Ricordi di un cacciatore di spazi spuri

Enrico Gregorio

Sommario

Parecchi trabocchetti ci attendono quando programmiamo in (LA)TEX. Esamineremo alcuni di essi in cerca di una soluzione. Troveremo la terra promessa? Forse, con `expl3`.

Abstract

Several pitfalls are waiting for us when programming in (LA)TEX. This paper will examine some and search for a solution. Shall we find the promised land? Maybe so, with `expl3`.

1 Introduzione

Di per sé la programmazione in (LA)TEX non è facile, principalmente per le idiosincrasie del linguaggio, ma anche per via di alcune sottigliezze dovute al fatto che il linguaggio è orientato alla *composizione tipografica* del testo. Knuth ha fatto del suo meglio per facilitare la scrittura di un documento senza preoccuparsi troppo degli spazi: in circostanze normali serie di spazi o di tabulazioni sono considerate come uno spazio singolo; la fine riga è convertita in uno spazio, purché non sia seguita da un'altra fine riga separata dalla precedente da soli spazi; lo spazio bianco all'inizio della riga è ignorato; gli spazi sono rimossi dalla fine di una riga e sostituiti da uno speciale marcatore.

Le regole precise sono spiegate nei dettagli nel *TEXbook* (Knuth, 1986) e in *TEX by Topic* (Eijkhout, 1992), questo articolo non è il posto dove entrare nelle minuzie. Ritornerò su alcuni dei punti descritti prima.

Alcuni dei lettori potrebbero conoscermi per la mia attività su TeX.StackExchange (come utente `egreg`) dove, secondo il parere di uno dei più stimati membri della comunità, ho accumulato gran parte della mia reputazione scovando spazi spuri in codice TEX.[1] A dire il vero, scovare spazi spuri è talvolta piuttosto complicato: possono nascondersi in angoli bui e può essere necessario ricorrere a `\tracingall` o altre armi pesanti per isolarli: le macro di TEX si richiamano a vicenda in modi talvolta molto contorti.

Nessuno che abbia intrapreso il compito di scrivere macro, semplici o anche terribilmente complicate, è mai stato immune dal morso di qualche spazio spurio lasciato nel codice. Succede! Una delle ragioni più frequenti è la risistemazione del codice: una riga è troppo lunga e vorremmo migliorare la leggibilità del codice, così la spezziamo e dimentichiamo il magico `%` a fine riga. Un altro aspetto importante del linguaggio di TEX è che in certe situazioni gli spazi sono *ignorati*: ci sono e TEX si comporta come se non ci fossero, ma eseguendo un'azione particolare che descriverò più avanti.

Dobbiamo distinguere con attenzione tra *battute spazio* e *token spazio*: solo le *battute spazio* (quando si spinge il pollice sulla barra spaziatrice scrivendo il documento) sono soggette alla regola di contrazione descritta prima, mentre i *token spazio* non lo sono. Un semplice esempio: `\space\space` produrrà *due spazi*, perché quando TEX espande `\space`, lo converte in un token spazio.

Questo articolo descriverà gli errori più comuni, con esempi presi da domande su TeX.StackExchange o da codice di pacchetti. Non darò riferimenti precisi, perché lo scopo non è di svergognarne gli autori.[2] Finirò con qualche considerazione sui metodi per evitare questi problemini.

L'articolo è una traduzione di "Recollections of a spurious space catcher" che apparirà negli atti del TUG meeting 2015, *TUGboat* 35 (2015).

2 Primi esempi

Un famoso *spaghetti western* è *Il buono, il brutto e il cattivo* di Sergio Leone, con Clint Eastwood, Lee Van Cleef e Eli Wallach. Mi piace presentare gli esempi in questa forma e così farò, per cominciare.

2.1 Il brutto

```
\providecommand{\sVert}[1][0]{
\ensuremath{\mathinner{
\ifthenelse{\equal{#1}{0}}{ % if
\rvert}{}
\ifthenelse{\equal{#1}{1}}{ % if
\bigr\rvert}{}
\ifthenelse{\equal{#1}{2}}{ % if
\Bigr\rvert}{}
\ifthenelse{\equal{#1}{3}}{ % if
\biggr\rvert}{}
\ifthenelse{\equal{#1}{4}}{ % if
\Biggr\rvert}{}
}} % \ensuremath{\mathinner{
}
```

Questo codice fa parte di un pacchetto nel quale le definizioni non sono separate da righe vuote e che è brutto sotto molti aspetti: non c'è alcun rientro che possa chiarire le relazioni fra le varie

1. Ci piace scherzare nella *chatroom* del sito; non sono però del tutto sicuro che quella osservazione sia uno scherzo.

2. Magari farò un paio di eccezioni.

```
\cref@addlanguagedefs{spanish}{%
 \PackageInfo{cleveref}{loaded 'spanish' language definitions}
 \renewcommand{\crefrangeconjunction}{ a\nobreakspace}%
 \renewcommand{\crefrangepreconjunction}{}%
 \renewcommand{\crefrangepostconjunction}{}%
 \renewcommand{\crefpairconjunction}{ y\nobreakspace}%
 [...]
```

Tabella 1: Codice da una vecchia versione di `cleveref.sty`

parti e l'annidamento dei condizionali; il codice è goffo e trascura molte protezioni a fine riga.

