Liste, cicli, LaTeX3

Enrico Gregorio

Sommario

Si esamina come eseguire cicli in LaTeX con la macro `\foreach` e come sostituirla con codice basato su LaTeX3.

Abstract

We examine how to execute loops in LaTeX with the macro `\foreach` and a new implementation based on LaTeX3 code.

1 Introduzione

In molte occasioni è necessario o conveniente ripetere un comando più volte su una lista di oggetti o su interi in un certo intervallo.

Con plain TeX è possibile eseguire cicli con `\loop`, che ha la forma

```
\loop
  ⟨codice⟩
⟨if⟩⟨test⟩
  ⟨codice⟩
\repeat
```

Ovviamente è anche possibile definire nuove macro per i propri cicli. Più esteso è il supporto del nucleo di LaTeX, che mette a disposizione

```
\@for
\@tfor
\@whilenum
\@whiledim
\@whilesw
```

Si vedano Goossens *et al.* (1994) e Braams *et al.* (2016).

Il pacchetto etoolbox fornisce parecchie macro dedicate allo scopo. È anche possibile trovare codice girando per la rete, in particolare per estendere le funzionalità di `\loop`.

Nell'articolo si presume l'uso di LaTeX, ma per alcune macro sarà adoperato codice plain TeX.

2 `\foreach`

Un comando molto comodo è fornito da PGF, con il pacchetto pgffor, caricato automaticamente da Ti*k*Z(Tantau, 2015). La sintassi di base è

```
\foreach \i in {⟨lista⟩}{⟨codice⟩}
```

oppure

```
\foreach \i in ⟨macro⟩ {⟨codice⟩}
```

ed è possibile dare alcune opzioni per ottenere effetti particolari.

Mi occuperò per ora del caso base, in cui la lista è di valori separati da virgole, in cui è anche possibile adoperare `...` per indicare ripetizione secondo uno schema; per esempio, invece di

```
\foreach \i in {1,3,5,7,9,11,13,15}{⟨codice⟩}
```

è più semplice scrivere

```
\foreach \i in {1,3,...,15}{⟨codice⟩}
```

specificando il valore iniziale, quello finale e il passo. Si può anche scrivere `{1,...,10}` e il passo sarà implicitamente uno. I valori possono anche essere numeri in virgola mobile, si dirà più avanti qualcosa al riguardo. La lista può anche essere arbitraria, come

```
\foreach \i in {cane,gatto,criceto}{%
  ⟨codice⟩%
}
```

In questo caso, ovviamente, non è possibile la notazione ellittica.

La seconda possibilità si ottiene con codice come

```
\newcommand{\intervallo}{1,3,...,17}
\newcommand{\lista}{cane,gatto,criceto}
\foreach \i in \intervallo {⟨codice⟩}
\foreach \i in \lista {⟨codice⟩}
```

In altre parole, se la parola chiave `in` è seguita da una graffa aperta, l'argomento denota la lista di valori, altrimenti viene espansa la macro che deve essere passata e che deve espandersi a una lista lecita.

Il ⟨*codice*⟩ normalmente conterrà il "segnaposto" `\i`. Va osservato che la macro, qui denotata con `\i` nell'esempio di sintassi, è del tutto arbitraria e può essere adoperata nel codice da ripetere, dove avrà come espansione il valore corrente preso successivamente dalla lista. Ovviamente bisogna assicurarsi che la macro scelta come segnaposto non venga adoperata nel codice.

Per consentire di adoperare qualsiasi macro come segnaposto, ciascun ciclo viene eseguito in un gruppo. Una tikzpicture è costruita un passo alla volta e ciascuna dichiarazione come `\draw` viene tradotta in linguaggio PGF e aggiunta *globalmente* a una macro interna che poi viene eseguita da `\end{tikzpicture}`.

Il principale difetto di `\foreach` è proprio di eseguire i cicli in un gruppo, anche se ciò consente grande libertà nel dare un nome alla variabile 'locale'. Un altro difetto, meno ovvio, è che gli spazi sono rilevanti. Per esempio

```
\foreach \i in { cane, gatto, criceto }
  {X\i X\par}
```

produrrà

XcaneX
XgattoX
Xcriceto X

da cui si vede che gli spazi iniziali sono ignorati, mentre quelli successivi alla stringa non lo sono. Il problema è mascherato nel caso in cui la lista è numerica, perché la sintassi di TEX ignora spazi dopo costanti nelle assegnazioni.

