Funzioni e expl3

Enrico Gregorio

Sommario

Esporremo alcune nozioni relative alle funzioni nel linguaggio expl3: il loro ruolo, come si definiscono, le varianti; per le applicazioni, si darà anche un cenno alle variabili.

Abstract

In this tutorial we discuss expl3 functions, their role, definition and variants, touching also variables.[1]

1 Introduzione

Il termine *funzione* non è usato nel modo di esprimersi usuale in LaTeX. Invece è un concetto molto importante nel linguaggio expl3, che sarà alla base del futuro LaTeX3. Il linguaggio non aveva un nome preciso fino a un paio d'anni fa, quando fu deciso di chiamarlo expl3 come il pacchetto che ne permette l'uso. Nelle versioni attuali di LaTeX il pacchetto è stato incorporato nel nucleo.

In questo linguaggio, si ha cura di distinguere fra *variabili* e *funzioni*. Alle variabili si dà un valore che può cambiare durante la compilazione, mentre le funzioni eseguono una certa azione.

Un tipo speciale di variabili sono le *costanti*, il cui valore non deve cambiare durante la compilazione di un documento. Vero, il nome sembra contraddittorio, ma i matematici sono abituati a questo tipo di estensione della terminologia: chi non è matematico, non si preoccupi e vada avanti, limitandosi a sorridere delle bizzarrie del modo di pensare dei matematici.

Saremo principalmente interessati alle funzioni; tuttavia le variabili possono essere quello che passiamo alle funzioni, quindi ci servirà qualche nozione anche su queste.

Facciamo un esempio con concetti 'tradizionali'. Le classi di documento standard, e anche molte altre, possiedono il *comando* `\title`. Come funziona? Questo stesso articolo ha

```
\title{Funzioni e \expliii}
```

all'inizio. Quando LaTeX elabora questa istruzione, eseguirà qualcosa come

```
\gdef\@title{Funzioni e \expliii}
```

così da poter usare `\@title` durante la successiva elaborazione di `\maketitle`.

C'è una grande differenza concettuale tra `\title` e `\@title`. Il primo esegue un'azione, il secondo è semplicemente un contenitore. Nella terminologia di expl3, il primo sarà una funzione, il secondo una variabile: l'azione della funzione è di registrare qualcosa nella variabile specificata.

Al livello dell'utente, la distinzione non è così netta. Se scrivo

```
\newcommand{\CC}{\mathbb{C}}
```

sto dicendo che `\CC` è una funzione o una variabile? Non è veramente importante deciderlo, perché è piuttosto un'*abbreviazione* dell'utente e a questo livello la distinzione è quasi irrilevante.

La distinzione diventa rilevante quando si *programma*. Nella programmazione expl3 'corretta', ci sarà un comando a livello utente che chiama una funzione, la quale eseguirà un'opportuna azione:

```
\NewDocumentCommand{\title}{m}
 {
  \example_title:n { #1 }
 }
\tl_new:N \g_example_title_tl
\cs_new_protected:Nn \example_title:n
 {
  \tl_gset:Nn \g_example_title_tl { #1 }
 }
```

In seguito, il comando a livello utente `\maketitle` farà uso del valore registrato nella variabile, adoperando certamente altre funzioni. Nel seguito, *comando* si riferirà a ciò che si vede a livello utente, come `\CC` di prima.

Questo esempio (immaginario) mostra un buon numero dei concetti che discuteremo. Definiamo un comando utente in termini di una funzione; questa funzione ha un argomento (il titolo) e il suo mestiere è di impostare una ben determinata variabile con il valore specificato. La variabile va dichiarata prima di essere usata, tipicamente prima che si definisca una funzione che la impieghi:

- `\title` è un comando per l'utente, fuori dallo scopo di questo articolo;
- `\example_title:n` è una funzione;
- `\g_example_title_tl` è una variabile.

2 Convenzioni sui nomi

Un problema comune con TeX è che non possiede il concetto di *namespace*, che si diffuse nell'informatica parecchio più tardi. Conflitti di nomi erano

1. This article is a translation of a forthcoming paper in TUGboat.

molto frequenti ai primi tempi di LaTeX e continuano ad apparire di tanto in tanto. Potrebbe essere accattivante per un pacchetto adoperare `\@x` e `\@y` come coordinates, tuttavia gli autori del pacchetto dovrebbero rendersi conto che se gli piace un nome semplice è probabile che sia già piaciuto a qualcun altro.

Le variabili devono avere un nome della forma

`\l_`⟨*prefisso*⟩`_`⟨*nome proprio*⟩`_`⟨*tipo*⟩
`\g_`⟨*prefisso*⟩`_`⟨*nome proprio*⟩`_`⟨*tipo*⟩
`\c_`⟨*prefisso*⟩`_`⟨*nome proprio*⟩`_`⟨*tipo*⟩

dove le varie parti sono *importanti* e *necessarie*:

- `l`, `g` o `c` dichiarano che la variabile that the variable è locale, globale o costante rispettivamente;
- ⟨*prefisso*⟩ dev'essere una stringa di lettere che identifichi univocamente il pacchetto o il codice nel documento;
- ⟨*nome proprio*⟩ è un'arbitraria stringa di lettere che è possibile dividere in parti separate da un *underscore*;
- ⟨*tipo*⟩ è il tipo di variabile.

