

Ridefinire i comandi primitivi di T_EX e applicazioni a L^AT_EX

Enrico Gregorio

Sommario

La composizione tipografica eseguita da L^AT_EX si basa su alcuni comandi primitivi direttamente codificati nel programma T_EX. È possibile, sapendo ciò che si fa, *ridefinire* questi comandi. Discuterò alcuni esempi di questo tipo che daranno idee per altre ridefinizioni di comandi.

1 Introduzione

T_EX conosce circa trecento comandi *primitivi*, cioè che sono direttamente codificati nel programma. È sulla base di questi che vengono costruite le estensioni a tutti note: prima fra tutte il formato Plain T_EX, scritto dallo stesso D. E. Knuth (K_NU_TH, 1984), e il più noto L^AT_EX, dovuto inizialmente a L. Lamport e poi sviluppato da un gruppo internazionale (BRAAMS *et al.*). In questo articolo assumerò che si lavori in L^AT_EX.

Il comando primitivo più usato è `\par`. Forse qualcuno dei lettori non ha mai scritto la sequenza di controllo `\par` nei suoi documenti; eppure ha usato questo comando innumerevoli volte: infatti è T_EX stesso che si occupa di convertire una riga vuota in un comando `\par`.

Un altro comando primitivo molto usato è `\input`, che ha una sintassi molto particolare: il suo argomento è la più corta stringa di caratteri alfanumerici che lo segue (spazi e caratteri speciali sono esclusi). “Un momento!” dirà più di qualcuno: si deve usare la sintassi

```
\input{<nomefile>}
```

come spiegato in tutti i manuali di L^AT_EX. Ecco: questo è un esempio di ridefinizione di un comando primitivo.

Il comando `\input` originale ha una sintassi peculiare perché deve adattarsi a vari sistemi operativi. D'altra parte l'autore di L^AT_EX voleva nascondere all'utente alcune difficoltà e, soprattutto, mantenere il più possibile l'uniformità della sintassi. Un modo per risolvere il problema sarebbe di definire

```
\newcommand{\Input}[1]{\input #1}
```

e quindi obbligare l'utente al comando `\Input`, che avrebbe una forma non usuale (i comandi ‘pubblici’ di L^AT_EX sono tutti in minuscolo, tranne alcuni per accenti o simboli).

Si noti l'uso di `\newcommand`, che è specifico di L^AT_EX; una delle sue funzioni è di controllare che

il comando che si vuole introdurre non sia già esistente; se è già definito un comando con quel nome, L^AT_EX dà un errore. Questo controllo non viene eseguito dal comando primitivo `\def` e dal suo parente `\let`, che vedremo più avanti.

Come fare? La risposta è: ridefinire `\input`. Solo che qualcosa come

```
\renewcommand{\input}[1]{\input #1}
```

è destinato a fallire miseramente. Vedremo fra poco come si risolve il problema.

2 Il carattere @ e il comando \let

In molti comandi ‘interni’ di L^AT_EX compare il carattere ‘@’. Questo carattere non può, normalmente, essere usato in questo modo: i nomi dei comandi consistono (1) di una sola “non lettera”, (2) di una o più lettere (a-z, A-Z). Tuttavia, quando vengono letti i pacchetti e le classi (`.sty` e `.cls`) L^AT_EX è in uno stato speciale nel quale il carattere @ è considerato come una lettera normale; a livello utente si può ottenere lo stesso con il comando `\makeatletter` che va poi annullato con il comando `\makeatother`. I programmatori di pacchetti usano questo trucco per scrivere comandi che non possano essere inavvertitamente modificati. Ogni comando interno ha almeno un carattere @ nel nome, proprio per questo.

Una delle prime righe di `latex.ltx` (che contiene le definizioni del nucleo di L^AT_EX) è

```
\let\@@input\input
```

Anche qui vale la convenzione che @ è una lettera. Che cosa succede? Il comando `\let` seguito da due nomi di comando

```
\let\comandob\comandoa
```

fa in modo che `\comandob` abbia lo stesso significato *attuale* di `\comandoa`; successive ridefinizioni di `\comandoa` non influenzano il significato di `\comandob`. Attenzione: `\let` non esegue controlli come `\newcommand` e quindi è *pericoloso*, se usato da mani non esperte.

