut a: Box<i32> = Box::new(100);
let p1 = &mut a;
let p2 = &mut a; // <- compiler error:
// cannot borrow `a` as mutable more than
// once at a time
```

oppure ancora non è possibile modificare un valore di cui già esiste un puntatore mutevole:

```
let mut a: Box<i32> = Box::new(100);
let p1 = &mut a;
*a += 1; // <- compiler error:
// cannot assign to `*a` because it is borrowed
```

Ne consegue che una funzione non può creare valori boxed e restituirne solamente un puntatore:

```
fn crea(n: i32) -> &'static i32 {
    let mut b: Box<i32> = Box::new(n);
    &b // <- compiler error:
    // `b` does not live long enough
}

fn main() {
    let a = crea(100);
}
```

dovrà invece restituire il riferimento boxed la cui proprietà passerà alla variabile nel contesto della chiamata della funzione:

```
fn crea(n: i32) -> Box<i32> {
    Box::new(n) // OK
}

fn main() {
    let b = crea(100); // 'b' is the new owner
    let p = &b;
}
```

Altra caratteristica di Rust utile da conoscere è la non esistenza del tipo nullo, fonte di problemi nei programmi in C. Viene sostituito dal tipo generico `Option<T>` implementato nella libreria standard come `enum`, tipo che ammette che si possa associare dati alle varianti. La particolare enumerazione `Option<T>` consta di due varianti: `Some<T>` e `None`, la prima per il caso di esistenza del dato e la seconda per il caso contrario.

Nella libreria standard esiste anche un'altra enumerazione che evita l'uso del tipo nullo, il tipo generico `Result<T, E>` definito nella libreria standard come:

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

con la prima variante si rappresenta un risultato corretto di tipo `T`, mentre con la seconda si rappresenta un errore di tipo `E`.

Per esempio, una funzione che restituisce il rapporto tra due interi potrebbe essere definita così[11]:

```
fn divide(n: i32, d: i32)
    -> Result<i32, &'static str> {
    if d == 0 {
        Err("Divisor is equal to zero")
    } else {
        Ok(n/d)
    }
}

fn main() {
    println!("{}", divide(518, 0).unwrap())
}
```

Il metodo `unwrap()` del tipo `Result` restituisce il valore associato alla variante `Ok` oppure produce l'interruzione del programma a runtime.

8.2.2 Risorse su Rust

La documentazione su Rust è già abbastanza nutrita anche se il primo riferimento disponibile, chiamato *the book* e sviluppato con il modello open source (The Rust Community, 2017), è attualmente in fase di completa riscrittura. Tra breve ne uscirà anche una versione a stampa (Klabnik e Nichols, 2017), come pure è quasi terminato il testo Blandy e Orendorff (2016).

8.2.3 Installare Rust

Per installare Rust su Linux — in questo lavoro ho utilizzato la versione 1.19 stabile — di solito si scarica lo script `rustup-init.sh` dal sito ufficiale `https://rustup.rs/` e lo si esegue oppure, più brevemente, si lancia il seguente unico comando da terminale:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Se di Rust avete già un'installazione allora aggiornatela all'ultima disponibile, sapendo che l'avanzamento di versione avviene con un ciclo semestrale, con il comando:

```
$ rustup update stable
```

Una volta completate le procedure automatiche d'installazione, oltre al compilatore `rustc`, si hanno a diposizione `cargo`, il gestore dei pacchetti e dei task di compilazione, `rustdoc` per la generazione della documentazione e `rustfmt`, un'utilità che formatta il codice secondo uno standard. È possibile leggere il *libro* della guida con il comando:

```
$ rustup doc --book
```

mentre la documentazione completa della libreria standard si apre altrettanto comodamente nel browser con il comando:

```
$ rustup doc --std
```

8.3 ZeroMQ

E veniamo all'ultimo dei nostri ingredienti, ZeroMQ, una libreria scritta in C che semplifica lo scambio di messaggi tra due programmi attraverso un protocollo comune. Questa comunicazione può avvenire con diversi tipi di schemi a seconda delle necessità, il più semplice dei quali è lo schema[12] client/server denominato REQUEST/REPLY: un programma server in ascolto a un certo indirizzo, risponde alle richieste dei client, che possono essere sorgenti LuaTEX in fase di compilazione.

Nelle prossime due sezioni vedremo come documentarsi e come installare i moduli necessari per usare ZeroMQ, questa incredibile libreria che è stata adottata anche dal CERN di Ginevra per creare alcuni componenti chiave di gestione degli esperimenti sugli acceleratori di particelle (Dworak *et al.*, 2011).

8.3.1 Risorse su ZeroMQ

Il riferimento principale che troviamo in diverse forme è la guida chiamata `zguide` e scritta da Pieter Hintjens, uno dei principali sviluppatori, purtroppo scomparso l'anno scorso. La guida si può leggere online alla pagina `http://zguide.zeromq.org/page:all`, oppure nel libro scaricabile nel formato PDF (Hintjens, 2013a). Ne esiste anche un libro pubblicato per i tipi di O'Reilly (Hintjens, 2013b).

8.3.2 Installare ZeroMQ

La libreria ZeroMQ è scritta in linguaggio C; molti progetti, tuttavia, mettono a disposizione il *binding*[13] per molti altri linguaggi e piattaforme compreso Lua.

La procedura che ho messo a punto su un sistema Linux Debian-based comincia con l'installazione dei compilatori C e C++, dei file di sviluppo di Lua 5.2, e di quelli di ZeroMQ, nel momento in cui scrivo alla versione 4.1.4, utilizzando i repository ufficiali della distribuzione:

```
$ sudo apt-get install build-essential
$ sudo apt-get install liblua5.2-dev
$ sudo apt-get install libzmq3-dev
```

Se uno di questi pacchetti fosse già installato sul vostro sistema, non c'è da preoccuparsi, il comando corrispondente non avrà effetto. Si deve invece fare attenzione alla versione del pacchetto di sviluppo

11. Il costrutto `if` in Rust è un'espressione, restituisce cioè un risultato. Il vincolo è che tutti i rami del costrutto restituiscano lo *stesso* tipo, cosa verificata al solito dal compilatore.

12. Questi schemi vengono chiamati *communication pattern* o *message pattern*.

13. In generale per *binding* s'intende un componente software in grado di accedere alle funzioni di una libreria sottostante mettendone a disposizione un'interfaccia per un diverso linguaggio di programmazione.

di Lua perché questa deve corrispondere a quella dell'interprete Lua interno a LuaTEX attualmente alla 5.2.

Procediamo adesso con il preparare il package manager di Lua chiamato Luarocks, che in passato mi ha dato qualche problema. Consiglio allora di utilizzare l'ultima versione disponibile, a oggi la 2.4.2, scaricando e compilando i sorgenti. I dettagli si trovano alla pagina https://github.com/luarocks/luarocks/wiki/Download.

Scarichiamo e decomprimiamo i sorgenti con i comandi da digitare su un unica riga:

```
$ wget http://luarocks.github.io/luarocks/ \
  releases/luarocks-2.4.2.tar.gz
$ tar -xvzf luarocks-2.4.2.tar.gz
```

Impostiamo la directory di lavoro della finestra di terminale alla cartella decompressa e diamo i classici comandi di costruzione:

```
$ cd luarocks-2.4.2
$ ./configure
$ make build
$ sudo make install
```

Sul sito di ZeroMQ alla pagina http://zeromq.org/bindings:lua vengono citati due progetti che si occupano di fornire agli utenti il binding per Lua. Dopo qualche riflessione ho scelto il progetto `lzmq` ospitato su GitHub alla pagina https://github.com/zeromq/lzmq. In poco tempo, leggendo la documentazione si riesce facilmente a trovare i due ultimi comandi:

```
$ sudo luarocks install lua-llthreads2
$ sudo luarocks install lzmq
```

8.3.3 Prova del funzionamento

Terminate le operazioni d'installazione, conviene eseguire un piccolo script in Lua che stampa la versione di ZeroMQ e i nomi delle funzioni del primo livello della tabella che contiene `lzmq`:

```
-- lzmq test script
local zmq = require "lzmq"

local zver = table.concat(zmq.version(), '.')
print("0MQ version: " .. zver)
print()
print(" -> Main level function:")
for k, v in pairs(zmq) do
    if type(v) == "function" then
        print(k.."()")
    end
end
```

Se si ottiene un output simile al seguente, è molto probabile che ZeroMQ con il suo binding Lua `lzmq` sia stato installato correttamente:

```
0MQ version: 4.1.4

 -> Main level function:
msg_init_data_array()
msg_init_data()
z85_decode()
init_socket()
curve_keypair()
error()
has()
strerror()
msg_init_data_multi()
poller()
assert()
init()
msg_init()
context()
proxy()
msg_init_size()
proxy_steerable()
init_ctx()
z85_encode()
version()
device()
```

8.3.4 Altre vie

Più facile è la procedura d'installazione di ZeroMQ e più lo sarà usare le sue stupefacenti funzionalità da parte degli utenti TEX. Per far si che ZeroMQ sia disponibile vi sono almeno altre due strade.

La prima è utilizzare `swiglib` http://www.luatex.org/swiglib.html un progetto del team LuaTEX creato appositamente per facilitare l'uso di librerie esterne con questo motore di composizione — maggiori informazioni sul progetto si trovano nell'articolo Scarso (2013a). Esso si basa sul programma Swig http://www.swig.org/, un tool di sviluppo che, sulla base dei file di header e di un file di configurazione — non semplicissimo da scrivere come è ovvio attendersi —, crea un sorgente in C pronto da compilare per la creazione del binding a una data libreria C o C++ e per un dato linguaggio.

La seconda strada è quella di avvalersi del modulo FFI presente in LuajitTEX attraverso il progetto `lua-zmq` https://github.com/Neopallium/lua-zmq.

8.4 Shell editors

Ho editato i sorgenti pdfLATEX di questo lavoro con un editor che ormai uso da qualche settimana: Visual Code Editor di Microsoft https://code.visualstudio.com/, installandone la versione per Linux.

In questi editor avanzati multipiattaforma ed estensibili per la scrittura di codice sorgente — Atom ne è un altro esempio — non ricerco particolari utilità rivolte alla compilazione TEX o alla *code completion* del formato LATEX, ma uno spazio di lavoro il più confortevole possibile per l'editazione del testo, ed è abbastanza sorprendente come gli sviluppatori siano riusciti in questi anni a incrementare l'esperienza d'uso degli editor per testi, un tipo di programma che si pensava non avesse più niente di nuovo da dire.

Non esiste l'editor ideale per il sistema TEX ma se ne possono sempre utilizzare due a seconda dell'attività che si sta svolgendo sui sorgenti: costruzione e modifica del testo con un editor avanzato, compilazione e revisione finale delle bozze con uno shell editor per TEX, e TEX Works funziona benissimo per questo compito.

9 FFI e Lua API

9.1 FFI di LuaTEX via Rust

Come esempio dimostrativo della tecnologia FFI mostrerò in questa sezione applicativa come compilare una libreria dinamica in Rust, il linguaggio relativamente nuovo per la programmazione di sistema patrocinato da Mozilla Foundation, per eseguirne l'unica funzione in LuaTEX.

Il sorgente in Rust ha solo quattro righe:

```
#[no_mangle]
pub extern fn double_input(n: i32) -> i32 {
    n * 2
}
```

L'unica funzione, chiamata `double_input()`, è stata marcata con la direttiva `no_mangle` in modo che il compilatore non ne modifichi il nome nel file binario, operazione che in pratica azzera il rischio di collisioni degli identificativi. La keyword `extern` indentifica la funzione come aderente alle specifiche ABI, acronimo di Application Binary Interface.

Caratteristica di Rust è che ogni cosa è un'espressione, persino l'intero corpo di una funzione, perciò non è necessario aggiungere l'istruzione `return` con il doppio dell'argomento, un intero con segno a 32 bit.

Per ottenere il file della libreria è sufficiente fare ricorso a `cargo`, il package manager ufficiale fornito assieme agli altri strumenti di Rust, scrivendo nel file `Cargo.toml` (posizionato nella directory principale del progetto) il codice nel formato TOML:

```
[package]
name = "lib"
version = "0.1.0"
authors = ["your name <yourmail@amail.com>"]

[lib]
name = "double_input"
crate-type = ["dylib"]
```

Di importante in questo *manifest* è il valore `dylib` che sta per *dynamic library*, fornito alla chiave `crate-type`.

Con il comando:

```
$ cargo build
```

il package manager si occuperà di risolvere le eventuali dipendenze — assenti nel nostro esempio —, di compilare il sorgente e di creare il file corrispondente alla libreria dinamica per il dato sistema operativo. Lavorando in Linux e Mac, questo file avrà estensione `.so` che sta per *shared object*, mentre sotto Windows avrà l'estensione `.dll` che sta per *dynamic link library*.

Tornando a LuaTEX, o meglio a LuaLATEX, possiamo adesso caricare questa libreria dinamica via FFI:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass{minimal}
\begin{document}
\directlua{
local ffi = require('ffi')

ffi.cdef[[
int32_t double_input(int32_t input);
]]

local lib = ffi.load("doubleinput.so")
local res = lib.double_input(12)

tex.print("12 * 2 = " .. res)
}
\end{document}
```

Come si può notare leggendo il listato, usare le funzionalità FFI è abbastanza semplice. Quel che occorre fare è fornire al modulo `ffi` la firma delle funzioni che si intendono usare della libreria con la funzione `cdef()`, e poi caricare la libreria con la funzione `load()` che restituisce il riferimento `lib`, contenente la funzione esterna nella chiave `double_input`.