I tipi di variabili più comuni sono:[2]

- `tl`, per *token list*;
- `seq`, per *sequence*;
- `clist`, per *comma list*;
- `prop`, per *property list*;
- `int`, per *integer*;
- `dim`, per *dimension*;
- `box`, per *box*;
- `fp`, per *floating point.*

Ce ne sono parecchie altre, ma lo scopo di quest'introduzione è discutere le funzioni, quindi non menzionerò le variabili più esoteriche, se non ce ne sarà bisogno.

I nomi delle funzioni sono simili:

`\`⟨*prefisso*⟩`_`⟨*nome proprio*⟩`:`⟨*segnatura*⟩

Il ⟨*prefisso*⟩ e il ⟨*nome proprio*⟩ sono come prima; la ⟨*segnatura*⟩ deve essere spiegata. Può essere un'arbitraria stringa di caratteri tra

`N n e f x T F c V v o w`

ciascuno dei quali, eccetto `w`, denota un argomento della funzione.

Le funzioni matematiche possono dipendere da uno o più argomenti (anche zero, quando sono funzioni costanti) e lo stesso vale per le funzioni di expl3. Lo scopo della segnatura è di dire precisamente da quanti argomenti dipende una funzione e anche il loro tipo. Per esempio, la funzione di uso frequente

```
\seq_set_split:Nnn
```

ha tre argomenti, uno di tipo `N` e due di tipo `n` *in quest'ordine*. Un argomento di tipo `N` dev'essere un solo token, la cui natura dipende ovviamente dalla funzione; nel caso in esame, dev'essere una variabile *sequence*. Un argomento di tipo `n` dev'essere una lista di token tra graffe. Per esempio, il titolo del documento è un argomento di tipo `n` a `\title`. Esempio un po' tirato per i capelli, ma dovrebbe spiegare il concetto.

Il tipo `w` è un'eccezione perché non dice quasi nulla, ma solo che gli argomenti della funzione sono *strani*, in inglese *weird*, e ci si deve riferire alla documentazione del pacchetto per sapere come regolarsi. In generale, argomenti di tipo `w` dovrebbero comparire solo in funzioni di basso livello.

Una chiamata della funzione menzionata prima sarà qualcosa come

```
\seq_set_split:Nnn
  \l_example_test_seq
  { || }
  { a || b || c }
```

(quasi certamente su una sola riga, qui è divisa per via del formato tipografico). Non ci interessa veramente sapere che cosa fa questo codice: questa funzione normalmente viene chiamata nel corso di altre elaborazioni e riceve gli argomenti da altre funzioni. L'aspetto importante è vedere che gli argomenti seguono le convenzioni: il primo è un singolo token, gli altri due liste di token tra graffe.

Una funzione può anche non avere argomenti, ma i due punti della segnatura sono necessari comunque. Per la verità, TeX potrebbe non protestare se definite una funzione con un nome non conforme alle convenzioni; seguirle aiuta a evitare conflitti e ad avere codice più leggibile. Una tipica funzione senza argomenti è `\scan_stop:`, nient'altro che il nostro vecchio amico `\relax`. Il nome in expl3 è forse meno poetico, ma esprime bene qual è lo scopo principale della funzione.

3 Come definire funzioni

Ci sono molte funzioni il cui mestiere è definire funzioni. Tutte, però, hanno lo stesso prefisso `cs` e le principali sono

```
\cs_new:Nn
\cs_new_protected:Nn
```

2. Non traduco i nomi perché avrebbe poco senso.

Discuteremo poi le altre. Secondo le convenzioni, queste due hanno due argomenti: il primo è un singolo token, cioè il nome della funzione da definire, il secondo è il *testo di sostituzione* (replacement text), cioè il codice che sarà sostituito quando la funzione è chiamata.

Sebbene expl3 cerchi di emulare un linguaggio funzionale, è pur sempre basato su TEX che conosce solo primitive, macro e registri. Questo dovrebbe essere sempre tenuto presente quando si programma. D'altra parte, il linguaggio permette costruzioni più semplici che evitano le goffe (o divertenti, a seconda dello spirito del programmatore) catene di `\expandafter` o `\noexpand`, che sono spesso difficili da seguire.

Un esempio semplice potrebbe essere la funzione interna per manipolare il titolo del documento:

```
\cs_new_protected:Nn \example_title:n
 {
  \tl_gset:Nn \g_example_title_tl { #1 }
 }
```

Siccome la funzione ha segnatura `:n` (per precisione, i due punti non sono parte della segnatura, ma sono convenienti come marcatore), expl3 sa che il testo di sostituzione può contenere `#1` per riferirsi all'argomento specificato alla chiamata della funzione.

Perché adopero l'istruzione con `protected` e non quella più semplice? Perché la nostra funzione deve *impostare* il valore di una variabile. Si tratta di un aspetto che ha frustrato generazioni di programmatori LATEX e ha richiesto la distinzione tra comandi robusti e fragili. Al giorno d'oggi il problema è meno rilevante perché quasi tutti i comandi fragili sono stati 'resi robusti', ma si può ancora presentare.

Qual è il problema? Se, in LATEX tradizionale, facciamo qualcosa come

```
\newcommand{\foo}[1]{%
  \renewcommand{\baz}{#1}%
}
```

che è il modo normale di immagazzinare qualcosa in un contenitore e per qualche motivo `\foo` finisce in `\write` o `\edef`, anche nelle forme `\protected@write` o `\protected@edef`, apparirà una lunga lista di messaggi di errore. Il modo tradizionale di evitare la faccenda è di adoperare `\DeclareRobustCommand` invece di `\newcommand`.