Dunque abbiamo a disposizione un comando equivalente a `\input`. Per inciso, questa è una convenzione usata in tutto il nucleo di L^AT_EX: i comandi che cominciano con due caratteri @ sono di solito equivalenti a uno dei comandi primitivi. Più in là in `latex.ltx` si trova qualcosa come

```
\renewcommand{\input}[1]{\@@input #1}
```

e questo adesso *funziona!* A dire il vero la definizione è un po' più complicata: non importa, la funzione essenziale che viene eseguita è questa.

Ora un comando come `\input{pippo}` è, agli occhi di T_EX, del tutto equivalente a

```
\@@input pippo
```

e, quando raggiunge le profondità del programma dove si eseguono solo comandi primitivi, fa ciò che ci si aspetta: T_EX legge il *file* `pippo.tex`.

Un altro comando primitivo ridefinito è `\end`. Il significato di questo comando è 'finisci qui': quando T_EX lo incontra, considera chiuso l'ultimo paragrafo ed emette tutte le pagine ancora da comporre (più parecchie altre cose che non sono rilevanti per la discussione).

Lamport decise che la struttura di L^AT_EX fosse ad *ambienti* e che ciascuno di essi fosse delimitato da `\begin{nome}` e da `\end{nome}`. Ovviamente si rese necessario ridefinire il comando primitivo con lo stesso nome. Dunque

```
\let\@@end\end
```

dopo di che siamo liberi di definire `\end` come ci pare, purché ci ricordiamo di usare `\@@end` per dire a T_EX di finire il lavoro. E infatti la definizione di `\enddocument` è

```
\def\enddocument{%
...
\@@end}
```

Già che ci sono, spiego a chi non lo sapesse che il comando `\begin{nome}` esegue certe funzioni e alla fine il comando `\nome`, mentre `\end{nome}` esegue altre funzioni di controllo della situazione e il comando `\endnome`. Per questo `\newcommand` emette un messaggio di errore quando si cerca di definire un comando il cui nome cominci con `end`.

Il comando `\def` è quello primitivo di T_EX per introdurre nuovi comandi. Il suo uso non è molto difficile, ma va evitato se non si sa quello che si sta facendo: come `\let`, infatti, non esegue alcun controllo sul nome del comando che si sta definendo. Tuttavia è spesso l'unico modo di procedere. Vediamo un esempio, sempre da `latex.ltx`:

```
\def\@setpar#1{\def\par{#1}\def\@par{#1}}
\def\@par{\let\par\@@par\par}
\def\@restorepar{\def\par{\@par}}
```

Che fa il primo comando? Ha un parametro, prima di tutto. Quando viene eseguito, cambia la definizione di `\par` e ridefinisce un comando `\@par` rendendolo equivalente a `\par` (ridefinito).

Il secondo comando definisce `\@par`; quando questa 'incarnazione' di `\@par` viene eseguita, il comando `\par` riprende il suo significato originale (che è ben conservato in `\@@par`) ed eseguito.

Il terzo comando, quando eseguito, fa diventare `\par` un comando che esegue `\@par`.

Confusi? Forse. L'idea è che `\@setpar` venga eseguito dentro un ambiente (nel caso particolare `trivlist`) e quindi le modifiche che fa a `\par` spariscono alla fine dell'ambiente stesso. Il comando `\@restorepar` viene usato nella definizione di `\vspace` e nelle funzioni di chiusura degli ambienti, per assicurarsi di 'disfare' eventuali modifiche apportate a `\par`.

Usare `\(re)newcommand` in queste situazioni non sarebbe possibile per vari motivi.

L'idea di dare significati particolari a `\par` è stata usata dallo stesso Knuth nelle macro usate per comporre il T_EXbook e gli altri volumi della serie *Computers and Typesetting*. Possiamo usare idee simili alle sue per fare una cosa un po' stupida:

```
\makeatletter
\let\ciao@@end\@@end
\def\@@end{\message{^^J^^JCiao, %
ciao^^J^^J}\ciao@@end}
\makeatother
```

nel preambolo fa in modo che alla fine di una compilazione T_EX ci saluti. Lo strano simbolo `^^J` serve a inserire nei messaggi a schermo un fine riga. Provatelo.

Che cosa abbiamo fatto? Semplicemente duplicato la funzione di `\@@end` nel comando `\ciao@@end` (un nome come questo è difficilmente già definito) e poi ridefinito `\@@end` in modo che faccia apparire il messaggio e poi esegua `\ciao@@end` che è del tutto equivalente al comando primitivo `\end` prima di qualsiasi ridefinizione.

