Condizionali in LaTeX

Enrico Gregorio

Sommario

Dopo un breve cenno ai condizionali in TeX e LaTeX e alle loro limitazioni, si introducono le strutture condizionali disponibili con expl3, l'interfaccia di programmazione del futuro LaTeX3.

Abstract

After briefly touching on conditionals in TeX and LaTeX, with their limitations, we introduce the conditional structures available with expl3, the programming interface of the future LaTeX3.

1 Condizionali in TeX

Non ci possono essere dubbi sul fatto che le strutture condizionali siano fondamentali in qualsiasi linguaggio di programmazione e, ovviamente, TeX non fa eccezione.

Tuttavia, i condizionali in TeX sono, come al solito, piuttosto peculiari. Un condizionale prevede la sintassi

⟨*condizionale*⟩⟨*test*⟩
  ⟨*codice per vero*⟩
`\else`
  ⟨*codice per falso*⟩
`\fi`

dove la parte `\else`⟨*codice per falso*⟩ è facoltativa.

Il ⟨*test*⟩ dipende dal ⟨*condizionale*⟩, secondo la seguente tabella:

- `\ifx`, il cui ⟨*test*⟩ consiste dei due token che seguono (senza espansione);
- `\if`, il cui ⟨*test*⟩ consiste dei due token che seguono (dopo espansione);
- `\ifcat`, il cui ⟨*test*⟩ consiste dei due token che seguono (dopo espansione);
- `\ifnum`, il cui ⟨*test*⟩ deve avere la forma ⟨*intero*⟩⟨*relazione*⟩⟨*intero*⟩;
- `\ifodd`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifdim`, il cui ⟨*test*⟩ deve avere la forma ⟨*lunghezza*⟩⟨*relazione*⟩⟨*lunghezza*⟩;
- `\ifvoid`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifhbox`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifvbox`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifeof`, il cui ⟨*test*⟩ deve essere un ⟨*intero*⟩;
- `\ifhmode`;
- `\ifvmode`;
- `\ifmmode`;
- `\ifinner`;
- `\iftrue`;
- `\iffalse`.

Per gli ultimi cinque il ⟨*test*⟩ è vuoto; per `\ifvoid`, `\ifhbox` e `\ifvbox` il numero ⟨*intero*⟩ è di solito definito con `\newsavebox` o `\newbox` (se non è il numero di uno dei registri provvisori); per `\ifeof` il numero ⟨*intero*⟩ è di solito definito con `\newread`.

La ⟨*relazione*⟩ è uno fra $<_{12}$, $=_{12}$ e $>_{12}$, dove il codice di categoria *deve* essere quello indicato. Siccome TeX esegue l'espansione per trovare la ⟨*relazione*⟩, in caso di dubbio si può scrivere `\string<` e simili.

La descrizione di questi condizionali può essere trovata nelle solite fonti, per esempio Gregorio (2009) o, ovviamente, Knuth (1986) e Eijkhout (1992). Esiste anche `\ifcase`, con una sintassi ancora più peculiare; ε-TeX e i vari motori ne definiscono altri ancora. Nel seguito, `\ifZ` rappresenterà uno dei condizionali primitivi.

I più importanti, oltre agli ovvi `\ifnum`, `\ifodd` e `\ifdim`, sono `\ifx` e `\if`. Il primo confronta il *significato* dei due token che seguono, il secondo il loro codice di carattere (un token simbolico è considerato come un carattere oltre il dominio possibile). Il condizionale seguirà il ramo 'vero' se i due token sono uguali, il ramo 'falso' altrimenti. Due token sono uguali per `\ifx` quando `\show` stampa la stessa cosa (con qualche eccezione in casi esoterici). Due token carattere daranno lo stesso risultato con `\if` e `\ifx`, ma occorre ricordare che il primo espande, il secondo no.

Va anche notato che i condizionali sono del tutto indipendenti dai *gruppi*, quindi `\ifx aa{\else}\fi` equivale a inserire una graffa aperta. Un trucco del genere è usato in LaTeX nella forma

```
{\ifnum0=‘}\fi
\ifnum0=‘{\fi}
```

per 'ingannare' il meccanismo delle tabelle; si noti che i due ⟨*test*⟩ sono entrambi falsi. L'analisi di questa costruzione e dei suoi usi è interessante, ma ci porterebbe troppo distante.