Ogni funzione il cui mestiere comprenda impostare variabili o chiamare altre funzioni 'protette' deve, in generale essere essa stessa 'protetta'. In caso di dubbio, si adotti la protezione.

Attenzione! La segnatura della funzione da definire può solo contenere i caratteri `N` o `n`. Per la verità, anche `T` o `F` sarebbero permessi, ma questo è un caso speciale che toccheremo più avanti. Come entrano allora gli altri caratteri ammessi nelle segnature? Ottima domanda!

3.1 Generare varianti

Supponiamo di dover scrivere una funzione generica per impostare variabili `tl` in modo che contengano materiale da usare in seguito. L'interfaccia utente avrà i comandi `\setvar` per impostare la variabile e `\usevar` per ottenerne il valore.

Abbiamo un dilemma: come fa l'utente a specificare il nome di una variabile all'interno del documento, dove i nomi expl3 non sono permessi? Infatti, nel testo normale l'underscore non può essere impiegato nel nome di un comando e quindi

```
\setvar{\l_example_var_a_tl}{something}
```

avrebbe effetti disastrosi. Preferiremmo, invece, che l'utente scriva

```
\setvar{a}{something}
\usevar{a}
```

(in punti diversi del documento, chiaro). Vediamo, a passo lento, come si può fare; l'interfaccia utente sarà sistemata dopo. Per prima cosa definiamo una funzione che allochi una variabile e la imposti a un valore; poi la funzione che fornisce il contenuto di una variabile:

```
\cs_new_protected:Nn \example_setvar:Nn
 {
  \tl_clear_new:N #1
  \tl_set:Nn #1 { #2 }
 }
\cs_new:Nn \example_usevar:N
 {
  \tl_use:N #1
 }
```

La prima istruzione cancella il valore della variabile specificata, se esistente, oppure ne alloca una nuova. La seconda funzione non deve essere protetta, perché semplicemente produce il valore e non fa nulla di pericoloso. Ma questo non risolve il dilemma. Ecco dove il concetto di *variante* entra in scena. Ora eseguiamo

```
\cs_generate_variant:Nn
  \example_setvar:Nn
  { cn }
\cs_generate_variant:Nn
  \example_usevar:N
  { c }
```

che definisce due *nuove* funzioni di nome

```
\example_setvar:cn
\example_usevar:c
```

Che significa `c`? Che la nuova funzione si aspetta un argomento tra graffe, dal quale costruirà un nome di comando (in questo caso sarà il nome di una variabile, in altri casi potrebbe essere il nome di una funzione). Ora possiamo definire l'interfaccia utente:

```
\NewDocumentCommand{\setvar}{mm}
 {
  \example_setvar:cn
    { l_example_var_#1_tl }
    { #2 }
 }
\NewExpandableDocumentCommand{\usevar}{m}
 {
  \example_usevar:c
    { l_example_var_#1_tl }
 }
```

I siti su LaTeX sono infestati da domande su codice che fa stranezze come `\def\c{something}` e che, chissà perché, produce errori. Con l'approccio appena delineato, stiamo effettivamente definendo un *namespace* per le nostre variabili, che quindi possono essere chiamate con il loro 'nome esterno', lasciando all'implementazione i dettagli su come evitare conflitti.

Potremmo naturalmente definire i comandi a livello utente in LaTeX tradizionale. Una tipica implementazione sarebbe

```
\newcommand{\setvar}[2]{%
  \expandafter
  \def\csname example@var@#1\endcsname{#2}%
}
\newcommand{\usevar}[1]{%
  \csname example@var@#1\endcsname
}
```

Non litigherò con chi insista che questo modo è più semplice. Ma rimarrò della mia opinione che non lo è.

Mettiamo qualcos'altro sul fuoco. Vogliamo permettere all'utente di copiare il contenuto di una variabile in un'altra con `\copyvar{b}{a}`, dove `b` è la nuova 'variabile' e `a` quella già esistente. Ci serve solo la nuova variante

```
\cs_generate_variant:Nn
  \example_setvar:Nn
  { cv }
```

e potremo definire

```
\NewDocumentCommand{\copyvar}{mm}
 {
  \example_setvar:cv
    { l_example_var_#1_tl }
    { l_example_var_#2_tl }
 }
```

Ora abbiamo a disposizione un'altra funzione, cioè `\example_setvar:cv` che prende due argomenti tra graffe. Il secondo è elaborato come `c` e produce un token simbolico che dev'essere una variabile della quale si userà *il contenuto* come se fosse un argomento tra graffe dato alla funzione principale.

Lascio come facile esercizio su `\expandafter` scrivere il codice corrispondente in LaTeX tradizionale (suggerimento: si usino `\let` e due `\expandafter` in posizioni opportune).

Si può anche riassumere la definizione di entrambe le varianti:

```
\cs_generate_variant:Nn
  \example_setvar:Nn
  { cn, cv }
```

La variante `v` è un caso speciale della `V` che è molto simile e la maiuscola ci ricorda che l'argomento dev'essere un singolo token (il nome di una variabile, per la precisione) senza graffe. Supponiamo di avere una funzione che faccia qualcosa con il suo argomento:

```
\cs_new:Nn \example_foo:n { -- #1 -- }
```

(solo un esempio stupido per illustrare il concetto). Tuttavia, in alcuni casi ci serve passare alla funzione qualcosa che è stato immagazzinato in una variabile `tl`. Niente di più semplice perché possiamo dire

```
\cs_generate_variant:Nn
  \example_foo:n
  { V }
```

e questo ci permette di chiamare

```
\example_foo:V \l_tmpa_tl
```

Se, per esempio, `\l_tmpa_tl` contiene `abc`, la chiamata `\example_foo:V \l_tmpa_tl` darà lo stesso risultato di `\example_foo:n {abc}`.