3 Il pacchetto *everyshi*

L'idea di ridefinire un comando primitivo è impiegata nel pacchetto *everyshi* sul quale sono poi basati *prelim2e* e *eso-pic*. Il comando è uno di quelli misteriosissimi, `\shipout`.

Si tratta di un comando che nessuno usa mai esplicitamente: è quello che scarica nel `.dvi` o nel `.pdf` una pagina quando è completa con tanto di materiale annesso (testatina, piè di pagina, note a margine e così via).

Il pacchetto è brevissimo, 31 righe di codice, ma molto efficiente e, soprattutto, indipendente dalle classi che si usano. Infatti la sua azione si svolge dopo che L^AT_EX ha composto una pagina secondo le impostazioni ricevute dalla classe e usa solo comandi di basso livello. Questo permette al pacchetto *eso-pic* di aggiungere alla pagina, per esempio, le filigrane (in inglese *watermarks*): un bel *Top secret* stampato in grigio sullo sfondo, se si vuole (si veda la documentazione di *eso-pic* reperibile nella propria distribuzione o su CTAN¹).

1. CTAN è l'acronimo di 'Comprehensive T_EX Archive Network', www.ctan.org.

Il pacchetto `prelim2e` scrive alcune cose nel piè di pagina, utili per identificare le versioni preliminari di un documento.

Un altro uso della ridefinizione di `\shipout` è quello che si impiegava alcuni anni fa, quando le possibilità di agire sui `.dvi` erano limitate e il PDF ancora non esisteva. Un ingegnoso insieme di macro permetteva di stampare due pagine, purché di dimensioni appropriate, sullo stesso foglio e rendeva possibile la produzione diretta di libretti in formato A5. Oppure di risparmiare sui costosi fogli di acetato che servivano a produrre le lastre per la stampa offset².

Al giorno d'oggi il problema è molto meno sentito: se si deve stampare un libro si manda al tipografo un documento PDF con le pagine nel formato finale e tutto finisce lì; con il pacchetto `geometry` è facile definire un formato di 'foglio' arbitrario.

4 Usi veri delle ridefinizioni

Tutto quanto visto prima può sembrare complicato (e lo è); ma ci dà idee per cose che ci possono servire molto più spesso.

Per esempio, vogliamo controllare se nel nostro lungo documento abbiamo abusato del corsivo di enfasi. Come fare perché ci salti agli occhi? Senza però modificare troppo il sorgente e senza dover andare in cerca dei comandi `\emph`.

Ridefiniamo `\emph`, questa è la risposta! Faremo in modo che il nostro corsivo di enfasi appaia in un vistoso rosso che a schermo balzerà evidente:

```
\usepackage{color}
\newcommand{\originalemph}{-}
\let\originalemph\emph
\renewcommand{\emph}[1]{%
  \originalemph{\color{red}#1}}
```

Ci sono altri modi per ottenere lo stesso risultato (BECCARI, 2006), ma certamente questo è semplice e molto efficace. A che cosa serve la seconda riga? Facile: siccome `\let` non controlla se il comando che segue sia già definito, usiamo `\newcommand` per fare questo lavoro; così, se per caso `\originalemph` fosse già definito, LATEX reagirebbe con un messaggio di errore.

Supponiamo ora di voler stampare il nostro documento senza che però compaiano le figure inserite con `\includegraphics`; ovviamente vogliamo che lo spazio necessario sia lasciato bianco, quindi l'opzione `draft` alla classe o al pacchetto `graphicx` non è adatta, perché al posto dell'immagine stampa un riquadro con il nome del *file* grafico.

2. I fogli di acetato sono trasparenti e simili a quelli per le trasparenze; servivano da mastro per l'incisione delle lastre metalliche su cui veniva steso l'inchiostro per la stampa. Fra l'altro, andavano stampati in modo speculare perché il *toner* doveva essere dal lato non a contatto con la lastra, sulla quale ovviamente andava incisa l'immagine speculare della pagina.

