

Codici di categoria

Enrico Gregorio

Sunto. \TeX lavora con liste di *token* che forma via via che legge i caratteri da un documento `.tex`. Capire la procedura di *divisione in token* è importante se si vuole modificare il comportamento usuale di \TeX . Sotto questo aspetto la nozione di *codice di categoria* assegnato a un carattere è fondamentale.

Di particolare interesse sono i *caratteri attivi*. Ne darò un'applicazione che sfrutta alcune delle nuove funzionalità di $\varepsilon\text{-}\TeX$.

Abstract. \TeX works with token lists formed when it reads characters from a `.tex` document. Understanding the *tokenization* procedure is important if one wants to modify the usual behavior of \TeX . Under this respect the notion of *category code* attached to a character is fundamental.

Chiefly interesting are the *active characters*. I will give an application of them, which exploits some of the new features of $\varepsilon\text{-}\TeX$.

Introduzione

Quando si lancia la composizione di un documento, all'interno di \TeX arrivano solo *token*, che potremmo pensare come unità indivisibili dotate di significato. Ciascun *token* può essere sviluppabile o no; a un certo livello del programma, i *token* del primo tipo sono sviluppati e questa operazione produce una nuova lista di *token*, che a sua volta viene sottoposta al procedimento in modo ricorsivo, finché rimangono solo *token* non sviluppabili che quindi raggiungono il livello del programma dove ha luogo la composizione tipografica vera e propria.

Da un altro punto di vista, i *token* possono essere di due tipi: *caratteri* e *sequenze di controllo*, che chiamerò per brevità *simbolici*. Vediamo l'esempio classico, lo sviluppo del comando `\TeX`:

```
T\kern-.1667em\lower.5ex
 \hbox{E}\kern-.125emX\@
```

è una lista formata da 29 *token*, 24 caratteri e 5 simbolici. Una differenza fondamentale fra il primo tipo e il secondo è che i caratteri hanno un significato immutabile, una volta che raggiungono l'interno di \TeX , mentre l'interpretazione dei *token* simbolici può cambiare anche dopo (ridefinendo un comando prima che sia sviluppato).

In un primo momento, questa analisi può essere sufficiente. Quando però si deve andare più in profondità, ci si accorge che la verità è un po' più complessa.

1 Codici di categoria

Analizzerò qui i *token* di tipo carattere; occorre ricordare che all'interno di \TeX arriva prima di tutto un flusso di caratteri che viene sottoposto alla *divisione in token*. Per esempio, una riga nel `.tex` come

```
\author{E. Gregorio}
```

è formata da 20 caratteri; quando il flusso di caratteri viene immesso in \TeX , è trasformato in 14 *token*:

```
\author • { • E • . • □ •
 G • r • e • e • g • o • o • r • i • o • }
```

(il simbolo `•` serve solo a separare visivamente un *token* dall'altro).

Che cosa permette di riunire i primi sette caratteri in un unico *token*? Per capirlo occorre dire che ogni carattere che entra nel procedimento di divisione in *token* è visto come una coppia di numeri¹

(*carattere, categoria*)

dove *carattere* è un intero fra 0 e 255, mentre *categoria* è un intero fra 0 e 15. Nella situazione usuale, i caratteri precedenti sono visti come le coppie

```
(92, 0), (97, 11), (117, 11), (116, 11), (104, 11),
(111, 11), (114, 11), (123, 1), (69, 11), (46, 12),
(32, 10), (71, 11), (114, 11), (101, 11), (103, 11),
(111, 11), (114, 11), (105, 11), (111, 11), (125, 2).
```

La trasformazione dei sette caratteri iniziali in un unico *token* dipende proprio dal fatto che il primo ha categoria 0. Non starò qui a specificare la regola precisa; basta dire che sono possibili due casi, quando \TeX incontra un carattere di categoria 0:

1. se il carattere che segue non ha categoria 11, il *token* è formato dal carattere di categoria 0 e da quel carattere seguente;
2. se il carattere che segue ha categoria 11, il *token* è formato dal carattere di categoria 0 e da tutti i caratteri consecutivi di categoria 11 successivi.

1. \TeX usa il codice ASCII per i caratteri da 0 a 127; sui caratteri da 128 a 255 non fa supposizioni, è questo che rende possibile usare diverse codifiche tramite il pacchetto `inputenc`.

La categoria assegnata a un carattere non è fissata e può essere modificata, ma solo fino a quando il carattere stesso non è ancora entrato nel meccanismo di lettura; dunque possiamo solo modificare la categoria dei caratteri che *entreranno*.

A che serve la categoria? A dire la *funzione* di ciascun carattere. Per esempio, secondo la convenzione usuale, B ha categoria 11, cioè la sua funzione è di rappresentare un carattere alfabetico che, se raggiunge la profondità ultima di T_EX, produrrà una bella ‘B’ nel .dvi. Invece { ha categoria 1, cioè serve da delimitatore di apertura dei gruppi o degli argomenti. Ecco la lista completa; dopo il nome c’è un esempio, secondo le convenzioni usuali:

- 0: carattere di *escape* (\);
- 1: inizio di gruppo ({);
- 2: fine di gruppo (});
- 3: inizio o fine di formula (\$);
- 4: punto di allineamento (&);
- 5: fine riga;
- 6: parametro (#);
- 7: esponente (^);
- 8: pedice (_);
- 9: carattere da ignorare;
- 10: spazio;
- 11: lettera (a);
- 12: carattere ‘altro’ (?);
- 13: carattere attivo (~);
- 14: commento (%);
- 15: carattere non valido.

Che significa ‘altro’? Che non appartiene a nessuno degli altri gruppi. Un carattere di categoria 9 non oltrepassa la fase di lettura dei caratteri; un carattere di categoria 14 esclude dalla lettura tutti i caratteri che seguono fino al primo di categoria 5 (compreso). Un carattere non valido produce un messaggio di errore e viene poi ignorato.