Quando si definisce un nuovo condizionale con `\newif`, in realtà si stabilisce l'equivalenza del nuovo condizionale con `\iftrue` o `\iffalse`. Qualsiasi istruzione del tipo

```
\let\cs=\ifZ
```

rende `\cs` del tutto equivalente al condizionale primitivo, in particolare per quanto riguarda l'accoppiamento con `\else` e `\fi`.

2 Espansione dei condizionali

I condizionali `\ifZ`, al pari di `\else` e `\fi`, sono *espandibili*. L'espansione di `\ifZ` funziona così:

1. viene deciso se il ⟨*test*⟩ è vero o falso;
2. nel caso sia vero, l'espansione di `\ifZ` è semplicemente ciò che rimane rimuovendo `\ifZ` e il ⟨*test*⟩;
3. nel caso sia falso, l'espansione di `\ifZ` è tutto ciò che va da `\else` (escluso) fino a `\fi`;
4. se non c'è alcun `\else`, l'espansione è il solo `\fi`.

Un contatore interno viene aumentato di uno per il `\ifZ` che ha avviato la procedura e anche per ogni `\ifZ` incontrato fino a che si trova un `\else` o `\fi` allo stesso livello. Ogni `\fi` diminuisce di uno il contatore interno, così che TEX sa quando finire. Il testo ignorato quando il ⟨*test*⟩ è falso non viene interpretato in alcun modo, ma si tiene solo conto dei condizionali trovati durante la *traversata*.

Per vedere questo tipo di espansione all'opera, si provi

```
\def\fdef#1{%
  \expandafter\def
  \expandafter#1\expandafter
}
\fdef\testa{\iftrue a\else b\fi}
\show\test
\fdef\testb{\iffalse a\else b\fi}
\show\test
```

L'idea è che `\fdef` definisce l'argomento come l'espansione di primo livello del testo seguente. Si otterrà

```
> \testa=macro:
->a\else b\fi .

> \testb=macro:
->b\fi .
```

come previsto. In questi casi il ⟨*test*⟩ è vuoto; si può verificare che il ⟨*test*⟩ viene rimosso con `\if TT` o `\if TF` invece di `\iftrue` o `\iffalse`

L'espansione di `\else` funziona allo stesso modo dell'espansione di `\ifZ`, eccetto che il contatore interno non viene modificato da `\else`, ma solo dai condizionali che si trovano fino al `\fi` corrispondente che viene rimosso anch'esso. Per finire, l'espansione di `\fi` è vuota, dopo che il contatore interno è diminuito di uno.

La rimanenza di `\else` o `\fi` è spesso causa di grattacapi, che si risolvono in vari modi. Uno dei più comuni è quello di adoperare `\@firstoftwo` e `\@secondoftwo`, definendo macro di livello più alto che prendono argomenti. Per esempio

```
\newcommand{\eqtest}[4]{%
  \ifx#1#2%
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {#3}%
  {#4}%
}
```

produce il terzo argomento se i primi due argomenti sono token uguali (per `\ifx`), il quarto altrimenti. I due `\expandafter` servono a togliere di mezzo `\else` fino a `\fi` oppure `\fi` *prima* che `\@firstoftwo` e `\@secondoftwo` siano espansi a loro volta. Perciò nel caso vero e falso si avrà, rispettivamente

```
\@firstoftwo{#3}{#4}
\@secondoftwo{#3}{#4}
```

(i parametri `#3` e `#4` sono già stati sostituiti con gli argomenti effettivi).

Confusi? Be', ci vuole un po' di studio e di pazienza per entrare nei meandri di `\ifZ`, `\else`, `\fi` e `\expandafter` e venirne fuori.

3 Condizionali complessi

Non c'è alcun supporto primitivo per espressioni booleane. Se volessimo implementare un condizionale che valuta un "e", potremmo fare così: ci interessa sapere se un certo numero è maggiore di 0 e minore di 42; nel caso vogliamo eseguire `A`, altrimenti `B`. Un modo è facile:

```
\ifnum#1>0
  \ifnum#1<42
    A%
  \else
    B%
\else
  B%
\fi
```

La duplicazione di codice è però indesiderabile. Un altro modo è

```
\ifnum
  \ifnum#1>0 0\else 1\fi
  \ifnum#1<42 0\else 1\fi
=0
  A%
```

```
\else
  B%
\fi
```

che funziona perché i due condizionali interni producono 0 solo se entrambi falsi. Si potrebbero avere condizioni aggiuntive, con la stessa idea; similmente si può implementare il connettivo "o".