Un altro difetto, meno evidente a un primo esame, è che il segnaposto non viene espanso nel codice da ripetere. Consideriamo il seguente esempio[1]

```
\def\list{}
\foreach \x in {1,2,3,4} {%
  \gappto\list{\color{color\x}}%
}
```

che ha lo scopo di costruire una macro che, alla fine, contenga

```
\color{color1}\color{color2}%
\color{color3}\color{color4}
```

Ovviamente si tratta di un esempio ridotto al minimo. Se però si esegue questo codice (che richiede etoolbox), la macro costruita avrà come espansione

```
\color{color\x}\color{color\x}
\color{color\x}\color{color\x}
```

e \x non sarà definita. C'è anche da dire che adoperare \list come nome della macro da costruire è un grave errore, ma questo è secondario. Il problema è di espandere \x, ma se provassimo con \expandafter scopriremmo che ne dovremmo scrivere una quantità inverosimile: in questo caso si devono saltare dieci token e quindi occorre scrivere $1023 = 2^{10} - 1$ \expandafter prima di \gappto, $511 = 2^9 - 1$ prima di \list e così via fino a uno prima di r. L'alternativa con \xappto invece di \gappto è ancora peggiore, perché causa una notevole quantità di errori, ma non è tanto meglio

```
\newcommand\mylist{}
\makeatletter
\foreach \x in {1,2,3,4} {
  \protected@xdef\mylist{\color{color\x}}
}
\makeatother
```

anche se è efficace. Non sarebbe troppo difficile correggere il problema, con una seconda macro

```
\newcommand\mylist{}
\makeatletter
\foreach \x in {1,2,3,4} {
  \def\y##1{%
    \g@addto@macro\mylist{%
      \color{color##1}%
    }%
  }
  \expandafter\y\expandafter{\x}
}
\makeatother
```

La macro ausiliaria potrebbe essere definita fuori dal ciclo; così è più semplice e forse più chiaro.

Questo problema affligge anche \@for e \@tfor; a parte la sintassi meno flessibile, si avrebbe il problema descritto prima anche con

```
\newcommand{\mylist}{}
\makeatletter
\@for\next:={1,2,3,4}\do{%
  \g@addto@macro\mylist{%
    \color{color\next}%
  }%
}
\makeatother
```

e il rimedio sarebbe molto simile a quanto visto per \foreach.

Al di fuori dell'impiego in tikzpicture, \foreach è spesso adoperato per costruire macro per "accumulo", come nel caso precedente. Un caso interessante è definire comandi personali per lettere speciali in formule matematiche; l'esempio è

```
\foreach \x in {A,B,...,Z} {%
  \let\xp\expandafter
  \xp\gdef\csname c\x\xp\endcsname\xp{%
      \xp\mathcal\xp{\x}%
  }%
}
```

che può evitare \expandafter con

```
\foreach \x in {A,B,...,Z} {%
  \expandafter\xdef\csname c\x\endcsname
    {\noexpand\mathcal{\x}}%
}
```

Qui si sfrutta l'abilità di \foreach di completare anche liste alfabetiche e non solo numeriche con la notazione ellittica. C'è sempre il problema dell'espansione del segnaposto, ma solo nel testo di sostituzione della macro, perché \csname esegue l'espansione completa da sé.

Non è difficile scrivere lo stesso con \@whilenum:

```
\makeatletter
\count@=\z@
\@whilenum\count@<26 \do{%
  \advance\count@ by \@ne
  \expandafter
  \edef\csname c\@Alph{\count@}\endcsname
    {\noexpand\mathcal{\@Alph{\count@}}}%
}
\makeatother
```

anche se questo potrebbe non funzionare se sono caricati certi moduli di babel che modificano la definizione di \@Alph e il metodo sarebbe più sicuro con

1. http://tex.stackexchange.com/q/211234

```
\makeatletter
\count@=`A \advance\count@\m@ne
\@whilenum\count@<`Z \do{%
  \advance\count@ by \@ne
  \begingroup\lccode`x=\count@
  \lowercase{\endgroup
    \@namedef{cx}{\mathcal{x}}%
  }%
}
\makeatother
```

Non c'è dubbio che `\foreach` sia più semplice, anche se non troppo.

3 Liste

Senza entrare troppo nei dettagli tecnici, una lista ordinata può essere realizzata in due modi: con un separatore fra un elemento e il successivo oppure dando ciascun elemento come argomento di una macro:

```
\def\listaA{cane,gatto,criceto}
\def\listaB{%
  \foo{cane}%
  \foo{gatto}%
  \foo{criceto}%
}
```

Sia `\@for` sia `\foreach` accettano solo liste del primo tipo; è facile capire però che le liste del secondo tipo sono molto più flessibili, perché il significato della macro adoperata per delimitare ciascun elemento può cambiare quando ci pare.

