Uno script bash di ausilio alla redazione di manoscritti

Gianluca Pignalberi

Sommario

La fase di redazione di manoscritti ci pone spesso di fronte a una serie di cattive pratiche reiterate dagli autori. La correzione interamente manuale può essere fonte di dimenticanze. Vediamo come uno script bash ci consente di minimizzarle.

Abstract

A manuscript editing session puts us in front of a series of authors' repeated bad practices. An entirely-by hand correction can be source of oversights. We will see how a bash script allows us to minimize them.

1 Introduzione

Il mio lavoro di impaginatore LaTeX mi mette molto più spesso di fronte a manoscritti redatti con un word processor che a manoscritti redatti con LyX o direttamente in LaTeX.[1] In base a quanto visto direttamente, è raro che gli utenti di word processor mantengano alta la propria attenzione a qualcosa che non sia il contenuto quando redigono i proprî manoscritti. La loro negligenza relativa alle regole tipografiche più elementari e logiche fa sì che ai redattori si presentino sovente dei documenti dall'aspetto rabberciato e povero, quando non confuso e incoerente, dal punto di vista tipografico.

1. Molti autori usano scorrettamente i glifi e ciò danneggia il testo agli occhi dei redattori e dei lettori più esigenti e pignoli.

2. Altri, non necessariamente diversi, non fanno caso all'interezza delle porzioni di testo alle quali applicano determinate proprietà. Ciò può introdurre dei problemi nell'eventuale ebook, oltre ad aumentare la dimensione del file `.tex` risultante dalla conversione.

3. Molti autori tendono a non usare gli stili: questo può far sì che l'aspetto di elementi di pari semantica non sia coerente.

4. Molti autori, o forse il word processor da essi utilizzato, tendono a impostare incoerentemente più lingue all'interno dei proprî documenti.

1. La percentuale attuale è circa il 2 % di manoscritti LaTeX e 0 % di manoscritti LyX e Libre/OpenOffice. Serve specificare quale programma è usato nel 98 % dei casi?

Il lavoro del redattore è già messo a dura prova dalla correzione del testo: come esposto in Rawlinson (1976) ed esemplificato in Polidoro (2012), il cervello umano è in grado di leggere correttamente le parole con la prima e l'ultima lettera poste correttamente e le altre mescolate e magari con qualche refuso, fungendo così da correttore ortografico e rendendo difficile la correzione delle bozze.[2] Quando si arriva al momento di controllare tutte quelle piccole minuzie relative alle più elementari regole tipografiche, tale lavoro può diventare ancora peggiore. Due famosi studî psicologici (Simons e Levin (1998) e Simons e Chabris (1999)) hanno mostrato come l'attenzione umana nei confronti di un compito ci renda ciechi ai cambiamenti anche macroscopici del mondo circostante. Posso supporre, non essendo stato in grado di reperire studî specifici, che anche l'attenzione selettiva, oltre alla capacità di "correzione inconscia" che forse funziona anche sui simboli, influisca negativamente sul lavoro del redattore.

L'articolo porterà avanti due finalità: 1) l'analisi di alcuni casi in cui un autore o un word processor mettono alla prova il lavoro del redattore (di alcuni casi spiegheremo la semantica o le cause, chiarendo ulteriormente la natura degli errori commessi) e 2) la scrittura di uno script bash che scovi per noi tutti i casi esplicitati nell'articolo e potenzialmente dannosi nei file LaTeX, nativi o convertiti dagli originali word processor. A causa della mia predilezione per i sistemi Unix-like la seconda finalità sarà principalmente indirizzata agli utenti di tali sistemi (come Linux e Mac OS X), ma non disdegna gli utenti di Windows che abbiano installato un interprete di comandi bash sul proprio computer (Cygnus Solution-Red Hat (2019) oppure Bruessow (2017) tra le soluzioni più note). Lo script avrà il solo compito di analisi perché in alcuni casi, discussi puntualmente nell'articolo, la correzione automatica non farebbe che aumentare i problemi.

L'articolo prosegue così: la sezione 2 presenta brevemente la struttura dell'algoritmo[3] alla base

2. Fortunatamente esistono i correttori ortografici che possono aiutare su questo fronte. Qualche word processor si spinge al controllo semantico...

3. Ricordiamo che i termini algoritmo e procedura non sono equivalenti. L'algoritmo (Bertossi, 1990, 11) «che significa procedimento [...] indica la descrizione precisa delle *azioni* che un *esecutore* deve compiere per giungere alla soluzione di qualsiasi *problema* computazionale. [...] [P]uò essere considerato come un manipolatore di dati che, a fronte di certi dati d'ingresso consistenti con la natura del problema da risolvere (dati di *input*), produce altri dati come risultato

dello script descrivendone le parti più significative del codice; le sezioni 3–6 discuteranno alcuni casi d'uso reali tra quelli che ricadono nei punti 1–4 del precedente elenco e il relativo codice bash. Poiché alcuni di questi casi sono significativi per la sola lingua italiana, adattare lo script per altre lingue comporterà modificare alcuni test oppure cancellarne alcuni per aggiungerne di significativi nella lingua prescelta. Prima di concludere, la sezione 7 presenterà per intero lo script, ne descriverà alcuni dettagli implementativi e lo metterà all'opera su alcuni file di prova per testarne il funzionamento e mostrarne i risultati.