Occorre ricordare che le varianti non nascono da sole senza che prima le generiamo. Le funzioni del nucleo sono già fornite con molte varianti che si sono dimostrate utili e che sono elencate insieme alla funzione principale in `interface3.pdf`. Che succede se non sappiamo se una variante è già stata definita? Nessun problema! La nuova generazione verrebbe silenziosamente ignorata e, anche se non lo fosse, il problema non esisterebbe comunque, perché le varianti sono generate in modo uniforme.

Potremmo anche evitare di generare varianti. Per esempio, il mestiere di `\example_setvar:cv` potrebbe essere assunto da

```
\exp_args:Ncv \example_setvar:Nn
```

(che, di fatto, è come la variante è definita), ma non c'è alcuna ragione di complicarci la vita così. Le implementazioni moderne di TeX e soci sono dotate di abbondante memoria e i tempi in cui la memoria era *molto* scarsa e trucchi che risparmiavano anche pochi token erano importanti se li ricordano solo i dinosauri come Frank, David, Chris e io. Ricordo bene quella volta in cui vidi il tremendo messaggio di errore "This can't happen" perché stavo adoperando PiCTeX.

Torniamo alla teoria. Ci sono varianti delle funzioni che definiscono funzioni? Certo che sì! Ci sono occasioni in cui desideriamo definire funzioni il cui nome è deciso durante la compilazione. L'esempio seguente è un po' scemo, ma dovrebbe dare l'idea:

```
\cs_new:cn { example_foo:n } { -- #1 -- }
```

è lo stesso della definizione precedente di `\example_foo:n`, perché il nome, compresa la segnatura, sarà formato prima che la funzione principale `\cs_new:Nn` svolga il suo compito. Si può usare qualsiasi cosa nelle graffe corrispondenti a un argomento `c` purché il risultato finale, dopo l'*espansione completa* consista solo di caratteri.

Oh! Espansione! Che cos'è? Siate pazienti. C'è ancora qualcosa da discutere prima.

3.2 Locale, globale e ⟨*extra*⟩

Tutti dovrebbero sapere qualcosa su *locale* e *globale*. In LaTeX, se definiamo un comando all'interno di un ambiente, la definizione è locale all'ambiente stesso e sparirà quando finisce. Altre assegnazioni di significati sono invece globali: le operazioni sui contatori, per esempio. Non ci interessa discutere in dettaglio la differenza tra locale e globale, ma ci sono aspetti della questione che sono rilevanti per le funzioni.

Tutte le dichiarazioni `\cs_new`⟨*extra*⟩`:Nn` sono sempre *globali*. Anche se eseguite all'interno di un gruppo, il loro effetto sarà presente ai livelli superiori. Inoltre quelle funzioni controllano se la funzione sia già definita e, nel caso, emetteranno un messaggio di errore. La parte ⟨*extra*⟩ sarà descritta fra poco. Va anche notato che anche l'allocazione di variabili (ma non l'impostazione) è sempre globale: l'istruzione

```
\tl_new:Nn \l_example_foo_tl
```

definisce la variabile a tutti i livelli e protesterà se la variabile esiste già.

Però qualche volta ci serve una funzione ausiliaria che sia *definita localmente*, che non abbia un compito fisso e vada ridefinita in base al contesto. Per queste funzioni c'è la famiglia

`\cs_set`⟨*extra*⟩`:Nn`

La sintassi è esattamente la stessa e anche le varianti.

Quale preferire? La famiglia `new` or la `set`? Facile: la prima, a meno che la funzione non sia una che *deve* cambiare definizione secondo il contesto. Raramente, direi anche mai, si definisce una funzione di alto livello con `set`.

La natura stessa del linguaggio invita i programmatori a scrivere codice "a strati". Per esempio, avremmo anche potuto definire

```
\NewDocumentCommand{\setvar}{mm}
 {
  \tl_clear_new:c
    { l_example_var_#1_tl }
  \tl_set:cn
    { l_example_var_#1_tl }
    { #2 }
 }
```

senza introdurre `\example_setvar:Nn`. Scoraggio questo stile di programmazione: il nostro code dovrebbe *appoggiarsi* a `expl3` e fornire API che il programmatore impiega. Come detto prima, non c'è ragione di risparmiare una funzione, tanto più se consideriamo che quando la nostra API è disponibile, possiamo facilmente generarne varianti per compiti particolari.

Qual è la lista completa degli 'extra' dopo `\cs_new` o `\cs_set`? Eccola:

```
\cs_new:Nn
\cs_new_protected:Nn
\cs_new_nopar:Nn
\cs_new_protected_nopar:Nn
\cs_set:Nn
\cs_set_protected:Nn
\cs_set_nopar:Nn
\cs_set_protected_nopar:Nn
\cs_gset:Nn
\cs_gset_protected:Nn
\cs_gset_nopar:Nn
\cs_gset_protected_nopar:Nn
```

Dovreste già avere un'idea dello scopo di `protected`: fa sì che la funzione non sia espansa quando la *piena espansione* è forzata. In particolare, una funzione `protected` non può essere adoperata in un argomento di tipo `c`, perché non sarebbe espansa e non è un carattere. La varietà `nopar` proibisce `\par` negli argomenti della funzione (quando è chiamata). A meno di non essere in situazioni speciali in cui `\par` non è sensato negli argomenti, si eviti la varietà *nopar* (ma non ci sono leggi che lo impongano).