Per esempio, possiamo facilmente trasformare la seconda lista nella prima, ma con elementi separati da ; invece che da una virgola, con

```
\def\listaB{%
  \foo{cane}%
  \foo{gatto}%
  \foo{criceto}%
}
\begingroup
\def\foo#1{;\unexpanded{#1}}
\def\gobble#1#2\gobble{\unexpanded{#2}}
\edef\next{\listaB}
\edef\next{\endgroup
  \def\noexpand\listaC{%
    \expandafter\gobble\next\gobble
  }%
}\next
```

Oppure possiamo contare gli elementi della lista con

```
\newcount\listcount
\begingroup
\listcount=0
\def\foo#1{%
  \global\advance\listcount by 1
}
\listaB
\endgroup
```

Se non desideriamo assegnazioni globali, si può adoperare un registro temporaneo

```
\newcount\listcount
\begingroup
\count0=0
\def\foo#1{\advance\count0 by 1 }
\listaB
\expandafter\endgroup
\expandafter\listcount\the\count0\relax
```

Qui il valore di `\count0` è ottenuto prima che il gruppo venga terminato; solo in seguito `\endgroup` riassegna al registro il valore precedente.

Non è difficile fare lo stesso con `\foreach` operando su `\listaA`, basta ricordare che occorrono assegnazioni globali:

```
\newcount\listcount
\foreach \x in \listaA {%
  \global\advance\listcount by 1
}
```

oppure usare un contatore LaTeX:

```
\newcounter{listcount}
\foreach \x in \listaA {%
  \stepcounter{listcount}%
}
```

Qui si sfrutta il fatto che `\stepcounter` esegue un'assegnazione globale.

Una lista può essere costruita passo passo; per le liste del secondo tipo la faccenda è semplice:

```
\newcommand{\apptolistII}[2]{%
  \toks0=\expandafter{#1}%
  \edef#1{%
    \the\toks0
    \unexpanded{\foo{#2}}%
  }%
}
\newcommand{\listaD}{}% init
\apptolistII{\listaD}{cane}
\apptolistII{\listaD}{gatto}
\apptolistII{\listaD}{criceto}
```

Per quelle del primo tipo è un po' più complicato perché occorre distinguere se la lista è vuota:

```
% syntactic sugar
\providecommand{\expandonce}[1]{%
  \unexpanded\expandafter{#1}%
}
\newcommand\emptylist{}

\newcommand{\apptolistI}[2]{%
  \ifx#1\emptylist
    \def#1{#2}%
  \else
    \edef#1{%
      \expandonce{#1},%
      \unexpanded{#2}%
    }
  \fi
}
\newcommand{\listaE}{}% init
\apptolistI{\listaE}{cane}
\apptolistI{\listaE}{gatto}
\apptolistI{\listaE}{criceto}
```

Più complicato è poi adoperare una lista. Supponiamo che si voglia stampare la lista della frutta che si è mangiata. Per ogni pasto, si può eseguire `\addtolistI{\fruits}{`⟨*frutto*⟩`}` e terminare con

```
Ho mangiato
\foreach \x in \fruits {%
  \x, % spazio
}.
```

con il problema di avere una virgola di troppo. Possibile soluzione: contare prima il numero di elementi e produrre una virgola se non siamo all'ultimo passo:

```
\documentclass{article}
\usepackage{pgffor}

\newcounter{fruits}
\newcommand{\fruits}{%
  pera,
  mela,
  banana,
  pesca%
}

\begin{document}

Ho mangiato
\setcounter{fruits}{0}%
\foreach \x in \fruits{%
  \stepcounter{fruits}%
}%
\foreach \x [count=\y]
  in \fruits{%
  \x
  \ifnum\y<\value{fruits}%
    , % spazio
  \fi
}.

\end{document}
```

L'opzione `[count=\y]` fa in modo che all'interno di ciascun ciclo, `\y` si riferisca al numero dell'iterazione.

Per esercizio, si estenda il codice in modo che ci sia una 'e' tra banana e pesca, invece della virgola come nell'esempio che segue.

> Ho mangiato pera, mela, banana e pesca.

Se però si volesse seguire la convenzione detta *Oxford comma*, secondo la quale si dovrebbe scrivere

> mela e pera
> mela, pera, banana, e pesca

il codice dovrebbe essere molto più complicato.

È venuto il momento di procurarsi una cassetta degli attrezzi più fornita, cioè expl3.