2 Impostazione dello script

Sebbene niente possa (ancora) sostituire l'accurato controllo umano di un testo scritto da un umano, un aiuto automatico nei compiti più ripetitivi è sempre benvenuto. Uno dei compiti più ripetitivi che si possa immaginare è controllare se tutte le voci di una *checklist* siano presenti o meno in un documento, specie se questo è costituito da più di qualche file (o di qualche pagina se siamo di quelli che lavorano coi fogli stampati). Ritengo che a qualunque redattore possa far comodo avere un programma che controlli la *checklist* in sua vece e gli faccia un rapporto su quali file contengano quali voci. La presenza di un rapporto riduce e circoscrive i controlli manuali e, soprattutto, funge da promemoria sulle eventuali correzioni da fare. Il programma avrà le sembianze di uno script bash che scriverà per noi il rapporto come conseguenza dell'analisi di uno o più file di testo (che ricordiamo essere il formato dei file `.tex`) alla ricerca dei casi discussi nelle prossime sezioni.[4]

Per prima cosa vogliamo che lo script usi bash indipendentemente dalla shell in uso.[5] Dunque la prima riga conterrà la sequenza di caratteri nota come *shebang*[6] e il comando completo di percorso assoluto:[7]

```
#!/bin/bash
```

Immediatamente dopo vogliamo controllare che l'analizzatore abbia ricevuto in input almeno un file:[8]

```
1  if␣[[␣$BASH_ARGC␣<␣1␣]];␣then
2  ␣␣echo␣"Uso:␣editanalyze␣<file\
3  ␣da␣analizzare>"
4  ␣␣echo␣"Es.:␣editanalyze␣*.tex\
5  ␣(controlla␣tutti␣i␣file␣con\
6  ␣estensione␣.tex)"
7  ␣␣echo␣"␣␣␣␣␣editanalyze\
8  ␣capitolo1.tex␣(controlla\
9  ␣il␣solo␣file␣capitolo1.tex)"
10 ␣␣echo␣"␣␣␣␣␣editanalyze\
11 ␣capitolo[1-5].tex␣(controlla\
12 ␣i␣file␣capitolo1-capitolo5\
13 .tex)"
14 ␣␣exit␣1
15 fi
```

Questo pezzo di codice controlla che il comando abbia ricevuto almeno un argomento (riga 1; `$BASH_ARGC` vale 0 se non abbiamo dato argomenti al comando); in caso negativo, stampa alcune righe riassuntive sull'uso dello script, mostrando esplicitamente la possibilità di ricorrere a *wildcard* e a espressioni regolari (Goyvaerts, 2019) (righe 2–13), quindi interrompe l'esecuzione dello script uscendo con codice di errore 1 (riga 14). Se il test precedente è negativo (cioè $\texttt{\$BASH\_ARGC} \geqslant 1$) abbiamo passato almeno un parametro e quindi lo script può continuare.

Vogliamo che il rapporto, memorizzato nel file `report.txt`, contenga l'elenco dei file in cui è stato riscontrato ognuno dei casi oggetto di analisi, elenco preceduto dall'indicazione del relativo caso, come per esempio:

```
Il carattere ° si trova in
1.tex
3.tex
```

Capiamo quindi che i casi da esaminare si riducono alla presenza o meno di uno o più caratteri (una stringa) in configurazioni più o meno lineari. Prima di vedere l'algoritmo, diamo due definizioni di comodo:

Definizione 1. Un file è *positivo* (all'analisi) se contiene almeno un'occorrenza del testo cercato.

Definizione 2. Un file è *negativo* (all'analisi) se non contiene neanche un'occorrenza del testo cercato.

del problema (dati di *output*)». «Un algoritmo descritto per mezzo di *costrutti* tipici di un linguaggio di programmazione è comunemente detto *procedura*» (Bertossi, 1990, 13).

4. L'idea di produrre un tale script è nata successivamente all'idea di un articolo che descrivesse gli errori più frequenti degli autori, argomento tutto sommato abbastanza circoscritto. La versione dello script qui presentata è embrionale, forse didattica e quasi per niente testata in casi reali. In una versione precedente dell'articolo il redattore ha giustamente evidenziato l'incapacità dello script di trattare nomi di file contenenti degli spazî. Tutte le altre mancanze dello script, dal controllo sull'esistenza dei file di input alla scelta arbitraria del nome del file contenente il rapporto passando per la suggerita inefficienza, verranno implementate in futuro.

5. Alcuni sistemi Unix usano, di default o per scelta amministrativa, shell diverse da bash, quali sh, [t]csh, ksh e molte altre.

6. Così viene chiamata, per esempio, su Powers *et al.* (2002). Altre fonti, quali Wikipedia (2019) ne danno altre versioni e possibili significati.

7. Bisogna comunque ricordarsi di rendere lo script eseguibile, altrimenti il sistema operativo non lo riconoscerà come comando e non lo eseguirà.

8. Da notare che i testi dentro le virgolette dopo ogni comando `echo` vanno scritti su una riga. Nel caso la riga sia troppo lunga, potremo interromperla col carattere *di fuga* (comunemente detto *di escape*) \. Facciamo attenzione al numero di spazî dopo di esso per mostrare sullo schermo il testo di aiuto pulito e ordinato. In tutti i successivi brani di codice eviteremo l'uso esplicito del carattere di fuga e lasceremo a LaTeX l'incombenza di mandare a capo (solo tipograficamente) le righe troppo lunghe.

L'algoritmo proposto è il seguente:

```
stampa sullo schermo il caso da analizzare
per ognuno dei file ricevuti in input
  il file è positivo?
    SÌ: stampa il caso nel rapporto
        stampa il nome del file nel rapporto
        esci dal ciclo
    NO: non fare niente
per ognuno dei rimanenti file da analizzare
  il file è positivo?
    SÌ: stampa il nome del file nel rapporto
    NO: non fare niente
```

Il primo ciclo assicura che il caso verrà stampato nel rapporto solo se c'è un file positivo, il cui nome verrà inserito nel rapporto. Il secondo ciclo controlla i rimanenti file e stampa nel rapporto i soli positivi. Scritto in *bash-like*, l'algoritmo si presenta così:

```
1  echo␣"ANALISI␣DEL␣CASO␣IN␣ESAME"
2  count=0
3  for␣i␣in␣"$@";␣do
4  ␣␣if␣(␣CONDIZIONE␣DI␣TEST␣);␣then
5  ␣␣␣␣echo␣"TESTO␣DEL␣CASO␣IN␣ESAME"␣
       >>␣report.txt
6  ␣␣␣␣echo␣"$i"␣>>␣report.txt
7  ␣␣␣␣break
8  ␣␣fi
9  ␣␣count=$[$count+1]
10 done
11 args=("$@")
12 for␣((␣count=$[$count+1];␣$count<
       $BASH_ARGC;␣count=$[$count+1]␣))
       ;␣do
13 ␣␣CONDIZIONE␣DI␣TEST
14 done
```

Le locuzioni "condizione di test", "analisi del caso in esame" e "testo del caso in esame" sono in linguaggio naturale e andranno sostituite con del codice o delle scritte significative che vedremo nelle prossime sezioni. Soffermiamoci brevemente sul significato del codice scritto finora.