Quel che invece è più impegnativo è capire quali tipi del C corrispondano a quelli della libreria dinamica. Nel caso dell'esempio si tratta di trovare la corrispondenza con il tipo intero a 32 bit con segno `int32_t`.

Come è logico aspettarsi, è un terreno questo che coinvolge la programmazione di basso livello e che richiede ottime conoscenze tecniche in particolare in quelle situazioni in cui gli oggetti binari producono crash del programma senza alcun messaggio d'errore.

9.2 L'API di Lua via Rust

Lavorando direttamente con l'API di Lua si nota che la maggior parte delle sue funzioni hanno come primo argomento il puntatore all'oggetto `Lua State` che offre l'accesso allo stack d'interfaccia. Inoltre, esse restituiscono sempre un intero che rappresenta il numero di dati inseriti nello stack.

Per l'esempio dimostrativo di questa tecnologia descriverò passo-passo il processo di creazione e compilazione della libreria affinché si possa facilmente ripetere l'esperimento una volta si sia installato Rust sul proprio sistema.

Nel seguente sorgente il codice ha lo scopo di estendere Lua con due funzioni chiamate `magic()` e `sin()`. La prima funzione dimostra la creazione di una tabella Lua attraverso le funzione dell'API e la seconda dimostra un calcolo basato sull'uso dei numeri in virgola mobile. Il listato è suddiviso

in tre parti: l'iniziale definizione dei simboli che corrispondono alle funzioni dell'API di Lua che saranno disponibili a runtime, la definizione delle funzioni utili e la predisposizione della funzione di inizializzazione dal formato fissato dalle specifiche Lua, che verrà eseguita al caricamento della libreria dinamica:

```
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(dead_code)]

extern crate libc;
use libc::{c_int, c_char, c_double};

use std::ffi::CString;

pub type lua_State = libc::c_void;

type lua_Integer = libc::ptrdiff_t;
type lua_Number = c_double;


// Type for native functions that can be
// passed to Lua.
type lua_CFunction = Option<
    unsafe extern "C"
    fn(L: *mut lua_State) -> c_int
>;

#[repr(C)]
struct luaL_Reg {
  name: *const c_char,
  func: lua_CFunction,
}

extern "C" {
    fn lua_createtable(
        L: *mut lua_State,
        narr: c_int,
        nrec: c_int
    );
    fn lua_pushinteger(
        L: *mut lua_State, n: lua_Integer
    );
    fn lua_settable(
        L: *mut lua_State, idx: c_int
    );
    fn lua_pushnumber(
        L: *mut lua_State, n: lua_Number
    );
    fn luaL_setfuncs(
        L: *mut lua_State,
        l: *const luaL_Reg,
        nup: c_int
    );
    fn luaL_checknumber(
        L: *mut lua_State,
        arg: c_int
    ) -> lua_Number;
}

// This test function creates the
// Lua table {5, 10, 15}
unsafe extern "C" fn magic(L: *mut lua_State)
    -> c_int {
    lua_createtable(L, 3, 0);

    lua_pushinteger(L, 1);
    lua_pushinteger(L, 5);// 5
    lua_settable(L, -3);

    lua_pushinteger(L, 2);
    lua_pushinteger(L, 10);// 10
    lua_settable(L, -3);

    lua_pushinteger(L, 3);
    lua_pushinteger(L, 15);// 15
    lua_settable(L, -3);
    1
}

unsafe extern "C" fn sin(L: *mut lua_State)
    -> c_int {
    let x = luaL_checknumber(L, 1);
    lua_pushnumber(L, x.sin());
    1
}

// the format of this function name is
// defined by the Lua manual; the Lua
// interpreter will call into this when you
// require() this library
#[no_mangle]
pub extern "C" fn luaopen_libmagic(
        L: *mut lua_State
    ) -> c_int {
    unsafe { lua_createtable(L, 0, 1); }

    let fn_magic = CString::new("magic")
        .unwrap();
    let fn_sin = CString::new("sin")
        .unwrap();

    let reg = [
        luaL_Reg {
            name : fn_sin.as_ptr(),
            func : Some(sin)
        },
        luaL_Reg {
            name: fn_magic.as_ptr(),
            func: Some(magic)
        },
        luaL_Reg {
            name: std::ptr::null(),
            func: None
        },
    ];
    unsafe {
        luaL_setfuncs(L, reg.as_ptr(), 0);
    }
    1
}
```

Al sorgente in Rust si aggiunge il seguente file di *manifest* `Cargo.toml` perché per mezzo del package manager si possa compilare correttamente la libreria dinamica:

```
[package]
name = "magic"
```

```
version = "0.1.0"
authors = ["yourname <yournick@mail.com>"]

[dependencies]
libc = "0.2.26"

[lib]
crate-type = ["dylib"]
```

Al termine della compilazione possiamo verificare in LuaTEX il funzionamento della libreria di prova scritta secondo l'API nativa di Lua. Per l'utente Lua esse appaiono come normali funzioni eseguibili attraverso i nomi assegnati nella fase di apertura del file binario da parte di `require()`.

```
% !TeX program = LuaTeX
\directlua{
local libmagic = require "libmagic"

local t = libmagic.magic()
tex.print(t)
tex.print([[ ]])
tex.print(libmagic.sin(0.56))
}
\bye
```

10 QRCode Server

In questa sezione e nelle successive due riporterò alcune osservazioni sull'implementazione dei server, ovvero dei programmi connessi tramite protocollo standard a oggetti Socket della libreria ZeroMQ. Tutti i server sono scritti in Rust, un linguaggio molto diverso da Lua, ma che è possibile far interagire con LuaTEX con la tecnologia dello scambio di messaggi neutri, per estenderne le capacità.

Passando alla costruzione del server, il punto di vista sarà quello opposto rispetto al client, che richiede elaborazioni senza conoscere il componente che effettivamente le esegue.

Il QRCode server, di cui è riportata una sessione di lavoro nella figura 8, utilizza il progetto `https://github.com/kennytm/qrcode-rust` per la codifica QRCode e il progetto `rust-zmq` già introdotto alla sezione 4.1 per l'esecuzione delle funzione native di ZeroMQ.

Per questo server, come per gli altri due, il formato dei messaggi multiparte è definito nella corrispondente sezione precedente che descrive gli esempi di utilizzo in LuaTEX.

10.1 Un server robusto

Il server è progettato per essere robusto, ovvero per continuare a funzionare anche nel caso in cui si verifichino errori di comunicazione oppure conseguente al contenuto del messaggio. Come in C, il punto d'ingresso dell'esecuzione del programma è la funzione `main()`. Essa si basa su un ciclo infinito, all'interno del quale per ogni messaggio ricevuto viene elaborata la risposta e inviata al mittente:

```
extern crate qrcode;
extern crate zmq;

use qrcode::QrCode;
use std::error::Error;
use zmq::{Context, Socket, Message, SNDMORE};

fn main() {
    let ctx = Context::new();
    let skt = ctx.socket(zmq::REP)
                    .unwrap();
    assert!(
        skt.bind("tcp://*:5556")
            .is_ok()
    );
    let mut msg = Message::new()
                          .unwrap();
    println!("Server ready on port 5556...");
    loop {
        // get the task object
        let task = match
        recv_and_build_task(&skt, &mut msg) {
            Ok(task) => task,
            Err(e) => {
                match
                send_err(&skt, e.as_str()) {
                    Ok(_) => {},
                    Err(e) =>
                    println!("Error [{}]", e)
                }
                continue
            }
        };

        match send_result(&skt, &task) {
            Ok(_) => {},
            Err(e) => println!("Err [{}]", e)
        }
    }
}
```

La funzione `recv_and_build_task()` si occupa di ricevere il messaggio dal client. Se non ci sono stati errori nella ricezione oppure nel contenuto del messaggio multiparte rispetto al formato atteso, essa restituisce l'oggetto di tipo `QrTask`, una `struct` con la sequenza di zeri e uno del testo già decodificato:

```
fn recv_and_build_task(
    sk: &Socket, msg: &mut Message
    ) -> Result<QrTask, String> {
    // get the message
    let mut multipart = vec![];
    loop {
        match sk.recv(msg, 0) {
            Ok(_) => {},
            Err(e) => return Err(
                e.description().to_string()
            )
        }

        let part =
        if let Some(s) = msg.as_str() {
            s.to_string()
```

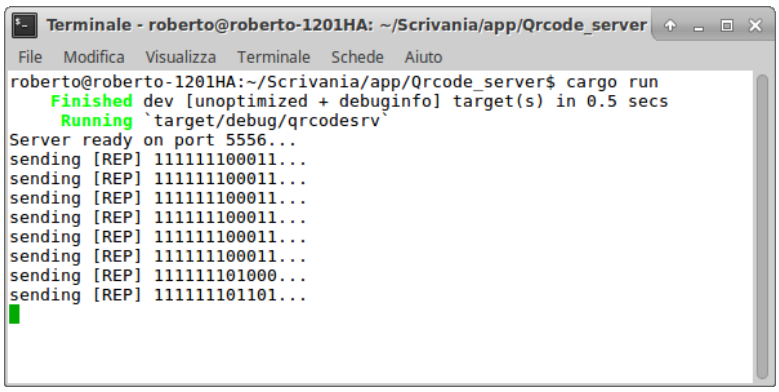


Figura 8: Immagine del terminale presa durante l'esecuzione del server che fornisce la codifica QRCode di stringhe di testo. Ogni volta che il server risponde a un messaggio, stampa in console un testo di notifica sia in caso di successo sia in caso di errore.

```
        } else {
            return Err(
                "Malformed UTF-8 command"
                .to_string()
            )
        };
        multipart.push(part);
        if !msg.get_more() {
            break
        }
    }
    if multipart.len() != 3 {
      return Err(
      "Unexpected number of message's frames"
      .to_string()
      )
    }
    if multipart[0] != "QRCODE" {
       return Err(
       "The first frame 'QRCODE' is required"
       .to_string()
       )
    }
    if multipart[2] != "END" {
       return Err(
           "The last frame 'END' is required"
           .to_string()
       )
    }
    QrTask::new(multipart[1].as_str())
}
```

In Rust, come anche in Go — altro moderno linguaggio sviluppato da Google —, non troviamo il paradigma della programmazione a oggetti. Tuttavia è possibile aggiungere molto semplicemente il metodo `new()` a `QrTask` tramite il costrutto `impl`. In questo caso il metodo *costruttore* decodifica il testo nella stringa di zero e uno a rappresentare i quadratini del simbolo QRCode rispettivamente spento o acceso, restituendo l'oggetto oppure un errore tramite il tipo `Result` di cui accenno al termine della sezione 8.2.1:

```
struct QrTask {
    qrenc : String,
}

impl QrTask {
    fn new(code: &str)
    -> Result<QrTask, String> {
        let qrcode = match
        QrCode::new(code.as_bytes()) {
            Ok(qrc) => qrc,
            Err(e) =>
                return Err(format!("{}", e))
        };
        let qrenc = qrcode.render::<char>()
                    .quiet_zone(false)
                    .module_dimensions(1, 1)
                    .light_color('0')
                    .dark_color('1')
                    .build();
        Ok(QrTask { qrenc : qrenc })
    }
}
```

Non rimangono che le funzioni di invio dei messaggi. In caso di esito corretto viene chiamata la funzione `send_result()` mentre in caso si sia verificato qualche errore viene chiamata `send_err()` con una descrizione breve di cosa è andato storto. Sarà responsabilità del client gestire o ignorare l'eventuale messaggio d'errore:

```
fn send_result(sk:&Socket, task: &QrTask)
    -> zmq::Result<()> {
    println!(
        "sending REPLAY {}...",
        &(task.qrenc)[..12]
    );

    // frame 0: <id>
    let e0 = sk.send_str("QRCODE", SNDMORE);
    if e0.is_err() {
        return e0
    }

    let e1 = sk.send_str(
        task.qrenc.as_str(), SNDMORE
    );
    if e1.is_err() {
        return e1
    }
```

```
    sk.send_str("END", 0)
}

fn send_err(sk: &Socket, errmsg: &str)
    -> zmq::Result<()> {
    println!("sending [ERR] {}", errmsg);
    // frame 0
    let e0 = sk.send_str("ERR", SNDMORE);
    if e0.is_err() {
        return e0
    }
    // frame 1
    let e1 = sk.send_str(errmsg, SNDMORE);
    if e1.is_err() {
        return e1
    }
    sk.send_str("END", 0) // frame 2
}
```

11 Prime Server

Il secondo server che descriverò in dettaglio sfrutta la genericità dei messaggi per fornire più di un servizio, in particolare per elaborare dati sui numeri primi. Anche in questo caso il server è implementato in Rust con lo scopo di mantenersi attivo anche in caso di errori.

Volendo evidenziare le parti di codice interessanti di questo server, ci concentreremo solamente su come sono implementate le tre diverse funzioni della verifica di primalità, del conteggio dei primi in un intervallo e la restituzione della lista dei primi in un dato intervallo.