La famiglia `gset` è quasi lo stesso di `new`, ma senza il controllo se la funzione da definire esista già.

Per i guru di TeX che leggono queste note„ `new` and `gset` usano `\gdef`, mentre `set` usa `\def`; il prefisso `\long` è aggiunto tranne che nella varietà `nopar`.

Quali sono le varianti disponibili? Questa è la lista completa delle segnature disponibili per tutte le funzioni elencate sopra:

```
Nn  cn  Nx  cx
```

Che fa la misteriosa `x`? Dovrebbe ricordare *fully expanded*. Ci sarà una sezione sull'argomento.

3.3 Parametri

Qualcuno potrebbe protestare dicendo che ha visto modi diversi di definire funzioni e avrebbe ragione. Infatti *c'è* un'intera altra famiglia come quella descritta, ma in cui la segnatura ha una bizzarra `p` tra `N` e `n`:

```
\cs_new:Npn
\cs_new_protected:Npn
\cs_new_nopar:Npn
\cs_new_protected_nopar:Npn
\cs_set:Npn
```

```
\cs_set_protected:Npn
\cs_set_nopar:Npn
\cs_set_protected_nopar:Npn
\cs_gset:Npn
\cs_gset_protected:Npn
\cs_gset_nopar:Npn
\cs_gset_protected_nopar:Npn
```

Il tipo `p` è riservato a queste funzioni e alle loro varianti

```
Npn  cpn  Npx  cpx
```

e sta per ***p**arameter text*. Le righe di codice seguenti sono del tutto equivalenti:

```
\cs_new:Nn \example_usevar:n {...}
\cs_new:Npn \example_usevar:n #1 {...}
```

Lo stesso per le altre funzioni. Nel secondo caso, il *parameter text* è scritto esplicitamente. Ricordiamo che quando sono in vigore le convenzioni di expl3 gli spazi sono ignorati, quindi per due parametri possiamo avere

```
\cs_new:Nn \example_foo:nn {...}
\cs_new:Npn \example_foo:nn #1#2{...}
\cs_new:Npn \example_foo:nn #1#2 {...}
\cs_new:Npn \example_foo:nn #1 #2{...}
\cs_new:Npn \example_foo:nn #1 #2 {...}
```

e le ultime quattro righe sono del tutto equivalenti. Personalmente preferisco la prima che mi appare 'più logica', altri preferiscono il secondo metodo. Attenzione! Il secondo metodo non controlla la coerenza della segnatura con il *parameter text* e permette anche segnature 'sbagliate', ma questo fatto non deve essere sfruttato: LaTeX non protesterà se scrivete

```
\cs_new_protected:Npn
  \example_setvar:cn #1 #2
  {...}
```

ma questo non significa che possiate evitare la procedura in due passi di prima definire `\example_setvar:Nn` e di generare poi la variante. Scrivere la definizione in quel modo sarebbe *sbagliato*. La seconda famiglia di funzioni (quelle con la `p`) ammette perfino che non ci sia la segnatura e quindi può essere adoperata per definire comandi a livello utente, sebbene il metodo con `\NewDocumentCommand` e simili sia raccomandato.[3]

Il metodo con `p` è necessario quando il *parameter text* è 'non standard', nel senso che stiamo definendo una funzione con *argomenti delimitati* e in questo caso la segnatura dovrebbe essere `:w`. Se vogliamo una funzione che imposti tre variabili all'anno, mese e giorno partendo da una data nel formato ISO come 2020-10-15, potremmo fare così:

3. expl3 può anche essere usato con plain TeX e, in tal caso, questo è l'unico modo per definire comandi a livello utente.

```
\int_new:N \l_example_year_int
\int_new:N \l_example_month_int
\int_new:N \l_example_day_int
\cs_new_protected:Nn \example_setdate:n
 {
  \__example_setdate:w #1 \q_stop
 }
\cs_new_protected:Npn
  \__example_setdate:w #1-#2-#3 \q_stop
 {
  \int_set:Nn \l_example_year_int { #1 }
  \int_set:Nn \l_example_month_int { #2 }
  \int_set:Nn \l_example_day_int { #3 }
 }
```

Qui introduco un'altra utile convenzione. Se il ⟨*prefisso*⟩ è preceduto da un doppio underscore, la funzione va considerata di livello più basso delle altre e non dovrebbe mai essere chiamata al di fuori degli usi specifici da parte delle funzioni 'normali' (senza il doppio underscore). L'idea è che le funzioni normali sono 'l'interfaccia del programmatore' mentre le altre sono ausiliarie e la loro implementazione non dovrebbe importare. La distinzione, quando si scrive codice per uso personale non è così importante: lo è per chi scrive pacchetti, però. Le funzioni normali (senza il doppio underscore) *possono* essere adoperate da altri pacchetti; al contrario, nessuno dovrebbe far conto che le funzioni di livello inferiore (con il doppio underscore) siano addirittura definite in versioni successive del pacchetto.