Le righe 1, 5 e 6 stampano qualcosa; a video se l'ultimo carattere della riga è `"`, su file in modalità *append* se dopo le virgolette di chiusura troviamo la sequenza `>>` seguita dal nome di un file. La riga 3 è un ciclo che assegna alla variabile `i` il contenuto di `"$@"`[9]. Quest'ultimo, stando a Ramey e Fox (2010, p. 24), «è un parametro speciale che si espande nei parametri posizionali partendo da uno. Quando l'espansione avviene entro i doppî apici, ogni parametro si espande in parole separate. Cioè, `"$@"` è equivalente a `"$1"`, `"$2"`...». Dunque `"$@"` è un vettore contenente i parametri passati allo script e `i` assumerà il contenuto di ognuna delle celle di detto vettore, cioè i nomi dei file da analizzare.

La riga 4, così come la riga 13, deve testare una condizione. Nelle prossime sezioni espliciteremo una gamma di esse.

Per fare in modo che il secondo ciclo inizi dal primo file non ancora analizzato sfruttiamo il contatore (`count`) inizializzato a 0 nella riga 2 e incrementato di 1 ogni volta che troviamo un file negativo nel primo ciclo. Nel secondo ciclo dobbiamo iniziare a valutare dal file successivo all'unico positivo trovato. Purtroppo non è possibile scandire `$@` indicizzandolo come un array, quindi dovremo assegnarne il contenuto a un array (che chiameremo `args` e che sappiamo partire dall'elemento 0, non da 1 come `$@`) e scandire quest'ultimo dalla posizione successiva a `count` fino alla fine.

I lettori più attenti saranno già insorti: se la riga 13 deve verificare una condizione di test per decidere cosa fare dopo, dove sono l'`if` e il `then` presenti nell'analoga riga 4? Risponderemo a questa domanda nella prossima sezione.

3 Caso 1: glifi errati

3.1 Analisi del caso

Il caso dei glifi errati è un caso in cui non sempre è possibile procedere a una correzione indiscriminata perché spesso non possiamo sapere se tale glifo è errato o no se non analizzando il contesto.

Un caso esemplare è l'uso del simbolo dei gradi (°) al posto della 'o' soprasegnata (º).[10] Agli occhi dei non amanti della tipografia i due simboli possono sembrare uguali, ma non lo sono e veicolano due significati diversi: la 'o' soprasegnata indica che il numerale immediatamente precedente dev'essere letto non come cardinale (uno, due, tre...) ma come ordinale maschile (primo, secondo, terzo...); il simbolo dei gradi va letto come "grado" o "gradi" se preceduto rispettivamente da 1 o dagli altri numeri e indica un angolo (normalmente espresso in gradi sessagesimali: un angolo giro vale 360°) o una temperatura se diversamente indicato.[11] In entrambi i casi (o soprasegnata e simbolo di gradi angolari) il simbolo va attaccato al numero che lo precede. Tipograficamente una 'o' soprasegnata mantiene le caratteristiche di forma, spessore e orientazione della 'o' del font in uso (alcuni font

9. Le virgolette servono a salvare la situazione nel caso il nome del file contenga degli spazî: senza le virgolette un unico nome contenente spazî verrebbe suddiviso in una sequenza di tanti nomi di file quanti sono gli spazî più uno. Nel caso questi "nomi" non corrispondano a niente avremo dell'output di `grep` che ci avverte dell'inesistenza del file. Ma se i nomi errati corrispondono a qualche file, l'analizzatore scandaglierà dei file che forse non erano da analizzare.

10. Per amor di *par condicio*, di recente mi è capitato di rileggere un libro (Polidoro, 2012) in cui la o soprasegnata è usata per indicare i gradi!

11. Per indicare una temperatura, il simbolo ° va seguito da una lettera senza alcuno spazio tra i due glifi: C a indicare i gradi Celsius (scala in cui 0 indica la temperatura di congelamento dell'acqua e 100 quella di ebollizione), F per i gradi Fahrenheit (in cui le due temperature precedenti sono, rispettivamente, 32 e 212). Inoltre il simbolo va staccato con uno spazio breve insecabile (`\,`) dal numero che lo precede. Attenzione a un errore comune: la temperatura indicata in gradi Kelvin usa il solo simbolo K, senza °.

le aggiungono una sottolineatura, ma non tutti, e comunque detta sottolineatura non ci sarà mai se usiamo il comando `\textsuperscript{o}` invece di `\textordmasculine`) mentre il simbolo dei gradi ha spessore uniforme e nessuna orientazione (di fatto è una piccola circonferenza; normalmente i manuali come Oetiker *et al.* (2018) consigliano l'uso del comando matematico `\circ` all'esponente o di `\textdegree` del pacchetto `textcomp`. Notiamo che possiamo usare il simbolo di gradi presente sulla tastiera solo usando `textcomp`; diversamente avremo un errore in compilazione con pdfLaTeX). Nella tabella 1 vediamo i glifi ingranditi per meglio apprezzarne le differenze. Mi si potrebbe obiettare che le 'o' dei font Sans Serif sono più facilmente confondibili, ma non necessariamente queste sono disegnate come circonferenze o sono di spessore uniforme (cose che avvengono, per esempio, nel font Futura).

Tabella 1: A sinistra la 'o' soprasegnata (tra parentesi quella ottenuta con comando di soprascrittura); a destra il simbolo dei gradi (tra parentesi quello ottenuto col comando matematico e quello col comando testuale, identico al simbolo di riferimento).

o̲ (o)	° (°, °)

Molti autori, però, ignorano o fingono di ignorare le differenze elencate e mostrate. Pertanto trovano comodo o lecito usare un simbolo presente sulla totalità delle tastiere. Tale simbolo *sembra* proprio essere quello desiderato e non occorre dover impazzire a cercare quello corretto all'interno di sterminate mappe di caratteri.