L'idea è che l'implementazione sia tale da consentire l'aggiunta di nuove funzioni al server senza alterare il corpo principale del codice. Ciò è realizzabile solamente se si rende astratto il tipo che rappresenta una funzione e questo in Rust è possibile utilizzando la tecnologia chiamata *object trait* per cui un oggetto è tenuto a mettere a disposizione certe funzioni identificate precisamente da un costrutto `trait`. La gestione di quest'oggetto sarà poi resa indiretta con l'uso di puntatori.

Il compilatore Rust impone vincoli piuttosto stretti al comportamento degli oggetti che si vuole siano oggetti interfaccia, mentre a runtime la tecnica chiamata *dynamic dispatch* basata su una tabella di funzioni, individuerà il metodo e l'oggetto nascosto dietro al tipo interfaccia.

Il *trait* che rappresenta la generica elaborazione sul server chiamato `TaskData` definisce due sole funzioni:

```
trait TaskData {
    fn get_name(&self) -> &str;
    fn resolve(&self, &[bool]) -> Vec<u64>;
}
```

Un oggetto qualsiasi che implementi `TaskData` dovrà definire le funzioni con gli stessi argomenti e con gli stessi tipi di uscita; in altre parole, con la stessa firma.

La funzione `get_name()` è utilizzata per definire il contenuto del primo frame del messaggio di risposta. Si tratta di una funzione perché a oggi Rust non consente che il *trait* possa definire tipi che non siano funzioni. Il metodo `resolve()` è invece quello centrale. Esso riceve come argomento uno *slice* di valori booleani che rappresenta la primalità dei numeri fino a un valore massimo e restituisce un vettore di interi a 64 bit senza segno.

Se per esempio è richiesta la lista dei primi in un intervallo, il vettore risultato conterrà in ordine crescente tale elenco mentre se è richiesto il conteggio dei primi, il vettore conterrà un solo numero corrispondente. In questo modo, non particolarmente elegante in Rust, siamo in grado di eseguire diverse elaborazioni mantenendo per le funzioni la stessa firma.

La verifica della primalità si può aggiungere al server introducendo il nuovo tipo `PCheck` — una `struct` che incorpora il numero che occorre verificare — definendone il costruttore `new()` che, per la regola fissata, si aspetta l'argomento incluso in un vettore di un unico elemento, e infine implementando il `trait TaskData`:

```
struct PCheck<'a> {
    name : &'a str,
    n    : u64
}
impl<'b> PCheck<'b> {
    fn new<'a>(args: Vec<u64>)
    -> Result<Box<PCheck<'a>>, &'a str> {
        if args.len() == 1 {
            Ok(Box::new(PCheck {
                name : "PRIME-CHECK",
                n : args[0]
            }))
        } else {
            Err("Wrong number of arguments")
        }
    }
}
impl<'c> TaskData for PCheck<'c> {
    fn get_name(&self) -> &str {
        self.name
    }
    fn resolve(&self, signal: &[bool])
    -> Vec<u64> {
        let isprime = match self.n {
            0 | 1 => false,
            2     => true,
            x if x % 2 == 0 => false,
            x => {
                signal[(x as usize - 3)/2]
            }
        };
        // [0] == false, [1] == true
        if isprime {
            vec![1]
        } else {
            vec![0]
        }
    }
}
```

Il metodo `resolve()`, in particolare, tratta prima il caso in cui si richieda di verificare 0 e 1, poi il numero 2 e i numeri pari e per ultimo il resto dei numeri dispari. La sequenza calcolata di valori booleani che traccia i numeri primi considera solo i valori dispari a partire da 3, perciò l'intero x è primo oppure no a seconda che nel vettore all'indice $(x-3)/2$ è contenuto `true` o `false`.

A questo punto non rimane che scrivere il metodo costruttore per l'oggetto interfaccia sulla base della chiave contenute nel frame 0 del messaggio:

```
impl TaskData {
    fn newtask(cmd: &str, arg: Vec<u64>)
    -> Result<Box<TaskData>, &str> {
        match cmd {
            "PRIME-CHECK" => {
                match PCheck::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            "PRIME-LIST" => {
                match PList::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            "PRIME-COUNT" => {
                match PCount::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            _ => {
                println!("{}", cmd);
                Err("Unrecognized command")
            }
        }
    }
}
```

Qualsiasi sia l'oggetto che implementa un dato tipo di elaborazione, la seguente funzione `send_result()` sarà sempre la stessa, ricevendo infatti un riferimento *trait object* risolto a runtime poiché a tempo di compilazione si conoscono solamente le funzioni definite dall'interfaccia:

```
fn send_result(
        sk   : &Socket,
        prime: &[bool],
        tk   : Box<TaskData>
    ) -> zmq::Result<()> {
    let res = tk.resolve(prime);
    let n = res.len();
    println!("[OK] sending {} numbers", n);

    // frame 0: <id>
    let e1 = sk.send_str(
        tk.get_name(), SNDMORE
    );
    if e1.is_err() {
        return e1
    }

    for n in res {// result
        let s = n.to_string();
        let e2 = sk.send_str(
            s.as_str(), SNDMORE
        );
        if e2.is_err() {
            return e2
        }
    }
    sk.send_str("END", 0)
}
```

Per completezza riporto la funzione che costruisce il vettore booleano che contiene la sequenza di primalità vero/falso sui numeri dispari maggiori di 3. Essa implementa l'algoritmo del setaccio di Eulero, descritto in fondo alla pagina web https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes:

```
// the primes' signal with the Euler's sieve
fn primes_signal(n: u64) -> Vec<bool> {
    let n = n as usize;
    let lim = (n as f64).sqrt() as usize;
    let mut p_nums = vec![true; (n-1)/2];
    for i in 0..(lim-1)/2 {
        if p_nums[i] {
            let p = 2*i + 3; // prime value
            for k in i..(n/p - 1)/2 {
                p_nums[(p*(2*k + 3) - 3)/2] =
                false;
            }
        }
    }
    p_nums
}
```

12 SQLite Server

La connessione al database SQLite dal programma che svolge il ruolo di server è implementata utilizzando il progetto https://github.com/jgallagher/rusqlite, oggi alla versione 0.12.

Nella figura 9 è riportato il frammento di codice Rust più interessante perché risolve brillantemente il problema di convertire i tipi che restituisce il database in conseguenza all'esecuzione di una query, nel tipo testo con chiavi iniziali fisse a quattro caratteri, per l'invio all'interno del messaggio di risposta al client.

Molto elegantemente è sufficiente implementare il `trait FromSql` previsto in `rusqlite`. Interessante è anche dare uno sguardo al codice che esegue il prepared statement verso il database, racchiuso all'interno di una funzione che in un'unica fase esegue la query e invia il messaggio al client:

```
fn send_query_result(
   sk  : &Socket,
   conn: &Connection,
```

```
struct UniType {
    s: String,
}
impl FromSql for UniType {
    fn column_result(value: ValueRef)
    -> FromSqlResult<Self> {
        let ut = match value {
            ValueRef::Null       => UniType {s : "NULL:".to_string()},
            ValueRef::Integer(i) => UniType {s : format!("INTE:{}", i.to_string())}, // i64
            ValueRef::Real(f)    => UniType {s : format!("REAL:{}", f.to_string())}, // f64
            ValueRef::Text(s)    => UniType {s : format!("TEXT:{}", s.to_string())}, // &str
            ValueRef::Blob(v)    => {
                let s = v.iter().fold(String::new(),
                        |acc, byte| acc + format!("{:03}", byte).as_str()
                );
                UniType {s : format!("BLOB:{}", s)} // &[u8]
            },
        };
        Ok(ut)
    }
}
```

Figura 9: Porzione di codice Rust per l'implementazione del server connesso a un database SQLite. Si tratta del semplice meccanismo con cui si realizza la conversione personalizzata dai cinque tipi previsti dal database al tipo testo adatto alla trasmissione via Socket.

```
   args: &[String]) {
   let query = args[0].as_str();
   // create prepared statement
   let mut stmt = match conn.prepare(query) {
       Ok(stmt) => stmt,
       Err(e) => {
           send_err(sk, e.description());
           return
       }
   };
   let cols = {
       let c = stmt.column_names();
       let mut cols = vec![];
       for cn in c {
           cols.push(cn.to_string());
       }
       cols
   };
   let cols_num = cols.len();
   let mut param: Vec<&ToSql> = vec![];
   for i in 1..args.len() {
       param.push(&args[i] as &ToSql);
   }
   let mut rows =
   match stmt.query(&param[..]) {
       Ok(r) => r,
       Err(e) => {
           send_err(sk, e.description());
           return
       }
   };
   let mut tbl: Vec<String> = vec![];
   while let Some(result_row) = rows.next() {
       match result_row {
           Ok(row) => {
               for i in 0..cols_num as i32 {
                   match row.get_checked(i) {
                       Ok(val) => {
                           let ut: UniType =
                           val;
                           tbl.push(ut.s);
                       },
                       Err(e) => {
                           send_err(&sk,
                             e.description()
                           );
                           return;
                       }
                   }
               }
           },
           Err(e) => {
               send_err(&sk,
                   e.description()
               );
               return;
           }
       }
   }
    // send frames
   send_table(sk, cols, tbl)
}
```

13 Conclusioni

Il programma di composizione LuaTEX è in grado di produrre report molto sofisticati per qualità tipografica e capacità di elaborare i contenuti grazie al linguaggio di alto livello Lua incorporato e ai moduli interni avanzati dedicati ai font e ai nodi.

Oltre a questo, LuaTEX è capace di avvalersi di tecnologie come quelle illustrate in questo lavoro, che mettono in grado il motore di composizione di reperire le informazioni da diverse fonti e in particolare da database relazionali.

Il vantaggio che ne ricava l'utente è duplice: da un lato è possibile ottenere documenti di ottima qualità che si adattano facilmente all'evoluzione delle necessità, e dall'altro si ha a disposizione uno strumento in grado di integrarsi all'interno dell'organizzazione produttiva, che sempre più fa dell'affidabilità e della sicurezza dei dati uno dei motori del proprio sviluppo.

In questo lavoro ho ripercorso alcune possibili tecnologie che estendono LuaTEX con funzioni di acquisizione dell'informazione, esplorando in particolare quella della comunicazione tra codice in esecuzione in differenti processi basata sulla libreria ZeroMQ.

Rendere il codice capace di comunicare indipendentemente da quale sia il linguaggio di programmazione, il sistema operativo o il luogo delle rete dove si trovano i servizi software, apre un entusiasmante e ricco scenario a cui oggi grazie a LuaTEX si aggiungono anche TEX e le sue eccezionali doti tipografiche.

14 Ringraziamenti

Per il suo aiuto, e soprattutto per la sua amicizia, ringrazio Luigi Scarso, senza il quale — lo so, sembra una frase fatta ma vi assicuro che non lo è — questo articolo non sarebbe stato scritto.

Ringrazio infine la redazione tutta di ArsTEXnica per l'impegno profuso nell'accettazione e nella revisione dell'articolo e gli organizzatori del meeting annuale dell'associazione.

Riferimenti bibliografici

Blandy, J. e Orendorff, J. (2016). *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, 1ª edizione. URL https://www.nostarch.com/Rust.

Dworak, A., Charrue, P., Ehm, F., Sliwinski, W. e Sobczak, M. (2011). «Middleware Trends And Market Leaders 2011». *Conf. Proc.*, **C111010** (CERN-ATS-2011-196), p. FRBHMULT05. 4 p. URL http://cds.cern.ch/record/1391410.

Giacomelli, R. (2016). «LuaTEX + pdfliteral». *ArsTEXnica*, (22), pp. 27–43. URL http://www.guitex.org/home/numero-22.

— (2017). «A database experiment with LuajitLATEX». *ArsTEXnica*, (23), pp. 12–34. URL http://www.guitex.org/home/numero-23.

Giacomelli, R. e Pignalberi, G. (2015). «Generare documenti LATEX con diversi linguaggi di programmazione». *ArsTEXnica*, (20), pp. 40–73. URL http://www.guitex.org/home/numero-20.

Hagen, H. e Scarso, L. (2017). «Foreign function interface in luaTEX». *ArsTEXnica*, (23), pp. 70–76. URL http://www.guitex.org/home/numero-23.

Hintjens, P. (2013a). *Code Connected Volume 1*. iMatix Corporation, 1ª edizione. URL http://hintjens.wdfiles.com/local--files/main:files/cc1pe.pdf.

— (2013b). *ZeroMQ Messaging for Many Applications*. O'Reilly Media, 1ª edizione.

Ierusalimschy, R. (2016). *Programming in Lua*. Lua.org, 3ª edizione.

Klabnik, S. e Nichols, C. (2017). *The Rust Programming Language*. No Starch Press Inc., 1ª edizione. URL https://www.nostarch.com/Rust.

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley Professional, 1ª edizione.

Scarso, L. (2013a). «Experiments with multitasking and multithreading in LUATEX». *ArsTEXnica*, (16), pp. 68–83. URL http://www.guitex.org/home/numero-16.

— (2013b). «LuajitTEX». *ArsTEXnica*, (15), pp. 59–69. URL http://www.guitex.org/home/numero-15.

The LuaTEX development team (2017). *LuaTEX Reference Manual*, 1.0.4 edizione.

The Rust Community (2017). *The Rust Programming Language*, 2ª edizione.