Un carattere di categoria 5 viene convertito in uno di categoria 10 oppure in un *token* \par secondo certe regole non facilissime; è la procedura che ci permette di andare a capo come ci pare e di lasciare una riga vuota per indicare la fine di un paragrafo².

Un carattere di categoria 10 (usualmente lo spazio e il carattere di tabulazione) può essere ignorato in certe situazioni, per esempio dopo un nome di

2. Il sistema operativo dice a T_EX dov’è la fine di una riga; a questo punto T_EX ignora l’eventuale carattere che la indica e lo sostituisce con un carattere di categoria 5.

comando oppure a inizio riga. Altrimenti funge da spazio, come è ovvio. In tutti i casi, durante la lettura, caratteri consecutivi di categoria 10 vengono ridotti a uno solo.

Un carattere attivo ha uno stato molto particolare: equivale a un comando che verrà sviluppato secondo le stesse regole dei *token* simbolici, quando non segue immediatamente un singolo carattere di categoria 0, dove T_EX sta cercando un nome di comando; la stessa osservazione vale per tutti gli altri codici. Perciò ~ è un comando, così come \~.

Il modo di assegnare un codice di categoria a un carattere è

```
\catcode<numero a 8 bit>=<numero a 4 bit>
```

Per esempio, `plain.tex` comincia assegnando il codice 1 alla graffa aperta e il codice 2 alla graffa chiusa³, ma lo fa in modo curioso:

```
\catcode'\{=1
\catcode'\}=2
```

Il *<numero a 8 bit>* può essere specificato: (1) come numero in notazione decimale (per esempio 123); (2) come numero in notazione ottale (per esempio ‘173’); (3) come numero in notazione esadecimale (per esempio “7B”); (4) come carattere ASCII (per esempio ‘\{’). L’ultimo modo è quello più pratico in questo caso: si scrive il carattere “accento grave”, la barra rovescia e il carattere corrispondente, così non c’è da ricordare qual è il suo numero nel codice ASCII (meglio però non giocare con i caratteri da 128 a 255, per via delle differenti codifiche). Non è obbligatorio precedere il carattere con la barra rovescia, se questo carattere ha categoria 11 o 12.

Molto spesso si trova, spulciando i *forum* di discussione su T_EX, il comando `\makeatletter` e il corrispondente `\makeatother`. Il loro sviluppo è

```
\catcode'\@=11
\catcode'\@=12
```

rispettivamente; è il primo comando che permette di usare @ come se fosse una lettera (categoria 11, per essere precisi) durante la divisione in *token*. Quando un comando il cui nome contiene un @ di categoria 11 è entrato nel meccanismo di lettura, il suo codice è immutabile. Perciò, anche dopo aver dato `\makeatother` il richiamo a quel comando sarà interpretato nel modo corretto.

Preciso meglio. Supponiamo di scrivere

```
\makeatletter
\newcommand{\@xxx}{xxx}
\newcommand{\xxx}{\@xxx}
\makeatother
```

3. Occorre dire che in T_EX ha incorporato il codice usuale per la barra rovescia, per il ‘per cento’ e per i caratteri alfabetici e numerici.

Lo sviluppo del *token* `\xxx` è allora `\@xxx`, che a sua volta viene sviluppato come i tre *token* `xxx`, indipendentemente dal codice di categoria del carattere `@` al momento in cui `\xxx` viene sviluppato. Tuttavia, quando `@` ha categoria 12 non ci si può riferire direttamente al comando `\@xxx`.

2 Caratteri attivi

In qualche punto di `latex.ltx` si trova⁴

```
\catcode'\~ =13
```

e, più avanti, si legge

```
\def~{\leavevmode\penalty10000 \ }
```

Come si vede un carattere attivo può essere definito allo stesso modo di un comando. L'idea è di (1) cominciare un paragrafo, se per caso si è in 'modo verticale'; (2) inibire la possibilità di spezzare una riga; (3) lasciare uno spazio.

I caratteri attivi sono molto usati da `babel`, per varie abbreviazioni. Bisogna usare una certa cautela, perché se scriviamo

```
\catcode'\a =13
```

non possiamo più usare nomi di comando che contengano la lettera 'a'. E non potremmo nemmeno tornare indietro, visto che il comando per l'assegnazione del codice di categoria contiene la 'a'!

Naturalmente le assegnazioni di categoria rispettano i gruppi; perciò se scriviamo

```
{\catcode'\a =13 \gdefa{xxx}}
```

dopo la fine del gruppo il carattere `a` ha ancora categoria 11. Nel gruppo abbiamo definito *globalmente* il comando `a` (quando il carattere è attivo). La definizione rimane silente, fino a che `TEX` incontra un carattere `a` attivo; in questo caso lo svilupperà secondo le solite regole. Faccio il solito esempio stupido.

```
\def\restorecc{\catcode'\a =11 }
{\catcode'\a =13 \gdefa{\restorecc}}
```

Quando usiamo `\restorecc`, il codice di categoria dei caratteri che compongono il corpo della definizione è già fissato. Perciò scrivendo

```
\catcode'\a =13 a
```

otterremo che il codice di categoria di `a` ridiventa 11. Infatti dopo aver fatto diventare `a` attivo, diamo il 'comando `a`' che viene sviluppato e ciò che viene eseguito alla fine è l'assegnazione al carattere `a` del codice 11.

4. Quasi mai scriverò esattamente ciò che si legge in `latex.ltx`, ma userò una forma semplificata e, agli effetti pratici, equivalente.

Ovviamente questo non è molto interessante, ma ci sono molte situazioni in cui un saggio uso di caratteri attivi facilita le cose.

Un esempio importante è il codice dell'ambiente `verbatim` che vedremo più avanti; è evidente che il concetto di cambio di codice di categoria è essenziale per poterlo definire. Una delle cose che in effetti cambia è il codice del carattere ' , per un motivo molto semplice: quando `TEX` incontra la successione di caratteri '?', produce il carattere `¿`. Invece, ponendo

```
\catcode'\ ' =13
\def'\{\leavevmode\kern0pt\char'\ '}
```

il problema è risolto, perché `TEX` non riconosce più la legatura. La stessa cosa viene eseguita per gli altri caratteri che hanno lo stesso problema. Fra parentesi: `\char<numero a 8 bit>` produce il carattere corrispondente al numero, ma solo come comando interno. Quindi non si può scrivere, per esempio,

```
\begin{tabular}{\char'\b}{cc}
```

e sperare che l'argomento opzionale sia interpretato come una lettera `b`.