Entrambi i simboli si trovano alla destra di numeri cardinali (scritti in cifre) e, dunque, solo la lettura del testo ci permette di capire se l'autore intendeva scrivere le cifre in gradi o le abbreviazioni di numeri ordinali. Quindi non è possibile applicare una sostituzione indiscriminata senza la certezza che tutti i casi ricadano in una e una sola delle due possibilità.

3.2 Codice di analisi

È arrivato il momento di svelare il "mistero" dei test introdotto alla pagina 3. Iniziamo proprio dalla ricerca del carattere °. All'interno dell'`if` della riga 4 scriveremo:

```
grep␣--silent␣-E␣°␣"$i"
```

quindi l'intera riga 4 sarà:

```
if␣(␣grep␣--silent␣-E␣°␣"$i"␣);␣then
```

Vediamone il significato, che potrebbe risultare oscuro a qualche lettore, a partire dalle singole componenti:
`if`: comando di bash che esegue un test di verità/falsità. Esegue i comandi seguenti la clausola `then` solo nel caso in cui il test nelle parentesi sia vero o valga 0;
`grep` (Magloire *et al.*, 2017): comando che stampa le righe di un file contenenti una stringa o pattern di riferimento (in pratica cerca la presenza di uno o più termini o di un'espressione regolare in uno o più file). Se tale stringa di riferimento è presente, lo stato di uscita di `grep` sarà 0. L'opzione `--silent` sopprime il normale output del programma (cioè la riga contenente la stringa di riferimento, eventualmente preceduta dal nome del file, che non vogliamo nell'output dello script) e l'eventuale codice 2 emesso in caso di errore (maggiori dettagli su Magloire *et al.* (2017, 12)) mentre l'opzione `-E` permette le espressioni regolari estese. Friedl (2006) dedica diverse pagine alle espressioni regolari di `egrep`. Questo comando è ormai deprecato e sostituito proprio da `grep -E`;
`°`: stringa o pattern di riferimento, in questo caso costituita da un solo carattere;
`"$i"`: contenuto della variabile `i` dentro cui (riga 2 del listato alla pagina 3) troviamo i nomi dei file passati come parametri al comando di analisi.

Il "test" da scrivere nella riga 13, invece, non prevede l'uso di `if` perché procederemo diversamente, usando l'espressività del sistema Unix e dei suoi comandi:

```
grep␣-l␣-E␣°␣"${args[$count]}"␣>>␣report.txt
```

Prima di analizzare per intero il "test", vediamo il significato delle singole parti:
`grep -l -E °`: di `grep` e `-E` già sappiamo; `-l` è l'opzione che sopprime il normale output di `grep` per stampare il nome dei file di input in cui sia stata trovata un'occorrenza della stringa di riferimento;
`"${args[$count]}"`: contenuto della cella `count`-esima dell'array `args` (ne sappiamo lo scopo dalla fine della sezione 2);
`>> report.txt`: ridirezione dell'output su file (qui chiamato `report.txt`) in modalità *append* (con scrittura di seguito a quanto già presente nel file; differisce dalla modalità *write* perché quest'ultima sovrascrive qualunque contenuto).

Alla luce di quanto appena visto, il "test" significa: stampa i nomi di tutti e soli i file specificati in input in cui trovi un'occorrenza della stringa ° e scrivili alla fine del file `report.txt`. Naturalmente non ci sarà alcuna scrittura nel rapporto se `grep` non trova alcuna stringa di riferimento in un file.

Ora abbiamo tutti (o quasi) gli elementi per arricchire lo script di tutti i controlli che riterremo necessarî.

3.3 Altri glifi errati

Due casi in cui l'errore è imputabile più al word processor che all'autore coinvolgono le virgolette "intelligenti". Il primo si ha quando scriviamo la forma abbreviata di un decennio, ad esempio *gli anni '20*. I normali word processor, vedendo che l'apostrofo è stato digitato dopo uno spazio, ritengono trattarsi di una virgoletta aperta e cambiano

glifo, col risultato di avere *gli anni ‘20*. Il secondo caso si ha, invece, quando a una parola terminante con l’apostrofo succede una parola virgolettata, ad esempio *l’“attor giovane”*. Qui il word processor fa il “ragionamento” inverso: le virgolette sono state digitate senza avere uno spazio precedente, quindi devono essere virgolette chiuse. Dunque avremo ottenuto la stringa *l’”attor giovane”*. Ovviamente non possiamo sapere a priori se ogni eventuale occorrenza ricade in quanto già esposto o nei possibili “virgoletta aperta + numero cardinale” e “virgoletta chiusa + virgolette chiuse”. Un controllo manuale dirimerà la questione.

Trovare questi casi è semplice: basta sostituire nel test il carattere `°` con la stringa `‘[0-9]` o con `’’’`. Insomma, sono tutte variazioni sul tema per cui funzionano anche le espressioni regolari.

Altri casi che potremmo voler controllare sono:

- spazî precedenti le interpunzioni. La complessità dell’analisi deriva dal fatto che gli autori possono aver compreso gli spazî in uno stile e nel risultante `.tex` dovremo tener conto delle parentesi graffe: `[␣]}*[␣]*[.,:;]`;
- spazî forzati (`\\\␣` è la stringa di riferimento da dare a `grep`), che il convertitore di un comune word processor inserisce anche nei casi in cui siano stati digitati più spazî consecutivi. Ciò inficia la proprietà di LaTeX di considerare più spazî consecutivi come un singolo spazio;
- trattini (dash, en-dash, em-dash). Qui la stringa di riferimento è più complicata perché gli autori tendono a “mescolare una gran quantità di… stili”. Diciamo che può essere utile iniziare il controllo da `[A-Za-z.,:;␣]-[A-Za-z.,:;␣]` e le sue varianti con en-dash (codice Unicode 2013) e em-dash (codice Unicode 2014). Naturalmente il compito del redattore è verificare che ogni trattino analizzato sia stato usato secondo le norme redazionali.