È chiaro però che espressioni booleane più complesse diventano difficili da gestire. Esistono altri modi, più o meno misteriosi.

Un'altra applicazione, con una strategia diversa per i tempi di espansione è

```
\newif\ifxetexorluatex
\begingroup
  \catcode94=7 % ASCII 94 is ^
  \catcode0=9 % ASCII 0 is ignored
  \catcode30=12 % ^^^ is ASCII 30
\expandafter\endgroup
\if\relax^^^^0000\relax
  \xetexorluatextrue
\else
  \xetexorluatexfalse
\fi
```

Qui l'espansione di `\if` avviene prima che termini il gruppo con le impostazioni dei codici di categoria date per precauzione. Se il motore è XeTeX o LuaTeX, `^^^^0000` diventa il carattere ASCII 0, quindi ignorato perché di categoria 9, e `\if` confronta `\relax` con un altro `\relax`. Altrimenti, `^^^` rappresenta il carattere ASCII 30 che non è uguale a `\relax`; i token `^000\relax\xetexorluatextrue` diventano il ⟨*codice per vero*⟩ e sono quindi ignorati. Più *code golfing* che altro, ma interessante lo stesso.

4 Condizionali in LaTeX

In LaTeX, per gli scopi della programmazione, esistono vari condizionali piuttosto utili:

```
\@ifnextchar
\@ifstar
\@ifpackageloaded
\@ifclassloaded
\@ifpackagelater
\@ifclasslater
\@ifpackagewith
\@ifclasswith
```

che seguono il paradigma

`\@ifZ`⟨*test*⟩`{`⟨*true*⟩`}{`⟨*false*⟩`}`

Per esempio, il ⟨*test*⟩ per `\@ifnextchar` è un singolo token; per `\@ifstar` è vuoto; per `\@ifpackagewith` è una coppia di argomenti tra graffe che rappresentano rispettivamente il nome di un pacchetto e un'opzione. Esiste anche `\@ifdefinable` che però accetta solo l'argomento per ⟨*true*⟩; nel caso il test risulti falso, cioè il token passato come primo argomento sia già definito, il codice ⟨*false*⟩ è implicito, cioè un messaggio di errore.

Un'importante estensione è fornita dal pacchetto etoolbox che definisce alcuni condizionali nello stile LaTeX e permette anche, con una sintassi nemmeno troppo complessa, di valutare espressioni booleane basate su altri condizionali. Anche ifthen ha la possibilità di valutare espressioni booleane, ma è meno flessibile e, a differenza di etoolbox, non prevede l'espandibilità.

Per esempio, la condizione `\ifxetexorluatex` può essere impostata con i pacchetti ifxetex e ifluatex scrivendo [1]

```
\newif\ifxetexorluatex
\ifboolexpr{
  bool{xetex} or bool{luatex}
}{%
  \xetexorluatextrue
}{%
  \xetexorluatexfalse
}
```

Per il problema precedente, la soluzione sarebbe

```
\newcommand{\ininterval}[1]{%
  \ifboolexpe{
    test{\ifnumcomp{#1}{>}{0}}
    and
    test{\ifnumcomp{#1}{<}{42}}
  }{#1: Yes}{#1: No}
}
```

La macro `\ifboolexpe` è espandibile, ma le espressioni ammesse sono limitate.

Definire macro ricorsive con etoolbox è più facile; per esempio, se volessimo inscatolare i termini di una stringa di caratteri, possiamo scrivere

```
\documentclass{article}
\usepackage{etoolbox}

\makeatletter
\newcommand{\boxit}[1]{%
  \boxit@aux#1\@nil
}
\def\boxit@aux#1#2\@nil{%
  \fbox{\strut#1}%
  \ifblank{#2}
    {}
    {\kern-\fboxrule\boxit@aux#2\@nil}%
}
\makeatother

\begin{document}

X\boxit{abc}X

\end{document}
```

1. https://tex.stackexchange.com/a/47711/

e il risultato sarebbe

X a b c X

In questo caso si potrebbe adoperare un altro trucco

```
\def\boxit@aux#1#2\@nil{%
  \fbox{\strut#1}%
  \if\relax\detokenize{#2}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\kern-\fboxrule\boxit@aux#2\@nil}%
}
```

ma `\ifblank` è più versatile e risulta vero anche se l'argomento contiene solo spazi.