Prima un po' di *syntactic sugar* per maneggiare liste. Si supporrà che i pacchetti expl3 e xparse siano caricati (basta il secondo). Per maggiori informazioni si consultino The LaTeX3 Project (2016b,a,c).

```
% nel preambolo
\ExplSyntaxOn
% definire una lista
\NewDocumentCommand{\newlist}{m}
 {
  \manual_list_new:n { #1 }
 }
% azzerare una lista
\NewDocumentCommand{\clearlist}{m}
 {
  \manual_list_clear:n { #1 }
 }
% aggiungere uno o più elementi
\NewDocumentCommand{\addtolist}{mm}
 {
  \clist_map_inline:nn { #2 }
   {
    \manual_list_append:nn { #1 } { ##1 }
   }
 }
\NewDocumentCommand{\uselist}{mmmO{#3}}
 {
  \seq_use:cnnn { g_manual_list_#1_seq }
   { #3 } { #2 } { #4 }
 }
% comandi interni
\cs_new_protected:Nn \manual_list_new:n
 {
  \seq_new:c { g_manual_list_#1_seq }
 }
\cs_new_protected:Nn \manual_list_clear:n
 {
  \seq_clear:c { g_manual_list_#1_seq }
 }
\cs_new_protected:Nn \manual_list_append:nn
 {
  \seq_gput_right:cn { g_manual_list_#1_seq }
   { #2 }
 }
\ExplSyntaxOff

% nel documento
\newlist{fruits}
\addtolist{fruits}{mela,pera}

Ho mangiato
\uselist{fruits}{, }{ e }.

\addtolist{fruits}{banana}
\addtolist{fruits}{pesca}

Ho mangiato
\uselist{fruits}{, }{ e }.

Ho mangiato
\uselist{fruits}{, }{ e }[, e ].
```

La lista viene chiamata "per nome"; il secondo argomento di `\uselist` è il separatore normale, il terzo il separatore fra gli ultimi due elementi; se viene specificato l'argomento facoltativo, il terzo argomento viene adoperato solo se la lista è di due soli elementi, altrimenti fra gli ultimi due apparirà l'argomento facoltativo. Si verifichi che il codice produce

Ho mangiato mela e pera
Ho mangiato mela, pera, banana e pesca.
Ho mangiato mela, pera, banana, e pesca.

Per divagare un po', si noti come, nella specificazione degli argomenti di `\uselist`, il valore di default per l'argomento facoltativo è identico al valore del terzo argomento. Una specificazione equivalente sarebbe

```
\NewDocumentCommand{\uselist}{mmmo}
 {
  \IfNoValueTF{#4}
   {
    \seq_use:cnnn { g_manual_list_#1_seq }
     { #3 } { #2 } { #3 }
   }
   {
    \seq_use:cnnn { g_manual_list_#1_seq }
     { #3 } { #2 } { #4 }
   }
 }
```

ed è chiaro il risparmio in termini di codice. Il metodo più lungo deve essere usato se si vuole che la macro sia espandibile:

```
\DeclareExpandableDocumentCommand{\uselist}
 {mmom}
 {
  \IfNoValueTF{#3}
   {
    \seq_use:cnnn { g_manual_list_#1_seq }
     { #4 } { #2 } { #4 }
   }
   {
    \seq_use:cnnn { g_manual_list_#1_seq }
     { #4 } { #2 } { #3 }
   }
 }
```

ma la sintassi sarà diversa, perché l'argomento facoltativo non può andare per ultimo in questa situazione.

La macro `\addtolist` accetta anche più di un elemento alla volta, rimuovendo gli spazi prima e dopo; quindi sono del tutto equivalenti

```
\addtolist{fruits}{mela,pera}
\addtolist{fruits}{mela, pera}
\addtolist{fruits}{mela , pera}
\addtolist{fruits}{ mela, pera }
```

e le altre possibili variazioni sul tema.

La struttura dati adoperata è quella delle "liste del secondo tipo", anche se non è necessario saperlo.

Viene subito in mente l'idea di costruire liste da dare in pasto a `\foreach` con metodi simili. In tal caso, basta adoperare la struttura dati *clist* e sostituire `\seq` con `\clist` e `_seq` con `_clist` nel codice mostrato.

4 Un nuovo `\foreach`, anzi due

La sintassi di `\foreach` per indicare cicli basati su interi è comoda, ma non più di una sintassi del tipo

```
\nforeach{
  start=1,
  step=2,
  end=13
}{<codice>}
```

invece di

```
\foreach \x in {1,3,...,13}{<codice>}
```

dove l'opzione `step` è facoltativa, nel caso il passo scelto sia 1.