Lascio al lettore trovare altri casi simili appartenenti alla stessa classe di problemi.

4 Caso 2: scorretta applicazione delle proprietà al testo

Spesso vedo, o mi vengono segnalate dagli editor che rivedono i miei impaginati, porzioni di testo di stile incoerente. Gli utenti di word processor hanno la comodità di poter selezionare una o più parole a cui applicare il grassetto, il corsivo e il sottolineato evidenziandole col mouse e poi premendo un pulsante (il maiuscoletto e l’inclinato sono normalmente di applicazione più farraginosa). Bene!, proprio questa comodità sembra incrementare la disattenzione: trovo sovente parole per metà in corsivo[12] e per metà in tondo e interpunzioni comprese nel corsivo quando dovrebbero essere in tondo. Come possiamo far esaminare questi due casi a `grep`?

Il primo caso, indice di trascuratezza, può essere esaminato abbastanza facilmente cercando quelle occorrenze di testo in cui qualche carattere precede o segue un comando `\text..`[13] senza alcuna interruzione. Questo è un caso (al pari di quelli discussi nella sezione 3.3) in cui l’uso delle espressioni regolari permette sintesi espressiva ed efficienza programmativa. Le stringhe da ricercare sono, rispettivamente:[14]

```
[0-9A-Za-z]\\\text(it|bf|sc|tt|sl)
```

e

```
\\\text(it|bf|sc|tt|sl){[^}]*}[0-9A-Za
    -z]
```

Il secondo caso è abbastanza simile: dobbiamo controllare la presenza di parti di testo con comandi `\text..` contenenti del testo che inizia e/o finisce con un segno di interpunzione. All’interno del `grep` possiamo scrivere le seguenti espressioni regolari entro una coppia di virgolette:

```
\\\text(it|bf|sc|tt|sl){[^}]*[␣
    ]*[.,:;][␣]*}
```

e

```
\\\text(it|bf|sc|tt|sl){[.,:;][␣
    ]*[^}]*}
```

rispettivamente per le interpunzioni alla fine e all’inizio di un `\text..`

Un terzo caso, non rilevabile da un redattore umano che guardi solo il PDF perché non porta conseguenze visibili ai documenti finali, riguarda sempre la scorretta applicazione delle proprietà al testo: una stringa viene resa in uno stile o peso in due o più riprese. Questo si traduce in due o più comandi `\text..` consecutivi, intercalati o no da spazî. Perché dovremmo voler rilevare questi casi? Perché nell’eventuale ebook prodotto a partire da quei sorgenti viene inserito uno spazio in corrispondenza dei due `\text..` consecutivi (anche non intervallati da uno spazio) e può capitare che detto spazio divida una parola, introducendo un refuso. Quindi sarà utile cercare la seguente stringa:

```
\\\text(it|bf|sc|tt|sl){[^}]*}[␣]*\\\
    text(it|bf|sc|tt|sl)
```

12. L’uso del termine corsivo è di comodo. Quanto detto vale per il grassetto e le altre forme elencate.

13. In questo caso il `.` è una *wild card* che indica *qualunque carattere* (dovrei specificare in questa sede “tra quelli leciti”).

14. Ho evitato di comprendere il sottolineato, il cui comando è `\underline`, per questione di inopportunità tipografica.

5 Caso 3: mancanza di stile

Raramente mi sono capitati manoscritti in cui gli autori abbiano usato gli stili. Nel linguaggio degli elaboratori di testo (ma anche degli editor HTML), gli stili sono delle proprietà visuali e strutturali da applicare a una porzione di testo. Per esempio, Intestazione 1, Titolo, Citazione, sono stili reperibili nel menù Stili di LibreOffice Writer e sono gli equivalenti di `\section`,[15] `\title`, `\begin{quote}` `\end{quote}` di LaTeX.

Poiché gli autori sono spesso restii a strutturare il loro testo, l'elemento da ricercare (meglio, di cui ricercare l'assenza) è proprio la traccia della struttura. Dunque in questo caso dovremo fare una ricerca "negata". Ad esempio, visto che normalmente un manoscritto convertito da un word processor sarà convertito come articolo, ci basterà cercare la presenza (o meglio l'assenza) di `\section`.

Come modifichiamo l'`if` della riga 4 per trovare il primo file in cui non sia presente il comando `\section`? È semplice: se il test visto in precedenza doveva testare una condizione di verità (la presenza di un simbolo), ci basta negare quella condizione per testare la falsità (l'assenza di un simbolo o di una stringa di testo):

```
if␣(␣!␣grep␣--silent␣-E␣\\\section␣$i␣
    );␣then
```

La negazione si esplicita col punto esclamativo (`!`).

Per il test successivo, quello della riga 13, sfruttiamo un flag di `grep` diverso da `-l` usato finora: `-L`. Questo flag permette a `grep` di elencare tutti e soli i file, tra quelli analizzati, che *non* contengono la stringa di riferimento:

```
grep␣-L␣-E␣\\\section␣${args[$count]}␣
    >>␣report.txt
```

Certamente un file non strutturato darà più lavoro al redattore rispetto a un file già strutturato anche se quest'ultimo, proprio in virtù di una strutturazione arbitraria da parte del convertitore o incompleta da parte dell'autore, non lo esenterà da un lavoro di "promozione" dei comandi di struttura (`\section`→`\chapter`, `\subsection`→`\section` e così via. Mi raccomando: non nell'ordine inverso. Perché? La risposta alla fine dell'articolo.) o dall'applicazione manuale del testo citato o altro.

6 Caso 4: documenti fintamente plurilingue

Mi capita spesso che i documenti convertiti da un word processor a LaTeX contengano una quantità da enorme a spropositata di `\selectlanguage` e `\foreignlanguage` (col termine "spropositata" intendo decine di `\selectlanguage` e centinaia di `\foreignlanguage` in manoscritti di non più di 50 cartelle). La presenza del primo è giustificabile senz'altro all'inizio del documento per impostare la lingua principale e, sporadicamente, all'interno di un documento in caso ci siano lunghe porzioni di testo di cui specificare la lingua. Un uso indiscriminato, spesso paragrafo per paragrafo nel caso di conversione automatica, non è giustificabile. La presenza del secondo comando, `\foreignlanguage`, è giustificabile in tutti i casi di porzioni di testo, possibilmente brevi, scritte in una lingua diversa da quella principale.