▷ Roberto Giacomelli
Carrara
giaconet dot mailbox at gmail dot com

Un DTD SGML per la generazione di codice NGSpice e CircuiTikZ

Renato Battistin

Sommario

La simulazione di un circuito elettronico e la sua rappresentazione grafica sono solitamente due operazioni distinte. Un DTD SGML che allo stesso tempo definisce i componenti dei circuiti, le loro connessioni, i comandi di simulazione e la rappresentazione grafica del circuito, consente di ottenere da un unico file SGML i codici NGSpice e CircuiTikZ.

Abstract

The simulation of an electronic circuit and its graphic representation are usually two distinct operations. An SGML DTD that simultaneously defines the circuit components, their connections, simulation commands, and graphical representation of the circuit, allows of to obtain the NGSpice and CircuiTikZ codes from a single SGML file.

1 Introduzione

La descrizione di un circuito elettronico e la sua simulazione è un problema affrontato dagli inizi dell'era informatica. I software sono innumerevoli e vanno dai primi simulatori per *mainframe* degli anni Settanta, come ad esempio *SPICE* di Laurence Nagel (Nagel), scritto ancora in Fortran, fino a quelli attuali, tra i quali possiamo annoverare

- **gSpiceUI**, motori di simulazione: ngSpice, GNU-Cap;
- **XCircuit**, schemi circuitali;
- **gEDA**, schemi circuitali in funzione PCB;
- **PartSim**, motore di simulazione e schema circuitale via interfaccia web;

solo per citarne alcuni, ma l'elenco esaustivo sarebbe lungo.

Questi simulatori hanno come scopo principale l'analisi del circuito che avviene dopo una descrizione, o meglio un'elencazione, dei componenti circuitali e delle loro connessioni elettriche. La rappresentazione grafica è un aspetto importante ma non strettamente necessario; come tale a volte è parte integrante del software di simulazione, a volte è derogata a programmi satellite. A seconda dei casi, l'utente può avere o meno la possibilità, più o meno ampia, di gestire la rappresentazione grafica, sia per i simboli circuitali che per gli altri elementi grafici come le etichette e le informazioni tecniche e gestionali del circuito elettronico.

Se guardiamo agli ambiti di impiego di questo tipo di software di simulazione, allora oltre a quello professionale è molto rilevante quello amatoriale, ma anche quello scolastico dove però sono richieste allo studente oltre alla conoscenza della materia elettronica, anche abilità di programmazione ed un minimo di capacità organizzativa per gestire la rappresentazione grafica circuitale.

Sia la programmazione della simulazione che la rappresentazione grafica hanno le loro specifiche esigenze. Ad esempio la descrizione circuitale al fine della simulazione generalmente richiede solamente una descrizione topologica dei componenti circuitali e non una loro descrizione topografica; in sostanza quello che importa sapere è a quali componenti elettronici è connesso un certo componente circuitale; più raramente se quei componenti sono fisicamente vicini o se il percorso di connessione è più o meno lungo[1]. Un simulatore circuitale richiede che vengano impartiti dei comandi appositi per eseguire una simulazione del comportamento fisico del circuito. Questi comandi dipendono dal tipo di simulazione richiesta: analisi dei piccoli segnali, transienti, analisi di Fourier, ecc.

Per contro la descrizione topografica del circuito elettronico richiede la rappresentazione delle connessioni fisiche oltre che quella dei singoli componenti circuitali; inoltre, affiché questa rappresentazione sia efficace, generalmente sono richiesti un certo numero di altri elementi grafici e tra questi senz'altro le etichette. Infine una serie di altre informazioni relative al circuito, ad esempio relative al suo impiego, alla sua progettazione, alla sua implementazione pratica, ecc., possono essere necessarie per completare la rappresentazione grafica del circuito elettronico.

In generale quindi i software di simulazione disponibili considerano la rappresentazione grafica un elemento, se non secondario, al più funzionale alla simulazione e seppure la resa grafica possa essere soddisfacente, ciò può non essere sufficiente in alcuni ambiti come ad esempio quello accademico.

Tra i pacchetti per il disegno tipografico dei circuiti elettronici ci sono:

1. Ovviamente dal punto di vista fisico la lunghezza di una connessione elettrica è importante in quanto può comportare ad esempio una capacità elettrica distribuita non trascurabile.

- la libreria `pst-circ` di PSTricks (Voss)
- il pacchetto MetaPost `mpcirc` di Tomasz J. Cholewo
- il pacchetto CircuiTikZ creato da Massimo A. Redaelli
- le macro M4 di Dwight Aplevich (Aplevich)

Questo articolo prende in considerazione solo CircuiTikZ per mostrare come un linguaggio di marcatura possa essere utilizzato per generare sia codice NGSpice, che codice integrabile direttamente in documenti *TEX. A tal scopo viene presentato un DTD SGML che consente la stesura in linguaggio marcato di codice che contestualmente ai comandi di simulazione permette di associare comandi per la generazione di codice CircuiTikZ direttamente utilizzabile in ambiente LATEX. Nel contempo la parte di codice SGML relativa alla simulazione può essere mappata in codice NGSpice, un dialetto *open source* molto diffuso di *Spice*, un linguaggio storico di eccellenza per la simulazione di circuiti elettronici.

Le sezioni seguenti mostrano le caratteristiche salienti di CircuiTikZ e di NGSpice, la loro organizzazione e resa nel DTD e le mappature adottate per traslitterare il codice SGML nei due linguaggi NGSpice e CircuiTikZ.

2 NGSpice

NGSpice, è un simulatore di circuiti elettronici conforme alle specifiche del linguaggio Spice. Il termine 'simulazione' ha un'accezione piuttosto ampia in quanto ammette varianti sulla base del problema circuitale studiato. Si va dalla semplice analisi delle condizioni stazionarie, ossia quello che in gergo elettronico viene detto 'punto di lavoro' del circuito; all'esame dei segnali transitori; all'analisi cosiddetta dei *piccoli segnali*;[2]; fino all'analisi termica, di Fourier, di MonteCarlo, del rumore e della sensibilità (*sensitivity*). Poiché software come NGSpice hanno la propria origine negli anni 70 del secolo scorso, non è inusuale trovarsi davanti a delle scarse capacità di rappresentazione tipografica dei risultati, in opposizione alle capacità di simulazione. Infatti spesso l'interazione con il software avviene mediante una *shell* testuale ed anche le moderne varianti di Spice normalmente si adeguano a questa impostazione. Se un'uscita grafica esiste, allora questa è sovente, come nel caso di NGSpice, solo una finestra *pop-up* che rappresenta l'andamento dei segnali ai nodi prescelti mentre non vi è alcuna rappresentazione grafica circuitale. Questo non significa che altre varianti di Spice non ce l'abbiano, anzi le più moderne varianti permettono di esaminare i segnali ai nodi a volte semplicemente usando il puntatore sulla rappresentazione grafica circuitale. Tuttavia il lavoro di costruzione circuitale in queste varianti di Spice moderne è comunque spesso delegato all'utente per via grafica, senza un corrispettivo strumento testuale alternativo, certamente molto più apprezzato, ad esempio, in ambito professionale.

Un codice NGSpice consiste in una serie di righe[3] che si distinguono fondamentalmente in due tipi: descrittore circuitale e direttiva di controllo. Le righe descrittrici generalmente sono costituite da un identificatore del componente circuitale, ad esempio un resistore, un transistor, un alimentatore; da una sequenza di nodi in base al numero di poli associati al componente elettronico, ad esempio due per un condensatore, tre per un transistor, ecc.; da un'eventuale stringa identificatrice del modello fisico del componente circuitale, come ad esempio nel caso di un diodo o, di nuovo, di un transistor; da un'eventuale successione di coppie di assegnazione parametro-valore, come ad esempio la temperatura di un resistore.

Le direttive di controllo sono contraddistinte da un punto, '.', come carattere iniziale; includono la prima riga di comando che obbligatoriamente deve essere quella del titolo, '.title'; l'ultima riga, '.end', che obbligatoriamente deve essere sempre riportata. Altre direttive di controllo hanno la funzione di indicare il tipo di modello, '.model', di un componente circuitale ove richiesto; l'esecuzione di un tipo di simulazione, ad esempio un transitorio, '.tran'; un tipo di analisi circuitale, ad esempio del rumore, '.noise'; la richiesta di un'uscita dati, ad esempio un grafico, '.plot'.

3 CircuiTikZ

Il pacchetto CircuiTikZ è utilizzabile per disegnare circuiti elettronici e fu scritto originariamente da Massimo Redaelli nel 2007 come strumento per la creazione di esercizi ed esami in ambito accademico; è ora mantenuto dall'autore e da altri due sviluppatori, Stefan Lindner e Stefan Erhardt (Redaelli *et al.*, 2017). Il pacchetto CircuiTikZ è compatibile con LATEX e ConTEXt. Le sue radici sono ben evidenti nella sua sintassi che sostanzialmente è quella di Ti*k*Z. Non è però del tutto compatibile con la libreria circuitale di Ti*k*Z (Tantau, 2015); l'uso contemporaneo dei due pacchetti richiede che venga specificata l'opzione `compatibility` di CircuiTikZ.

2. Un piccolo segnale è una variazione del segnale sufficientemente piccola da consentire un modello lineare per tutti i componenti del circuito (l'autore ringrazia Claudio Beccari per questa chiara e efficace definizione). Un piccolo segnale può essere aggiunto ad un segnale finito (ad esempio quello corrispondente al punto di lavoro), eventualmente in forma di segnale periodico e in seguito viene simulato il comportamento del circuito alla sua variazione, solitamente analizzando la risposta in frequenza ed in ampiezza.

3. Detta in gergo *netlist*.

Gran parte del circuito elettronico viene disegnato con CircuiTikZ percorrendo la rete circuitale per collocare lungo il percorso i nodi e tra di essi le connessioni mediante delle linee, oppure dei componenti elettronici[4]. Attributi dei nodi e delle connessioni completano le informazioni quali le etichette, le loro posizioni, la forma delle connessioni, aspetti grafici come colore ed evidenziazione, ecc.

4 Sintassi

Le sintassi di NGSpice e di CircuiTikZ sono piuttosto diverse tra loro. La prima è incentrata sulla descrizione del componente elettronico richiedendo la dichiarazione esplicita di tutti i suoi nodi oltre che della sua natura fisica:

```
componente1 nodo1 nodo2 ...
componente2 nodo2 nodo3 ...
...
```

La seconda, essendo incentrata sulla descrizione del percorso, può essere ridotta alla forma minimale

```
nodo1 componente1 nodo2 componente2 ...
```

dove in quest'ultimo caso il vocabolo `componente` include anche componenti grafici che non hanno un corrispettivo componente elettronico in NGSpice come, ad esempio, la semplice linea di collegamento tra due nodi e il nodo di terra.

I vincoli maggiori per impostare una sorta di sintassi comune tra i due linguaggi sono presentati da NGSpice, che richiede obbligatoriamente che ciascun componente sia descritto su una, ed una sola, riga di codice. Per contro CircuiTikZ, ereditando la versatilità della sintassi di Ti*k*Z, può accettare del codice sia distribuito su più righe, sia ridondante, ragion per cui possiamo adattare la sintassi di CircuiTikZ alla forma seguente[5]:

```
nodo1 componente nodo2 nodo2 \
componente2 nodo3 nodo3 ...
```

Questa sintassi verrà quindi sfruttata nel DTD per definire i componenti circuitali in modo tale che il medesimo codice SGML sia traslitterabile in entrambe le sintassi NGSpice e CircuiTikZ.

5 Il DTD

5.1 Scopo

Il DTD è rivolto principalmente alla simulazione. Questo significa che la rappresentazione grafica è opzionale per tutti i componenti circuitali. Questa scelta è dettata dalla considerazione che la simulazione è un processo generalmente volto al miglioramento delle prestazioni circuitali e, come tale, modifica progressivamente il numero ed il tipo di componenti circuitali impiegati, ed anche il tipo ed il numero di simulazioni richieste; solo al termine di questo percorso sorge eventualmente l'esigenza di ottenere una rappresentazione grafica. Questa forma del DTD può penalizzare leggermente il suo impiego in ambito scolastico nel caso in cui la rappresentazione grafica del circuito sia l'unica esigenza, in questo caso può essere sufficiente impiegare direttamente CircuiTikZ.

5.2 Traslitterazione

La traslitterazione del codice SGML in codice NGSpice oppure CircuiTikZ viene eseguita mediante dei file qui chiamati di *mappatura*. Questi file hanno una sintassi conforme ai file di sostituzione dell'*Amsterdam SGML Parser*[6] e sono forniti come argomento al software `sgmlsaps` per traslitterare il suo input standard generato a sua volta dal *parser* `nsgmls`, ad esempio[7]:

```
cat doc.sgml | nsgmls -c catalog | \
sgmlsasp map > doc.tex
```

dove `doc.sgml` è il file SGML da traslitterare, `map` è il file di mappatura, `doc.tex` è il file LaTeX generato; il file `catalog` come dice il nome è un cosiddetto file di catalogo che generalmente viene fornito come opzione a `nsgmls` per includere nella traslitterazione alcuni file SGML standard come ad esempio un file di entità SGML predefinite.