Ricordo che una fine riga è convertito in un token spazio; in questo esempio particolare non sono particolarmente importanti, perché la macro deve essere usata in modo matematico, nel quale i token spazio sono ignorati. Ma un utente *potrebbe* adoperare `\sVert` in modo testo per via di `\ensuremath`, finendo con spazi in eccesso: in questo caso la prima fine riga e lo spazio tra `}}` e `%` alla fine *non* sono ignorati.

L'uso di `\providecommand` è ovviamente sbagliato: un utente che carichi il pacchetto si aspetta che il comando `\sVert` esegua l'azione descritta, non qualche altra. Una definizione migliore può essere

```
\newcommand*{\sVert}[1][0]{%
  \ifcase#1\relax
    \rvert % 0
  \or
    \bigr\rvert % 1
  \or
    \Bigr\rvert % 2
  \or
    \biggr\rvert % 3
  \or
    \Biggr\rvert % 4
  \fi
}
```

Considero cattivo stile di programmazione avere `%` accanto a una parola di controllo, ma è solo la mia opinione. I caratteri `%` dopo `\rvert` non sarebbero necessari se non ci fossero i commenti.

2.2 Il cattivo

```
\def\@wrqbar#1{%
\ifnum\value{page}<10\def\X{\string\X}\else%
\ifnum\value{page}<100\def\X{\string\Y}\else%
\def\X{\string\Z}\fi\fi%
\def\F{\string\F}\def\E{\string\E}%
\stepcounter{arts}%
\iffootnote%
\edef\@tempa{\write\@barfile{\string%
\quellentry{#1\X{}{\thepage}}{\F{}{\thefootnote}}}}%
\else%
\edef\@tempa{\write\@barfile{\string%
\quellentry{#1\X{}{\thepage}}{\E{}{\thearts}}}}%
\fi%
\expandafter\endgroup\@tempa%
\if@nobreak \ifvmode\nobreak\fi\fi\@esphack}
```

Qui di sicuro non troveremo spazi spuri! Ci sono perfino troppi caratteri `%`! Il problema è che è impossibile leggere il codice, tanto meno comprenderlo.

2.3 Il buono

```
\def\deleterightmost#1{%
  \edef#1{\expandafter\xyzzy#1\xyzzy}}
\long\def\xyzzy\\#1#2{\ifx#2\xyzzy\yzzyx
  \else\noexpand\\{#1}\fi\xyzzy#2}
\long\def\yzzyx#1\xyzzy\xyzzy{\fi}
```

Questo codice del Grand Wizard può essere preso come modello di chiarezza. Non è altrettanto chiaro che cosa faccia: per capirlo è necessario leggersi l'appendice D del *TEXbook*.

2.4 Sorpresa!

```
{\tt A'C B}

\def\adef#1{\catcode`#1=13 \begingroup
 \lccode`\~=`#1\lowercase{\endgroup\def~}}
\let\oldtt\tt\def\tt{\adef'{\char"0D}\oldtt}

{\tt A'C B}

{\tt A'c B}
```

Fate andare plain TEX su questo codice (aggiungendo `\bye` alla fine) e ne riceverete una bella sorpresa. Lo scopo è di sostituire l'apostrofo curvo con quello diritto che il font `cmtt10` ha alla posizione `"0D`. La sorpresa è che viene stampato solo `AB` senza alcuno spazio tra le lettere, né apostrofo, né `C`. Però troviamo un avviso misterioso nel log:

```
Missing character:
There is no ^^dc in font cmtt10!
```

Questo esempio non ha spazi in più, invece ne *manca* uno!

2.5 I grandi programmatori non sono immuni

La tabella 1 mostra un estratto da `cleveref.sty`, che è certamente un software notevole.

Forse l'autore aveva provato il supporto per le lingue solo in alcuni casi: lo spazio spurio si notava solo quando un utente provava qualcosa come

```
The Spanish word for Spain is
\foreignlanguage{spanish}{Espa\~na}
```

e riscontrava la presenza di *due spazi* tra 'is' e 'España'. Isolare il problema non fu affatto facile,

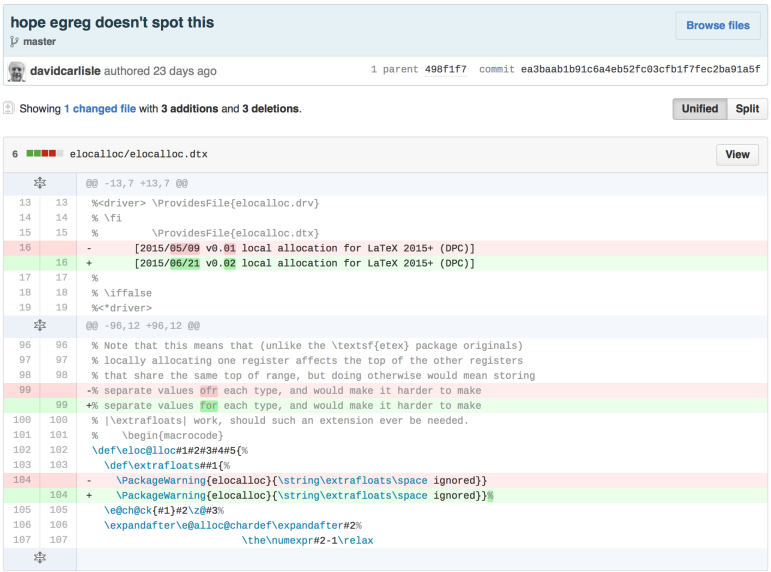

FIGURA 1: Un altro esempio che anche grandi TEXperts non sono immuni

perché cleveref non sembrava nemmeno coinvolto. La macro `\cref@addlanguagedefs` aggiunge codice a `\extrasspanish`, che viene eseguita ogni volta che la lingua passa allo spagnolo: a ogni cambio di lingua veniva prodotto uno spazio.