Un modo potrebbe essere quello di definirsi un comando `\myig` per fare ciò che si desidera. Ma il documento conterrebbe questo comando e non l'usuale. Si può fare? Certo che si può.

```
\newcommand{\origig}{-}
\let\origig\includegraphics
\renewcommand{\includegraphics}[2] []
  {\phantom{\origig[#1]{#2}}}
```

Uso qui la possibilità di definire comandi con un argomento opzionale; nel nostro caso `\includegraphics` viene ridefinito con un argomento opzionale e uno obbligatorio. Ciò che viene sostituito all'argomento opzionale quando non sia specificato è *niente*. Come in precedenza, `\newcommand{\origig}{-}` serve a evitare di ridefinire un comando che abbia già un significato.

Esaminiamo ora una chiamata tipica:

```
\includegraphics[scale=.5]{pippo}
```

Questa viene tradotta in

```
\phantom{\origig[scale=.5]{pippo}}
```

Sappiamo che `\phantom` compone ciò che ha come argomento, ma poi lascia solo lo spazio richiesto. Siccome `\origig` ha lo stesso significato originale di `\includegraphics` abbiamo ottenuto quello che desideravamo. Se scriviamo

```
\includegraphics{pippo}
```

questo risulta in

```
\phantom{\origig[] {pippo}}
```

ma a `\includegraphics` originale non dà alcun fastidio avere un argomento opzionale vuoto. *Voi-là*. Il documento ha gli stessi comandi che avrebbe normalmente e basterà cancellare le due righe in cui abbiamo definito `\origig` e ridefinito `\includegraphics` per avere un documento che non usa alcun comando strano.

Un problema apparentemente più complicato è quello di colorare tutte le tabelle di verde; precisamente vogliamo che ogni elemento della tabella sia verde, testo e linee. Ovviamente vogliamo mantenere la possibilità di specificare per un ambiente `tabular` l'argomento opzionale di allineamento e specificare l'ambiente con il suo nome usuale: si tratta di un espediente temporaneo per mettere in evidenza le tabelle.

Qui viene a proposito il modo con cui LATEX comincia l'ambiente `nome`, ricordate? Viene eseguita una serie di azioni, l'ultima delle quali è l'esecuzione del comando `\nome`.

```
\newcommand{\origtabular}{-}
\let\origtabular\tabular
\renewcommand{\tabular}{%
  \color{green}\origtabular}
```

Che succede quando si chiama `tabular` dopo aver dato questi comandi? Le solite azioni di inizio ambiente vengono svolte, per finire viene emesso il comando `\tabular` che ora significa

```
\color{green}\origtabular
```

Quindi \LaTeX si prepara a scrivere in verde (meglio, istruisce il *driver* a farlo) ed esegue il comando `\origtabular` che ha lo stesso significato dell'originale `\tabular`: questo allora va in cerca dei suoi argomenti al modo solito e quindi la sintassi è a posto.

Si noti che la dichiarazione di 'scrivere in verde' avviene all'interno di un gruppo (che il comando `\begin` ha già aperto); quindi il corrispondente `\end{tabular}` ne annulla l'effetto: fuori da un ambiente `tabular` il testo sarà nel colore normale.

Funziona tutto sempre? Be', no. Supponiamo di non voler scrivere ogni volta `\hline` dopo ogni riga di una tabella. Sfrutterò ora il fatto che all'interno di un ambiente `tabular` il comando `\tabularnewline` è equivalente a `\\`. Si potrebbe pensare di scrivere

```
\newcommand{\origtabular}{}
\renewcommand{\tabular}{%
  \renewcommand{\\}{\tabularnewline
    \hline}%
  \origtabular}
```

cioè ridefinire `\\` dopo aver cominciato l'ambiente, per avere ciò che ci aspettiamo. Giusto? No. Provare per credere: nessun messaggio di errore, ma niente linee orizzontali. Perché? Non è così facile: il fatto è che lo stesso comando (originale) `\tabular` chiama un altro comando il quale ridefinisce a sua volta il comando `\\`. E ridefinire questo, che è di quelli che possono avere l'asterisco e un argomento opzionale richiederebbe di ridefinirne un altro e di perdersi nel mare di definizioni: meglio scrivere `\hline`, dopo tutto (o magari evitarlo, in generale).

Ovviamente la strategia di definirsi nuovi ambienti sulla base di ambienti dati rimane utilissima: se per caso abbiamo molte tabelle con un tipo fisso di struttura, è certamente conveniente definire

```
\newenvironment{fixtab}
  {\begin{tabular}{ccc}}
  {\end{tabular}}
```

dove, ovviamente, `ccc` è il tipo di allineamento desiderato. Questo ci permette, per esempio, di modificare la struttura delle colonne di tutte le tabelle di quel tipo agendo in un solo punto del sorgente.