5.3 Organizzazione

Il DTD, essendo orientato principalmente alla simulazione, presenta una struttura essenzialmente consona, con un titolo, una descrizione topologica del circuito e quindi un'analisi ed una stampa dei risultati:

```
<!ELEMENT ngspice - o (title, circuit,
    control?, analysis?,
print-results?, end-line)
    +(includefile|comment|tikz)>
```

In pratica attualmente solo la parte di codice SGML che descrive il circuito, delimitata dall'elemento[8] `<circuit>`, può produrre codice CircuiTikZ mentre le altre parti del codice sono essenzialmente funzionali alla simulazione e quindi vengono mappate esclusivamente in codice NGSpice. Tuttavia il DTD prevede la possibilità di includere direttamente del codice LaTeX e Ti*k*Z a qualsiasi livello del documento SGML:

4. Con un meccanismo che ricorda la *turtle graphics* del linguaggio di programmazione Logo.

5. Il carattere '\' ha il classico significato di indicare che il codice seguente continua in effetti sulla medesima riga.

6. https://web.cs.wpi.edu/~kal/elecdoc/sgml/ASPhead.html.

7. Per un esempio applicativo si confrontino i sempre utili *Appunti di Informatica Libera* di Daniele Giacomini, http://wwwcdf.pd.infn.it/AppuntiLinux/a2558.htm.

8. Con un minimo di abuso di linguaggio identifichiamo l'elemento con il suo `tag` di apertura.

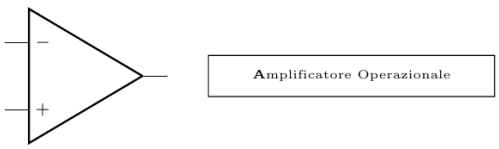


Figura 1: Il classico simbolo di un amplificatore operazionale assieme ad un'etichetta ottenuta mediante inclusione di codice Ti*k*Z.

```
<!NOTATION TikZ system "">
<!ELEMENT tikz - - CDATA >
<!ATTLIST tikz format NOTATION (Tikz)
    "Tikz">
```

In tal caso il codice viene gestito come una notazione[9] e quindi al di fuori dell'ambito di validazione SGML. Ad esempio il codice SGML

```
<!doctype ngspice system 'ngspice.dtd'>
<ngspice>
  <title>Testo accanto a componente
      circuitale
   gestito via notazione 'TikZ'</title>
  <circuit>
    <opamp name="myopamp" node="(2,2)">
    <right-angle-circuit>
      <xy-minus>(myopamp.out)</xy-minus>
      <!-- one of (HVLINE VHLINE
          STRAIGHT) -->
      <straight>
      <xy-plus>(3,2)</xy-plus>
    </right-angle-circuit>
    <tikz>(3.5,1.75) rectangle +(2.5,
        0.5) +(1.25, 0.25)
 node {\tiny \textbf Amplificatore
     Operazionale}</tikz>
  </circuit>
  <end-line>
</ngspice>
```

viene mappato al sequente codice CircuiTikZ

```
%
% Testo accanto a componente circuitale
    gestito via notazione 'TikZ'
%
\begin{circuitikz}[scale=1.2]\draw[gray]
(2,2) node [op amp] (myopamp) {}
(myopamp.out) -- (3,2)(3.5,1.75)
    rectangle +(2.5, 0.5) +(1.25, 0.25)
 node {\tiny \textbf Amplificatore
     Operazionale}
;\end{circuitikz}
```

che può essere incluso direttamente nel proprio documento LaTeX e compilato (Fig. 1).

5.4 Componenti circuitali ed elementi grafici

L'elemento `<circuit>`:

```
<!ELEMENT circuit - -
    (%ngspice-elements; |
    %circuitikz-elements; |
    %tikzgraphics; | subcircuit |
    model)+ >
```

attualmente permette di inserire una serie di componenti circuitali NGSpice:

```
<!ENTITY % ngspice-elements
    'xspace-code-model |
    behavioral-source | capacitor |
    diode |
    linear-voltage-controlled-current |
    linear-voltage-controlled-voltage |
    linear-current-controlled-current |
    linear-current-controlled-voltage |
    jfet | coupled-inductors | inductor
    | mosfet | num-dev-gss |
    lossy-trams-line | bjt | resistor |
    switch | single-lossy-trasm-line |
    mesfet | voltage | current'>
```

intercalati eventualmente al loro modello e ad eventuali sotto-circuiti. Inoltre consente l'inserimento di alcuni elementi "grafici" di CircuiTikZ

```
<!ENTITY % circuitikz-elements 'opamp |
    block-diagram-component |
    short-circuit | right-angle-circuit
    | open-circuit'>
```

che ovviamente non essendo componenti circuitali non vengono mappati a codice NGSpice.
L'elemento `<block-diagram-component>` rappresenta uno qualsiasi dei diagrammi a blocchi previsti da CircuiTikZ, escluso l'amplificatore operazionale al quale è riservato l'elemento apposito `<opamp>`:

```
<!ELEMENT block-diagram-component - o
    EMPTY >
<!ATTLIST block-diagram-component
        name cdata #required
        node cdata #required
        symbol cdata #required
        label cdata #implied
        label-position cdata #implied
        more-options cdata #implied >
```

dove la distinzione viene fatta mediante l'attributo `symbol` all'atto della mappatura:

```
<block-diagram-component> + "[NODE]␣node␣
    \[[SYMBOL]\]␣([NAME])␣
    {[MORE-OPTIONS]}"
</block-diagram-component>
```

5.5 Sezioni marcate nel DTD

La separazione mediante l'impiego di entità parametriche tra componenti circuitali da una parte e modello e sotto-circuito dall'altra è utile quando si desidera utilizzare il DTD solo per rappresentare schemi circuitali e non per la simulazione. Implementando infatti nel DTD una coppia di *sezioni marcate*,[10]

```
<!-- SGML variant for mapping to
    CircuiTikz -->
<![ %ngspice; [ <!ELEMENT ngspice - o
    (title, circuit, control?,
    analysis?, print-results?, end-line)
    +(includefile & comment & latex &
    tikz)> ]]>
```

9. La parola chiave `NOTATION` in sostanza afferma che il testo delimitato dell'elemento <tikz> non viene sottoposto alle usuali regole di *parsing* ma viene passato tale e quale, all'atto della mappatura del codice SGML, al fantomatico programma 'Ti*k*Z' che se ne prenderà auspicabilmente carico.

10. Le sezioni marcate hanno una funzione simile ad un condizionale e consentono di includere o escludere alcune sezioni del DTD per cambiarne la destinazione d'uso.

```
<!-- SGML variant for mapping to NGSpice
    -->
<![ %circuitikz; [ <!ELEMENT ngspice - o
    (title, circuit) +(latex & tikz)> ]]>
```

è possibile attivare la sezione del DTD relativa alla sola descrizione circuitale, essenzialmente l'unica ad essere mappata a codice CircuiTikZ come detto in precedenza:

```
<!doctype ngspice system 'ngspice.dtd' [
<!ENTITY % ngspice "IGNORE">
<!ENTITY % circuitikz "INCLUDE">
]>
```

5.6 Rappresentazione circuitale in CircuiTikZ

CircuiTikZ può rappresentare un intero circuito elettronico mediante un unico comando `draw` ed in effetti la mappatura dell'elemento `<circuit>` segue questa impostazione:

```
<circuit> +
    "\\begin\{circuitikz\}\[[OPTIONS]\]
\\draw"
</circuit> + ";\\end{circuitikz}" +
```

Tuttavia in alcuni casi, per esempio se si vuole evidenziare una parte del circuito con un colore oppure una grafica differente, può essere utile inserire nell'ambiente `circuitikz` ulteriori comandi `draw`. Il DTD attualmente supporta i comandi `draw` e `filldraw`, ciascuno con la possibilità di introdurre tutte le opzioni desiderate:

```
<!ENTITY % tikzgraphics 'draw | filldraw
    | node-labels' >
<!ELEMENT (%tikzgraphics;) - o EMPTY >
<!ATTLIST (%tikzgraphics;)
        options cdata #implied >
```

I due elementi, una volta mappati, chiudono il comando precedente con un carattere ';' ed iniziano quello nuovo, nuove opzioni incluse:

```
<draw> ";␣\n\\draw\[[OPTIONS]\]␣"
</draw>
<filldraw> ";␣\n\\filldraw\[[OPTIONS]\]␣"
</filldraw>
```

Ad esempio consideriamo il codice SGML di un circuito per il raddoppio della tensione di alimentazione (Roberts e Sedra, 1997, p. 97) mostrato in Fig. 3:

```
<!doctype ngspice system 'ngspice.dtd'>
<ngspice>
  <title>Voltage Doubler - (Roberts
      &amp; Sedra, Spice, 2nd Ed.,
      Oxford)</title>
  <circuit>
    <voltage name="in" node-plus="1"
        node-minus="0" xy-plus="(0,2)"
        xy-minus="(0,0)" symbol="sV"
        label="10V" to-options="-*">
      <sinusoidal vo="0" va="10"
          freq="1kHz">
    </voltage>
    <short-circuit xy-minus="(0,0)"
        xy-plus="(2,0)">
    <capacitor name="1" node-plus="2"
        node-minus="1" symbol="eC"
        value="1u" xy-minus="(0,2)"
        xy-plus="(2,2)" label="$C_1␣
        (1\micro\farad)$"
        label-position="_"
        to-options="*-">
    </capacitor>
    <diode name="1" node-plus="0"
        node-minus="2" model="DIN4148"
        label="$D_1$" xy-plus="(2,0)"
        xy-minus="(2,2)">
    </diode>
    <ground xy-node="(2,0)">
    <draw options="dashed">
    <diode name="2" node-plus="2"
        node-minus="3" model="DIN4148"
        label="$D_2$" label-position="^"
        xy-plus="(2,2)" xy-minus="(4,2)">
    </diode>
    <capacitor name="2" node-plus="3"
        node-minus="0" symbol="eC"
        value="1u" xy-minus="(4,0)"
        xy-plus="(4,2)" label="$C_2␣
        (1\micro\farad)$"
        label-position="_">
    </capacitor>
    <short-circuit xy-plus="(4,0)"
        xy-minus="(2,0)">
    <draw>
    <short-circuit xy-plus="(5,0)"
        xy-minus="(4,0)">
    <short-circuit xy-plus="(5,2)"
        xy-minus="(4,2)">
    <open-circuit xy-plus="(5,2)"
        xy-minus="(5,0)"
        to-options="o-o">
    <node-labels
        options="(0,2)node[anchor=south]␣
        {1}␣(2,2)node[anchor=south]{2}␣
        (5,2)node[anchor=south]␣{3}">
    <model mname="DIN4148" type="D">
      <parameter pname="Is" pval="0.1pA">
      <parameter pname="Rs" pval="16">
      <parameter pname="CJO" pval="2p">
      <parameter pname="Tt" pval="12n">
      <parameter pname="Bv" pval="100">
      <parameter pname="Ibv" pval="0.1p">
    </model>
  </circuit>
  <analysis>
    <tran-analysis time-step="100u"
        time-stop="10m" time-start="0m"
        time-step-maximum="100u">
  </analysis>
  <end-line>
</ngspice>
```

La sua rappresentazione grafica di questo codice, una volta mappato, è mostrata in Fig. 2, dove la parte di circuito tratteggiata è stata introdotta da un comando `draw`; il codice CircuiTikZ è riportato qui di seguito:

```
%
% Voltage Doubler - (Roberts \& Sedra,
    Spice, 2nd Ed., Oxford)
%
\begin{circuitikz}[scale=1.2,american]
\draw(0,0) to[sV, l=10V, -* ] (0,2)
(0,0) to[short, l=,  ] (2,0)
(2,2) to[eC, l_=$C_1 (1\micro\farad)$,
    *- ] (0,2)
(2,0) to[Do, l=$D_1$,  ] (2,2)
(2,0) node[ground] {};
```

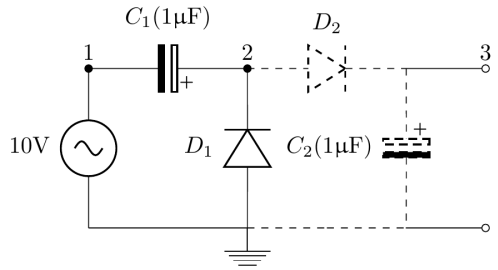


FIGURA 2: Immagine del raddoppiatore di tensione.

```
\draw[dashed]
(2,2) to[Do, l^=$D_2$,  ] (4,2)
(4,2) to[eC, l_=$C_2 (1\micro\farad)$,
    ] (4,0)
(2,0) to[short, l=,  ] (4,0);
\draw[]
(4,0) to[short, l=,  ] (5,0)
(4,2) to[short, l=,  ] (5,2)
(5,0) to[open, l=, o-o ] (5,2)
    {(0,2)node[anchor=south] {1}
    (2,2)node[anchor=south]{2}
    (5,2)node[anchor=south] {3}}
;\end{circuitikz}
```