Si veda la figura 1 per un altro esempio.

3 Che succede?

La regola è semplice: una fine riga è convertita in una *battuta spazio* che *può* diventare un token spazio, ma non se segue una parola di controllo. Quindi, nella riga

```
\iffootnote%
```

il carattere di commento non è necessario perché la battuta spazio che risulta dalla fine riga sarà ignorata, proprio perché segue una parola di controllo. Al contrario la fine riga in

```
\PackageInfo{cleveref}{loaded `spanish'
  language definitions}
```

diventa un normale token spazio nel testo di sostituzione della macro che si sta definendo e sparirà solo se TEX è di buon umore quando la macro viene usata.

TEX è di buon umore nei confronti dei token spazio quando il modo corrente è verticale (tra capoversi, essenzialmente) o matematico: in questi casi i token spazio non fanno nulla. Se TEX sta componendo un normale capoverso (o una box orizzontale), i token spazio non sono di solito ignorati e questa è la causa del misterioso effetto presentato nella sezione 2.5.

Ecco alcuni esempi di codice scritto da persone probabilmente abituate ad altri linguaggi di programmazione, in cui gli spazi sono usati molto più liberamente per separare i token e sono per lo più irrilevanti (purché non in una stringa).

```
\newcommand{\smallx}[1]{
     \begin{center}
        \begin{Verbatim}[commandchars=\\\{\}]

            \code{#1}

        \end{Verbatim}
    \end{center}
 }
============================
\include{fp}
\newcommand\entryOne[1]{
\ifnum #1 = 100 #1 \fi
}
\newcommand\entryTwo[1]{
\FPeval{\result}{#1}
\ifnum \result = 100 \result \fi
}
============================
\newcommand{\trellis}[4]{
  \def \STATES {#1}
  \def \PSK {#2}
  \def \XDISTANCE {#3}
  \def \YDISTANCE {#4}
  \FPupn\NGROUPS{\STATES{} \PSK{} div 0 trunc}
  \multido{\ryA=0+-\YDISTANCE,\nA=1+1}{\STATES}{%
    \dotnode(0,\ryA){dotA\nA}
    \dotnode(\XDISTANCE,\ryA){dotB\nA}
  }
  \multido{\nG=1+1,\nOffset=1+\PSK}{\NGROUPS}{%
    \multido{\nStart=\nG+\NGROUPS}{\PSK}{%
      \multido{\nArrows=\nOffset+1}{\PSK}{%
        \ncline{dotA\nStart}{dotB\nArrows}
      }
    }
  }
}
```

Gli autori dei primi due esempi non sanno nulla di spazi spuri e scrivono il codice TEX come se fosse C. Il terzo esempio mescola le protezioni a

SENATVSPOPVLVSQVEROMANVS
IMPCAESARIDIVINERVAEFNERVAE
TRAIANOAVGGERMDACICOPONTIF
MAXIMOTRIBPOTXVIIIMPVICOSVIPP
ADDECLARANDVMQVANTAEALTITVDINIS
MONSETLOCVSTANTISOPERIBVSSITEGESTVS

FIGURA 2: L'iscrizione sul basamento della *Columna Traiana* a Roma

fine riga con *free form code*. Contare il numero degli spazi spuri è divertente.

Sfortunatamente TEX non è *free form*! Gli spazi sono importanti nella composizione, a meno che non vogliamo scrivere i nostri testi come facevano gli antichi romani: nella figura 2 possiamo vedere un'emulazione dell'iscrizione sul basamento della *Columna Traiana* a Roma.[3] Il font è Trajan di Peter Wilson.

Gli antichi avevano buone ragioni per non usare spazi: i suffissi facilitavano la divisione in parole, ma forse più importante era risparmiare spazio: il marmo e la pergamena erano molto costosi. Solo l'introduzione della stampa e di carta più economica permise di usare spazi tra le parole per maggiore chiarezza e facilità di lettura.

Bene, uno potrebbe dire, perché non mettiamo `%` alla fine di ogni riga nel codice di programmazione e smettiamo di preoccuparci, anche se il codice sarà un po' più arduo da leggere?

Spiacente, non funziona. Ecco un altro esempio che lo dimostra: ci sono due malattie che chiamo 'sindrome da spazio spurio' e 'sindrome da spazio mancante'. La seconda è più grave.

```
\documentclass{article}
\newcount\monthlycount
\newcommand{\monthlytodo}[1]{\par%
  \fbox{%
    \parbox{10cm}{%
      \monthlycount=1%
      \loop\ifnum\monthlycount<13%
        #1--\number\monthlycount\hrulefill\par%
        \advance\monthlycount by 1%
      \repeat%
    }%
  }%
}
\begin{document}
\monthlytodo{2013}
\end{document}
```

Questo codice dovrebbe servire a stampare una lista numerata e incorniciata, con gli elementi della lista preceduti da anno e numero del mese. Se proviamo a compilarlo, dopo un bel po' di tempo senza segni di attività, TEX si fermerà con il messaggio

```
! TeX capacity exceeded, sorry
[main memory size=5000000].
<to be read again>
                   -
l.15 \monthlytodo{2013}
```

Sorprendente, no? L'autore del codice era stato, in precedenza, vittima della sindrome da spazio spurio, quindi cominciò ad aggiungere `%` alla fine di ogni riga, perfino nel documento stesso, non solo nel codice del preambolo.