Il lettore attento noterà una differenza importante: nel codice per `\nforeach` non è menzionata la 'variabile' che riceve il valore corrispondente in ciascun ciclo. La spiegazione è semplice: invece di

```
\begin{tikzpicture}
 \foreach \x in {1,...,10}{
   \draw (\x,0) circle (0.4cm);
 }
\end{tikzpicture}
```

(il passo è 1), possiamo scrivere

```
\begin{tikzpicture}
\nforeach{start=1,end=10}{
  \draw (#1,0) circle (0.4cm);
}
\end{tikzpicture}
```

La variabile è semplicemente indicata con `#1`, che diventerà `##1` in un ciclo annidato, `####1` al terzo livello e così via raddoppiando.

Potremmo facilmente definire una versione con argomenti invece della sintassi chiave-valore con

```
\NewDocumentCommand{\simpleforeach}{mmmm}
 {
  \int_step_inline:nnnn { #1 } { #2 } { #3 }
   { #4 }
 }
```

dove gli argomenti sono, nell'ordine, il punto di partenza, il passo, il punto finale e il codice da eseguire. Tuttavia la sintassi proposta prima permette di estendere la macro per avere a disposizione cicli basati su numerazione alfabetica o incrementi su numeri a virgola mobile. Un esempio:

```
\nforeach{start=0,end=10}{%
  $2^{#1}=\fpeval{2^#1}$, %
}
```

produce

$2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$,

dove `\fpeval` è un'ovvia versione 'utente' di `\fp_eval:n`.

Il caso di numeri in virgola mobile:

Tabella 1: Logaritmi naturali

$\log 1 = 0$	$\log 2 = 0.6931$	$\log 3 = 1.0986$	$\log 4 = 1.3863$	$\log 5 = 1.6094$
$\log 1.1 = 0.0953$	$\log 2.1 = 0.7419$	$\log 3.1 = 1.1314$	$\log 4.1 = 1.411$	$\log 5.1 = 1.6292$
$\log 1.2 = 0.1823$	$\log 2.2 = 0.7885$	$\log 3.2 = 1.1632$	$\log 4.2 = 1.4351$	$\log 5.2 = 1.6487$
$\log 1.3 = 0.2624$	$\log 2.3 = 0.8329$	$\log 3.3 = 1.1939$	$\log 4.3 = 1.4586$	$\log 5.3 = 1.6677$
$\log 1.4 = 0.3365$	$\log 2.4 = 0.8755$	$\log 3.4 = 1.2238$	$\log 4.4 = 1.4816$	$\log 5.4 = 1.6864$
$\log 1.5 = 0.4055$	$\log 2.5 = 0.9163$	$\log 3.5 = 1.2528$	$\log 4.5 = 1.5041$	$\log 5.5 = 1.7047$
$\log 1.6 = 0.47$	$\log 2.6 = 0.9555$	$\log 3.6 = 1.2809$	$\log 4.6 = 1.5261$	$\log 5.6 = 1.7228$
$\log 1.7 = 0.5306$	$\log 2.7 = 0.9933$	$\log 3.7 = 1.3083$	$\log 4.7 = 1.5476$	$\log 5.7 = 1.7405$
$\log 1.8 = 0.5878$	$\log 2.8 = 1.0296$	$\log 3.8 = 1.335$	$\log 4.8 = 1.5686$	$\log 5.8 = 1.7579$
$\log 1.9 = 0.6419$	$\log 2.9 = 1.0647$	$\log 3.9 = 1.361$	$\log 4.9 = 1.5892$	$\log 5.9 = 1.775$

```
\nforeach{
  type=fp,
  start=1.1,
  step=0.1,
  end=1.9
}
{ciclo~#1, }
```

ha come risultato

ciclo 1.1, ciclo 1.2, ciclo 1.3, ciclo 1.4, ciclo 1.5, ciclo 1.6, ciclo 1.7, ciclo 1.8, ciclo 1.9,

Nella tabella 1 c'è il risultato di

```
\nforeach{
  start=1,
  end=5,
  step=1,
}{%
  \nforeach{
    type=fp,
    start=0,
    end=0.9,
    step=0.1
  }{%
    $\log\fpeval{#1+##1}=
     \fpeval{round(ln(#1+##1),4)}$%
    \par
   }%
}
```

cioè la tabella dei logaritmi dei numeri da 1 a 5.9 con un passo di 0.1 arrotondati alla quarta cifra decimale. Si sarebbe potuto scrivere un unico ciclo, ma in questo modo si dimostra come i cicli possano essere annidati. Non occorre l'opzione `type=integers` nel caso la variabile sia intera, ma può essere scritta per maggiore chiarezza.