Questo dovrebbe chiarire perché il codice precedente divide il lavoro a due livelli; abbiamo la funzione di alto livello `\example_setdate:n` che si appoggia a una di livello inferiore che faccia il lavoro sporco. L'autore del pacchetto potrebbe scoprire in seguito metodi migliori per quello scopo, ma questo avrebbe influenza solo sul livello inferiore e non sulla funzione principale che si potrà adoperare sempre e allo stesso modo. Magari la definizione di `\example_setdate:n` cambierà in futuro, ma questo non avrà effetto sul codice che la adoperi.

4 Espansione

Non si può capire a fondo TeX senza avere un minimo di conoscenza del suo funzionamento. Nei linguaggi di programmazione funzionali, se `g` è una funzione di una variabile che restituisce un array di tre dati, mentre `f` è una funzione di tre variabili, un'istruzione

```
f(g(x))
```

sarebbe permessa; lo è dal punto di vista della matematica, il particolare linguaggio di programmazione potrebbe richiedere qualche aggiustamento. Questo non succede con TeX che procede dall'esterno verso l'interno e non viceversa.

Se abbiamo la funzione con un argomento `\example_a:n` che restituisce tre liste di token tra graffe e la funzione `\example_b:nnn` che prende tre argomenti, una chiamata come

```
\example_b:nnn { \example_a:n {x} }
```

fallirebbe in modo miserabile. TEX funziona così e nessun codice, per quanto perfezionato e furbo può cambiare la situazione. La funzione esterna cercherà tre argomenti: si trova una sola lista di token tra graffe e si arrangerà in altro modo, con risultati disastrosi. Per fare un esempio, supponiamo che a `\example_a:n` possa essere dato come argomento una data in formato ISO format e che da `2020-10-15` restituisca `{2020}{10}{15}`, mentre `\example_b:nnn` prende tre argomenti e produce la data in un formato diverso, diciamo "giorno 'nome del mese' anno"; in questo caso dovrebbe stampare "15 ottobre 2020".

Ci serve un approccio indiretto se vogliamo permettere di fornire una data ISO alla funzione che stampa la data nel formato tradizionale. Vediamo come si può definire la funzione generale:

```
\cs_new:Nn \example_date:nnn
 { % #1 = year, #2 = month, #3 = day
  #3~
  \int_case:nn { #2 }
   {
    {1}{gennaio}
    {2}{febbraio}
    ...
    {12}{dicembre}
   }
  ,~#1
 }
```

Il codice non è completo, dovrebbe essere chiaro come riempire il buco. La funzione `\int_case:nn` compara il suo primo argomento, un intero, con la lista data come secondo argomento che consiste di coppie di oggetti tra graffe; il primo oggetto è un intero, il secondo il testo da restituire se gli interi coincidono. Notiamo che `~`, nell'ambiente di programmazione `expl3` è un semplice spazio.

Abbiamo anche la funzione che prende una data nel formato ISO come `2020-10-15` e restituisce `{2020}{10}{15}`; potrebbe essere

```
\cs_new:Nn \__example_isodate:n
 {
  \__example_isodate:w #1 \q_stop
 }
\cs_new:Npn
  \__example_isodate:w #1-#2-#3 \q_stop
 {
  {#1}{#2}{#3}
 }
```

L'input della seconda funzione è diviso ai trattini e al terminatore, cioè stiamo adoperando *argomenti delimitati*, dettaglio su cui non insisto: ci basta sapere che la chiamata `\__example_isodate:n {2020-10-15}` in qualche modo restituirà `{2020}{10}{15}` nel flusso di input di TEX.

Come le combiniamo? Ci sono parecchi modi, ma tutti richiedono di capire il concetto di *piena espansione*. TEX conosce solo le macro; quando ne trova una, sa quanti argomenti vuole e li cerca nel flusso di input; quando li ha trovati, sostituisce l'intero insieme di token con il testo di sostituzione della macro.

Il problama primario è che il più delle volte non abbiamo la minima idea di quanti passi di questo processo di espansione ci sia bisogno per passare da `\__example_isodate:n {2020-10-15}` a `{2020}{10}{15}`. In questo caso non sarebbe difficile contarli, ma è solo un esempio semplice. Se lo sapessimo, un'opportuna catena di `\expandafter` basterebbe, ma sarebbe facilissimo commettere errori e non garantirebbe nulla se per caso l'implementazione della funzione di livello inferiore cambiasse.

Vorremmo dunque che `\__example_isodate:n` arrivasse al risultato finale in una sola mossa. Ecco un modo:

```
\exp_last_unbraced:Ne
  \example_date:nnn
  { \__example_isodate:n {2020-10-15} }
```

La prima funzione ha lo scopo di produrre la *piena espansione* del suo secondo argomento e inserisce il risultato nel flusso di input senza graffe attorno. Ci sono altri modi: uno è di definire una funzione ausiliaria

```
\cs_new:Nn \example_date:n
 {
  \__example_date:Ne
    \example_date:nnn
    { \__example_isodate:n { #1 } }
 }
\cs_new:Nn \__example_date:Nn
 {
  #1 #2
 }
\cs_generate_variant:Nn
  \__example_date:Nn { Ne }
```

e così `\example_date:n {2020-10-15}` produrrebbe il risultato voluto.

Ci sono certamente modi migliori per gestire le date, ma l'idea era di presentare come sfruttare le varianti che producono piena espansione.

Ce ne sono di tre tipi: `e`, `x` e `f`. L'ultima è la più 'restrittiva' perché esegue un'espansione ricorsiva dei token non appena appaiono nel flusso di input e termina al primo token non espandibile che trova. Nonostante questa limitazione, ha parecchi usi.