Vediamo che succede: il testo di sostituzione della macro verrebbe mostrato da TEX come

```
... \loop \ifnum \monthlycount
  <13#1--\number \monthlycount ...
```

e, quando `\monthlytodo{2013}` è chiamato, il test diventa

```
... \loop \ifnum \monthlycount
  <132013--\number \monthlycount ...
```

Non ci meravigliamo più che TEX ci metta tanto a finire il ciclo e che esaurisca la memoria, perché sta tentando di costruire una `\fbox`.

Un caso conclamato di sindrome da spazio mancante. La soluzione è di *non* avere `%` dopo `13`. Questa costante è parte di un test numerico e quindi siamo in uno dei casi speciali in cui TEX *ignora* un token spazio che la segua.

Il codice della sezione 2.4 soffre della stessa sindrome; quando `{\tt A'C B}` è sviluppato, il carattere apostrofo diventa attivo ed espanso come una macro, producendo la lista di token

```
A\char"0DC B
```

ed è qui che va tutto a pallino: `\char` si aspetta un numero in formato esadecimale per via di `"` e trova le cifre `DC`; dal momento che in quella posizione del font non c'è alcun carattere, compare l'avviso di `Missing character ^^dc|`. Il codice corretto deve avere uno spazio oppure `\relax`:

```
\let\oldtt\tt
\def\tt{\adef'{\char"0D }\oldtt}

\let\oldtt\tt
\def\tt{\adef'{\char"0D\relax}\oldtt}
```

Quale dei due è una decisione stilistica: sia lo spazio sia `\relax` terminano la ricerca di altre cifre; `\relax` non fa nulla mentre lo spazio è ignorato. C'è un'altra possibilità:

```
\let\oldtt\tt\def\tt{\adef'{\active}\oldtt}
```

ma qui si adopera il fatto accidentale che `"0D` = 13 and che `\active` è definita con `\chardef` per puntare al carattere 13: non un esempio di buona programmazione.

Un esempio più perfido, scoperto da Frank Mittelbach qualche giorno dopo il convegno del TUG.[4]

3. Mentre scrivevo l'articolo, sono stato vittima di uno spazio spurio che si è inserito nel testo dell'iscrizione, ma fortunatamente l'ho notato prima di inviarlo.

4. http://tex.stackexchange.com/questions/257100/varioref-and-previous-page

```
2337      \advance\@tempcnta-2%
2338      \ifnum \thevpagerefnum =\@tempcnta%
```

Ricordate lo spazio spurio in `cleveref.sty` discusso nella sezione 2.5? Dopo che segnalai il baco, l'autore aggiunse `%` alla fine di ogni riga. Ma il risultato è un grave caso di sindrome da spazio mancante: quando TEX si aspetta una costante numerica, guarda se trova un token spazio o un token non espandibile che non possa essere interpretato come cifra. Per questa ricerca, i token vengono *espansi*, in particolare il condizionale è valutato prima che un valore sia assegnato alla variabile `\@tempcnta` perché TEX ancora non sa se il numero è terminato. Il risultato è che il riferimento non sarà corretto. Il codice è parte della ridefinizione di un comando in `varioref.sty`; il codice originale (quasi identico) non ha `%` dopo `-2`.

La regola precisa è che quando un costrutto sintattico permette ⟨*one optional space*⟩, TEX lo cercherà eseguendo l'espansione. Cito il *TEXbook* (Knuth, 1986, p. 208)

> Per un risultato ottimale, mettete sempre uno spazio bianco dopo una costante numerica; questo spazio bianco dice a TEX che la costante è completa e questo spazio non apparirà in stampa. Di fatto, se non mettete lo spazio bianco dopo una costante, TEX deve fare più lavoro, perché una costante continua fino a che una non-cifra viene letta; se questa non-cifra non è uno spazio, TEX la rimette al suo posto, pronta per essere letta di nuovo.

Si potrebbe obiettare che chi usa `\loop` dovrebbe conoscere i dettagli della programmazione TEX, dal momento che questa macro non è ufficialmente supportata da LATEX e quindi il problema dell'esaurimento della memoria non si presenterebbe. Una stima grossolana del numero di pacchetti che usano `\loop` è 378. Non ho nemmeno provato a controllare quanti usino `\ifnum`: entrambi sono essenziali nella cassetta degli attrezzi di qualsiasi programmatore (LA)TEX.

È evidente che un aspirante autore di macro dovrebbe essere a conoscenza di questi problemi, ma sono molto sottili e, come abbiamo visto, anche programmatori molto esperti possono cadere vittima della grave sindrome. Potrei dare molti altri esempi.

4 Soluzioni

Una via d'uscita dal problema delle sindromi collegate agli spazi potrebbe essere quella di delimitare la parte di programmazione con qualcosa che sia equivalente a

```
\catcode` =9 \endlinechar=-1 \makeatletter
```