Se vogliamo definire tutti insieme i comandi come `\calA` per `\mathcal{A}`, problema che abbiamo già risolto prima,

```
% syntactic sugar
\newcommand\namenewcommand[1]{%
  \expandafter\newcommand\csname #1\endcsname
}
\nforeach{type=Alph,start=A,end=Z}{%
  \namenewcommand{cal#1}{\mathcal{#1}}%
}
$\calA$, $\calC$, $\calZ$.
```

e il risultato è

$\mathcal{A}$, $\mathcal{C}$, $\mathcal{Z}$.

Il codice è nelle tabelle 2 e 3. Non è possibile commentarlo completamente, ma l'idea fondamentale è che con

`\nforeach{`⟨*opzioni*⟩`}{`⟨*codice*⟩`}`

il ⟨*codice*⟩ diventa il testo di sostituzione di una macro con un argomento il cui nome dipende dal livello di annidamento e che viene applicata a ciascun numero (o lettera) che viene passato successivamente dalla funzione che ripete i cicli. Il livello di annidamento è conservato in una variabile globale che viene incrementata a ogni chiamata e riportata indietro quando il ciclo termina. A dire il vero la faccenda è un po' più complessa, ma non è il caso di entrare nei dettagli. Per i cicli base, con interi, viene adoperata la funzione `\int_step_inline:nnnn` già fornita da expl3, per i cicli con numeri a virgola mobile e alfabetici l'intera serie viene memorizzata in una variabile di tipo *sequence* e si impiega la funzione `\seq_map_inline:Nn`.

Nel manuale di Ti*k*Z, pagina 902, c'è l'esempio

```
\foreach \x in {0,0.1,...,0.5} {\x, }
```

che produce

0, 0.1, 0.20001, 0.30002, 0.40002,

Con `\nforeach` avremmo

```
\nforeach{
  type=fp,
  start=0,
  step=0.1,
  end=0.5}{#1, }
```

che produce

0, 0.1, 0.2, 0.3, 0.4, 0.5,

e si vede come la precisione sia notevolmente maggiore.

Il codice è in una versione preliminare e richiederà parecchi perfezionamenti; per esempio non è ancora possibile sapere in quale passo del ciclo si sia, ma verrà aggiunta una chiave del tipo `cyclecount` e, con `cyclecount=var` si potrà adoperare `\cyclecount{var}` per ottenere (dopo l'espansione) il numero corrispondente al numero del ciclo attuale.

Per i cicli basati su liste è prevista una macro diversa, chiamata `\lforeach`; sarebbe forse possibile adoperare una sola macro, ma la mia opinione è che sia meglio separarle. Le opzioni sono molto diverse.

La macro ha una variante asterisco per indicare che il primo argomento è a sua volta una macro che contiene la lista da elaborare; a parte questo, la sintassi è

`\lforeach[`⟨*opzione*⟩`]{`⟨*lista*⟩`}{`⟨*azione*⟩`}`

dove, come per `\nforeach`, ci si può riferire al valore attuale nel ciclo con `#1`. Quindi

```
\lforeach{cane,gatto,criceto}{Animale: #1\par}
```

produrrà

Animale: cane
Animale: gatto
Animale: criceto

Le opzioni sono `single`, `double`, `triple` e `format`. La prima è implicita e indica che è presente solo una variabile, come nell'esempio precedente. Per illustrare l'opzione `double` si consideri

```
\lforeach[double]{
  cane/bau,gatto/miao,criceto/squit
}{Il #1 fa #2\par}
```

che produce

Il cane fa bau
Il gatto fa miao
Il criceto fa squit

Questo ciclo ha due variabili, i cui valori sono indicati separandoli con una barra. L'opzione `triple` si applica in modo analogo a liste i cui elementi siano del tipo `A/B/C`. L'opzione `format` permette invece di indicare le variabili (massimo nove) in modo molto libero:

```
\begin{tikzpicture}
\lforeach[format=(#1;#2)]{
  (0;0),(0;1),(1;1),(1;0)
}{\draw (#1,#2) circle (0.1cm);}
\end{tikzpicture}
```

per ottenere

Si noti che le coordinate non possono essere separate da una virgola, per non confondere il *parser* della lista. Con `\foreach` si potrebbe scrivere lo stesso codice come

```
\begin{tikzpicture}
\foreach \x in {(0,0),(0,1),(1,1),(1,0)}{
  \draw \x circle (0.1cm);
}
\end{tikzpicture}
```

che però limita la possibilità di adoperare separatamente le due variabili. Se la lista di coordinate è ottenuta da qualche sorgente esterna, non sarà difficile modificarla in modo da poter adoperare `\lforeach`.