Si noti che il condensatore polarizzato[11], così come il diodo, sono componenti bipolari che, a differenza di altri quali il resistore, ammettono un polo positivo ed uno negativo. Il disegno del componente bipolare "inizia" dal suo polo positivo e quindi la mappatura assume l'aspetto seguente:

```
<diode> + "[XY-PLUS] to\[[SYMBOL], 
    l[LABEL-POSITION]=[LABEL], 
    [TO-OPTIONS] \] [XY-MINUS]"
</diode>
...
<capacitor> + "[XY-PLUS] to\[[SYMBOL], 
    l[LABEL-POSITION]=[LABEL], 
    [TO-OPTIONS] \] [XY-MINUS]"
</capacitor>
```

dove apparentemente sembra che due componenti elettronici fisicamente distinti vengano mappati in maniera identica. Questo ovviamente non è vero, in quanto il valore dell'attributo `SYMBOL` viene gestito in forma predefinita entro il DTD al momento della definizione degli attributi dell'elemento rappresentante il componente elettronico, ad esempio:

```
<!ATTLIST diode
        name cdata #required
        node-plus cdata #required
        node-minus cdata #required
        model cdata #required
        symbol cdata "Do"
        label cdata #implied
        label-position cdata #implied
        xy-plus cdata #implied
        xy-minus cdata #implied
        to-options cdata #implied >
```

Le etichette dei tre nodi principali del circuito sono inserite come codice CircuiTikZ prima della chiusura dell'ambiente `circuitikz`; il codice viene generato mediante mappatura dell'elemento `<node-labels>`:

```
<node-labels> "{[OPTIONS]}"
</node-labels>
```

Gli elementi SGML per così dire 'comuni' sia a CircuiTikZ che a NGSpice sono stati definiti nel DTD in modo tale che i loro attributi siano nell'ordine prima tutti quelli caratteristici di NGSpice, come gli identificatori alfanumerici dei poli di connessione ed il valore della grandezza fisica caratterizzante il componente elettronico, ad esempio 50kΩ; poi tutti gli attributi caratteristici di CircuiTikZ e, più in generale, di Ti*k*Z come le coppie ordinate cartesiane dei poli circuitali, le etichette, la loro posizione, il tipo di connessione, ecc.:

```
<capacitor name="2" node-plus="3"
    node-minus="0" value="1u"
    xy-plus="(4,2)" xy-minus="(4,0)"
    label="$C_2$">
```

Come accennato in precedenza, essendo il DTD principalmente orientato a NGSpice, solo gli attributi relativi a quest'ultimo sono obbligatori:

```
<!ELEMENT capacitor - o  (par-model?,
    par-ac?, par-m?, par-scale?,
    par-temp?, par-dtemp?, par-tc1?,
    par-tc2?, ic?)>
<!ATTLIST capacitor
        name cdata #required
        node-plus cdata #required
        node-minus cdata #required
        symbol cdata "C" -- circuital
            symbol --
        value cdata #required --
            capacitance value --
        xy-plus cdata #implied
        xy-minus cdata #implied
        label cdata #implied
        label-position cdata #implied
        to-options cdata #implied >
```

Un medesimo sorgente SGML può essere mappato sia a codice CircuiTikZ, che a codice NGSpice; questo, una volta eseguito, permette di ottenere, ad esempio, l'andamento transitorio iniziale della tensione al nodo di uscita '3' confermando il raddoppio della tensione di alimentazione (Fig. 3).

5.7 Unità di misura

Il pacchetto CircuiTikZ integra il pacchetto `siunitx`[12]; tuttavia è possibile impiegare anche il pacchetto `SIunits` ad esempio direttamente entro l'attributo `label` dell'elemento `capacitor` come si può vedere nel codice SGML del raddoppiatore di tensione.

5.8 Elementi multipolari

Il codice più sotto riportato rappresenta un circuito elettronico mostrato in Fig. 4 comprendente anche un elemento multipolare, un transistor:

11. Fisicamente può essere un condensatore elettrolitico.

12. Anche se nelle ultime versioni è disabilitato nelle impostazioni predefinite.

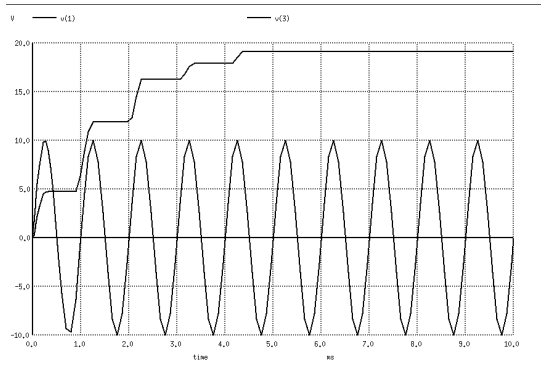


Figura 3: Transitorio iniziale della tensione di uscita al nodo '3' del circuito raddoppiatore di tensione così come mostrato da NGSpice assieme all'andamento della tensione d'ingresso al nodo '1'.

```
<!doctype ngspice system 'ngspice.dtd'>
<ngspice>
  <title>Voltage Gain of a Transistor
      Amplifier</title>
  <circuit>
    <ground xy-node="(0,-0.5)">
    <comment>DC supply</comment>
    <voltage name="BB" node-plus="5"
        node-minus="0" xy-plus="(0,0.5)"
        xy-minus="(0,-0.5)"
        symbol="battery1" label="$V_{BB}␣
        (3\volt)$" label-position="_">
      <dc-tran value="3V">
    </voltage>
    <comment>Small signals
        source</comment>
    <voltage name="i" node-plus="4"
        node-minus="5" xy-plus="(0,2)"
        xy-minus="(0,0.5)" symbol="sV"
        label="$v_i␣(1\milli\volt)$"
        to-options="*-*">
      <ac acmag="1mV">
    </voltage>
     <short-circuit xy-plus="(0,2.5)"
         xy-minus="(0,2)">
    <resistor name="B" node-plus="3"
        node-minus="4" value="100K"
        xy-plus="(2,2.5)"
        xy-minus="(0,2.5)"
        label="$R_{BB}␣(100\kilo\ohm)$">
    </resistor>
    <bjt name="1" collector-node="2"
        base-node="3" emitter-node="0"
        model="npn_transistor"
        xy-node="(2.5,2.5)"
        id-node="bjt1" >
    </bjt>
    <short-circuit xy-plus="(bjt1.base)"
        xy-minus="(2,2.5)"
        to-options="*-">
    <short-circuit
        xy-plus="(bjt1.emitter)"
        xy-minus="(2.5,2)"
        to-options="-*">
    <model mname="npn_transistor"
        type="npn">
      <parameter pname="Is"
          pval="1.8104e-15">
      <parameter pname="Bf" pval="100">
    </model>
    <ground xy-node="(2.5,2)">
    <resistor name="C" node-plus="1"
        node-minus="2" value="3K"
        xy-plus="(2.5,4.5)"
        xy-minus="(bjt1.collector)"
        label="$R_C␣(3\kilo\ohm)$"
        to-options="*-">
    </resistor>
    <voltage name="CC" node-plus="10"
        node-minus="1" xy-plus="(2.5,6)"
        xy-minus="(2.5,4.5)"
        label="$V_{CC}␣(10\volt)$"
        to-options="*-">
      <dc-tran value="10V">
    </voltage>
    <short-circuit xy-plus="(2.5,6)"
        xy-minus="(3.5,6)"
        to-options="-*">
    <ground xy-node="(3.5,6)">
    <node-label node="(2.5,6)" label="1"
        anchor="east">
    <node-label node="(bjt1.collector)"
        label="2" anchor="east">
    <node-label node="(bjt1.collector)"
        label="$v_o$" anchor="west">
    <node-label node="(2.5,2)" label="0"
        anchor="east">
    <node-label node="(2,2.5)" label="3"
        anchor="south">
    <node-label node="(0,2)" label="4"
        anchor="east">
    <node-label node="(0,0.5)" label="5"
        anchor="east">
    <comment>transistor collector source
        polarity symbols</comment>
    <node-label node="(2.5,5.25)"
        label="-" anchor="east">
    <node-label node="(2.5,5.25)"
        label="+" anchor="east">
    <comment>transistor base source
        polarity symbols</comment>
    <node-label node="(0,0)" label="+"
        anchor="east">
    <node-label node="(0,0)" label="-"
        anchor="east">
  </circuit>
  <end-line>
</ngspice>
```

Un transistor è rappresentato in CircuiTikZ mediante un singolo nodo dotato però di tre oppure quattro connettori,[13] quindi per identificarne la posizione è sufficiente una singola coppia di coordinate cartesiane:

```
<bjt name="1" collector-node="2"
    base-node="3" emitter-node="0"
    model="npn_transistor"
    xy-node="(2.5,2.5)" id-node="bjt1" >
```

I connettori invece sono individuati tramite dei suffissi che richiamano la loro funzione; ad esempio, se il nodo rappresentante un transistor è identificato da CircuiTikZ con la stringa `bjt1`, allora il riferimento alla sua base è dato dalla stringa `bjt1.base`.

Sempre nel codice SGML precedente si osservi l'impiego dell'elemento `<comment>` di ovvia interpretazione e come viene mappato nel codice CircuiTikZ riportato più sotto; si tratta comuque di una singola linea di commento ed in tale forma

13. Un transistor ammette a volte un quarto connettore che viene utilizzato per collegare a terra il cosiddetto *case* ossia l'involucro.

viene mappata anche nel codice NGSpice. Un'altra osservazione riguarda i segni di polarità delle sorgenti di tensione costante che sono stati apposti *ad-hoc* anche se a livello normativo può essere tranquillamente usato il corrispodente simbolo senza i segni. Si noti inoltre al termine del codice SGML la sequenza di istanze dell'elemento `<node-label>` a completamento delle informazioni e dell'aspetto del circuito elettronico. Ecco il codice CircuiTikZ:

```
%
% Voltage Gain of a Transistor Amplifier
%
\begin{circuitikz}[scale=1.2,american]
\draw(0,-0.5) node[ground] {}
% DC supply
(0,-0.5) to[battery1, l=$V_{BB}
    (3\volt)$,  ] (0,0.5)
% Small signals source
(0,0.5) to[sV, l=$v_i (1\milli\volt)$,
    *-* ] (0,2)
(0,2) to[short, l=,  ] (0,2.5)
(0,2.5) to[R, l=$R_{BB} (100\kilo\ohm)$,
     ] (2,2.5)
(2.5,2.5) node[npn] (bjt1) {}
(2,2.5) to[short, l=, *- ] (bjt1.base)
(2.5,2) to[short, l=, -* ] (bjt1.emitter)
(2.5,2) node[ground] {}
(bjt1.collector) to[R, l=$R_C
    (3\kilo\ohm)$, *- ] (2.5,4.5)
% (2.5,4.5) to[V, l=$V_{CC} (+10\volt)$,
    -o ] (2.5,6)
(2.5,6) to[battery1, l_=$V_{CC}
    (10\volt)$, *- ] (2.5,4.5)
(2.5,6) to[short, l=,] (3.5,6)
(3.5,6) node[ground] {}
(2.5,6) node[anchor=east] {1}
(bjt1.collector) node[anchor=east] {2}
(bjt1.collector) node[anchor=west]
    {$v_o$}
(2.5,2) node[anchor=east] {0}
(2,2.5) node[anchor=south] {3}
(0,2) node[anchor=east] {4}
(0,0.5) node[anchor=east] {5}
% transistor collector source polarity
    symbols
(2.5,5.25) node[anchor=south east] {-}
(2.5,5.25) node[anchor=north east] {+}
% transistor base source polarity symbols
(0,0) node[anchor=south east] {+}
(0,0) node[anchor=north east] {-}
;\end{circuitikz}
```

5.9 Componenti non-NGSpice

Esiste un'intera classe di componenti elettronici non previsti in forma nativa in NGSpice ma che ciononostante vengono rappresentati in CircuiTikZ, come le porte logiche, i doppi dipoli (es. il trasformatore), ecc. In NGSpice sono composti di, o riconducibili a, componenti elettronici elementari che possono essere resi disponibili mediante librerie. In CircuiTikZ questi componenti sono identificati come entità e quindi nel DTD non hanno un elemento dedicato. Il DTD mette a disposizione un elemento generico `<node-component>` che può essere impiegato per definire volta per volta il componente desiderato, come ad esempio una porta logica oppure un filtro passa-basso, ed inol-

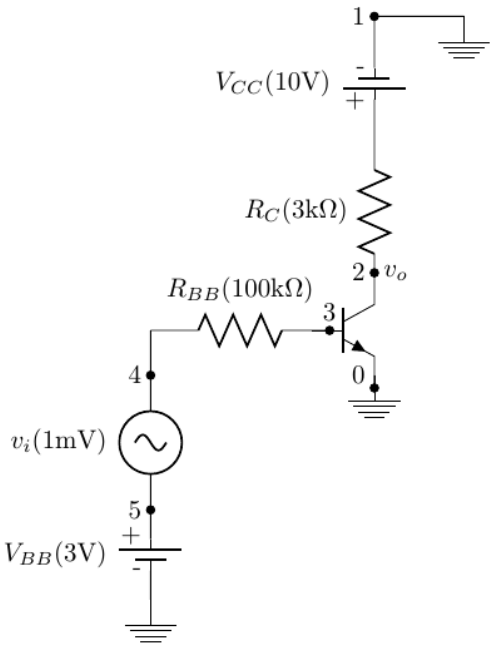


Figura 4: L'immagine dell'amplificatore di tensione a singolo stadio.