Una presenza spropositata dei due comandi in un manoscritto convertito da un word processor può indicare che l'autore non abbia etichettato correttamente il testo in base alla lingua, o che abbia lasciato la lingua preimpostata per redigere un manoscritto in un'altra lingua. Naturalmente il convertitore non può conoscere le intenzioni dell'autore e dunque si limiterà a porre quei comandi nei punti esatti in cui ne ravvisa la necessità. Facciamo un esempio. Supponiamo che l'autore abbia scritto il testo della figura 1 evidenziandolo allo stesso modo. Quando selezionerà l'inglese, questo verrà applicato a tutto il titolo evidenziato.

Allo stesso modo di come molti autori mettono i grassetti e i corsivi, anche mettere le lingue col mouse comporta degli inconvenienti. Volendo applicare l'inglese allo stesso testo, ma evidenziato come nella figura 2 comporterà impostarlo per ognuno dei due pezzi evidenziati; il risultato della conversione sarà avere due `\foreignlanguage` consecutivi anziché uno.

Quello appena discusso sarebbe un caso già ottimale o subottimale. Purtroppo la maggior parte delle volte gli esiti delle conversioni sono ben peggiori, con lingue impostate con qualche criterio oscuro e sbagliate (ignoro il motivo per cui trovai impostato il polacco per contrassegnare dei titoli di libri in francese e inglese...)

Comunque, basta fare un paio di ricerche sulle sole stringhe `selectlanguage` e `foreignlanguage`. Non è detto che la presenza di tali comandi in un documento sia indice di errore e quindi starà al redattore analizzare se e cosa correggere.

7 Il test finale

L'algoritmo bash-like così come scritto alla pagina 3 non è granché utile se tradotto pedissequamente nel programma: andrebbe ripetuto e adattato per ognuno dei casi da analizzare. Ciò sarebbe uno spreco di memoria (limitato, ma spreco), di tempo di digitazione (pure al netto del copia e incolla) e, soprattutto, di manutenzione (immaginate di aver scritto male un test dal punto di vista semantico *prima* del copia e incolla; lo script è stato fatto per analizzare centinaia di casi e dovete correggere il test per tutte le centinaia di casi). Possiamo trovare un automatismo che ci permetta di usare lo stesso codice per tutti i casi simili, un po' come le

15. Sembra che il convertitore di Writer non contempli altro tipo di documento che non sia un articolo, da lì l'equivalenza Intestazione 1-`\section`.

... nel loro fondamentale *A Programming Approach to Computability* gli autori...

Figura 1: L'autore selezionerà l'inglese per il testo evidenziato.

... nel loro fondamentale *A Programming* *Approach to Computability* gli autori...

Figura 2: L'autore dovrà selezionare l'inglese per ognuna delle porzioni di testo evidenziate.

funzioni del linguaggio C? Sì, esiste: basta sfruttare gli array come già fatto per i parametri di input.

Le figure 3 e 4 mostrano l'intero codice dello script di analisi (`editanalyze`), suddiviso per motivi di spazio ma in maniera significativa: una figura contiene la parte "dichiarativa", l'altra quella "imperativa".[16] Nel codice mostrato nella figura 3 riempiamo una per una le celle di sei array. Questi sei vettori contengono il pattern di ricerca (`stringa`), il testo di output a video (`caso`) e il testo da scrivere nel rapporto (`testo`) per il test di positività e gli analoghi per il test di negatività (`stringan`, `cason`, `teston`). Tali array, espandibili man mano che ci si presentano nuovi casi da voler includere nell'analisi, ci permettono di parametrizzare gli elementi variabili nel codice, che non dovremo più duplicare per ogni caso da analizzare. Per ognuno degli array scriveremo una variabile col valore delle celle riempite, così da non dover cambiare valori all'interno del ciclo che testa in sequenza tutti i casi noti (trattiamo una variabile come fosse una costante).

Sempre nella figura 4, dopo il test sugli argomenti, notiamo che viene cancellata un'eventuale vecchia versione di `report.txt`, quindi ci sono due blocchi analitici: quello per tutti i casi di positività rispetto a un pattern e quello per verificare l'unico caso studiato in cui ci interessa che il pattern sia assente. Avremmo potuto accorpare in un unico codice i due casi? In definitiva si tratta di aggiungere o togliere un punto esclamativo a un test e sostituire un `-l` con un `-L` o viceversa. La cosa sarebbe fattibile in diversi modi: mi viene in mente di usare `sed` e su Unix & Linux Stack Exchange suggeriscono, in aggiunta, di usare uno script Perl per modificare quanto necessario o di sostituire lo script con uno script esterno. Ma perché vogliamo complicarci la vita, complicando anche quella del sistema operativo (quelli moderni sono progettati per evitare programmi con codice automodificante perché fonte di potenziali problemi di sicurezza e di determinismo esecutivo oltre che, aggiungerei, di leggibilità del codice), solo per risparmiare 1 KB di codice e provare a imitare i puntatori a funzione del C? Personalmente non mi avventurerò per questa via.

Passiamo al test. Abbiamo costruito quattro file contenenti ognuno alcuni casi tra quelli elencati nell'articolo. Il loro contenuto è mostrato nella tabella 2.

La figura 5 riporta il contenuto del rapporto generato dall'esecuzione dello script sui quattro file (`editanalyzer *tex`).

16. Le virgolette ai termini dichiarativa e imperativa indicano un significato forzato. Bash, a differenza di linguaggi come il C, non distingue le due fasi e la suddivisione data qui è più formale che sostanziale.