Naturalmente è possibile annidare `\lforeach` e `\nforeach` in ogni combinazione:

```
\lforeach{cane,gatto,criceto}{%
  \nforeach{start=1,end=3}{%
    #1--##1 }\par
}
```

produce

cane–1 cane–2 cane–3
gatto–1 gatto–2 gatto–3
criceto–1 criceto–2 criceto–3

Riferimenti bibliografici

Braams, J., Carlisle, D., Jeffrey, A., Lamport, L., Mittelbach, F., Rowley, C. e Schöpf, R. (2016). «The LaTeX 2ε sources, 2016/03/31 patch level 3». `texdoc source2e`.

Goossens, M., Mittelbach, F. e Samarin, A. (1994). *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA.

Tantau, T. (2015). «The TikZ and PGF packages (version 3.0.1a)». `texdoc tikz`.

The LaTeX3 Project (2016a). «The LaTeX3 interfaces». `texdoc interface3`.

— (2016b). «The expl3 package and LaTeX3 programming». `texdoc expl3`.

— (2016c). «The xparse package; document command parser». `texdoc xparse`.

▷ Enrico Gregorio
Dipartimento di Informatica
Università di Verona
`enrico DOT gregorio AT univr DOT it`

Tabella 2: Il codice per `\nforeach` e `\lforeach`

```
\ExplSyntaxOn
\providecommand\fpeval{\fp_eval:n}

\NewDocumentCommand{\nforeach}{ m +m }
 {
  \int_gincr:N \g__manual_foreach_map_int
  \tl_clear:N \l__manual_nforeach_type_tl
  \keys_set:nn { manual/nforeach }
   {
    type=integers,start = 1, step = 1, end = 0,
   }
  \keys_set:nn { manual/nforeach } { #1 }
  \__manual_nforeach_exec:n { #2 }
  \int_gdecr:N \g__manual_foreach_map_int
 }

\int_new:N \g__manual_foreach_map_int
\int_new:N \g__manual_fp_map_int
\tl_new:N \l__manual_nforeach_type_tl

\keys_define:nn { manual/nforeach }
 {
  type .choice:,
  type .value_required:n = true,
  type/integers .code:n = \tl_set:Nn \l__manual_nforeach_type_tl { integers },
  type/fp       .code:n = \tl_set:Nn \l__manual_nforeach_type_tl { fp },
  type/alph     .code:n = \tl_set:Nn \l__manual_nforeach_type_tl { alph },
  type/Alph     .code:n = \tl_set:Nn \l__manual_nforeach_type_tl { Alph },
  start .tl_set:N = \l__manual_nforeach_start_tl,
  step  .tl_set:N = \l__manual_nforeach_step_tl,
  end   .tl_set:N = \l__manual_nforeach_end_tl,
 }

\cs_new_protected:Nn \__manual_nforeach_exec:n
 {
  \str_case:Vn \l__manual_nforeach_type_tl
   {
    {integers}{\__manual_nforeach_exec_integers:n { #1 }}
    {fp}      {\__manual_nforeach_exec_fp:n { #1 }}
    {alph}    {\__manual_nforeach_exec_alph:Nn \int_to_alph:n { #1 }}
    {Alph}    {\__manual_nforeach_exec_alph:Nn \int_to_Alph:n { #1 }}
   }
 }
\cs_generate_variant:Nn \str_case:nn { V }

\cs_new_protected:Nn \__manual_nforeach_exec_integers:n
 {
  \int_step_inline:nnnn
   { \l__manual_nforeach_start_tl }
   { \l__manual_nforeach_step_tl }
   { \l__manual_nforeach_end_tl }
   { #1 }
 }
\cs_new_protected:Nn \__manual_nforeach_exec_alph:Nn
 {
  \cs_set:cn { __manual_nforeach_alph_ \int_use:N \g__manual_foreach_map_int :n } { #2 }
  \cs_generate_variant:cn
   { __manual_nforeach_alph_ \int_use:N \g__manual_foreach_map_int :n }
   { f }
  \int_step_inline:nnnn
   { \int_from_alph:f { \l__manual_nforeach_start_tl } }
   { \l__manual_nforeach_step_tl }
   { \int_from_alph:f { \l__manual_nforeach_end_tl } }
   {
    \use:c { __manual_nforeach_alph_ \int_use:N \g__manual_foreach_map_int :f }
     { #1 { ##1 } }
   }
 }
\cs_generate_variant:Nn \cs_generate_variant:Nn { c }
\cs_generate_variant:Nn \int_from_alph:n { f }

\cs_new_protected:Nn \__manual_nforeach_exec_fp:n
 {
  \manual_fp_step_inline:nnnn
   { \l__manual_nforeach_start_tl }
   { \l__manual_nforeach_step_tl }
   { \l__manual_nforeach_end_tl }
   { #1 }
 }
```