5 Parametri

Molti comandi hanno parametri (e quindi vanno in cerca, quando chiamati, dei loro argomenti). Quando si vogliono ridefinire, questi argomenti possono dare problemi.

Uno dei consigli che vanno maggiormente ascoltati è quello di definirsi *sempre* comandi astratti per le strutture logicamente distinte nel nostro documento. Per esempio, volendo usare notazioni uniformi per gli insiemi numerici (naturali, interi, reali), è bene definirsi una struttura astratta e definire i comandi in termini di quella:

```
\newcommand{\nf}[1]{\mathbf{#1}}
\newcommand{\N}{\nf{N}} % naturali
\newcommand{\Z}{\nf{Z}} % interi
...
```

Ma qui c'è qualcosa di troppo che potremmo eliminare: ci si creda o no, la prima riga può diventare semplicemente

```
\newcommand{\nf}{\mathbf{}}
```

(o addirittura, se si vuole essere un *macho* \LaTeX programmer, `\let\nf\mathbf`). In generale, è meglio evitare di leggere gli argomenti quando non è necessario, useremmo male la memoria di \TeX e lo rallenteremmo. Vediamo infatti che cosa succede con la prima definizione quando \LaTeX vede `\nf{X}`:

1. per prima cosa si cerca qual è l'argomento; nel caso specifico, `X`;
2. viene sostituito `\nf{X}` con il suo sviluppo, cioè `\mathbf{X}`;
3. ora viene sviluppato `\mathbf` che va in cerca del suo argomento³.

C'è un passaggio di troppo: irrilevante qui, dove l'argomento è un singolo carattere. Può diventarlo quando si deve valutare un argomento complesso e magari lungo parecchie righe.

Lo sviluppo della seconda definizione di `\nf` evita il problema.

Supponiamo, sempre per esempio, di voler fare in modo che ogni sezione cominci una nuova pagina e, come al solito, di non voler modificare sostanzialmente il sorgente per poter tornare indietro facilmente.

Il comando `\section` è ben diverso da `\includegraphics`: dare un argomento opzionale vuoto non è come non darlo. C'è anche il problema che l'argomento di `\section` è *mobile* e non lo si deve 'agitare troppo' per evitare problemi durante la composizione dell'indice. Per non parlare della possibilità che un asterisco segua il comando.

La soluzione dovrebbe essere ovvia, ormai:

```
\newcommand{\oldsection}{}
\let\oldsection\section
\renewcommand{\section}{%
  \clearpage\oldsection}
```

3. Chi sa di *inner* \LaTeX obietterà che `\mathbf` non è veramente un comando con un argomento; di fatto quasi nessuno dei comandi pubblici di \LaTeX ha argomenti. Ma, agli scopi pratici, questo è poco importante.

Lasciamo a `\oldsection` il problema di cercare l'asterisco, di vedere se c'è l'argomento opzionale e di valutare l'argomento obbligatorio. Notiamo che questo funziona anche quando si siano caricati pacchetti come `titlesec` o `sectsty` che modificano lo stesso comando `\section`, purché la nostra ridefinizione avvenga *dopo* il caricamento del pacchetto e la relativa scelta del formato dei titoli di sezione.

6 Come distinguere le situazioni?

Abbiamo visto due diverse situazioni in cui è necessario adottare strategie diverse per ottenere lo scopo voluto. Se però ci pensiamo, non è così complicato scegliere quella più opportuna.

Nel caso di `\includegraphics` siamo forzati a scegliere di definire un comando con argomenti perché dobbiamo fornire il risultato come argomento di `\phantom`; siamo fortunati perché il comando `\includegraphics` accetta un argomento opzionale vuoto senza alcun problema. Nel caso di `\section`, dobbiamo solo aggiungere qualcosa *prima* di eseguire il comando stesso.

Entriamo ora un po' più a fondo nei meandri del nucleo di \LaTeX per vedere una diversa applicazione. Come sempre sarà un'applicazione inutile, ma l'idea può forse essere usata per cose più aderenti alla realtà.

Usiamo la classe `book`; vogliamo che i titoli dei capitoli *non numerati* siano scritti in rosso.