3 L'ambiente verbatim

Come ho già detto, parlando di codici di categoria vengono subito in mente l'ambiente `verbatim` e il comando `\verb`. Prenderò in esame le parti principali della definizione dell'ambiente.

```
{\catcode'\ =\active%
\gdef@\vobeyspaces{\catcode'\ \active
\let \@xobeysp}}
```

Si comincia un gruppo nel quale lo spazio è reso attivo, per poter dare la definizione di `\vobeyspaces`; questo comando, quando eseguito, rende attivo lo spazio e lo definisce come `\xobeysp`, che è un comando interno equivalente a `\nobeyspace` e al comando `~` (cioè uno spazio non impiegabile per spezzare una riga). Siccome siamo in un gruppo, la definizione deve essere *globale*, questo è il motivo di `\gdef`.

Le righe successive sono le più importanti per capire il funzionamento dell'ambiente.

```
\begingroup \catcode'| =0
\catcode' [= 1 \catcode'| =2
\catcode'_{ =12 \catcode'\} =12
\catcode'\ =12
\gdef|@xverbatim#1\end{verbatim}
[#1\end[verbatim]]
\endgroup
```

Qui viene aperto un nuovo gruppo, nel quale il carattere `|` diventa di categoria 0, `[` di categoria 1 e `]` di categoria 2. I caratteri `\`, `{` e `}` diventano

di categoria 12, quindi stampabili normalmente. Viene poi definito il comando `\@xverbatim`.

Occorre una precisazione. Ci può essere un numero qualunque di caratteri di categoria 0; se, per esempio, `\` e `|` sono di categoria 0, scrivere `\mbox` o `\lbox` è equivalente. Tanto che Knuth, nel `TeXbook`, usa la notazione `\mbox` per indicare quel *token* quando è stato letto da `TeX`. Per evitare ambiguità con il carattere di categoria 0 in uso, applicherò la convenzione che `\nome` indica il comando con quel nome, almeno per un po'.

Usualmente solo `\` ha categoria 0, ma in questa particolare applicazione è necessario usare un altro carattere. Infatti il comando `\@xverbatim` è un comando con *argomento delimitato*.

La definizione rigorosa e completa di che cosa sia un argomento delimitato è troppo lunga da dare; mostrerò solo un esempio. Scrivendo `\def\Sc#1/{\scshape #1}` (non è una definizione particolarmente utile) possiamo usare

```
\Sc Donald E. Knuth/ è un nome famoso
```

e l'argomento al comando `\Sc` è tutto ciò che viene dopo il nome del comando (spazio escluso, naturalmente: viene ignorato come sempre) fino al primo carattere `/`. Per la precisione fino al primo carattere `/` di categoria 12, se non abbiamo fatto modifiche prima della definizione. Perciò scrivendo

```
\def\Sc#1/{\scshape #1}
\catcode'\/=11
```

```
\Sc Donald E. Knuth/ è un nome famoso
```

otterremmo un messaggio di errore, perché nella definizione `/` ha categoria 12, mentre il carattere `/` incontrato nell'uso del comando ha categoria 11. Provare, se non ci si crede. Eventuali spazi che seguono il delimitatore *non* sono ignorati, se questo delimitatore non è una sequenza di controllo. Si possono ottenere particolari effetti, perché il delimitatore può anche essere un *token* non definito.

Torniamo però al caso in esame, `\@xverbatim`. Tutto ciò che sta fra questo comando e la successione di caratteri

```
\end{verbatim}
```

è preso come argomento. Lo sviluppo di questo comando è l'argomento seguito da `\end{verbatim}`.

Facciamo attenzione: non ho scritto `\end{verbatim}`. Infatti il delimitatore del comando è la successione di 14 caratteri

```
\_12 e n d { _12 v e r b a t i m } _12
```

dove ho segnato accanto ai caratteri speciali la categoria che devono avere per essere riconosciuti come delimitatori in questo caso (le lettere hanno categoria 11).

Come vedremo fra poco, il comando `\verbatim` che fa cominciare tutta la faccenda (ed emesso

tramite `\begin{verbatim}`) rende quei caratteri di categoria 12, come richiesto. È per questo motivo che non è possibile (almeno senza ridefinire l'ambiente) lasciare uno spazio fra `\end` e `{verbatim}`.

Proseguiamo nell'analisi; ciò che segue è

```
\def\@verbatim{\trivlist \item\relax
...
\let\do\@makeother \dospecials
\obeylines
...}
```

Ho omesso la parte più noiosa; questo comando apre un ambiente `trivlist`, fa un po' di cose non troppo importanti e poi emette il comando `\dospecials`, dopo aver definito `\do` come `\@makeother`.

Che significa tutto ciò? I caratteri speciali sono 'conservati' in una lista nel modo seguente:

```
\def\dospecials{\do\ \do\\\do{\do}%
\do$\do&\do#\do~\do\_do%\do~}
```

Il comando `\do` non è nemmeno definito; possiamo usare questa lista per particolari azioni sui suoi elementi. Siccome viene dato

```
\def\@makeother#1{\catcode'#1=12 }
```

e `\do` è stato reso equivalente a esso, il comando `\dospecials` viene sviluppato eseguendo `\do\` che quindi diventa

```
\catcode'\ =12
```

e così via per tutti i caratteri speciali nella lista che diventano di categoria 12.

Il comando `\obeylines` fa in modo che ogni fine riga emetta un comando `\par`. Ci sono altri comandi che trascuriamo.