8 Conclusioni

Il lavoro del redattore può essere molto pesante quando arriva il momento di controllare tutte quelle piccole minuzie relative alle più elementari regole tipografiche. Uno strumento automatico di analisi delle minuzie può essere di grande aiuto, sia dal punto di vista dell'esaustività, sia da quello della velocità.

Questo articolo ha voluto fornire il suo contributo alla spiegazione degli errori commessi dagli autori, chiarendone la semantica e i motivi quando opportuno, e alla costruzione di uno strumento automatico di controllo.

Quest'ultimo punto si concretizza in uno script bash che permette di controllare uno o più file di testo alla ricerca di una serie di errori tipici e che genera un rapporto descrittivo della corrispondenza file-caso analizzato. Starà poi al redattore determinare quali occorrenze dovranno essere corrette e come correggerle.

Infine, questo lavoro potrebbe essere considerato un primo passo verso il controllo esaustivo automatico della corretta applicazione delle norme tipografiche in un manoscritto.

Ringraziamenti

L'obbligatorio ringraziamento va al Consiglio Scientifico del GUIT per aver valutato positivamente il lavoro e per i consigli datimi. Ma il ringraziamento grande va all'ottimo redattore che mi ha fatto notare alcune imprecisioni, dimenticanze e lungaggini nel testo e alcune dimenticanze nel codice del programma. La sua osservazione — in puro stile Pierre de Fermat — sulla dubbia efficienza dello script porterà in un prossimo futuro all'eventuale adozione di comandi alternativi a `grep` in base al risultato di test sulla velocità di esecuzione su una serie di testi di diverse dimensioni.

Risposta al quesito proposto alla pagina 6

Se iniziamo la promozione dal livello più basso, dunque sostituendo `\subparagraph` con `\paragraph`,

```
#! /bin/bash

# Parte da modificare in base ai casi da analizzare
# Array dei pattern di ricerca "positiva" e dei messaggi per l'utente
stringa[1]="°"
caso[1]="Analisi della presenza del simbolo °..."
testo[1]="\nIl carattere ° si trova in"
stringa[2]="‘[0-9]"
caso[2]="Analisi della presenza dell'apostrofo sbagliato prima degli anni..."
testo[2]="\nL'apostrofo sbagliato prima degli anni si trova in"
stringa[3]="’’’"
caso[3]="Analisi della presenza della sequenza ’’’"
testo[3]="\nLa sequenza ’’’ si trova in"
stringa[4]="[ ]}*[ ]*[.,:;]"
caso[4]="Analisi della presenza di spazi prima delle interpunzioni..."
testo[4]="\nSpazi prima delle interpunzioni si trovano in"
stringa[5]="\\\ "
caso[5]="Analisi della presenza di spazi forzati..."
testo[5]="\nSpazi forzati si trovano in"
stringa[6]="[0-9A-Za-z.,:; ]-[0-9A-Za-z.,:; ]"
caso[6]="Analisi della presenza di trattini brevi..."
testo[6]="\nTrattini brevi si trovano in"
stringa[7]="[0-9A-Za-z.,:; ]-[0-9A-Za-z.,:; ]"
caso[7]="Analisi della presenza di trattini medi..."
testo[7]="\nTrattini medi si trovano in"
stringa[8]="[0-9A-Za-z.,:; ]-[0-9A-Za-z.,:; ]"
caso[8]="Analisi della presenza di trattini lunghi..."
testo[8]="\nTrattini lunghi si trovano in"
stringa[9]="[0-9A-Za-z]\\\text(it|bf|sc|tt|sl)"
caso[9]="Analisi della presenza di lettere o numeri prefissi a un comando \textxx..."
testo[9]="\nLettere o numeri prefissi a un comando \\\textxx si trovano in"
stringa[10]="\\\text(it|bf|sc|tt|sl){[^}]*}[0-9A-Za-z]"
caso[10]="Analisi della presenza di lettere o numeri postfissi a un comando \textxx..."
testo[10]="\nLettere o numeri postfissi a un comando \\\textxx si trovano in"
stringa[11]="\\\text(it|bf|sc|tt|sl){[^}]*[ ]*[.,:;][ ]*}"
caso[11]="Analisi della presenza di interpunzioni alla fine di un comando \textxx..."
testo[11]="\nInterpunzioni alla fine di un comando \\\textxx si trovano in"
stringa[12]="\\\text(it|bf|sc|tt|sl){[.,:;][ ]*[^}]*}"
caso[12]="Analisi della presenza di interpunzioni all'inizio di un comando \textxx..."
testo[12]="\nInterpunzioni all'inizio di un comando \\\textxx si trovano in"
stringa[13]="\text(it|bf|sc|tt|sl){[^}]*}[ ]*\\\text(it|bf|sc|tt|sl)"
caso[13]="Analisi della presenza di comandi \textxx consecutivi..."
testo[13]="\nComandi \\\textxx consecutivi si trovano in"
stringa[14]="selectlanguage"
caso[14]="Analisi della presenza di comandi \selectlanguage..."
testo[14]="\n\selectlanguage si trova in"
stringa[15]="foreignlanguage"
caso[15]="Analisi della presenza di comandi \foreignlanguage..."
testo[15]="\n\\\foreignlanguage si trova in"
casip=15

# Array dei pattern di ricerca "negativa" e dei messaggi per l'utente
stringan[1]="\\\section"
cason[1]="Analisi dell'assenza di \section..."
teston[1]="\nIl comando \section non si trova in"
casin=1
# Fine parte da modificare
```