(l'ultimo comando è per maneggiare le macro interne di LATEX) con un codice di riallineamento alla fine, qualcosa come

```
\edef\restorecodes{%
  \catcode32=\the\catcode32
  \endlinechar=\the\endlinechar
  \noexpand\makeatother
}
\catcode32=9 \endlinechar=-1 \makeatletter

<macro code>

\restorecodes
```

ma questo non curerebbe la sindrome da spazio mancante; anzi, la aggraverebbe perché non potremmo più aggiungere uno spazio alla fine di una costante! Ancora. Talvolta i token spazio servono nel codice di una macro. Potremmo scrivere `\space` dove ci serve uno spazio, ma sarebbe goffo. Ci viene in mente allora una brillante idea: usare `~` come spazio. Proviamo.

```
\edef\restorecodes{%
  \catcode32=\the\catcode32
  \catcode126=\the\catcode126
  \endlinechar=\the\endlinechar
  \noexpand\makeatother
}
\catcode32=9 \catcode126=10
\endlinechar=-1 \makeatletter

\newcount\monthlycount
\newcommand{\monthlytodo}[1]{\par
  \fbox{
    \parbox{10cm}{
      \monthlycount=1~
      \loop\ifnum\monthlycount<13~
        #1--\number\monthlycount\hrulefill\par
        \advance\monthlycount by 1~
      \repeat
    }
  }
}

\restorecodes
```

Molto più *free form*. Ricordo che il codice di categoria 9 significa 'ignorato' e 10 significa 'spazio'. Un momento! La regola dice che gli spazi vengono rimossi alla fine di una riga e sostituiti con il marcatore interno, in questo caso nulla perché il parametro `\endlinechar` ha il valore -1. E forse non avremmo 'veri spazi' nel testo di sostituzione delle nostre macro. Un paio di citazioni dal *TEXbook* ci tolgono i dubbi. La prima riguarda la rimozione degli spazi (Knuth, 1986, p. 46)

> TEX cancella ogni carattere ⟨*spazio*⟩ (numero 32) che capiti alla fine di una riga di input. Poi inserisce un carattere ⟨*return*⟩ (numero 13) alla fine della riga [...]

A questo va aggiunto il fatto che `\endlinechar` di solito ha il valore 13. Quindi la tilde con codice

di categoria 10 non viene rimosso perché non ha codice carattere 32.[5]

Per il secondo problema, vediamo (Knuth, 1986, p. 47)

> Se TEX vede un carattere di categoria 10 (spazio), l'azione dipende dallo stato corrente. Se TEX è in stato N o S, il carattere è semplicemente saltato e TEX rimane nello stesso stato. Altrimenti TEX è in stato M; il carattere è convertito in un token di categoria 10 e con codice di carattere 32, dopo di che TEX va in stato S. Il codice di carattere in un token spazio è sempre 32.

Possiamo dunque contare sul fatto che `~` venga convertito in un 'vero token spazio' quando il testo di sostituzione di una macro è messo in memoria. Certo, perdiamo il carattere che indica uno spazio senza possibilità di spezzare una riga, ma questo ci serve raramente, scrivendo macro, e possiamo sempre adoperare `\nobreakspace`.

Perché non usare `\relax`, invece? Può certamente essere usato, ma perderemmo l'espandibilità 'completa' che ci serve in molte situazioni. Notiamo anche che i registri numerici e le costanti definite con `\chardef` o `\mathchardef` sono già numeri 'completi' e non c'è la ricerca di spazi dopo di essi.

5 LaTeX3 e expl3, un nuovo modo per evitare questi problemi, ma anche molto di più

Chiunque abbia la gentilezza di seguire le mie risposte su TeX.StackExchange avrà già capito dove mi sto dirigendo: il progetto LaTeX3 fu avviato più di due decenni fa, ma è rimasto inerte a lungo, fino a pochi anni fa, quando divenne effettivamente possibile usare il nuovo paradigma di programmazione che aveva introdotto. Quando il LaTeX team cominciò a studiarlo, le risorse di calcolo erano davvero limitate: per esempio, LaTeX 2ε su emTEX lasciava pochissimo spazio per etichette e macro personali. Al giorno d'oggi, invece, possiamo produrre presentazioni, diagrammi e disegni complicati con tempi di calcolo molto brevi. Ricordo senza alcuna nostalgia i tempi in cui anche un diagramma non troppo complicato in PICTEX richiedeva parecchi minuti! Produrre una presentazione con 37 schermate da questo articolo richiede pochi secondi, nonostante la doppia compilazione con un processo PythonTEX in mezzo.

Il rallentamento dovuto a dover leggere qualche migliaio di righe di codice non è più un problema come anni fa; quando queste righe di codice verranno incorporate nel formato, il tempo di caricamento diventerà di pochi millisecondi.

5. Occorre menzionare che le attuali implementazioni di TEX in TEX Live e MiKTEX rimuovono anche i tabulatori (codice di carattere 9).

L'ambiente di programmazione `expl3` fornisce delimitatori dei blocchi di codice simili a quelli che ho descritto prima, ma tratta anche `_` e `:` come caratteri che possono essere usati nei nomi delle sequenze di controllo. Si può pensare a `_` come al tradizionale `@` (che invece in quei nomi non è lecito); l'uso dei due punti è interessante, ci tornerò dopo aver presentato alcuni esempi.

Sono ben conscio del fatto che cambiare un paradigma di programmazione può essere difficile all'inizio, perché è duro liberarsi delle abitudini. Un paio di problemini di assaggio potrebbero essere utili per attirare l'attenzione.

Primo problema: desideriamo ottenere il rapporto tra due lunghezze in modo espandibile da usare, per esempio, come fattore moltiplicativo di un'altra lunghezza, e vorremmo che il rapporto fosse espresso con la migliore approssimazione possibile. Il codice si vede nella tabella 2.