Il pacchetto etoolbox fornisce anche `whileexpr` che prende due argomenti: il primo è un'espressione booleana, il secondo il codice da eseguire finché l'espressione è vera. Molto più flessibile di `\@whilenum`, `\@whiledim` e `\@whilesw`.

Ci sarebbero molte altre soluzioni al problema, ma è tempo di guardare avanti.

5 Ora c'è expl3

L'ambiente di programmazione expl3 (The LaTeX Project, 2015) rinomina i condizionali primitivi secondo le sue convenzioni:

- `\ifx` diventa `\if_meaning:w`;
- `\if` diventa `\if_charcode:w` (ma anche `\if:w`);
- `\ifcat` diventa `\if_catcode:w`;
- `\ifnum` diventa `\if_int_compare:w`;
- `\ifodd` diventa `\if_int_odd:w`;
- `\ifdim` diventa `\if_dim:w`;
- `\ifvoid` diventa `\if_box_empty:N`;
- `\ifhbox` diventa `\if_hbox:N`;
- `\ifvbox` diventa `\if_vbox:N`;
- `\ifeof` diventa `\if_eof:w`;
- `\ifhmode` diventa `\if_mode_horizontal:`;
- `\ifvmode` diventa `\if_mode_vertical:`;
- `\ifmmode` diventa `\if_mode_math:`;
- `\ifinner` diventa `\if_mode_inner:`;
- `\iftrue` diventa `\if_true:`;
- `\iffalse` diventa `\if_false:`.

Vanno aggiunti `\if_case:w`, `\else:` e `\fi:`.

Meglio chiarire subito che l'uso diretto dei condizionali che hanno `w` come *signature* è sconsigliato per la programmazione ad alto livello. Tuttavia, anche gli altri condizionali che richiedono `\fi:` (con il facoltativo `\else:`) hanno una controparte con argomenti che sono la versione preferita.

Per esempio, se non ci serve davvero la capacità di `\if_charcode:w` di espandere i token che seguono, è meglio adoperare `\token_if_eq_meaning:NNTF` secondo la sintassi

```
\token_if_eq_meaning:NNTF
  ⟨token1⟩
  ⟨token2⟩
  {⟨codice per vero⟩}
  {⟨codice per falso⟩}
```

Esistono anche

```
\token_if_eq_charcode:NNTF
\token_if_eq_catcode:NNTF
\token_if_group_begin:NTF
\token_if_group_end:NTF
\token_if_math_toggle:NTF
```

e molti altri per esaminare la natura del token passato come primo argomento.

In tutti i casi di condizionali predefiniti, gli argomenti di tipo `TF` sono facoltativi, nel senso che esistono anche

```
\token_if_eq_meaning:NNT
\token_if_eq_meaning:NNF
```

per avere codice più compatto. Gli argomenti di tipo `T` e `F` non differiscono da quelli di tipo `n` per quanto riguarda la sintassi, ma vengono distinti proprio per applicare queste abbreviazioni e per maggiore chiarezza sul loro impiego.

Per comparazioni di interi, ci sono due possibilità:

```
\int_compare:nNnTF
\int_compare:nTF
```

La prima è più efficiente perché è un'interfaccia diretta con `\if_int_compare:w`

```
\int_compare:nNnTF { 2 } < { 3 }
 { A }
 { B }
```

che però nel primo e terzo argomento accetta *espressioni intere* senza bisogno di `\int_eval:n`.

Il secondo, seppure un po' meno efficiente, è di gran lunga più flessibile: il problema dell'intervallo analizzato prima si risolve con

```
\int_compare:nTF { 0 < #1 < 42 }
 { A }
 { B }
```

Il numero di relazioni ammesse è arbitrario; non solo, anche le relazioni sono più flessibili. Se, per esempio, gli estremi dell'intervallo sono accettabili, nella programmazione classica si dovrebbero adoperare -1 e 43. Qui, invece, possiamo più facilmente scrivere

```
\int_compare:nTF { 0 <= #1 <= 42 }
 { A }
 { B }
```

Le relazioni ammesse sono `=`, `<`, `<=`, `>`, `>=` e `!=` (che sta per $\neq$). Anche in questo caso è possibile usare *espressioni intere*:

```
\int_compare:nTF
 { \int_mod:nn {6}{3} <= #1 <= 6*7 }
 { A }
 { B }
```

Il prezzo da pagare in termini di efficienza è elevato,[2] ma la flessibilità è enormemente maggiore.