Tabella 3: Il codice per `\nforeach` e `\lforeach` (continua)

```
% a replacement for \fp_step_inline:nnnn
\seq_new:N \l__manual_fp_step_seq
\fp_new:N \l__manual_fp_step_start_fp

\cs_new_protected:Nn \manual_fp_step_inline:nnnn
 {
  \int_gincr:N \g__manual_fp_map_int
  \seq_clear_new:c { l__manual_fp_step_ \int_use:N \g__manual_fp_map_int _seq }
  \fp_compare:nTF { #2 < \c_zero_fp }
   {
    \__manual_fp_step_make_neg:nnn { #1 } { #2 } { #3 }
   }
   {
    \__manual_fp_step_make_pos:nnn { #1 } { #2 } { #3 }
   }
  \seq_map_inline:cn { l__manual_fp_step_ \int_use:N \g__manual_fp_map_int _seq } { #4 }
  \int_gdecr:N \g__manual_fp_map_int
 }
\cs_new_protected:Nn \__manual_fp_step_make_neg:nnn
 {
  \fp_set:Nn \l__manual_fp_step_start_fp { #1 }
  \fp_do_while:nn { \l__manual_fp_step_start_fp >= #3 }
   {
    \seq_put_right:cx { l__manual_fp_step_ \int_use:N \g__manual_fp_map_int _seq }
     { \fp_eval:n { \l__manual_fp_step_start_fp } }
    \fp_add:Nn \l__manual_fp_step_start_fp { #2 }
   }
 }
\cs_new_protected:Nn \__manual_fp_step_make_pos:nnn
 {
  \fp_set:Nn \l__manual_fp_step_start_fp { #1 }
  \fp_do_while:nn { \l__manual_fp_step_start_fp <= #3 }
   {
    \seq_put_right:cx { l__manual_fp_step_ \int_use:N \g__manual_fp_map_int _seq }
     { \fp_eval:n { \l__manual_fp_step_start_fp } }
    \fp_add:Nn \l__manual_fp_step_start_fp { #2 }
   }
 }

\NewDocumentCommand{\lforeach}{ s O{} m +m }
 {
  \IfBooleanTF{#1}
   {
    \manual_lforeach:non { #2 } { #3 } { #4 }
   }
   {
    \manual_lforeach:nnn { #2 } { #3 } { #4 }
   }
 }

\cs_new_protected:Nn \manual_lforeach:nnn
 {
  \int_gincr:N \g__manual_foreach_map_int
  \keys_set:nn { manual/lforeach } { single }
  \keys_set:nn { manual/lforeach } { #1 }
  \clist_set:Nn \l__manual_lforeach_list_clist { #2 }
  \__manual_lforeach_define:n { #3 }
  \clist_map_inline:Nn \l__manual_lforeach_list_clist
   {
    \use:c { __manual_lforeach_ \int_use:N \g__manual_foreach_map_int _action:w } ##1 \q_stop
   }
  \int_gdecr:N \g__manual_foreach_map_int
 }
\cs_generate_variant:Nn \manual_lforeach:nnn { no }

\cs_new_protected:Nn \__manual_lforeach_define:n
 {
  \exp_last_unbraced:NcV
   \cs_set:Npn
   { __manual_lforeach_ \int_use:N \g__manual_foreach_map_int _action:w }
   \l__manual_lforeach_format_tl
   \q_stop
   {#1}
 }

\keys_define:nn { manual/lforeach }
 {
  format .tl_set:N = \l__manual_lforeach_format_tl,
  single .code:n = \tl_set:Nn \l__manual_lforeach_format_tl { ##1 },
  double .code:n = \tl_set:Nn \l__manual_lforeach_format_tl { ##1/##2 },
  triple .code:n = \tl_set:Nn \l__manual_lforeach_format_tl { ##1/##2/##3 },
 }
\ExplSyntaxOff
```