Il codice è su più righe per maggiore leggibilità, ma la parte importante è essenzialmente una riga di codice! E può perfino essere *free form*! Si adopera la funzione `\fp_eval:n` che produce il risultato del calcolo, insieme a una funzione che può convertire da un tipo di dati a un altro. Se proviamo `\dimratio{\textwidth}{\textheight}` o `\dimratio[2]{\textwidth}{\textheight}` otteniamo, rispettivamente, `0.62727` e `0.63`; potremmo anche dire

```
\setlength{\mylength}{%
  \dimratio{\textwidth}{\textheight}\mylength
}
```

per scalare del dato rapporto il valore del parametro `\mylength`.

Secondo problema. Vorremmo isolare e trattare gli elementi in una lista in cui il separatore è il punto e virgola. Prima un interessante codice di Petr Olšák:

```
\def\ls#1{\lsA#1;;}
\def\lsA#1;{%
  \ifx;#1;%
  \else
    \dosomething{#1}\expandafter\lsA
 \fi
}
\def\dosomething#1{%
  \message{I am doing something with #1}%
}

\ls{(a,b);(c,d);(e,f)}
```

Si tratta di una tecnica ben nota che fa bene il suo mestiere, ma ha alcuni difetti che illustrerò dopo aver mostrato il corrispondente codice `expl3`:

```
\ExplSyntaxOn
\NewDocumentCommand{\dosomething}{m}
 {
  I ~ am ~ doing ~ something ~ with ~ #1
 }
```

```
\documentclass{article}
\usepackage{xparse}

\ExplSyntaxOn % start the programming environment
\DeclareExpandableDocumentCommand{\dimratio}{ O{5} m m }
 {
  \fp_eval:n
   {
    round ( \dim_to_fp:n { #2 } / \dim_to_fp:n { #3 } , #1 )
   }
 }

\ExplSyntaxOff % end the programming environment
```

Tabella 2: Codice per la macro `\dimratio`

```
\seq_new:N \l_manual_ls_items_seq

\NewDocumentCommand{\ls}{m}
 {
  \seq_set_split:Nnn
    \l_manual_ls_items_seq
    { ; }
    { #1 }
  \seq_map_function:NN
    \l_manual_ls_items_seq
    \dosomething
 }
\ExplSyntaxOff

\ls{(a,b); (c,d) ;(e,f)}
```

Uno dei nuovi tipi di dati introdotti da `expl3` è il tipo *sequence*: una lista ordinata di elementi ai quali si può accedere globalmente o per numero d'ordine. La prima funzione separa gli elementi e li ordina nella *sequence*; poi `\seq_map_function:NN` passa ogni elemento come argomento alla funzione `\dosomething`, esattamente lo stesso che fanno le macro di Petr Olšák.

Un momento! Se si guarda con attenzione, nel mio esempio ci sono spazi attorno all'elemento centrale che nel codice di Petr passerebbero come parte dell'argomento a `\dosomething`. Questo non succede nel codice `expl3`, perché la funzione di separazione automaticamente rimuove spazi che precedono o seguono un elemento. In particolare, si può perfino scrivere

```
\ls{
  (a,b);
  (c,d);
  (e,f)
}
```

se lo si ritiene conveniente.

Torniamo al codice del Grand Wizard mostrato nella sezione 2.3

```
\def\deleterightmost#1{%
  \edef#1{\expandafter\xyzzy#1\xyzzy}}
\long\def\xyzzy\\#1#2{\ifx#2\xyzzy\yzzyx
  \else\noexpand\\{#1}\fi\xyzzy#2}
\long\def\yzzyx#1\xyzzy\xyzzy{\fi}
```

Lo scopo è di rimuovere l'ultimo elemento da una *sequence*. Nell'appendice D (Knuth, 1986, p. 378) vengono chiamate *list macros* e sono implementate come macro con un testo di sostituzione come `\\{(a,b)}\\{(c,d)}\\{(e,f)}`; la definizione della *sequence* sarebbe

```
\def\myitems{\\{(a,b)}\\{(c,d)}\\{(e,f)}}
```

e la macro `\deleterightmost` deve essere chiamata come

```
\deleterightmost\myitems
```

per rimuovere l'ultimo elemento, come se la prima definizione fosse stata

```
\def\myitems{\\{(a,b)}\\{(c,d)}}
```

Bene, paragonate il codice di Knuth con quello `expl3`

```
\seq_pop_right:NN
  \l_manual_ls_items_seq
  \l_tmpa_tl
```

il quale ha anche il grande vantaggio che l'ultimo elemento è ancora disponibile nella variabile *token list* temporanea `\l_tmpa_tl` (o qualsiasi altra variabile dello stesso tipo decidiamo di usare). La macro di Knuth invece lo scarta.

Ecco una nuova versione della macro `\monthlytodo`:

```
\documentclass{article}
\usepackage{xparse}

\ExplSyntaxOn
\NewDocumentCommand{\monthlytodo}{ O{10cm} m }
 {
  \par
  \noindent
  \fbox
   {
    \manual_monthlytodo:nn { #1 } { #2 }
   }
```

```
 }
\cs_new_protected:Nn \manual_monthlytodo:nn
 {
  \parbox
   {
    \dim_eval:n { #1 - 2\fboxsep - 2\fboxrule }
   }
   {
    \int_step_inline:nnnn { 1 } { 1 } { 12 }
     {
      #2--\int_to_arabic:n { ##1 }
      \hrulefill
      \par
     }
   }
 }
\ExplSyntaxOff

\begin{document}

\monthlytodo{2013}

\monthlytodo[\textwidth]{2015}

\end{document}
```

Notiamo che non c'è più `\loop`, ma si adopera la funzione `\int_step_inline:nnnn`, molto più maneggevole, che ha come argomenti il punto di partenza, il passo, il punto di arrivo e il codice da eseguire, nel quale il valore corrente è disponibile come `#1`, che, in questo caso, deve diventare `##1` perché stiamo dando una definizione. Un argomento facoltativo permette di decidere la larghezza della box: un espediente per dimostrare l'uso di `\dim_eval:n`.