Non è il caso di elencare tutti i condizionali disponibili, perché sono moltissimi. Vanno citati

```
\dim_compare:nNnTF
\dim_compare:nTF
\fp_compare:nNnTF
\fp_compare:nTF
```

che funzionano in modo analogo ai precedenti.

Naturalmente expl3 fornisce anche *variabili booleane*:

```
\bool_new:N \l_manual_foo_bool
\bool_new:N \g_manual_foo_bool
```

definiscono nuove variabili booleane, una *locale* e una *globale* (valgono le solite convenzioni), inizialmente con valore 'falso'. Queste possono essere modificate con

```
\bool_set_true:N \l_manual_foo_bool
\bool_set_false:N \l_manual_foo_bool
\bool_gset_true:N \g_manual_foo_bool
\bool_gset_false:N \g_manual_foo_bool
```

e adoperate come primo argomento di `\bool_if:NTF`. Per esempio

```
\bool_if:NTF \l_manual_foo_bool
 { A }
 { B }
```

è la controparte del classico

```
\iffoo A\else B\fi
```

Dove però expl3 sbaraglia la concorrenza è nella possibilità di sfruttare *espressioni booleane*. Un'espressione booleana *atomica* è una variabile booleana oppure un *predicato*. Ogni condizionale espandibile predefinito fornisce anche la forma predicativa, per esempio

```
\int_compare_p:n { #1 > 0 }
```

è un predicato che darà 'vero' se l'argomento della macro è un intero positivo, 'falso' altrimenti. Il nome della forma predicativa è facilmente prevedibile, perché si ottiene togliendo `TF` e aggiungendo `_p` prima della *signature*.

Un'espressione booleana si ottiene applicando nel modo tradizionale connettivi e parentesi. I connettivi ammessi sono `!`, `&&` e `||`, per 'non', 'e', 'o' rispettivamente, in ordine di precedenza. L'esempio nella documentazione è

```
\int_compare_p:n { 1 = 1 } &&
  (
   \int_compare_p:n { 2 = 3 } ||
   \int_compare_p:n { 4 <= 4 } ||
   \str_if_eq_p:nn { abc } { def }
  ) &&
  ! \int_compare_p:n { 2 = 4 }
```

La funzione `\bool_if:nTF` prende come primo argomento un'espressione booleana.

Va notato che *tutte* le espressioni atomiche sono valutate, per poi stabilire la verità o falsità dell'intera espressione. Fino a qualche tempo fa, si era tentato di permettere valutazioni *lazy*, cioè solo delle espressioni atomiche necessarie, ma questo si è rivelato troppo complicato. Un esempio è un baco di fontspec venuto alla luce quando il cambio è diventato operativo;[3] il codice adoperava

```
\bool_if:nTF
 {
  \tl_if_single_p:n {##1}
  &&
  \token_if_cs_p:N ##1
 }
 {true}{false}
```

che fallisce quando `##1` è `'A` (e in molti altri casi): infatti la prima espressione atomica dà falso, ma la seconda è illecita, perché lascia `A` nella lista di token da esaminare. Questo si risolve con `\bool_lazy_and:nnTF` che appunto fornisce una valutazione *lazy*:

```
\bool_lazy_and:nnTF
 { \tl_if_single_p:n {##1} }
 { \token_if_cs_p:N ##1 }
 {true}{false}
```

(i codici per vero e falso non sono rilevanti). Con questa funzione, se la prima espressione risulta falsa la seconda non viene nemmeno esaminata, perché l'espressione globale con il connettivo 'e' è certamente falsa. Il risultato dà vero solo se la prima espressione è vera, cioè `##1` è un singolo token, e anche la seconda, cioè il token è un token simbolico. Quando la prima espressione risulta falsa, la seconda viene semplicemente ignorata (che sia valida sintatticamente o no).

2. Un rapido test fatto eseguendo dieci milioni di istruzioni dice che la seconda forma è cinque volte più lenta della prima: 50 secondi contro 10.

