Espressioni regolari in LaTeX

Enrico Gregorio

Sommario

Con expl3, l'ambiente di programmazione per LaTeX3, e il pacchetto l3regex possiamo usare espressioni regolari come in Perl e altri linguaggi.

Abstract

With expl3, the programming environment for LaTeX3, and the l3regex package we can use regular expressions like in Perl e other languages.

1 Introduzione

Cominciamo con il solito *toy problem*. Vogliamo stampare in neretto tutte le 'a' di una parola data come argomento a una macro. Questo è abbastanza facile con una macro ricorsiva

```
\makeatletter
\def\bolda#1{\bolda@aux#1a\bolda@aux}
\def\bolda@aux#1a#2\bolda@aux{%
  #1%
  \if\relax#2\relax
   \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\textbf{a}\bolda@aux#2\bolda@aux}%
}
\makeatother
\bolda{aiutare}
\bolda{aiuola}
```

aiut**a**re **a**iuol**a**

Più complicato sarebbe stampare tutte le vocali, maiuscole e minuscole comprese, in neretto. Si può fare, usando macro di van der Laan (van der Laan, 1993).

```
\def\fifo#1{%
  \ifx\ofif#1\ofif\fi
  \process{#1}\fifo
}
\def\ofif#1\fifo{\fi}
\def\loc#1#2{%locate #1 in #2
  \def\locate##1#1##2\end{%
    \ifx\empty##2\empty
      \foundfalse
    \else
      \foundtrue
    \fi
  }%
  \locate#2.#1\end
}
\newif\iffound
\def\boldvowels#1{%
  \def\process##1{%
    \uppercase{\loc##1}{AEIOU}%
    \iffound\textbf{##1}\else##1\fi
  }%
\fifo #1\ofif
}

\boldvowels{aiutare}
\boldvowels{Aiuola}
```

aiuta**re A**iuol**a**

Vediamo subito qualcosa di più maneggevole.

```
% \usepackage{xparse,l3regex}
\ExplSyntaxOn
\NewDocumentCommand{\boldvowels}{m}
 {
  \manual_boldvowels:n { #1 }
 }
\tl_new:N \l_manual_boldvowels_tl
\cs_new_protected:Nn \manual_boldvowels:n
 {
  \tl_set:Nn \l_manual_boldvowels_tl { #1 }
  \regex_replace_all:nnN
   { ([AEIOUaeiou]) }
   { \c{textbf}\cB\{\1\cE\} }
   \l_manual_boldvowels_tl
  \tl_use:N \l_manual_boldvowels_tl
 }
\ExplSyntaxOff

\boldvowels{aiutare Aiuola}
```

aiuta**re A**iuol**a**

Due fatti sono indiscutibili: il primo è che questo codice è centinaia di volte meno efficiente dell'altro, il secondo è che questo codice è infinite volte più potente. Il pacchetto l3regex (The LaTeX team, 2014b) fa parte di expl3 (The LaTeX team, 2014a), il livello di programmazione di LaTeX3. È ancora classificato sperimentale, ma è ormai piuttosto stabile.

2 Introduzione a l3regex

Vediamo in dettaglio il codice, sistemando prima i dettagli. I comandi `\ExplSyntaxOn` e `\ExplSyntaxOff` servono a entrare nell'ambiente di programmazione di LaTeX3 e a uscirne. Il comando a livello utente passa il controllo a una funzione interna; è buona norma farlo perché spesso permette di evitare duplicazioni di codice. Le funzioni per eseguire sostituzioni con espressioni regolari richiedono poi che il testo da esaminare sia contenuto in una variabile *token list*, che quindi definiamo. Quin-

di la parte più importante, cioè la funzione interna, del tipo `protected` perché contiene assegnazioni.