Fine delle ossessioni sugli spazi dopo le costanti, perché gli oggetti sono chiaramente separati gli uni dagli altri. Se una funzione vuole un argomento numerico, questo sarà tra graffe. La *segnatura* della funzione (cioè la parte dopo i due punti) ci dice quanti argomenti sono richiesti (ma ovviamente bisogna anche sapere che tipo di argomento è atteso).

Mostriamo la potenza di `expl3` costruendo un calendario bimensile:

```
\cs_new_protected:Nn \manual_bimonthlytodo:nn
 {
  \parbox
   {
    \dim_eval:n { #1 - 2\fboxsep - 2\fboxrule }
   }
   {
    \int_step_inline:nnnn { 1 } { 2 } { 12 }
     {
      #2 --
      (\int_to_arabic:n { ##1 } --
      \int_to_arabic:n { ##1 + 1 })
      \hrulefill\par
     }
   }
 }
```

Qui il passo è due; verranno eseguiti solo sei passi, perché il successivo andrebbe oltre il limite superiore stabilito: lasciamo i calcoli a TEX.

In `expl3` c'è un'attenta distinzione tra *funzioni* e *variabili*. Internamente sono ovviamente realizzate con macro o registri, ma la distinzione è nell'uso: le funzioni eseguono qualcosa, le variabili contengono token o valori.

Lo schema di nomi rende facile la distinzione: una funzione ha una segnatura formata da zero o più caratteri, separati dal nome con i due punti. Il manuale introduttivo (The LaTeX Project, 2015b) spiega quali caratteri sono leciti e che cosa significano. Il nome di una variabile deve cominciare con `l_`, `g_` o `c_`, a significare *locale*, *globale* o *costante*; `expl3` fornisce funzioni distinte per agire sulle variabili locali o globali; le costanti vanno solo definite dando loro un valore. Un modo facile per trovarsi nelle peste è di esaurire la memoria perché si è riempito completamente il *save stack*; agire sempre correttamente sulle variabili evita il problema. Il manuale sull'interfaccia (The LaTeX Project, 2015a) descrive tutte le funzioni rese disponibili dal *kernel*; ci sono anche altri manuali per i pacchetti ancora allo stato sperimentale, per esempio quello sulle espressioni regolari (The LaTeX Project, 2015c). Non va trascurato il pacchetto per la definizione di comandi di alto livello `xparse` (The LaTeX Project, 2015d).

6 Vantaggi e svantaggi

Attrezzi potenti hanno sempre i loro pro e contro. Ecco una breve lista di ciò di buono che ho trovato in `expl3`.

1. Interfaccia coerente: il team si sforza di fornire un insieme di funzioni di base in modo che il nome suggerisca l'azione.
2. Nuovi tipi di dati: TEX ha, essenzialmente, solo macro, ma `expl3` costruisce strutture di più alto livello che aiutano nel tenere separati i concetti.
3. Centinaia di funzioni predefinite: le azioni più comuni sulle varie strutture di dati sono coperte e il team è di solito pronto a rispondere quando nuovi casi d'uso vengono dimostrati.
4. Sviluppo in corso: lo stato di `expl3` è abbastanza stabile; il buono è che `expl3` nasconde i dettagli implementativi delle azioni di base e dei tipi di dati, in modo che cambi a basso livello non abbiano alcun effetto sui costrutti di più alto livello (efficienza e velocità a parte).

Una breve lista dei nuovi tipi di dati:

token list è un contenitore generico di token (si pensi a `\chaptername`);

sequence è un insieme ordinato di *token list*;

comma separated list è quasi lo stesso, ma più orientato verso l'interfaccia utente;

property list è un insieme disordinato di *token list*, con indirizzamento per *chiave*;

floating point number secondo lo standard IEEE;

regular expression è il più possibile vicino allo standard POSIX;

coffin è una normale box di TEX, ma con molte più maniglie.

I tipi di dati comprendono anche *boolean*, *integer*, *box*, *length*, *skip* (cioè le lunghezze elastiche) oltre ai flussi di input e output come nello standard TEX.

Un altro esempio. Molti articoli su TUGboat riguardano il problema di macro per eseguire il *case switching*; ecco come si può fare in `expl3`:

```
\str_case:nnTF { <stringa> }
 {
  {a}{Caso~a}
  {b}{Caso~b}
  {z}{Caso~z}
 }
 {
  Codice aggiuntivo se
  c'è una corrispondenza
 }
 {
  Codice da eseguire se non
  c'è una corrispondenza
 }
```

La ⟨*stringa*⟩ nel primo argomento proverrà di solito dall'argomento di una funzione/macro. L'esempio mostra anche lo scopo della segnatura delle funzioni: questa si aspetta quattro argomenti tra graffe, gli ultimi due indicano codice da eseguire se un test viene valutato vero o falso (rispettivamente `T` e `F`), in questo caso se c'è una corrispondenza tra il primo argomento e uno degli elementi nel secondo. Sono forniti anche `\str_case:nn`, `\str_case:nnT` e `\str_case:nnF` per i casi in cui il codice per i casi vero e falso non siano necessari: un buon esempio di ciò che considero un'interfaccia coerente.