Ecco ora il comando che dà inizio all'ambiente:

```
\def\verbatim{\@verbatim
\frenchspacing\@vobeyspaces
\@xverbatim}
\def\endverbatim{\if@newlist
\leavevmode\fi\endtrivlist}
```

Quando `TeX` incontra `\begin{verbatim}`, esegue il comando `\verbatim`; ciò a sua volta richiama `\@verbatim`; una volta eseguito questo vengono sviluppati `\frenchspacing`, per rendere uniformi tutti gli spazi fra parole, e `\@vobeyspaces`, per attivare lo spazio come abbiamo visto.

Infine si esegue `\@xverbatim` che trova l'argomento, lo compone tipograficamente e richiama `\endverbatim` il quale chiude l'ambiente `trivlist` aperto in precedenza; il condizionale serve a evitare certi errori.

Ora possiamo rinunciare alla convenzione e tornare al solito modo di scrivere i comandi.

Ci rimane solo da discutere `\obeylines`. Ricordiamo che il carattere di fine riga ha categoria 5. La definizione di `\obeylines` è

```
{\catcode'\^^M=13 %
 \gdef\obeylines{\catcode'\^^M=13}%
 \let^^M\par}}
```

Una combinazione `^^{carattere ASCII}` è un modo per denotare caratteri non stampabili; siccome M ha codice ASCII 77, la combinazione `^^M` significa il carattere di codice $77 - 64 = 13$, che è proprio il carattere di fine riga (almeno per T_EX). Il compito di `\obeylines` è di renderlo attivo ed equivalente a `\par`, che è proprio l'ufficio voluto.

Va notato che questo modo di definire l'ambiente *verbatim* non è molto efficiente: un lungo listato potrebbe esaurire la memoria di T_EX. Il pacchetto *verbatim* migliora la definizione, facendo una lettura di ciò che si vuole scrivere *verbatim* riga per riga. Il pacchetto *fancyvrb* fa molto di più.

4 Un terzo tipo di *token*?

Per completezza, occorre segnalare che T_EX conosce un terzo tipo di *token*: *parametro*.

Un *token* parametro è una successione di due caratteri, il primo di categoria 6, il secondo uno dei caratteri 1...9 purché di categoria 12. Questi *token* sono però legali solo in certe situazioni: quando T_EX sta leggendo i parametri di una definizione e quando sta leggendo i *token* delle sostituzioni di una definizione.

Per esempio, in una riga come

```
X #1Y
```

T_EX legge cinque *token*:

```
X • □ • # • 1 • Y
```

e il *token* # produce un errore; invece in una riga come

```
\def\x#1{X #1Y}
```

T_EX vede nove *token*:

```
\def • \x • #1 • { • X • □ • #1 • Y • }
```

Per gli scopi di questa regola, un comando di definizione è `\def`, `\gdef`, `\edef` oppure `\xdef`.

Attenzione: il concetto di *token* parametro è valido solo per la *sintassi* dei comandi di definizione. In effetti se scriviamo

```
\def\a{1}
\edef\b#1{\expandafter#\a}
```

T_EX non si lamenta e un successivo `\show\b` mostra

```
> \b=macro:
#1->#1.
```

Da ciò segue che # è in realtà considerato come *token* unico per quanto riguarda la divisione in *token*.

5 Un problema

Scrivendo una dispensa per la parte di logica di un corso di base del primo anno, mi sono posto il problema di scrivere formule in modo un po' diverso dal solito, in modo da farle risaltare rispetto al testo e, soprattutto, per distinguere i simboli formali da quelli del linguaggio corrente; per intenderci, il simbolo dello zero deve essere distinto dal modo di denotare il numero zero.

Un trucco usato in alcuni testi di logica è di scrivere i simboli del linguaggio formale in nero. Siccome poi voglio usare la notazione polacca diretta, ogni simbolo va considerato come ordinario, con una spaziatura fissa fra uno e l'altro. Inoltre i pacchetti usuali non danno risultati molto soddisfacenti e quindi ho deciso di ricorrere al 'poor man's bold' per i simboli e al nero matematico per le lettere, riservandomi di vedere in seguito se ci siano caratteri più adatti.

Proviamo a scrivere la formula che esprime che un certo numero è divisibile per un altro:

$$\exists v_2 = v_0 \times v_1 v_2$$

Il codice per scrivere questa cosa è, dopo aver caricato il pacchetto *ambsy*,

```
{\pmb{\exists}}\;\mathbf{v}_-{2}\;
{\pmb{=}}\;\mathbf{v}_-{0}\;
{\pmb{\times}}\;\mathbf{v}_-{1}\;
\mathbf{v}_-{2}
```

È pensabile di scrivere decine di cose di questo tipo? E poi riuscire a leggerle nel *.tex*? Mi sono detto che un codice come

```
\formula{E v2 s= v0 s\times v1 v2}
```

sarebbe probabilmente più leggibile. Qui 'E' sta per \exists , 's' è un prefisso per un simbolo, 'v' indica una variabile con il suo indice. Per indicare \forall ho deciso di usare 'A' e un prefisso 'p' per i simboli di funzioni e relazioni che sono normalmente lettere. In più uso 'a' e 'o' per i connettivi 'et' e 'vel', 'i' per 'implica', 'j' per 'se e solo se' e 'n' per 'non'.

Viene subito l'idea di usare caratteri attivi.

La sintassi di `\formula` dovrebbe assomigliare a un comando con argomento, ma non può esserlo veramente, perché altrimenti i codici di categoria dell'argomento sarebbero letti e non più mutabili. Il problema è noto e la soluzione anche:

```
\def\formula{\bgroup\logicactivate
 \let\next= }
```

Lo sviluppo di `\formula` apre un gruppo con `\bgroup`; il comando `\logicactivate` cambia i codici di categoria necessari e quindi rende disponibili le definizioni dei caratteri attivati; infine gli ultimi comandi ingoiano la graffa aperta. Tutto ciò che viene eseguito qui sarà annullato quando si incontra la graffa chiusa, che corrisponde al

comando `\bgroup` inserito da `\formula`. Va ricordato che `\bgroup` è, in certi contesti, equivalente a una graffa aperta.