Infatti, l'argomento viene memorizzato nella variabile e quindi entra in azione la funzione davvero importante: `\regex_replace_all:nnN`, come dice la sua firma, richiede tre argomenti:

```
\regex_replace_all:nnN
  ⟨regex search⟩
  ⟨regex replace⟩
  ⟨token list variable⟩
```

Il terzo argomento è una variabile di tipo *token list* e non ne parliamo più; questa variabile, dopo il massaggio eseguito dall'istruzione `\regex_replace:nnN` verrà usata alla fine. Il primo e il secondo argomento sono invece due *espressioni regolari*. Chiunque abbia un minimo di esperienza al riguardo riconoscerà nella prima una *classe*, cioè un gruppo di caratteri che verranno cercati nel contenuto della variabile; qui sono le vocali, cioè

```
[AEIOUaeiou]
```

Le parentesi intorno alla classe servono a "ricordare" il carattere trovato in modo che possa essere usato nella sostituzione. Infatti l'espressione regolare per la sostituzione contiene `\1`, che sta per la sottoespressione che costituisce un *match*. Se nell'espressione di sostituzione si fosse usato `\1\1`, le vocali sarebbero raddoppiate.

L'analisi dell'espressione di sostituzione è un po' più complicata e la rimandiamo a dopo. Per ora descriveremo le principali caratteristiche delle espressioni regolari. Lo standard POSIX è rispettato il più possibile, quindi

- `.` rappresenta un carattere qualsiasi, più precisamente un *token* qualsiasi;
- ogni carattere alfabetico rappresenta sé stesso;
- ogni cifra rappresenta sé stessa;
- ogni carattere non alfabetico preceduto dalla barra rovescia rappresenta sé stesso (per esempio, `\?` rappresenta `?` e `\.` rappresenta il punto);
- `\d` rappresenta una cifra qualsiasi, `\D` qualsiasi token che non sia una cifra;
- `\s` rappresenta uno spazio, `\S` qualsiasi token che non sia uno spazio;
- `\w` rappresenta una lettera, una linea bassa `_` oppure una cifra, mentre `\W` qualsiasi altro token.

Ci sono altre scorciatoie, più che altro per compatibilità con lo standard, ma che sono meno importanti per l'uso con LaTeX. Si possono naturalmente esprimere classi:

- `[...]` indica una classe 'positiva';
- `[^...]` indica una classe 'negativa', per esempio `[^AEIOUaeiou]` rappresenta ogni token che non sia una vocale;
- `[:`⟨*nome*⟩`:]` indica una classe predefinita, i nomi sono gli stessi dello standard POSIX;
- `[:^`⟨*nome*⟩`:]` indica il complementare della classe predefinita;
- `[x-y]` indica un *intervallo* per esempio `[a-z]` indica le lettere minuscole.

Molto importanti sono gli operatori di ripetizione che sono identici allo standard e che vengono trasformati da 'ingordi' (*greedy*) in 'pigri' (*lazy*) con l'aggiunta di `?`:

- `?` e `??` indicano nessuna o una occorrenza;
- `*` e `*?` indicano zero o più occorrenze;
- `+` e `+?` indicano una o più occorrenze;
- `{`⟨n⟩`}` indica esattamente ⟨n⟩ occorrenze (la versione pigra non ha senso);
- `{`⟨n⟩`,}` e `{`⟨n⟩`,}?` indicano ⟨n⟩ o più occorrenze;
- `{`⟨n_1⟩`,`⟨n_2⟩`}` e `{`⟨n_1⟩`,`⟨n_2⟩`}?` indicano almeno ⟨n_1⟩ e non più di ⟨n_2⟩ occorrenze.

Si possono denotare punti particolari:

- `\b` indica inizio o fine di una parola, cioè un punto che è preceduto da un token `\w` e seguito da un token `\W` o viceversa; l'inizio e la fine del testo sono considerati come se appartenessero alla classe `\W`;
- `^` o `\A` indicano l'inizio del testo;
- `$`, `\Z` o `\z` indicano la fine del testo.

È possibile usare `(...)`, `(?:...)` e `(?|...)` per 'catturare' (o no) token da usare poi nell'espressione di sostituzione con `\1`, `\2` e così via esattamente come nello standard POSIX.

Attenzione! Un'importante deviazione dallo standard, inevitabile con TeX, è che gli spazi nelle espressioni regolari sono *ignorati*. Uno spazio va espresso con `\␣` (o con `\s`).