3. https://tex.stackexchange.com/questions/382880/

Le valutazioni *lazy* sono disponibili solo per espressioni che coinvolgono i connettivi 'e' e 'o':

```
\bool_lazy_and:nnTF
\bool_lazy_or:nnTF
\bool_lazy_all:nTF
\bool_lazy_any:nTF
```

Il condizionale precedente potrebbe essere scritto come

```
\bool_lazy_and:nTF
 {
  { \tl_if_single_p:n {##1} }
  { \token_if_cs_p:N ##1 }
 }
 {true}{false}
```

ma quando le espressioni atomiche sono solo due è più efficiente la versione a due argomenti.

Possiamo ovviamente definire nuovi condizionali. Prendiamo come esempio proprio una funzione che produca vero se l'argomento è una lista di token che consista di un singolo token simbolico. Ecco una possibilità:

```
\prg_new_conditional:Nnn
 \manual_tl_if_singlecs:n
  { TF, T, F, p }
  {
   \bool_lazy_and:nnTF
    { \tl_if_single_p:n {#1} }
    { \token_if_cs_p:N #1 }
   { \prg_return_true: }
   { \prg_return_false: }
  }
```

che definirà

```
\manual_tl_if_singlecs:nTF
\manual_tl_if_singlecs:nT
\manual_tl_if_singlecs:nF
\manual_tl_if_singlecs_p:n
```

Il primo argomento è il nome del nuovo condizionale (meglio se contiene `if`), con una *signature* provvisoria che indichi quanti argomenti verranno passati per costruire il condizionale; il secondo è l'elenco delle funzioni effettivamente definite (dove `p` sta per la forma predicativa).

La versione expl3 del condizionale per 'motori Unicode' sarebbe

```
\prg_new_conditional:Nnn
 \manual_if_unicode_engine:
  { TF, T, F, p }
  {
   \bool_if:nTF
    {
     \sys_if_engine_xetex_p:
     ||
     \sys_if_engine_luatex_p:
    }
    { \prg_return_true: }
    { \prg_return_false: }
  }
```

Non è possibile specificare la forma predicativa nella definizione se il codice nel terzo argomento contiene parti non espandibili (per esempio `\tl_if_in:nnTF`). Nel manuale di expl3 le funzioni espandibili sono marcate con una stellina rossa. Se il codice non è espandibile, si userà

```
\prg_new_protected_conditional:Nnn
```

6 *Case switch*

Un *case switch* è un particolare tipo di condizionale 'multiplo'. Alcuni linguaggi forniscono *else if* e direttamente funzioni per *case switch* più complessi che però non esistono in TEX. Ovviamente si possono programmare e si trovano parecchi esempi su TUGboat.

In TEX si trova una funzione primitiva in tutti i sensi, cioè `\ifcase`:

```
\ifcase⟨intero⟩
  ⟨caso 0⟩\or
  ⟨caso 1⟩\or
  ...
  ⟨caso n⟩\else
  ⟨altri casi⟩\fi
```

L'espansione è analoga a quella dei condizionali: TEX esamina l'intero e scarta tutto ciò che trova fino all'`\or` (compreso) che precede il caso corrispondente, oppure a `\else` o a `\fi`. L'espansione di `\or` è vuota ed elimina tutto quanto si trova fino a `\fi`.

Un *case switch* elementare si può ottenere assegnando valori interi a macro opportune. Per esempio

```
\def\cs@apple{0}
\def\cs@peach{1}
\def\cs@apricot{2}
\def\frutto#1{%
  \ifcase\csname cs@#1\endcsname
    pomo\or
    persego\or
    armelin\else
    fruto\fi
}
```

Le difficoltà di questo approccio sono evidenti, perché occorre far corrispondere manualmente il codice della macro con la lista dei numeri assegnati.

La stessa cosa in expl3:

```
\cs_new:Nn \manual_fruit:n
 {
  \str_case:nnF { #1 }
   {
    {apple}{pomo}
    {peach}{persego}
```

```
    {apricot}{armelin}
   }
   {fruto}
 }
```

Esistono varie altre funzioni predefinite:

```
\str_case_x:nnTF
\tl_case:NnTF
\int_case:nnTF
\dim_case:nnTF
```

In tutte l'argomento corrispondente a `T` può essere omesso come abbiamo fatto nell'esempio precedente; se adoperato, il testo viene inserito dopo quello corrispondente al caso *matched*. Normalmente non lo si adopera.