tre anche un elemento generico `<node-label>` per etichettare i nodi:

```
<!doctype ngspice system 'ngspice.dtd'>
<ngspice>
  <title>Logic gates</title>
  <circuit>
    <node-component xy-node="(0,2)"
        component="and␣port"
        id-node="myand1">
    <node-component xy-node="(0,0)"
        component="and␣port"
        id-node="myand2">
    <node-component xy-node="(2,1)"
        component="xnor␣port"
        id-node="myxnor">
    <right-angle-circuit>
      <xy-plus>(myxnor.in 1)</xy-plus>
      <hvline>
      <xy-minus>(myand1.out)</xy-minus>
    </right-angle-circuit>
    <right-angle-circuit>
      <xy-plus>(myxnor.in 2)</xy-plus>
      <hvline>
      <xy-minus>(myand2.out)</xy-minus>
    </right-angle-circuit>
    <node-label id-node="myand1"
        sub-id-node="out" anchor="south␣
        west" label="AND␣1␣out">
    <node-label id-node="myand2"
        sub-id-node="out" anchor="north␣
        west" label="AND␣2␣out">
    <node-label id-node="myxnor"
        sub-id-node="out" anchor="west"
        label="XNOR␣out">
  </circuit>
  <end-line>
</ngspice>
```

Il codice SGML una volta mappato

```
%
% Logic gates
```

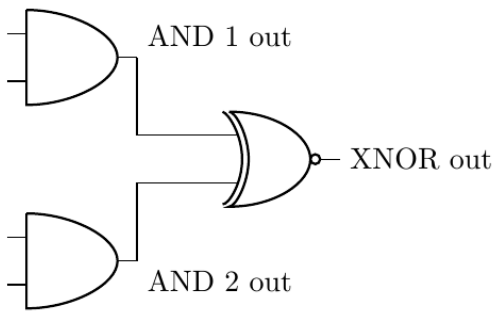


Figura 5: Porte logiche.

```
%
\begin{circuitikz}[scale=1.2,american]
\draw(0,2) node[and port] (myand1) {}
(0,0) node[and port] (myand2) {}
(2,1) node[xnor port] (myxnor) {}
(myxnor.in 1) -| (myand1.out)
(myxnor.in 2) -| (myand2.out)
(myand1.out) node[anchor=south west]
    {AND 1 out}
(myand2.out) node[anchor=north west]
    {AND 2 out}
(myxnor.out) node[anchor=west] {XNOR out}
;\end{circuitikz}
```

e compilato, fornisce l'immagine in Fig. 5.

6 Conclusioni

L'attuale DTD è da considerarsi ancora in fase di sviluppo seppure la struttura generale delle parti finora implementate possa definirsi stabile. La parte tipografica attualmente supporta la maggioranza dei componenti elettronici non-NGSpice di CircuiTikZ in forma generica, come visto nel caso delle porte logiche. Questo comporta inevitabilmente un minore supporto in fase di stesura del codice SGML per l'utente, il quale deve ricorrere alle proprie conoscenze di CircuiTikZ, e può eventualmente dare origine a problemi di validazione. Il DTD attuale è anche privo di riferimenti a comandi CircuiTikZ quali `ctikzset` che consentirebbe di adeguare parametricamente la rappresentazione grafica dei componenti elettronici; l'ambiente `scope`, seppure concettualmente semplice da implementare; i tripoli come il tiristore, il potenziometro e il deviatore; anche i cosiddetti *percorsi di transistor* (*transistor paths*) non sono stati ancora implementati. Tutti questi componenti e comandi possono attualmente essere descritti in SGML, ma solo scrivendo direttamente il codice CircuiTikZ protetto entro l'elemento `<tikz>`. Complessivamente si può affermare che, seppure utilizzabile, il DTD attuale richiede comunque una conoscenza piuttosto approfondita di CircuiTikZ e in qualche misura anche di Ti*k*Z, soprattutto per quanto concerne le sue regole di disegno.

7 Ringraziamenti

L'autore desidera ringraziare Claudio Beccari per le sue preziose osservazioni relative ai circuiti elettronici citati nell'articolo.

Riferimenti bibliografici

Aplevich, D. *M4 Macros for Electric Circuit Diagrams in LaTeX Documents*. URL http://ctan.mirror.garr.it/mirrors/CTAN/graphics/circuit_macros/doc/CMman.pdf.

Nagel, L. W. «The origins of spice». URL http://www.omega-enterprises.net/The%20Origins%20of%20SPICE.html.

Redaelli, M. A., Lindner, S. e Erhardt, S. (2017). *CircuiTikZ*. URL http://ctan.mirror.garr.it/mirrors/CTAN/graphics/pgf/contrib/circuitikz/doc/circuitikzmanual.pdf.

Roberts, G. W. e Sedra, A. S. (1997). *SPICE*. Oxford University Press.

Tantau, T. (2015). *The TikZ and PGF Packages Manual*. Institut fur Theoretische Informatik Universitat zu Lubeck. URL http://ctan.mirror.garr.it/mirrors/CTAN/graphics/pgf/base/doc/pgfmanual.pdf.

Voss, H. *pst-circ - A PSTricks package for drawing electric circuits*. URL http://ctan.mirror.garr.it/mirrors/CTAN/graphics/pstricks/contrib/pst-circ/doc/pst-circ-doc.pdf.

▷ Renato Battistin
rbattistin at apf dot it

A Template Engine with TEX and Friends

Paulo Roberto Massa Cereda

Abstract

This article presents some insights towards using template engines with TEX and friends with the help of a command line tool written for the sole purpose of merging data sources and templates.

Sommario

Questo lavoro presenta alcune idee sull'uso dei template con il sistema TEX, per mezzo di strumenti a riga di comando con lo scopo di unire i dati ai modelli di template.

1 Introduction

Repetition constitutes a fundamental part of one fair day's work for a fair day's pay. Very often, we end up with the very same tasks exhaustively executed. As a means to achieving time and resource optimizations, such repetitive patterns could be factorized and reused whenever needed.

Reports, letters and certificates are instances of document patterns that are repeated over and over. Office suites such as Libre Office and Microsoft Office offer features to merge documents containing special elements (also known as place holders) with external data sources (a database or a spreadsheet), resulting in new documents composed of both structure and data.

Beyond academic usage, TEX and friends offer an interesting approach for providing document patterns. The advent of the great datatool package brought unlimited possibilities for creating documents based on the provided data (represented as a table in the CSV format) without worrying about the structure and layout. For example, consider the following code:

letter.tex

```
\documentclass{article}
\usepackage{datatool}
\DTLloaddb{db}{list.csv}

\begin{document}
\DTLforeach{db}{\name=name,\gift=gift}{
Dear Santa, I've been a good boy
this year, please bring me a \gift.

Merry Christmas from \name!
\newpage}
\end{document}
```

In this example, given a list of people and their respective gifts, it will be easy to generate letters for Santa Claus (of course, the demand for gifts will increase greatly, but little we can do about it). A sample list in the CSV format is presented as follows:

list.csv

```
name,gift
Enrico,motorbike
David,pineapple pizza
Paulo,rubber duck
```

Based on both document structure and data, three letters to Santa will be generated in a single LATEX run, one letter per page. However, I have a feeling that only two gifts from that list will be effectively delivered[1].

A TEX-based solution is surely desirable in most cases. In this paper, however, I advocate for an alternative approach: template engines. I must point out that I am an avid user of datatool and the package really does wonders. The motivation for taking a different route relies on the fact that template engines are generic enough to be used beyond the TEX world. Nicola Talbot provides an important note on the datatool manual cover:

> The datatool bundle is provided to help perform repetitive commands, such as mail merging, but since TEX is designed as a typesetting language, don't expect this bundle to perform as efficiently as custom database systems or a dedicated mathematical or scripting language. If the provided packages take a frustratingly long time to compile your document, use another language to perform your calculations or data manipulation and save the results in a file that can be input into your document.
>
> — The datatool user manual

For this paper, the implementation and deployment details of a template engine from a programming language perspective will not be covered. Instead, I plan to keep the discussion solely at the user level. In order to make things easier, the narrative will be assisted by a command line tool, which I forgot I wrote it a couple of years ago! I wish you all a great reading.

1. I am sure David does not like pineapple pizza at all, but I cannot lose this opportunity of shocking Italian readers on his behalf.

2 My kingdom for a tool

Back in 2012, I was working with template engines for a report generation. So far, I only had come up with ad hoc solutions, but that mindset was about to change: what if I had a tool that merges a list of data sources into a template and generates the corresponding output? At first, my primary goal was a tool specifically designed for TEX and friends, but as time went by, the concept was extended to virtual any textual transformation.

Java was my language of choice at that time (and, not surprisingly, it still is), so the programming language in which the tool would be written was settled. But one question was still open:

> Which template language is suitable for my needs? Would it cover most scenarios?

It would be desirable to be a template language with an easy syntax and potentially small footprint. Additionally, being Java-friendly would be a great bonus, as integration during the development phase had to be less traumatic as possible.

Being myself a fan of the Apache Foundation, I suddenly found the answer to my inquiries while browsing the list of active projects: I would use Velocity as my template language. This particular project aims at providing a simple yet powerful language known as Velocity Template Language (VTL), as a means to generate any sort of text-based content according to a predefined structure and corresponding data source.

2.1 The Velocity language: a primer

A brief introduction to the Velocity Template Language is advisable. In short, VTL uses references to embed content in a document. Let us see how the traditional `Hello world` example would be:

```
#set( $foo = "Velocity" )
Hello $foo world!
```

Everything that starts with `#` is a statement. In our example, `set` is a directive that sets the value of a reference; `$foo` is a variable and gets the literal string `Velocity`. The output will be:

```
Hello Velocity world!
```

Observe that variables are preceded by `$` and are available after being declared. A variable can be explicitly declared with the `#set` directive or through a map of variables coming from the programming side. References `$foo` and `${foo}` denote the same variable. The second notation is advisable when handling complex operations.

Internally, a variable represents a Java object and thus all corresponding methods are available. For example, consider the following snippet:

```
#set( $foo = "Velocity" )
Hello ${foo.toUpperCase()} world!
```

Since `$foo` holds a string value, we can call the `toUpperCase()` method available in the `String` class from Java. The new output will be:

```
Hello VELOCITY world!
```

When rendering a reference, the value is automatically converted to a string. For instance, if `$foo` holds an object that represents an integer value, Velocity will call its `toString()` method and resolve the object into a string.

Variables can also be arrays, so an element can be accessed through its index, e.g., `$foo[0]` and so on. It is also important to observe that all array references are treated as if they are fixed length lists, meaning that all methods from the `List` class are available.

The `#if` directive allows for text to be included in the output on the conditional that the statement holds true. Consider the following example:

```
#set( $user = "Enrico" )
#if( $user == "Enrico" )
Forza Juve!
#else
Go Palmeiras!
#end
```

The variable `$user` is evaluated to determine whether the condition (that is, if `$user` holds the string value `Enrico`) is true. In our example, the condition is true, then the output will be:

```
Forza Juve!
```

The `#foreach` directive allows looping over the elements of a Java `Vector`, `Hashtable` or an array. For example, if the variable `$stooges` contains the names of the Three Stooges, the following code will iterate through them:

```
#set( $stooges = [ "Moe",
        "Curly", "Harry" ] )
#foreach ( $stooge in $stooges )
$foreach.count - $stooge
#end
```

Note that there is a special variable `$foreach` holding several details about the particular looping execution. In our example, `$foreach.count` holds the current loop counter. The new output will be:

```
1 - Moe
2 - Curly
3 - Harry
```

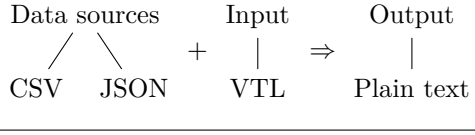
There is much more about VTL, but I believe this introduction covered all the bits we need to proceed in our narrative. More information about Apache Velocity can be found in the project website:

`https://velocity.apache.org`

2.2 The merging tool

Once the template language was chosen, it was a matter of time until I had a merging tool ready to be used in production: `duckity`. The culprit for the command line tool name is no one other than Enrico Gregorio himself, as he gave me the suggestion in the chat room of the TEX community at StackExchange. Figure 1 shows the default output of `duckity` when executed in the terminal without any command line parameters.

Let us start with the basics: `duckity` can merge any template written in VTL with any combination of CSV or JSON data sources. In short:



The data sources information (namely, file and identifier) can be described inside the input file through a specific JSON header, as seen in the following example:

```
{ "datasources": [
  {
    "file"       : "grades.csv",
    "identifier" : "students"
  }
] }
```

Observe that `datasources` is a list, so `duckity` expects as many pairs containing the file name and the identifier as the template requires for a proper merging. The identifier plays an important role in the template, as it will act as a reference to its corresponding file content (for example, `students` will become a variable named `$students` that refers to the CSV data obtained from `grades.csv`). It is also important to note that the JSON format always encode entries as strings (both keys and values), and `duckity` will parse the file references accordingly based on the file extension.

Once the JSON header is defined, there is a special mark that divides header and template, and `duckity` will always expect this mark (followed by a line break) to set up the template before merging:

```
[TEMPLATE]
```

Once the mark is found, `duckity` will consider every text after it to be written in VTL. Similarly, everything before the mark is ignored in the resulting output.