Un'altra possibilità è quella di adoperare il prefisso `\u`; precisamente, se `\l_manual_foo_tl` è una variabile token list (può anche essere globale o costante, ovviamente), con `\u{l_manual_foo_tl}` possiamo indicare il valore della variabile in un'espressione regolare. Per esempio, se volessimo cercare

```
\begin{minipage}[t]{\textwidth}
```

invece di una complicata espressione come

```
\c{begin}\cB.minipage\cE\[t\]
  \cB.\c{textwidth}\cE.
```

possiamo più semplicemente assegnare quel complesso a una variabile con

```
\tl_set:Nn \l_manual_foo_tl
 {\begin{minipage}[t]{\textwidth}}
```

e specificare `\u{l_manual_foo_tl}` nell'espressione regolare.

3 Novità in l3regex

La sostanziale differenza con il trattamento delle espressioni regolari in Perl o `sed` è che in LaTeX abbiamo *token* e non semplici successioni di caratteri. Per fare un esempio facile, l'espressione regolare `foo` non troverà nulla se il testo da esaminare contiene `abc\foo`, perché `\foo` è un singolo oggetto agli occhi di TeX che ha poco a che vedere, internamente, con il nome della macro. Analogamente, se ci interessa trovare con `\{` una 'vera' parentesi graffa e non una che sia, per qualche motivo, un carattere stampabile.

Andando un po' più in profondità, i confronti sono eseguiti tramite `\if`, che ignora il codice di categoria. Per ottenere risultati più precisi che considerano anche la duplice natura dei caratteri in TeX sono disponibili prefissi che indicano che tipo di token si sta effettivamente cercando o si vuole sostituire. I prefissi sono tutti della forma `\cX`, dove `X` è una lettera tra `CBEMTPUDSLOA` con le seguenti corrispondenze, dove useremo le solite convenzioni riguardo ai caratteri speciali:

- `\cC` per una sequenza di controllo;
- `\cB` per `{`;
- `\cE` per `}`;
- `\cM` per `$`;
- `\cT` per `&`;
- `\cP` per `#`;
- `\cU` per `_`;
- `\cD` per `^`;
- `\cS` per uno spazio;
- `\cL` per una lettera;
- `\cO` per una 'non lettera';
- `\cA` per un carattere attivo.

Con il prefisso `\c` seguito da un argomento tra graffe contenente caratteri si indica quella precisa sequenza di controllo. I prefissi possono essere combinati anche con le classi; per esempio, l'espressione

```
[\c0\d \c[LO][A-F]]
```

corrisponde alla specifica delle cifre esadecimali in TeX: sono le cifre, con codice di categoria 12, oppure le lettere maiuscole ABCDEF con categoria 11 oppure 12 (si veda l'esercizio 24.1 del TeXbook).

Tipicamente si userà `\cB.` per indicare in un'espressione di ricerca una graffa aperta o `\cC.` per una qualsiasi sequenza di controllo. Nelle espressioni di sostituzione invece faremo seguire al prefisso il carattere che effettivamente vogliamo. Si capisce dunque perché nell'esempio abbiamo

```
\c{textbf}\cB\{\1\cE\}
```

che significa: 'il comando `\textbf`, una graffa aperta, ciò che hai trovato, una graffa chiusa'. Perciò la 'A' di 'Aiuola' verrà sostituita da `\textbf{A}` e così per gli altri caratteri che corrispondono all'espressione di ricerca, cioè le vocali. Se avessimo adoperato semplicemente `\\textbf\{\1\}` il risultato sarebbe stato completamente diverso, perché avremmo sostituito semplici stringhe di caratteri (per la precisione di categoria 12) e non i token che ci servono. Ecco la prova:

```
% \usepackage{xparse,l3regex}
\ExplSyntaxOn
\NewDocumentCommand{\oops}{m}
 {
  \manual_oops:n { #1 }
 }
\tl_new:N \l_manual_oops_tl
\cs_new_protected:Nn \manual_oops:n
 {
  \tl_set:Nn \l_manual_oops_tl { #1 }
  \regex_replace_all:nnN
   { ([AEIOUaeiou]) }
   { \\textbf\{\1\} }
   \l_manual_oops_tl
  \tl_use:N \l_manual_oops_tl
 }
\ExplSyntaxOff

\oops{abc}
```

```
\textbf{a}bc
```

L'esistenza di `\regex_replace_all:nnN` suggerisce quella di `\regex_replace_once:nnN` che infatti esiste e ha la stessa sintassi, ma esegue solo la sostituzione della prima occorrenza dell'espressione di ricerca. Entrambe non fanno nulla se la ricerca non ha successo.