La versione `\str_case_x:nnTF` espande sia il primo argomento sia la prima parte dei casi. Un esempio d'uso[4] è nella tavola 1. Si noti che

```
\str_case:nnF
 {\languagename}
 {
  {english}{EN}
  {spanish}{SP}
 }
 {??}
}
```

produrrebbe `??` in ogni caso perché la stringa data come primo argomento non viene interpretata in alcun modo; con `\str_case_x:nnF` invece il primo argomento viene espanso prima del confronto.

Un'interessante macro che implementa i vari *case switch* a livello utente può essere[5]

```
\NewExpandableDocumentCommand
 {\switchcondition}
 {O{string}mmO{}}
 {
  \use:c { manual_#1_switch:nnn }
   { #2 }
   { #3 }
   { #4 }
 }

\cs_new:Nn \manual_string_switch:nnn
 {
  \str_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_stringx_switch:nnn
 {
  \str_case_x:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_token_switch:nnn
 {
  \tl_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_integer_switch:nnn
```

4. https://tex.stackexchange.com/a/348677/4427
5. https://tex.stackexchange.com/a/386307/4427

```
 {
  \int_case:nnF { #1 } { #2 } { #3 }
 }
\cs_new:Nn \manual_dimen_switch:nnn
 {
  \dim_case:nnF { #1 } { #2 } { #3 }
 }
\ExplSyntaxOff
```

Un esempio d'uso è

```
\newcommand{\placement}[1]{%
  \switchcondition{#1}{
    {ul}{\let\position\AtPageUpperLeft}
    {ll}{\let\position\AtPageLowerLeft}
    {ur}{\let\position\AtPageUpperRight}
    {lr}{\let\position\AtPageLowerRight}
  }[\let\position\ERROR]%
}
```

L'argomento facoltativo di `\switchcondition` decide il tipo di *case switch*.

Riferimenti bibliografici

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

Gregorio, E. (2009). «Appunti di programmazione in TEX e LATEX». http://profs.scienze.univr.it/~gregorio/introtex.pdf.

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

The LATEX Project (2015). «The expl3 package and LATEX3 programming». http://ctan.org/pkg/l3kernel.

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

```
\documentclass{article}
\usepackage[english,ngerman]{babel}
\usepackage{amsmath}
\usepackage{xparse}

\ExplSyntaxOn
\NewDocumentCommand\DeclareBabelMathOperator{mmO{???}}
 {
  \DeclareMathOperator{#1}
   {
    \str_case_x:nnF { \use:c { bbl@main@language } } { #2 } { #3 }
   }
 }
\NewDocumentCommand\DeclareVariableBabelMathOperator{mmO{???}}
 {
  \DeclareMathOperator{#1}
   {
    \str_case_x:nnF { \languagename }{ #2 } { #3 }
   }
 }
\ExplSyntaxOff


\DeclareBabelMathOperator{\range}{
  {english}{ran}
  {ngerman}{Bild}
}
\DeclareBabelMathOperator{\kernel}{
  {english}{ker}
  {ngerman}{Kern}
}[ker]
\DeclareVariableBabelMathOperator{\vrange}{
  {english}{ran}
  {ngerman}{Bild}
}
\DeclareVariableBabelMathOperator{\vkernel}{
  {english}{ker}
  {ngerman}{Kern}
}[ker]

\begin{document}

\section{Fixed names}
\[
  \dim(V) = \dim(\range(T)) + \dim(\kernel(T))
\]
\begin{otherlanguage*}{english}
\[
  \dim(V) = \dim(\range(T)) + \dim(\kernel(T))
\]
\end{otherlanguage*}

\section{Variable names}
\[
  \dim(V) = \dim(\vrange(T)) + \dim(\vkernel(T))
\]
\begin{otherlanguage*}{english}
\[
  \dim(V) = \dim(\vrange(T)) + \dim(\vkernel(T))
\]
\end{otherlanguage*}

\end{document}
```

1 Fixed names

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

2 Variable names

$$\dim(V) = \dim(\mathrm{Bild}(T)) + \dim(\mathrm{Kern}(T))$$

$$\dim(V) = \dim(\mathrm{ran}(T)) + \dim(\mathrm{ker}(T))$$

Tavola 1: Esempio d'uso di `\str _case_x:nnF`