Ora si tratta di definire `\logicactivate`:

```
\def\logicactivate{\catcode'\A=13
\catcode'\E=13
...
}
```

Dobbiamo definire i comandi; `A` attivo deve stare per `{\pmb{\forall}}`; e vediamo subito un problemino: siccome voglio attivare anche `p`, non posso più dare il comando `\pmb`!

La soluzione è semplice: prima definisco dei comandi ausiliari e poi rendo i caratteri attivi equivalenti a questi. Mi serve poi un modo di dire `\global\let` senza usare caratteri proibiti.

```
{% apro un gruppo
\def\GL{\global\let}
\def\A{{\pmb{\forall}}\;}
...
\def\v#1{\mathbf{v}_{#1}\;}
\logicactivate
\GL A\A
...
\GL v\v
}
```

Se notate, nella prima parte ho usato nomi di comandi già definiti in \LaTeX ; tuttavia, quando il gruppo verrà chiuso, queste definizioni spariranno, ma non svanisce il significato dei caratteri quando attivati, poiché ho usato `\global\let`.

Rimane un problema: l'ultimo carattere inserisce una spaziatura di troppo che andrebbe tolta. Per ora non mi interessa, anche perché la soluzione definitiva non è questa.

Fin qui infatti tutto sembra funzionare: scrivendo `\formula{A v0}` ho quello che mi serviva (a parte le spaziature). E una ciliegia tira l'altra: ora mi viene in mente di scrivere all'interno di una formula anche *metavariabili*, cioè cose del tipo

$$\forall v_0 \varphi$$

dove φ sta a indicare una formula qualunque. Non è poi così difficile, dopo tutto, una volta che si capisce ciò che c'è da fare: disattivare i caratteri che abbiamo attivato, ma dentro un gruppo, cosicché ritornano attivi quando il gruppo termina. La sintassi che mi propongo è, per esempio,

```
\formula{A v0 <\varphi>}
```

Siccome dobbiamo attivare e disattivare caratteri, è bene sciversi una *lista* dei caratteri speciali, esattamente come `\dospecials` per *verbatim*, e comandi ausiliari per poter eseguire la lista in modi diversi:

```
\def\logiclist{\do\A\do\E\do\a\do\i%
\do\j\do\n\do\o\do\p\do\s\do\v}
\def\logicactivate{\let\do\logicactive
\logiclist\logicactive<}
\def\logicinactivate{\let\do\logicletter
\logiclist\logicother<}
\def\logicletter#1{\catcode'#1=11 }
\def\logicother#1{\catcode'#1=12 }
\def\logicactive#1{\catcode'#1=13 }
```

Il comando `\logicactivate` *esegue* la lista dei caratteri e li attiva; poi attivo anche il carattere `<`. Questo carattere non va nella lista principale, perché la sua categoria usuale è 12. Il comando `\logicinactivate` invece fa l'operazione contraria.

Ora devo definire la funzione del carattere `<`; suppongo che il carattere sia già attivato e aggiungerò ai comandi precedenti

```
\def<{\bgroup\logicinactivate
\logicescape}
```

prima di `\logicactivate` e, dopo,

```
\GL <<
```

Si tratta ora di definire `\logicescape`:

```
\def\logicescape#1>{#1\egroup\;}}
```

Il comando considera come suo argomento tutto ciò che viene fino al carattere `>` (si noti la sintassi con un argomento delimitato), e lo passa invariato aggiungendo `\egroup` e la spaziatura. La chiusura del gruppo fa tornare attivi i caratteri giusti e la composizione della formula può continuare come prima; dunque è lecito scrivere cose del tipo

```
\formula{i <\alpha> i <\beta> <\gamma>}
```

che produce

$$\rightarrow \alpha \rightarrow \beta \gamma$$

Molto bene, le cose vanno alla grande. Posso sbizzarrirmi a scrivere formule in modo facile e intuitivo.

L'inghippo si vede immediatamente quando però provo a scrivere queste formule in uno degli ambienti di allineamento di `amsmath`: non vanno! L'ultima formula scritta dà come risultato

$$i <\alpha> i <\beta> <\gamma>$$

che non è proprio quello sperato!

Qual è il motivo? Gli ambienti di allineamento di `amsmath`, per esempio `align`, leggono tutto il contenuto dell'ambiente e poi eseguono varie composizioni per ottenere il miglior risultato possibile. Ottimo dal punto di vista tipografico, certo; pessimo per me: quando entra in azione `\formula` i caratteri all'interno delle graffe sono già passati per la *bocca* di \TeX e quindi hanno già ricevuto la loro categoria. Ormai è troppo tardi per cambiarla.

Che fare? Arrendersi e scrivere le formule per esteso, magari inventandosi qualche abbreviazione? No: con \TeX si può fare tutto!

6 ε -TEX

Da un po' di tempo, chi ha una distribuzione aggiornata del sistema TEX basata su Web2C (le più diffuse lo sono) e invoca `latex` o `pdflatex` sta in realtà usando `pdfetex`, cioè la versione di ε -TEX capace di scrivere anche in formato PDF oltre che `.dvi`.

Il progetto ε -TEX ha studiato un programma compatibile con TEX ma con alcune estensioni. Una fra queste è quella che rende possibile adattare i comandi per le formule in modo da evitare il problema con `amsmath`.

L'estensione di cui avevo bisogno c'è e si chiama `\scantokens`. Questo comando primitivo aggiunto da ε -TEX ha la seguente azione: legge la lista di *token* che lo segue fra parentesi graffe come se si trattasse di un *file* esterno richiamato con `\input`. Lo stesso risultato si può ottenere con TEX tradizionale scrivendo su un *file* esterno e poi leggendolo, con certe limitazioni.

I caratteri che vengono immessi tramite `\scantokens` ricevono in quel momento la categoria, perché rientrano nel meccanismo di lettura.