Ci sono parecchie altre funzioni, qualcuna la vedremo. Per ora va menzionata `\regex_match:nnTF` che ha la sintassi

```
\regex_match:nnTF
  ⟨regex⟩
  ⟨token list⟩
  ⟨true⟩
  ⟨false⟩
```

e che, a differenza delle due descritte, prende come primo argomento un'espressione regolare e come secondo un qualsiasi testo. Se l'espressione regolare ha una corrispondenza nel testo, sarà eseguito il codice ⟨*true*⟩, altrimenti il codice ⟨*false*⟩. Come al solito in expl3, sono definite anche le varianti `\regex_match:nnT` e `\regex_match:nnF` per evitare di dover specificare un argomento vuoto. Per esempio, se volessimo emettere un avviso nel caso una variabile *token list* non ha corrispondenze con l'espressione regolare, potremmo dire

```
% \usepackage{xparse,l3regex}
% \ExplSyntaxOn
\cs_generate_variant:Nn
  \regex_match:nnF { nV }
\regex_match:nVF
 { [AEIOUaeiou] }
 \l_manual_test_tl
 { \msg_warning:nn { test } { novowels } }
%\ExplSyntaxOff
```

avendo definito l'opportuno messaggio di avviso.

Ecco un'applicazione più seria. Vogliamo un'espressione regolare per riconoscere se una sequenza di caratteri soddisfa la definizione di numero decimale secondo TEX:[1]

```
\regex_const:Nn \c_manual_number_regex
 {\A \s* (\+|\-)* \cO\d* \.? \cO\d* \s? \Z}
```

dove, con `\A` e `\Z` diciamo che vogliamo esaminare tutta la sequenza di caratteri che può cominciare con qualsiasi numero di spazi e terminare con uno spazio; inoltre può avere un numero qualsiasi di caratteri `+` e `-`, seguiti da zero o più cifre (di categoria 12), poi al più da un punto decimale e quindi altre cifre. In realtà la sintassi di TEX dà la possibilità di usare la virgola, ma è bene evitarlo, perché la virgola è usatissima come separatore di liste di token nelle opzioni. Se si vuole ammettere anche la virgola, basta sostituire `\.` con `(\.|\,)`, come per i segni iniziali facoltativi. Come si vede, è un'espressione regolare del tutto standard (a parte gli spazi che però danno maggiore leggibilità).

Ecco vista una nuova proprietà di l3regex: la possibilità di definire variabili e costanti di tipo `regex`. Le funzioni del pacchetto hanno tutte la variante in cui il primo argomento è di tipo `N` e richiede una variabile o costante di tipo `regex`. Così, per vedere se un argomento della nostra macro è un numero decimale, possiamo usare

```
\regex_match:NnTF \c_manual_unit_regex
 { #1 }
 { a~number }
 { not~a~number }
```

Sono analogamente disponibili

- `\regex_replace_once:NnN` e
- `\regex_replace_all:NnN`.

L'argomento che contiene l'espressione di sostituzione non può mai essere una variabile o costante di tipo `regex`; il motivo è che, al momento, non sembra utile. Se casi d'uso dovessero emergere, non sarà difficile aggiungere le varianti. La funzione per definire una costante l'abbiamo vista; per definire e impostare variabili si adoperano

```
\regex_new:N \l_manual_vara_regex
\regex_set:Nn \l_manual_vara_regex
 { ⟨regex⟩ }
\regex_clear:N \l_manual_vara_regex
\regex_new:N \g_manual_varb_regex
\regex_gset:Nn \g_manual_varb_regex
 { ⟨regex⟩ }
\regex_gclear:N \g_manual_varb_regex
```

dove la prima variabile è da usare localmente, la seconda globalmente. Lo scopo è di avere a disposizione espressioni regolari di ricerca di uso frequente; il risparmio è nel fatto che l'espressione è già predigerita. Un esempio è nella tabella 1, in cui si mostra il risultato al terminale del comando `\regex_show:N \c_manual_number_regex`.