Il tipo `x` è al giorno d'oggi meno importante perché tutti i motori TEX possiedono la primitiva

`\expanded` che è essa stessa espandibile. expandable. Solo TEX 'originale Knuthiano' (quello che si lancia con `tex` dalla linea di comando) non ce l'ha, perché è mantenuto senza estensioni, secondo i desideri di Knuth.

Che fa `\expanded`? Fa essenzialmente lo stesso di `\edef`, con la differenza che non viene definita una macro. L'argomento subisce la piena espansione ricorsiva che *non* termina quando si trova un token espandibile: questo viene superato e lasciato com'è e si procede sul successivo, fino al termine della lista di token passata inizialmente. Il risultato va nel flusso di input (tutto insieme e senza graffe).

4.1 Piena espansione con e

Il tipo di argomento `e` dice a LATEX che prima deve eseguire la piena espansione e poi passare il risultato alla funzione principale. Questo è molto importante se per caso l'argomento contiene una variabile di cui vogliamo adoperare il valore al momento in cui la funzione è chiamata.

Ecco un esempio tratto da una domanda posta su TEX.StackExchange.[4] La domanda riguarda aggiungere a una 'nota finale' (*endnote*, pacchetto `endnote`) il numero di pagina dove la nota finale è richiamata. Chiaramente ci serve `\pageref` attraverso una `\label` generata automaticamente:

```
\NewDocumentCommand{\MyEndNote}{m}
 {
  \polyv_myendnote:ne
    { #1 }
    { \int_eval:n { \arabic{endnote}+1 } }
 }

\cs_new_protected:Nn \polyv_myendnote:nn
 {
  \endnote
   {
    #1~(page\nobreakspace
    \pageref{#2:endnote})
   }
  \label{\arabic{endnote}:endnote}
 }
\cs_generate_variant:Nn
  \polyv_myendnote:nn
  { ne }
```

Qual è il problema da risolvere? Il numero della nota finale è incrementato solo dopo che il testo della nota è stato elaborato, in modo che una `\label` successiva si riferisca al numero giusto. Quindi non possiamo usare

```
\pageref{\arabic{endnote}:endnote}
```

perché questo si riferirebbe alla nota *precedente*. Quindi la funzione interna impiega l'espansione `e` in modo da generare il successore del valore attuale del contatore `endnote`. Senza questa piena espansione, tutte le note finali dichiarate con `\MyEndNote` conterrebbero l'equivalente di

```
\pageref{%
  \int_eval:n{\arabic{endnote}+1}:endnote
}
```

e perciò il risultato finale sarebbero riferimenti incrociati indefiniti perché `\arabic{endnote}` fornirebbe l'*ultimo* valore del contatore. Se l'ultima nota finale avesse il numero 10, otterremmo `\pageref{11:endnote}`. Invece, con la piena espansione, viene usato il valore al momento della chiamata. Alla prima nota il contatore ha il valore 0, così il risultato sarà lo stesso di

```
\endnote{The text of the endnote
  (page\nobreakspace\pageref{1:endnote})}
\label{1:endnote}
```

Avremmo anche potuto usare `f` o `x` per questa particolare applicazione, ma `e` è il più efficiente dei tre. La differenza con `x` è che le funzioni che usano questa variante non sono espandibili, quindi devono essere del tipo `protected`. Infatti il processo richiede due passi: prima viene impostata una variabile `tl` con

```
\tl_set:Nx \l__exp_internal_tl {...}
```

che sfrutta internamente il buon vecchio `\edef`, poi la funzione è definita adoperando il valore della variabile.

L'introduzione della piena espansione `e` è stato un grande avanzamento perché permette cose che prima erano quasi impossibili.

Ci sono comunque impieghi utili di `x`: per esempio non ci sono varianti come `\cs_new:Ne` che sarebbe *meno* efficiente di `\cs_new:Nx` (che è esattamente `\edef`).

5 Un'altra utile variante

Nell'elenco dei tipi di argomento ci sono `V` e `v` su cui è stato dato qualche cenno. È ora di discutere il primo tipo in maggiore profondità.

Il tipo `v` è essenzialmente lo stesso e aggiunge solo la possibilità di costruire il nome della variabile durante la compilazione. Il tipo principale è `V`.

Supponiamo di avere la nostra funzione preferita che divide una data ISO nelle sue parti e la restituisce in forma leggibile. Chiamiamola `\example_date:n`, non ci interessa la sua implementazione.

Supponiamo ora che una data sia stata immagazzinata in una variabile `tl`. Siccome si tratta di una macro sotto travestimento, ai primordi di expl3 il modo di agire era

```
\cs_generate_variant:Nn
  \example_date:n { o }
```

4. https://tex.stackexchange.com/a/438715/. Il codice nella risposta adopera `f` perché `e` non era ancora disponibile.

e si doveva chiamare

```
\example_date:o { \l_tmpa_tl }
```

Funziona, ma ha il difetto di dipendere dalla conoscenza dell'implementazione delle variabili `tl` e non è generalizzabile ad altri tipi di variabili. La variante `o` esegue un singolo passo di espansione all'interno dell'argomento tra graffe; fa il suo dovere con una `tl`, ma fallirebbe in modo spettacolare con una variabile `fp` (che contiene la rappresentazione di un numero in virgola mobile).