Niente più problemi, dunque. Quasi, a dire il vero; ma lo vedremo più avanti. La nuova definizione di `\formula` è

```
\def\formula#1{\begingroup
  \logicactivate
  \scantokens{#1}%
  \unskip\endgroup}
```

Se invece di scrivere `\;` nelle definizioni dei simboli uso un comando esplicito di spaziatura, ho anche risolto il problema della spaziatura finale. La decisione finale è stata di usare uno spazio di 0,2em; il comando `\unskip` elimina l'ultima spaziatura.

Però c'è un altro problema, adesso: leggendo l'argomento di `\formula` e passandolo a `\scantokens`, non posso più uscire dal modo 'formula' per comporre le metavariable. Questo all'inizio mi sorprese, ma una breve riflessione, aiutata dai messaggi di errore, mi fece capire ciò che stava succedendo.

Quando si passa una lista di *token* a `\scantokens`, questo legge uno *pseudofile* che contiene esattamente quei *token*; e infatti il messaggio riguardava il comando `\p` non definito, mentre io avevo dato `\phi` come argomento. Il problema sta nel fatto che l'apparato di lettura non vede i quattro caratteri

```
\ p h i
```

ma solo tre oggetti: il *token* `\p` seguito dai caratteri `h` e `i`. Questo perché il carattere `p` ha già ricevuto la categoria 13 quando è stato valutato l'argomento di `\scantokens` nello sviluppo di `\formula`.

Gli sviluppatori di ε -TEX hanno pensato anche a questo e hanno messo a disposizione `\detokenize`. Che cosa fa questo comando primitivo? Va seguito

da una lista di *token* fra graffe ed esegue l'operazione inversa della lettura: spezza ciascun *token* nei suoi componenti iniziali. Una lettera rimane una lettera, ma un *token* simbolico come `\author` ridiventa una successione di sette caratteri che posso passare a `\scantokens` per farli leggere con le regole di assegnazione della categoria in vigore in quel momento.

Basta quindi ridefinire `\logicscape`:

```
\def\logicscape#1>{%
  \scantokens\expandafter{%
    \detokenize{#1}}%
  \endgroup\hskip.1em\relax}
```

Qui `\scantokens` va a cercare la lista che deve seguirlo e per fare questo sviluppa i *token* che seguono fino a trovare una parentesi graffa aperta; trova `\expandafter` che quindi sviluppa ciò che segue la parentesi graffa, cioè `\detokenize`. Il *token* `\expandafter` viene eliminato dal suo stesso sviluppo e `\scantokens` trova la graffa aperta; quindi legge la sequenza di caratteri ritrasformandola in *token*, ma con i codici di categoria giusti, perché `\logicscape` è stato chiamato dopo `\logicinactivate`.

C'è un altro piccolo problema quando si usa `\scantokens`. Se scriviamo `\scantokens{a}b` ci aspetteremmo che ε -TEX stampasse semplicemente 'ab', invece esce 'a b'. Questo perché, essendo i *token* letti come se venissero da un *file* esterno, a ogni 'riga' viene aggiunto il carattere di fine riga. Nel mio caso il problema non si pone, perché sto componendo in modo matematico; in altri potrebbe dare problemi che si possono curare mettendo come ultimo *token* nella lista data a `\scantokens` il comando `\noexpand`.

7 Il codice finale

Le idee di prima vengono utili anche per usare argomenti qualunque alle lettere `p` e `s`: posso usare anche per queste il ritorno ai caratteri non attivi in modo da poter scrivere anche

```
\formula{pA}
```

che non sarebbe stato concesso nella versione precedente. La lettera 'p' va seguita da un carattere alfabetico, mentre la lettera 's' da un simbolo qualunque.

Ecco il codice completo, con qualche raffinamento che invito a studiare. In particolare, invece che con `\pmb` o `\mathsfb` espliciti, ho definito i comandi in termini di 'prefissi' astratti che quindi è possibile modificare facilmente. Inoltre uso invece che 'logic' un prefisso 'lf@' per i comandi privati. Naturalmente il tutto va circondato da `\makeatletter` e `\makeatother` oppure letto da un pacchetto (`.sty`).

Si noti che ho definito un nuovo alfabeto matematico che usa i caratteri Computer Modern

Sans Serif neri (o, comunque, quelli della famiglia `\sfdefault`). Inoltre, se è caricato il pacchetto `graphicx`, i simboli dei quantificatori \forall e \exists si ottengono ruotando le lettere **A** e **E**.

Un altro problema che ho cercato di risolvere è quello di far cooperare queste macro con altre che attivino caratteri. Usare comandi come

```
{\catcode'\A=13 \gdef A{...}}
```

può essere pericoloso; forse non con 'A', ma magari con '<' sì. La faccenda si risolve usando un altro `\scantokens`, come vedremo.

Dividerò la descrizione del codice in vari brani, dopo il codice ci sarà il commento. Il codice è presentato sotto forma di un pacchetto, formula.

Identificazione e opzioni

```
\ProvidesPackage{formula}
\newif\iflf@graph
\@ifpackageloaded{graphicx}
  {\lf@graphtrue}
  {\lf@graphfalse}
  \PackageWarningNoLine{formula}{%
    It is best to load this package
    after 'graphicx'}%
}

\newif\iflf@tc
\@ifpackageloaded{textcomp}
  {\lf@tctrue}
  \PackageWarningNoLine{formula}{%
    Arrow symbols can be not available
    in some fonts;\MessageBreak
    in this case use the package with
    the 'notc' option}%
}
{\lf@tcfalse}

\DeclareOption{notc}{\lf@tcfalse}
\ProcessOptions\relax
```

Identifico il pacchetto e introduco due condizionali per verificare se sono stati caricati certi pacchetti. C'è anche un'opzione `notc` per evitare di usare i *font* nella codifica TS1; certi, infatti, non possiedono i caratteri necessari.

Preliminari

```
\DeclareMathAlphabet{\mathsfb}
  {\encodingdefault}{\sfdefault}{bx}{n}
\setbox0=\hbox{${\mathsfb{A}}$}
\iflf@tc
  \def\sfbs{%
    \usefont{TS1}{\sfdefault}{bx}{n}}
  \setbox0=\hbox{\sfbs 0}
\fi
```

Se ho a disposizione `graphicx` e `textcomp`, posso definire meglio certe costruzioni; altrimenti uso un ripiego. Definisco anche due *font* particolari, il

primo per comporre i simboli del linguaggio che consistono di lettere, il secondo per certi simboli che esistono nella codifica TS1.