4 Altri esempi

Gli esempi che presenteremo sono forse banali, ma dovrebbero dare l'idea delle potenzialità del pacchetto.

Primo problema: dividere una stringa in parti uguali, cominciando da destra; per esempio, la stringa `1234567` divisa a due a due deve diventare `1,23,45,67`. Ecco il codice, con solo la parte 'interna': l'interfaccia utente è facile da scrivere.[2]

```
% \usepackage{l3regex}
\ExplSyntaxOn
\tl_new:N \l_manual_string_tl
\cs_new_protected:Nn \manual_split:nn
 {
  \tl_set:Nn \l_manual_string_tl { #2 }
  % we need to start from the end,
  % so we reverse the string
  \tl_reverse:N \l_manual_string_tl
  % add a comma after any group of #1
     tokens
  \regex_replace_all:nnN
   { (.{#1}) }
   { \1\, }
   \l_manual_string_tl
  % if the length of the string is a
  % multiple of #1 a trailing comma is
  % added, so we remove it
  \regex_replace_once:nnN
   { \,\Z }
   { }
   \l_manual_string_tl
  % reverse back
  \tl_reverse:N \l_manual_string_tl
 }
```

1. Si veda `http://tex.stackexchange.com/questions/138737/`.

2. Si veda `http://tex.stackexchange.com/questions/171007/`.

Tabella 1: Esempio di sviluppo di un'espressione regolare

```
% \regex_const:Nn \c_manual_number_regex
%  { \A \s* (\+|\-)* \c0\d* \.? \c0\d* \s? \Z }
% \regex_show:N \c_manual_number_regex

> Compiled regex variable \c_manual_number_regex:
+-branch
  assertion: anchor at start (\A)
  Match, repeated 0 or more times, greedy
    char code 32
    char code 9
    char code 10
    char code 12
    char code 13
  ,-group begin
  | char code 43
  +-branch
  | char code 45
  `-group end, repeated 0 or more times, greedy
  Match, repeated 0 or more times, greedy
    categories 0, class
      range [48,57]
  char code 46, repeated between 0 and 1 times, greedy
  Match, repeated 0 or more times, greedy
    categories 0, class
      range [48,57]
  Match, repeated between 0 and 1 times, greedy
    char code 32
    char code 9
    char code 10
    char code 12
    char code 13
  assertion: anchor at end (\Z).
```

```
\manual_split:nn { 2 } { 1234567 }
\tl_use:N \l_manual_string_tl\par
\manual_split:nn { 3 } { aaabbbccc }
\tl_use:N \l_manual_string_tl\par

\ExplSyntaxOff
```

1,23,45,67
aaa,bbb,ccc

Per ovviare al problema dell'inizio, la stringa viene rovesciata e dopo ogni gruppo di n caratteri (dove n rappresenta il primo argomento) viene aggiunta la virgola con l'espressione `(.{`⟨n⟩`})` che cattura il *match* e può adoperarlo nell'espressione di sostituzione. Il caso in cui la lunghezza della stringa è un multiplo di n produce una virgola in più che rimuoviamo ancorando la seconda espressione regolare alla fine del testo con `\Z`. Per finire, rovesciamo di nuovo la stringa.

Naturalmente questo si potrebbe ottenere anche con una macro ricorsiva, tuttavia con l3regex è possibile controllare in anticipo se la stringa contiene solo caratteri permessi, per esempio cifre.

Un'altra applicazione che evita complicate definizioni con caratteri attivi è quella di caricare parti di documenti scritte secondo le convenzioni di Markdown. Per semplicità, supporremo che siano usate solo `*parola*` per il corsivo e `**parola**` per il neretto. L'idea è di adoperare catchfile per assegnare a una variabile *token list* l'intero contenuto del file e su questa variabile operare con `\regex_replace_all:nnN`. In realtà `\CatchFileDef` non è necessaria, perché la funzione è già stata incorporata in expl3 come `\tl_set_from_file:Nnn`.