Un altro esempio di interfaccia coerente. Supponiamo di dover fare qualcosa con due liste di token, una delle quali è esplicita e l'altra è talvolta solo disponibile come valore di una variabile *token list*.

```
\cs_new_protected:Nn \manual_foo:nn
 {
  Do~something~with~#1~and~#2
 }
\cs_generate_variant:Nn
  \manual_foo:nn
  { nV , Vn , VV }

\manual_foo:nn { Bar } { Foo }

\manual_foo:nV { Bar } \l_manual_whatever_tl

\manual_foo:Vn \l_manual_whatever_tl { Foo }

\manual_foo:VV
  \l_manual_whatever_a_tl
  \l_manual_whatever_b_tl
```

La *variante* è definita in modo coerente, ciò che ci dà quattro funzioni con nomi simili che eseguono la stessa azione su argomenti specificati diversamente. Fare lo stesso in LATEX classico richiede giochi di prestigio con gli argomenti e qualche `\expandafter` ben piazzato: un gioco divertente, che però conduce spesso a indesiderate duplicazioni di codice. L'argomento di tipo `V` significa che l'argomento deve essere una variabile, il cui valore è estratto e sostituito con un argomento tra graffe per poi essere trattato dalla funzione di base.

Ecco i contro. Il codice è molto più verboso, come il codice C è molto più verboso dell'Assembler; è il prezzo da pagare quando ci sono molte più funzioni di base disponibili. Non si deve mai usare `\expandafter`, il giocattolo preferito di ogni programmatore TEX; in particolare, non c'è più `\expandafter\@firstofone`.[6] Scrivere codice `expl3` richiede ancora di capire come funziona l'espansione di macro e i messaggi di errore sono spesso molto criptici perché spesso vengono dal livello più basso di TEX; va rilevato però che anche programmare in plain TEX può portare a messaggi di errore incomprensibili.

7 I veri vantaggi

L'albero `.../tex/latex` in TEX Live comprende più di 2000 cartelle. Se guardiamo dentro quei pacchetti, vediamo facilmente che reinventano la ruota molte volte. Non è affatto raro vedere reimplementate anche funzioni già disponibili nel nucleo di LATEX. La macro `\ls` descritta prima, con la sua ausiliaria `\lsA` è un esempio: il codice è buono, ma può essere ritrovato in un numero indefinito di piccole variazioni, ogni volta che deve essere eseguita un'operazione di *parsing*. Lo stesso può dirsi di dozzine di costrutti tipici: avere un'ampia base di funzioni per le azioni di uso comune renderà il codice più leggibile e comprensibile. Studiare l'esempio di `\xyzzy` è certamente divertente e istruttivo per afferrare concetti connessi con le macro ricorsive, in particolare la *tail recursion*. Ma perché rifare il lavoro che il LATEX team ha già fatto per noi?

Gli articoli di Frank Mittelbach (con Rainer Schöpf o Chris Rowley) (Mittelbach e Schöpf, 1991; Mittelbach e Rowley, 1992, 1997) su LATEX3 sono letture molto interessanti per capire gli scopi del progetto. Non è solo 'evitare gli spazi spuri' o 'non dimenticare spazi necessari': abbiamo a disposizione, in uno stato piuttosto maturo, un ambiente di programmazione quasi completo ed estendibile che ci libera dal dover pensare alle

6. Sto scherzando, è chiaro.

faccende di basso livello e permette di concentrarci sulla vera programmazione.[7]

Poiché il convegno del TUG si è svolto a Darmstadt, Germania, colgo l'occasione per chiudere in tedesco, con scuse a David Hilbert:

Aus dem Paradies,
das das LaTeX3 Team uns geschaffen,
soll uns niemand vertreiben können.

(Nessuno ci espellerà dal paradiso che il LaTeX3 team ha creato per noi.)

Riferimenti bibliografici

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

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

Mittelbach, F. e Rowley, C. (1992). «LaTeX 2.09 ↪ LaTeX3». *TUGboat*, **13** (1), pp. 96–101. http://tug.org/TUGboat/tb13-1/tb34mittl3.pdf.

— (1997). «The LaTeX3 Project». *TUGboat*, **18** (3), pp. 195–198. http://tug.org/TUGboat/tb18-3/l3project.pdf.

Mittelbach, F. e Schöpf, R. (1991). «Towards LaTeX 3.0». *TUGboat*, **12** (1), pp. 74–79. http://tug.org/TUGboat/tb12-1/tb31mitt.pdf.

The LaTeX Project (2015a). «The LaTeX3 interfaces». http://mirror.ctan.org/macros/latex/contrib/l3kernel/interface3.pdf.

— (2015b). «The expl3 package and LaTeX3 programming». http://ctan.org/pkg/l3kernel.

— (2015c). «The l3regex package: Regular expressions in TeX». http://ctan.org/pkg/l3regex.

— (2015d). «The xparse package: Document command parser». http://ctan.org/pkg/xparse.

7. In una conversazione durante il convegno del TUG, Stefan Kottwitz mi chiese di dire quali siano i più importanti passi avanti fatti da `expl3`; risposi 'i calcoli con virgola mobile e le espressioni regolari'. Sono ovviamente implementati con le primitive di TeX, ma la mia opinione è che non sarebbero così potenti se `expl3` non fosse già stato sviluppato.

▷ Enrico Gregorio
Università di Verona
Dipartimento di Informatica
Strada le Grazie 15
Verona (Italy)