Possiamo ridefinire semplicemente `\chapter`? Forse, ma dovremmo cominciare a vedere se il comando è seguito dall'asterisco oppure no. \LaTeX permette di definire facilmente comandi con un argomento opzionale, ma non comandi con la forma variata.

Come possiamo agire? Guardiamo `book.cls` e cerchiamo la definizione di `\chapter`. Vedremo che alla fine di quelle righe c'è

```
\secdef\@chapter\@schapter
```

Un po' di intuizione ci dice che `\@chapter` è il comando che viene chiamato quando non c'è l'asterisco. Quello che ci serve allora è `\@schapter`. Un'altra scorsa a `book.cls` rivela che `\@schapter` è un comando con un argomento.

La ricerca è finita: basterà dare

```
\makeatletter
\let\old@schapter\@schapter
\renewcommand{\@schapter}[1]{%
  \old@schapter{\color{red}#1}}
```

Un'applicazione certamente più intelligente è quella di aggiungere automaticamente una riga `\addcontentsline` basata sul titolo del capitolo e la lascio come esercizio al lettore. Una piccola complicazione: vogliamo che l'indice vada nell'indice? Il suggerimento è di rendere attiva una modifica del genere dopo aver composto l'indice e gli elenchi di tabelle e figure.

7 Evitare di leggere un argomento inutilmente

Mi cimento ora con una ridefinizione più complicata di quelle viste prima. Il problema è quello di scrivere in rosso il contenuto di ogni `\mbox`.

Userò il fatto che l'argomento di `\mbox` può essere delimitato anche da `\bgroup` e `\egroup` oltre che da `{` e `}`; inoltre possiamo usare sia un tipo di delimitatore che l'altro. Do subito il codice.

```
\let\redmbox\mbox
\def\mbox{\afterassignment\doredmbox
  \let\next=}
\def\doredmbox{\redmbox\bgroup
  \color{red}}
```

Qui uso `\def` perché sto facendo cose piuttosto complicate e perciò mi servo di comandi a basso livello (come i grandi programmatori che usano l'Assembler). In questo caso `\newcommand` sarebbe andato bene lo stesso.

La procedura è sempre la stessa: si definisce un equivalente del comando da modificare. Il problema qui è che vogliamo entrare nell'argomento di `\mbox`, aggiungere la specifica del colore rosso e poi far eseguire il comando `\mbox` originale sul nuovo argomento.

Entra in gioco di nuovo `\let`. La descrizione della sintassi di `\let` indicata in precedenza non è completa. In realtà

1. `\let` deve essere seguito da un nome di comando;
2. subito dopo può esserci un carattere `=` il quale può essere seguito da uno spazio;
3. infine ci vuole un *token* (vedi BECCARI (2006) per sapere che cos'è un *token*).

È perfettamente lecito scrivere `\let\xxx=a`; in tal caso il comando `\xxx` si comporta quasi sempre come il carattere `'a'`. Siccome il carattere `=` è opzionale, andrebbe bene anche `\let\xxx a`, ma non sarebbe altrettanto leggibile.

Nel nostro caso il comando `\redmbox`, dopo aver eseguito

```
\afterassignment\doredmbox
```

esegue `\let\next=` e quindi fa diventare `\next` equivalente alla parentesi graffa che segue `\mbox`⁴.

Che cosa fa `\afterassignment\doredmbox`? Il suo scopo è di eseguire `\doredmbox` *dopo* che è stato eseguito `\let` (che è un comando di assegnazione). Che cosa fa `\doredmbox`? Il suo sviluppo è

4. Qui il carattere `=` è necessario; infatti, nel caso (improbabile) che uno scriva `\mbox=` (che è lecito), \LaTeX assegnerebbe a `\next` il significato del *token* che segue. È un \TeX nicismo, ma è meglio essere precisi; insomma, `\let\next==` assegna a `\next` il significato del carattere `=`, mentre `\let\next=` è ancora *incompleto*.

```
\redmbox\bgroup\color{red}
```

cosicché `\redmbox` ‘vede’ un argomento che è quello dato ma con aggiunta all’inizio la specifica del colore rosso.

In definitiva, la parentesi graffa aperta che segue `\mbox` viene ‘mangiata’ con `\let\next={`, quindi T_EX vede `\redmbox` che è equivalente a `\mbox` originale, poi la graffa aperta (`\bgroup`) e il comando di scrivere in rosso. È come se scrivessimo ogni volta

```
\mbox{\color{red}...}
```

invece che `\mbox{...}`, che è ciò che volevamo.