La riga `\setbox0=\hbox{\sfbs 0}` e l'analoga per `\mathsfb` servono a far caricare i *font* subito, per evitare che il *file* `.fd` corrispondente sia letto quando i caratteri sono attivi, con gli ovvi problemi che ne risulterebbero: un `.fd` che non sia precaricato viene letto infatti al primo uso di un carattere da comporre in quel *font*.

Liste

```
\def\lf@letterlist{\do\A\do\E\do\alpha%
  \do\i\do\j\do\n\do\o\do\p\do\s\do\v}
\def\lf@otherlist{\do\<}

\def\lf@activate{\let\do\lf@active
  \lf@letterlist\lf@otherlist}
\def\lf@inactivate{%
  \let\do\lf@letter\lf@letterlist
  \let\do\lf@other\lf@otherlist}

\def\lf@letter#1{\catcode'#1=11 }
\def\lf@other#1{\catcode'#1=12 }
\def\lf@active#1{\catcode'#1=13 }
```

Definisco due liste, una con i caratteri di categoria 11 e una con quelli di categoria 12 da attivare in seguito. Nella seconda ce n'è solo uno, ma meglio essere previdenti: il pacchetto potrebbe essere esteso.

Poi introduco i comandi per eseguire le liste e assegnare le corrette categorie.

Comandi astratti

```
\def\lf@prefi{\pmb}
\def\lf@genI#1{%
  {\lf@prefi{#1}}\lf@space}
\def\lf@prefii{\mathsfb}
\def\lf@genII#1{%
  {\lf@prefii{#1}}\lf@space}
\iflf@tc
  \def\lf@prefiii{\sfbs}
  \def\lf@genIII#1{%
    \hbox{\lf@prefiii{#1}}\lf@space}
\fi
\def\lf@space{\mskip 2mu minus 1mu}
```

Definisco comandi astratti, in modo da non dover fare troppe modifiche se decido di cambiare come trattare tipograficamente i simboli. Per esempio, `\lf@space` inserisce una 'lunghezza matematica elastica' di 2 unità, che può diminuire fino a 1. Diciotto unità μ sono uno spazio quadratone (`\quad`).

Quantificatori

```
\iflf@graph
\def\lf@rot#1{%
  \rotatebox[origin=c]{180}{#1}}
\def\lf@A{%
```

```

\lf@rot{\lf@prefii{A}}\lf@space}
\def\lf@E{%
\lf@rot{\lf@prefii{E}}\lf@space}
\else
\def\lf@A{\lf@prefii{\forall}}\lf@space}
\def\lf@E{\lf@prefii{\exists}}\lf@space}
\fi

```

Comincio a definire i nuovi comandi. Per ‘esiste’ e ‘per ogni’, prevedo due modalità diverse, se è caricato `graphicx` o no. Anche qui definisco un comando generico, `\lf@rot`, che può venire utile per eventuali estensioni del pacchetto.

Altri simboli

```

\def\lf@a{\lf@genI{\textstyle\bigwedge}}
\def\lf@o{\lf@genI{\textstyle\bigvee}}
\iflftc
\def\lf@i{\lf@genIII{\char'031}}
\def\lf@j{\lf@genIII{%
\rlap{\char'030}\kern.1em\char'031}}
\def\lf@n{\lf@genIII{\char'254}}
\else
\def\lf@i{\lf@genI{\to}}
\def\lf@j{\lf@genI{\leftrightharpoon}}
\def\lf@n{\lf@genI{\lnot}}
\fi

```

Proseguo nelle definizioni. Per ‘implica’, ‘se e solo se’ e ‘non’, uso i caratteri nella codifica TS1, se è caricato `textcomp`; il ‘se e solo se’ è ottenuto sovrapponendo la freccia verso sinistra a quella verso destra, con un piccolo spostamento. I simboli per ‘e’ e ‘o’ sono più grandi per aiutare a distinguerli.

Simboli per relazioni e funzioni

```

\def\lf@p{\bgroup\lf@inactivate\lf@pred}
\def\lf@pred#1{%
\lf@prefii{#1}\egroup\lf@space}

\def\lf@s{\bgroup\lf@inactivate\lf@sym}
\def\lf@sym#1{%
{\lf@prefii{#1}}\egroup\lf@space}

\def\lf@v#1{\lf@prefii{v}_#1}\lf@space}

```

Definisco i comandi con argomenti; il primo per comporre una lettera, il secondo per un simbolo: in entrambi i casi apro un gruppo, disattivo i caratteri e chiamo la macro che esegue la composizione corretta. Nel terzo caso semplicemente compongo la lettera ‘v’ con il pedice indicato, nella supposizione che sia un numero e che i numeri non vengano attivati. Posso scrivere `v0` oppure, se necessario, `v{10}`.

Uscita dal modo formula

```

\def\lf@lt{\bgroup\lf@inactivate
\lf@escape}
\def\lf@escape#1{%
\scantokens\expandafter{%

```

```

\detokenize{#1}}%
\egroup\lf@space}

```

Definisco il comando per ‘uscire’ temporaneamente dal modo formula. La descrizione della macro è già stata data. Uso `\bgroup` perché così la sottoformula che compongo è considerata come un simbolo ordinario in ogni caso.

Definizione locale

```

\def\lf@doequiv{\let\\\let
\let\A\lf@A \let\E\lf@E \let\a\lf@a
\let\i\lf@i \let\j\lf@j \let\n\lf@n
\let\o\lf@o \let\p\lf@p \let\s\lf@s
\let\v\lf@v \let\<\lf@lt}

```

Definisco un comando che rende equivalenti i comandi precedenti con sequenze di controllo più brevi, nelle quali si usano solo i caratteri che vengono attivati nel modo formula. Questo perché una volta che li ho attivati non posso più usare comandi a più lettere che contengano quei caratteri! Naturalmente questo comando viene emesso dentro un gruppo e quindi le ridefinizioni che opera spariscono alla chiusura del gruppo. Il comando `\\` viene reso equivalente a `\let`.