Figura 3: Codice dello script di ausilio ai redattori.

```
if [[ $BASH_ARGC < 1 ]]; then
  echo "Uso: editanalyze <file\
 da analizzare>"
  echo "Es.: editanalyze *.tex\
 (controlla tutti i file con\
 estensione .tex)"
  echo "     editanalyze\
 capitolo1.tex (controlla\
 il solo file capitolo1.tex)"
  echo "     editanalyze\
 capitolo[1-5].tex (controlla\
 i file capitolo1-capitolo5.tex)"
  exit 1
fi

rm report.txt

# Analisi dei casi positivi (presenza di un pattern nei file)
for (( n=1 ; n<=$casip; n=n+1 )); do
  echo ${caso[$n]}
  echo "Pattern di ricerca: " ${stringa[$n]}
  count=0
  for i in "$@"; do
    if ( grep --silent -E "${stringa[$n]}" "$i" ); then
      echo -e ${testo[$n]} >> report.txt
      echo "$i" >> report.txt
      break
    fi
    count=$[$count+1]
  done
  args=("$@")
  for (( count=$[$count+1]; $count<$BASH_ARGC; count=$[$count+1] )); do
    grep -l -E "${stringa[$n]}" "${args[$count]}" >> report.txt
  done
done

# Analisi dei casi negativi (assenza di un pattern nei file)
for (( n=1 ; n<=$casin; n=n+1 )); do
  echo ${cason[$n]}
  echo "Pattern di ricerca: " ${stringan[$n]}
  count=0
  for i in "$@"; do
    if ( ! grep --silent -E "${stringan[$n]}" "$i" ); then
      echo -e ${teston[$n]} >> report.txt
      echo "$i" >> report.txt
      break
    fi
    count=$[$count+1]
  done
  args=("$@")
  for (( count=$[$count+1]; $count<$BASH_ARGC; count=$[$count+1] )); do
    grep -L -E "${stringan[$n]}" "${args[$count]}" >> report.txt
  done
done

exit 0
```

Figura 4: Seguito del codice dello script di ausilio ai redattori.

```
Il carattere ° si trova in
1.tex
3.tex

L'apostrofo sbagliato prima degli anni si trova in
2.tex
3.tex

La sequenza ''' si trova in
1.tex
3.tex

Spazi prima delle interpunzioni si trovano in
2.tex
3.tex

Spazi forzati si trovano in
1.tex
3.tex

Trattini brevi si trovano in
2.tex
3.tex

Trattini medi si trovano in
1.tex
3.tex

Trattini lunghi si trovano in
2.tex
3.tex

Lettere o numeri prefissi a un comando \textxx si trovano in
1.tex
4.tex

Lettere o numeri postfissi a un comando \textxx si trovano in
2.tex
4.tex

Interpunzioni alla fine di un comando \textxx si trovano in
1.tex
4.tex

Interpunzioni all'inizio di un comando \textxx si trovano in
2.tex
4.tex

Comandi \textxx consecutivi si trovano in
1.tex
4.tex

\selectlanguage si trova in
2.tex
4.tex

\foreignlanguage si trova in
1.tex
4.tex

Il comando \section non si trova in
1.tex
3.tex
```

Figura 5: Contenuto del rapporto sull'analisi dei quattro file riportati nella tabella 2.

Tabella 2: Contenuto dei quattro file testuali usati per provare lo script.

1.tex	2.tex
10°	‘20
’’’	abc ,
\	10-20
10-20	30-40
a\textit{abd}	\textit{abc}0
\textbf{abc,}	\textit{, abc}
\textsc{abc}\textsc{def}	\selectlanguage{italian}
\foreignlanguage{italian}{ciao}	\section{}
3.tex	**4.tex**
50°	T\textit{he fog}
‘70	\textit{Essi vivo}no
l’’’etica	\textsc{John Carpenter,}
\textit{allorquando },	\textsc{, John Carpenter}
sono \ qui	\textit{La} \textit{Cosa}
quando -travolti -	\selectlanguage{english}
quando - travolti -	\foreignlanguage{french}{aussi}
quando-travolti-	\section{}

Per qualche “oscuro” motivo, en-dash (–) e em-dash (—) vengono mostrati come dei normali dash (-) nel carattere monospaziato. Tecnicamente è chiaro che i tre diversi caratteri sono stati disegnati allo stesso modo e con le stesse dimensioni per rispettare la principale proprietà dei caratteri monospaziati, cioè l’uniforme dimensione orizzontale.

come potremo distinguere i `\paragraph` originali da promuovere a `\subsubsection` da quelli appena promossi da `\subparagraph`?

Riferimenti bibliografici

Bertossi, Alan. A. (1990). *Strutture Algoritmi Complessità*. ECIG (Edizioni Culturali Internazionali Genova), Genova.

Bruessow, Christian V.J. (2017). «win-bash - bash port for windows». `http://win-bash.sourceforge.net/`.

Cygnus Solution-Red Hat (2019). «Cygwin». `http://www.cygwin.com/`.

Friedl, Jeffrey E.F. (2006). *Mastering Regular Expressions*. O’Reilly, Sebastopol, 3ª edizione.

Goyvaerts, Jan (2019). «Regular-Expression.info - Regex Tutorial, Examples and Reference-Regexp Patterns». `https://www.regular-expressions.info/`.

Magloire *et al.*, Alain (2017). *GNU Grep: Print lines matching a pattern*. `http://www.gnu.org/software/grep/manual/`.

Oetiker, Tobias, Hubert Partl, Irene Hyna e Elisabeth Schlegl (2018). *The Not So Short Introduction to LaTeX 2ε*. Accessibile da terminale con `texdoc lshort`.

Polidoro, Massimo (2012). *Il sesto senso*. Gruner+Jahr/Mondadori, Milano.

Powers, Shelley, Jerry Peek, Tim O’Reilly e Mike Loukides (2002). *Unix Power Tools*. Sebastopol, CA.

Ramey, Chet e Brian Fox (2010). *Bash Reference Manual*. Boston, MA.

Rawlinson, Graham Ernest (1976). *The Significance of Letter Position in Word Recognition*. Tesi di Dottorato, University of Nottingham.

Simons, Daniel J. e Christopher F. Chabris (1999). «Gorillas in our midst: Sustained inattentional blindness for dynamic events». *Perception*, **28** (9), pp. 1059–1074. `https://doi.org/10.1068/p281059`.

Simons, Daniel J. e Daniel T. Levin (1998). «Failure to detect changes to people during a real-world interaction». *Psychonomic Bulletin & Review*, **5** (4), pp. 644–649. `https://doi.org/10.3758/BF03208840`.

Wikipedia (2019). «Shebang (Unix)». `https://en.wikipedia.org/wiki/Shebang_(Unix)`.

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