Il comando `\next` viene definito, ma non fa niente, se non ingoiare la graffa. È tradizione usarlo proprio per scopi come questo, Knuth lo chiama *scratch macro*, cioè ‘macro per scarabocchi’, come se fosse un pezzo di carta per prendere appunti.

8 Impacchettare le modifiche

Supponiamo di aver deciso di modificare alcuni comandi a nostro uso durante la composizione (per esempio il rosso per il corsivo). Desideriamo rendere ancora più semplice l’operazione di attivazione o disattivazione delle modifiche.

Il modo più indolore per ottenere lo scopo è di definirsi una propria classe nella quale incorporare queste modifiche. A documento terminato, sarà così necessario solo cambiare *una riga* del sorgente, scegliendo la classe definitiva, diciamo `arstexnica`.

Il compito sembra imponente, ma vedremo che è semplicissimo. Apriamo un *file* `myarstexnica.cls` nel quale scriveremo

```
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesClass{myarstexnica}
\DeclareOption*{%
  \PassOptionsToClass{\CurrentOption}%
  {arstexnica}}
\ProcessOptions\relax

\LoadClass{arstexnica}
\RequirePackage{color}

\let\myat@emph\emph
\renewcommand{\emph}[1]{%
  \myat@emph{\color{red}#1}}

\let\myat@redmbox\mbox
\def\mbox{\afterassignment\myat@doredmbox
  \let\next= }
\def\myat@doredmbox{\myat@redmbox\bgroup
  \color{red}}
```

A seguire scriveremo il codice delle altre modifiche desiderate e termineremo con `\endinput`. La nostra classe è pronta: non era poi così difficile. Possiamo tenere questo *file* nella stessa cartella

dove abbiamo il documento da comporre oppure trasferirlo in una delle aree (locali) dove T_EX cerca classi e pacchetti.

Un paio di annotazioni. Le prime due righe sono di identificazione, la terza e la quarta servono a passare le opzioni date alla classe `myarstexnica` nell’argomento opzionale di `\documentclass` alla classe `arstexnica` che viene caricata in seguito. La nostra classe privata non definisce alcuna nuova opzione, anche se sarebbe possibile farlo.

La riga successiva è necessaria per elaborare le opzioni. Viene poi caricata la classe sulla quale la nostra si basa e anche il pacchetto `color` essenziale se vogliamo colorare di rosso qualcosa.

I caratteri ‘%’ servono per evitare di lasciare spazi bianchi non voluti nel documento finale; in generale è meglio usarli, quando si scrivono definizioni, nelle righe che non finiscono con un nome di comando. Qui e in altri punti dell’articolo le righe sono spezzate per esigenze tipografiche, nella pratica molte non lo sarebbero.

Ultima nota: quando si scrive una classe o un pacchetto, è conveniente usare come nomi privati di comandi un prefisso seguito dal carattere ‘@’; se il prefisso è scelto bene, sarà improbabile che si ridefiniscano comandi già esistenti. In questo caso ho scelto come prefisso `myat`; a parte la differenza di nomi, la tecnica è quella spiegata in precedenza. Per esempio, `\myat@emph` corrisponde a `\originalemph` della sezione 4.

A questo punto il nostro documento comincerà semplicemente con

```
\documentclass[{opzioni}]{myarstexnica}
```

e ci basterà cancellare due lettere per ottenere il documento finale.

La classe `arstexnica` è quella usata per comporre questa stessa rivista, la si può scaricare dal sito di G_UI_T all’indirizzo <http://www.guit.sssup.it/downloads/arstexnica.zip>

Riferimenti bibliografici

BECCARI, C. (2006). «I registri *token*: questi sconosciuti». *ArsT_EXnica*.

BRAAMS, J., CARLISLE, D., JEFFREY, A., LAMPORT, L., MITTELBACH, F., ROWLEY, C. e SCHÖPF, R. «The L^AT_EX 2_ε sources». Compreso nella distribuzione di L^AT_EX.

KNUTH, D. E. (1984). *The T_EXbook*. Addison-Wesley, Reading, MA, USA.

- ▷ Enrico Gregorio
Dipartimento di Informatica, Università di Verona
Enrico.Gregorio@univr.it