Attivazione locale

```

\def\lf@doactivechars{%
\\ A\A \\ E\E \\ a\A \\ i|i
\\ j|j \\ n\n \\ o\o \\ p|p
\\ s\s \\ v\v \\ <|<}

```

Questo è il comando che rende i caratteri attivi equivalenti ai comandi definiti prima. Il fatto che ‘A’ e gli altri caratteri siano al momento di categoria 11 o 12 è irrilevante, vedremo perché.

Il comando principale

```

\DeclareRobustCommand\formula[1]{%
\begingroup
\lf@doequiv
\lf@activate
\scantokens\expandafter{%
\lf@doactivechars}%
\scantokens{#1}%
\unskip
\endgroup}

```

Questo è il comando principale, reso *robusto* per poterlo usare anche nei titoli di capitolo o sezione.

1. Si apre un gruppo, con `\begingroup` per motivi TEXnici.
2. Eseguo il comando che rende i comandi con nomi lunghi equivalenti ai comandi con nomi brevi.
3. Attivo i caratteri del modo formula.
4. Eseguo il comando `\lf@doactivechars`, ma rileggendone i *token* tramite `\scantokens`; in questo modo i caratteri dopo `\\` (che equivale a `\let`) sono attivi.

5. Leggo l'argomento, ma tramite `\scantokens`, in modo che i codici di categoria siano assegnati correttamente.

6. Elimino lo spazio esplicito dopo l'ultimo simbolo della formula e chiudo il gruppo.

Comandi ausiliari

```
\newcommand{\lvar}[1]{\lf@prefii{v}_{#1}}
\newcommand{\lrf}[1]{\{\lf@prefii{#1}\}}
\newcommand{\lsym}[1]{\{\lf@prefi{#1}\}}
```

`\makeatother`

Definisco, già che ci sono, anche abbreviazioni per poter usare i simboli anche senza abilitare il modo formula. Le parentesi graffe che sembrano in più servono a far considerare il simbolo come ordinario, in modo che non segua le sue spaziature proprie, come succederebbe senza.

8 Esempi d'uso

Ecco alcuni esempi; non occorre conoscere la logica per leggerli, basta confrontare ciò che viene composto dal codice che mostro subito dopo.

Sia **P** un simbolo di relazione binaria; l'interpretazione di **P** è una relazione di equivalenza se e solo se l'interpretazione delle seguenti tre formule è vera:

$$\forall v_0 P v_0 v_0 \tag{1}$$

$$\forall v_0 \forall v_1 \rightarrow P v_0 v_1 P v_1 v_0 \tag{2}$$

$$\forall v_0 \forall v_1 \forall v_2 \rightarrow \bigwedge P v_0 v_1 P v_1 v_2 P v_0 v_2 \tag{3}$$

Il codice per questa terna è

```
\begin{gather}
\formula{A v0 pP v0 v0}\
\formula{A v0 A v1
  i pP v0 v1 pP v1 v0}\
\formula{A v0 A v1 A v2
  i a pP v0 v1 pP v1 v2 pP v0 v2}
\end{gather}
```

Ed ecco la formula per la divisibilità:

$$\exists v_2 = v_0 \times v_1 v_2 \tag{4}$$

che ha come codice

```
\begin{equation}
\formula{E v2 s= v0 s\times v1 v2}
\end{equation}
```

L'aritmetica di Peano al primo ordine ha i seguenti assiomi:

$$\begin{aligned}
&\forall v_0 \neg = 0 S v_0 \\
&\forall v_0 \forall v_1 \rightarrow = S v_0 S v_1 = v_0 v_1 \\
&\forall v_0 = + 0 v_0 v_0 \\
&\forall v_0 \forall v_1 = + v_0 S v_1 S + v_0 v_1 \\
&\forall v_0 = \times 0 v_0 v_0 \\
&\forall v_0 \forall v_1 = \times v_0 S v_1 + \times v_0 v_1 v_0 \\
&\rightarrow \bigwedge \varphi(0) \forall v_0 \rightarrow \varphi(v_0) \varphi(S v_0) \forall v_0 \varphi(v_0)
\end{aligned}$$

dove φ è una qualunque formula con una variabile libera e $\varphi(x)$ indica la formula che si ottiene sostituendo tutte le occorrenze della variabile libera con il termine x .

Riporto solo il codice per l'ultima formula:

```
\formula{i a <\varphi>(p0)
  A v0 i <\varphi>(v0) <\varphi>(pSv0)
  A v0 <\varphi>(v0)}
```

Una formula con una variabile libera la cui interpretazione sia 'il numero naturale denotato è primo', può essere

$$\bigwedge < 1 v_0 \forall v_1 \rightarrow \exists v_2 = v_0 \times v_1 v_2 \bigvee = 1 v_1 = v_0 v_1$$

che va scritta

```
\formula{ a s< p1 v0 A v1 i
  E v2 s= v0 s\times v1 v2
  o s= s1 v1 s= v0 v1 }
```

La lettura della formula composta sembra impervia; ma con un po' di pazienza si fa l'abitudine alla notazione polacca. Ci sono due sottoformule collegate da un 'e': la prima dice che il numero dato è maggiore di 1; la seconda dice che, se un numero divide il numero dato, allora è 1 oppure il numero dato. Certamente però il modo di scrivere la formula in L^AT_EX è semplice e chiaro.

Facciamo l'ultima prova per vedere che effettivamente lo spazio finale viene cancellato:

```
\begin{gather*}
|\formula{pA}|\quad|\formula{<\psi>}|\
|\lrf{A}|\quad|\psi|
\end{gather*}
```

produce le seguenti formule:

$$\begin{aligned}
|A| &|\psi| \\
|A| &|\psi|
\end{aligned}$$

che sono effettivamente identiche.

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