The JSON header can be omitted and replaced by command line flags representing the same information. The following command line flags represent the same information of our JSON header from the last example:

```
-d grades.csv -i students
```

There can be multiple command line flags representing multiple data sources and identifiers, and it is required that they always come in pairs (that is, the same number of identifiers and data sources). Note that the special mark is not needed in this scenario, as the entire file content will be considered as a VTL template.

And that is it, the tool has a very straightforward behavior: get the data sources information, retrieve the corresponding contents, process the template and write the output (that is, the merged template) to a file. In Section 3, we will take a look at some examples.

3 Examples

For our first example, let us consider a table of student grades in the CSV format. The table is presented as follows:

grades.csv

```
Alice,8.0,7.3
Bob,2.2,6.7
Carl,10.0,9.3
David,9.2,10.0
```

Now, let us focus on the table presentation. We need to iterate through this table (a list of rows) and we can use the `#foreach` statement previously seen to achieve this goal:

```
#foreach( $student in $students )
Notes of $student[0]:
- First exam: $student[1]
- Second exam: $student[2]
#end
```

Note that the indices represent the columns (starting from zero) in our table. The template is a plain text, but we can easily make it TEX-aware:

```
[paulo@nineveh ~] $ duckity
  _|      _ |  o _|_
 (_| |_| (_ |< |  |_ \/
                    /
duckity 1.1 - The template helper
Copyright (c) 2012, Paulo Roberto Massa Cereda
All rights reserved.

usage: duckity [file [--datasource D --identifier I]*
        --output file | --help | --version]
 -d,--datasource <arg>   set the datasource
 -h,--help               print the help message
 -i,--identifier <arg>   set the identifier
 -o,--output <arg>       set the output file
 -v,--version            print the application version
```

Figure 1: The default output of `duckity` when executed without any command line parameters.

```
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
#foreach( $st in $students )
$st[0] & $st[1] & $st[2] \\
#end\bottomrule
\end{tabular}
```

Cool, we got our first template done (note that the `S` column type comes from siunitx). Now let us see the entire template (plus header), ready for merging:

ex1.tex

```
{ "datasources": [
  {
    "file"       : "grades.csv",
    "identifier" : "students"
  }
] }
[TEMPLATE]
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{siunitx}
\usepackage{booktabs}
\begin{document}
\begin{table}[h]
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
#foreach( $st in $students )
$st[0] & $st[1] & $st[2] \\
#end\bottomrule
\end{tabular}
\end{table}
\end{document}
```

And that is it! Now `duckity` can process `ex1.tex` and merge `grades.csv` with the underlying template. Do not worry with the `$` syntax of VTL, as they will not mess with your mathematical formulas! The execution of `duckity` is as follows, it only suffices to provide the output flag:

```
[paulo@nineveh ~] $ duckity ex1.tex \
-o out1.tex
  _|      _ |  o _|_
 (_| |_| (_ |< |  |_ \/
                    /
Done.
```

Checking `out1.tex`, we can observe the merging was successful, as the grades were correctly typeset! Highlighting just the important bits:

```
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
Alice & 8.0 & 7.3 \\
Bob & 2.2 & 6.7 \\
Carl & 10.0 & 9.3 \\
David & 9.2 & 10.0 \\
\bottomrule
\end{tabular}
```

Now let us add a different scenario: what if we want to display only the students that did not fail on the first exam? The solution is simple, it is a matter of adding a simple check inside the `#foreach` statement:

```
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) > 5.0 )
- $st[0]
#end
#end
```

Observe that we had to make use of a helper object (referenced by `$math`), since the CSV and JSON parsers resolve all entries to strings. A TEX version of the plain text excerpt is presented as follows:

```
\begin{itemize}
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) >= 5.0 )
\item $st[0]
#end
#end
\end{itemize}
```

Now let us see a complete example based on the previous code. It is important to note that if every student failed on exam 1, the generated TEX file will contain an error: an empty itemize environment! In this case, the solution would be populating an auxiliary variable (through the `#set` directive) with the students who succeeded on exam 1 and the check the length of this variable. That will be left as an exercise to the reader.

ex2.tex

```
{ "datasources": [
  {
    "file"       : "grades.csv",
    "identifier" : "students"
  }
] }
[TEMPLATE]
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\begin{document}
\begin{itemize}
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) >= 5.0 )
\item $st[0]
#end
#end
\end{itemize}
\end{document}
```

After running `duckity` on `ex2.tex`, the list of students that succeeded on exam 1 is correctly generated:

```
\begin{itemize}
\item Alice
\item Carl
\item David
\end{itemize}
```

If `ex2.tex` did not have the JSON header (nor the special template mark), the command line invocation would be:

```
[paulo@nineveh ~] $ duckity ex2.tex \
-d grades.csv -i students -o out2.tex
  _|      _ |  o _|_
 (_| |_| (_ |< |  |_ \/
                   /
Done.
```

So far, our examples contemplated data sources in the CSV format. For the sake of completeness, consider the following file specified in the JSON format (note that the square brackets denote a list of elements):

person.json

```
{
  "name"      : "Paulo",
  "location"  : "Brazil",
  "interests" : [ "ducks", "Pringles" ]
}
```

While elements from CSV references are indexed by positional values (that is, integers representing the columns starting from zero), elements from JSON references are indexed by their corresponding identifiers (also known as keys). If the data source identifier is `person` (thus the reference will be `$person`), in order to access the `name` element, it suffices to invoke `$person.name` in VTL. The following example makes use of such keys in order to build a sentence:

ex3.tex

```
{ "datasources": [
  {
    "file"       : "person.json",
    "identifier" : "person"
  }
] }
[TEMPLATE]
My name is $person.name, I am from
$person.location and my interests
include #foreach( $i in
$person.interests )$i#if(
$foreach.hasNext() )#if(
$foreach.count ==
$person.interests.size() - 1)
 and #else, #end#end#end.
\bye
```

Observe the presence of `$foreach.hasNext()`, which indicates if the loop still have elements to iterate, plus an evaluation on the list size. This

is a trick to display elements from a list, so the last element is preceded by the word `and` and not by a comma (the result looks more natural and visually appealing). Velocity statements can be nested in order to generate an elaborated output (also note that, in some cases, spaces do matter!). The generated TEX code is presented as follows:

out3.tex

```
My name is Paulo, I am from Brazil
and my interests include ducks
and Pringles.
\bye
```

We can have multiple data sources in the same template, provided that they have different identifiers. The next example covers this scenario:

ex4.tex

```
{ "datasources": [
  {
    "file"       : "grades.csv",
    "identifier" : "students"
  },
  {
    "file"       : "person.json",
    "identifier" : "person"
  }
] }
[TEMPLATE]
I am $person.name and my friend
is $students[0][0]!
\bye
```

The first row from the CSV file is directly accessed through `$students[0]` (no need for a `#foreach` statement here), and the first index gives the student name. The output is as follows:

out4.tex

```
I am Paulo and my friend is Alice!
\bye
```

It is worth mentioning that version 4.0 of `arara` has support for VTL through a `velocity` rule. The corresponding directive takes the following parameters:

- `input`: optional, it refers to the input file containing the VTL template. If not specified, `arara` will assume the current file being processed holds the template.
- `output`: mandatory, it refers to the output file, generated from the template merging.
- `context`: mandatory, it contains a map that represents all references to be included in the template. It is very similar to a JSON file, but only the keys are expected to be strings, values can be anything that maps to a Java object (from primitives to complex data structures).

The next example illustrates how `arara` handles templates using VTL. Observe that, contrary to `duckity`, there is only one data source, that is, the map represented by the `context` parameter (similar to JSON syntax, but it is still YAML). Map entries are directly mapped inside the template with no prefix.

ex5.tex

```
% arara: velocity: {
% arara: --> output: out5.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
Hello, my name is $name.
\bye
```

Now, it is just a matter of invoking `arara` on `ex5.tex` and the merged file `out5.tex` (set as output in the `velocity` directive) will be automatically generated:

```
[paulo@nineveh ~] $ arara ex5.tex
```

The resulting file `out5.tex` is as follows. Observe that the directives from `ex5.tex` were transposed to the merged file as well, as they are also part of the VTL template:

out5.tex

```
% arara: velocity: {
% arara: --> output: out5.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
Hello, my name is Paulo.
\bye
```

We can exploit the use of conditionals in order to make `arara` process those two files differently, even if directives are transposed. Consider the new example presented as follows:

ex6.tex

```
% arara: velocity: {
% arara: --> output: out6.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> } if currentFile().
% arara: --> getName() == "ex6.tex"
% arara: pdftex if currentFile().
% arara: --> getName() == "out6.tex"
Hello, my name is $name.
\bye
```

The new `velocity` directive has a conditional now, so `arara` will only execute the corresponding rule if, and only if, the current file being processed is `ex6.tex`. Otherwise, the tool will simply ignore this directive and move on. We also included a `pdftex` directive using the same trick, now checking if the current file is the proper one

```
[paulo@nineveh ~] $ arara ex7.tex
  __ _ _ __ __ _ _ __ __ _
 / _‘ | ’__/ _‘ | ’__/ _‘ |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing ’ex7.tex’ (size: 178 bytes, last modified: 09/17/2017
13:08:39), please wait.

(Velocity) The Velocity engine ......................... SUCCESS
(PDFTeX) PDFTeX engine ................................. SUCCESS

Total: 0.37 seconds
```

Figure 2: Execution of `arara` in the command line.

(i.e, `out6.tex`). In this scenario, two independent runs of `arara` are required, one for each file.

However, we can do better than this. Let us forget about conditionals and use the `files` parameter on our `pdftex` directive and get our result with a single run. Consider the following example:

ex7.tex

```
% arara: velocity: {
% arara: --> output: out7.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
% arara: pdftex: {
% arara: --> files: [ out7.tex ]
% arara: --> }
Hello, my name is $name.
\bye
```

A nice improvement indeed! On a single run of `arara`, we get `out7.tex` from our template and compile it using the `pdftex` rule. The output of this particular run is shown in Figure 2.

It is worth mentioning that `arara` is not exactly the best way of handling templates regarding data sources. For this matter, `duckity` is far superior, as it offers two data source formats, namely CSV and JSON. However, both tools are powered by VTL, which is a fantastic template language, regardless of which back-end is used. Your mileage may vary. For `arara`, I might include support for data sources in the future, so please let me know if this feature would be interesting. For now, `duckity` is the best option if you want to take a direct approach to generating any sort of text-based content according to a predefined structure and corresponding data source. For a general approach, `arara` might help.

Unfortunately, `arara` 4.0 is not officially released yet, as the user manual is being written at the moment. Hopefully, the tool will be available soon. For now, you can easily build a development version, provided that you have an instance of a Java Development Kit (also known as JDK) and a tool named Maven (from the Apache Foundation). The source code is hosted at GitHub:

`https://github.com/cereda/arara`

Hopefully, these examples covered most of the common scenarios when handling data sources with templates. It is highly advisable to take a closer look at the VTL guide in order to learn the advanced features that particular scripting language has to offer.

4 Final remarks

This article discussed the use of template engines as an alternative approach to TEX-based solutions on reading data sources. Template engines are generic enough in order to cover different domains of application. The discussion was kept at the level user, with the assistance of a command line tool named `duckity`.

The command line tool is open source and released under the New BSD license. Java binaries, as well as the source code, are available in the project repository at GitHub:

`https://github.com/cereda/duckity`

Happy TEXing with templates!

Acknowledgments

The author wishes to thank Claudio Beccari, Enrico Gregorio, Carla Maggi and all friends from GUIT for the opportunity of writing this humble article for ArsTEXnica.

▷ Paulo Roberto Massa Cereda
Università di San Paolo, Brasile
`paulo dot cereda at usp dot br`

Questa rivista è stata prodotta
dal Gruppo Utilizzatori Italiani di TEX
usando esclusivamente software libero.

Versione elettronica per la diffusione via web.

ArsTEXnica – Call for Paper

La rivista è aperta al contributo di tutti coloro che vogliano partecipare con un proprio articolo. Questo dovrà essere inviato alla redazione di ArsTEXnica, per essere sottoposto alla valutazione di recensori entro e non oltre il 14 Febbraio 2018. È necessario che gli autori utilizzino la classe di documento ufficiale della rivista; l'autore troverà raccomandazioni e istruzioni più dettagliate all'interno del file d'esempio (.tex).

Gli articoli potranno trattare di qualsiasi argomento inerente al mondo di LaTeX e non dovranno necessariamente essere indirizzati ad un pubblico esperto. In particolare tutorial, rassegne e analisi comparate di pacchetti di uso comune, studi di applicazioni reali, saranno bene accetti, così come articoli riguardanti l'interazione con altre tecnologie correlate.

Di volta in volta verrà fissato, e reso pubblico sulla pagina web `http://www.guitex.org/arstexnica/`, un termine di scadenza per la presentazione degli articoli da pubblicare nel numero in preparazione della rivista. Tuttavia gli articoli potranno essere inviati in qualsiasi momento e troveranno collocazione, eventualmente, nei numeri seguenti.

Chiunque, poi, volesse collaborare con la rivista a qualsiasi titolo (recensore, revisore di bozze, grafico, etc.) può contattare la redazione all'indirizzo arstexnica@guitex.org.

ArsTEXnica

Rivista italiana di TEX e LATEX

Numero 24, Ottobre 2017

70024
9 771828 236001