Il metodo migliore e raccomandato è

```
\cs_generate_variant:Nn
  \example_date:n { V }
```

e la nuova funzione va chiamata come

```
\example_date:V \l_tmpa_tl
```

L'effetto è di passare alla funzione principale il *contenuto* della variabile, con le graffe al loro posto. Dopo

```
\tl_set:Nn \l_tmpa_tl { 2020-10-15 }
```

la chiamata `\example_date:V \l_tmpa_tl` sarà equivalente a `\example_date:n {2020-10-15}`.

Ecco un esempio con un tipo diverso di variabile, qui `int`. Vogliamo passare la rappresentazione decimale del valore e al risultato vanno aggiunti zeri a sinistra in modo da avere quattro cifre. Definiamo la funzione principale che genera tanti zeri quanti ne mancano, contando il numero di cifre del numero passato; poi generiamo la variante `V` e impostiamo una variabile `int` a un valore

```
\cs_new:Nn \example_pad:n
 {
  \prg_replicate:nn
    { 4 - \tl_count:n { #1 } }
    { 0 }
  #1
 }
\cs_generate_variant:Nn
  \example_pad:n { V }

\int_set:Nn \l_tmpa_int { 43 }
\example_pad:V \l_tmpa_int
```

Otterremo 0043. Forse non sembra così interessante, ma se usiamo il cugino `v` possiamo trasformare in

```
\cs_generate_variant:Nn
  \example_pad:n { v }

\example_pad:v { c@page }
```

sapendo, ovviamente che `\c@page` è il nome in LaTeX del registro che contiene il numero di pagina: si può immediatamente pensare a un'applicazione. Qui sfruttiamo il fatto che i contatori TeX sono proprio come le variabili `int` di expl3. Tramite le varianti `V` o `v` stiamo passando alla funzione principale il valore corrente come lista di cifre e possiamo contare quante ce ne sono, il che sarebbe impossibile con il valore 'astratto'.

Quali tipi di variabili possono essere adoperate in questo modo? Parecchie: `tl`, `int`, `fp`, ma anche `clist` e altre più esoteriche. In sostanza, tutte le variabili che possono restituire un output sensato. Non ce lo possiamo aspettare da variabili `seq` o `prop` e con queste otterremmo errori.

Ho trovato talvolta utile definire la variante `\cs_set:NV` a `\cs_set:Nn` per poter adoperare una variabile `tl` che è stata impostata con il testo di sostituzione desiderato, ma modificato con l'aiuto delle espressioni regolari.[5] Si noti che non è possibile definire la variante `\cs_set:NpV`, perché `\cs_generate_variant:Nn` accetta funzioni la cui segnatura contiene solo i caratteri `N` o `n`.

6 Vero o falso?

Ci sono altri due tipi di argomento interessanti: `T` e `F`. Il titolo della sezione dovrebbe suggerire che sono correlati a verità e falsità.

Giusto! Sono specificatori di argomento per le segnature delle funzioni *condizionali*. Esempio:

```
\int_compare:nTF
```

è una funzione che prende tre normali argomenti tra graffe; il primo è una relazione numerica tra interi che viene esaminata, il secondo il codice da eseguire se la relazione è vera, il terzo il codice da eseguire se la relazione è falsa. Quindi

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

stamperà, rispettivamente, A e B. Perché non è, più semplicemente, `\int_compare:nnn`? In realtà era proprio così nelle prime versioni di expl3, ma si comprese che gli specificatori `T` e `F` sono migliori: possiamo anche avere

```
\int_compare:nT
\int_compare:nF
```

per i casi in cui non ci sia da eseguire nulla nei casi falso e vero rispettivamente. Certo,

```
\int_compare:nF { ⟨relation⟩ } { B }
\int_compare:nTF { ⟨relation⟩ } { } { B }
```

sono del tutto equivalenti, ma il primo mostra più chiaramente che non vogliamo fare nulla se la relazione è vera. Con la segnatura `:nnn` sarebbe comunque necessario avere argomenti vuoti. Inoltre, la presenza di `T` or `F` (o entrambi) ci fa vedere bene che la funzione è condizionale.

5. https://tex.stackexchange.com/a/355576/

Tutte le funzioni condizionali del nucleo sono disponibili con `TF`, `T` or `F`; alcune funzioni pseudo-condizionali hanno perfino la versione senza entrambi. Ne troviamo un esempio in `interface3.pdf` dove ci sono `\str_case:nn` e anche `\str_case:nn`*`TF`*. La strana notazione *`TF`* significa che sono disponibili le tre combinazioni `TF`, `T` e `F`.

Perché pseudo-condizionale? È chiaro che `\str_case:nn` non è un condizionale, ma la versione estesa permette di eseguire qualcosa in più se c'è o non c'è un *match*. Probabilmente si impiega di più la versione `\str_case:nnF` per eseguire qualcosa in caso di *no match* che può essere un messaggio di errore o qualche *default*.

Va notato che per i condizionali (anche pseudo) le varianti vanno generate con

```
\prg_generate_conditional_variant:Nnn
```

invece che con `\cs_generate_variant:Nn`. Per esempio se pensiamo di immagazzinare relazioni numeriche per `\int_compare:n`*`TF`* in una variabile `tl`, il modo corretto di generare la variante è

```
\prg_generate_conditional_variant:Nnn
  \int_compare:n { V } { p, TF, T, F }
```

Questo genera tutte le varianti

```
\int_compare:VTF
\int_compare:VT
\int_compare:VF
```

e anche la *forma predicativa*

```
\int_compare_p:V
```

che si può usare nelle espressioni booleane. Ma questo va oltre gli scopi di questo articolo.

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