Questo è il file di esempio

```
A **test** with longer **bold face
also split** across lines.

Now also *italic*.
```

e questo è il codice completo

```
% \usepackage{xparse,l3regex}

\ExplSyntaxOn
\NewDocumentCommand{\mdinput}{m}
 {
  \manual_mdinput:n { #1 }
 }

\tl_new:N \l__manual_mdfile_tl

\cs_new_protected:Nn \manual_mdinput:n
 {
  \tl_set_from_file:Nnn
     \l__manual_mdfile_tl
```

```
    { } { #1 }
   \regex_replace_all:nnN
    { \*\*([^\*]*?)\*\* }
    { \c{textbf}\cB\{ \1 \cE\} }
    \l__manual_mdfile_tl
   \regex_replace_all:nnN
    { \*([^\*]*?)\* }
    { \c{emph}\cB\{ \1 \cE\} }
    \l__manual_mdfile_tl
   \tl_use:N \l__manual_mdfile_tl
  }

\ExplSyntaxOff

\mdinput{test.md}
```

A **test** with longer **bold face also split** across lines.
Now also *italic*.

Si potrebbe migliorare imponendo nell'espressione di ricerca controlli che vietino `\par` nel testo tra asterischi. Ovviamente il codice è inefficiente se i testi da importare sono molto lunghi.

5 Emulare la sostituzione di argomenti

Supponiamo di voler simulare un *here file*.[3] L'idea è di avere una struttura del tipo

```
\tofile{⟨file⟩}{⟨argomenti⟩}<<EOF
⟨codice⟩
<<EOF
```

dove ⟨*codice*⟩ contiene `#1`, `#2` e così via, da sostituire con gli argomenti specificati e ⟨*file*⟩ è il nome del file su cui scrivere; per esempio, vorremmo

```
\tofile{test.c}{{One}{Two}{Three}}<<EOF
/* #1, #2 */
#include <stdio.h>
main() {
  printf("#3",\n);
}
<<EOF
```

dove si noti il carattere `#` da scrivere così com'è. La stringa finale è arbitraria e va ripetuta alla fine per indicare dove si deve smettere.

L'idea è abbastanza semplice: gli argomenti sono memorizzati in una variabile *sequence*; il codice viene letto una riga alla volta (sorvolo sulla complicazione tecnica) e sulla riga vengono operate le necessarie sostituzioni con

```
\regex_replace_all:nnN
  { \# i }
  { ⟨i-esimo argomento⟩ }
  \l_manual_line_tl
```

Lento, ma efficace. Naturalmente, prima di leggere le righe, i caratteri speciali vengono tutti neutralizzati.

Nella tabella 2 si può leggere il codice che è interessante anche per altri usi di funzioni di expl3; di seguito è riportato il risultato.

6 Un pacchetto basato su l3regex

La possibilità di adoperare espressioni regolari ha subito suscitato l'idea di un nuovo pacchetto per *rappezzare* comandi e risolvere certe difficoltà che sorgono con etoolbox. Già xpatch è basato su expl3 per la procedura di decisione sul tipo di macro da rappezzare; infatti ce ne sono ben sette, perché occorre distinguere fra quelle definite con `\newcommand`, con `\DeclareRobustCommand` e `\newrobustcmd`; per le prime due è diverso il caso di parole di controllo (come `\foo`) o simboli di controllo (come `\?`). Il nuovo pacchetto si chiama regexpatch.

La verifica si esegue sull'espansione di primo livello della macro e con espressioni regolari è più facile controllarne la *firma*. Per esempio, la firma della macro `\foo` definita con `\newcommand` e un argomento facoltativo è

```
\A\\@protected@testopt\ \\foo\ \\\\
```

e il nome della macro può essere memorizzato in una variabile *token list*, da specificare con il prefisso `\u`. Sembra complicato, ma è solo questione di pazienza, perché le firme sono davvero univoche, una volta che l'espansione di primo livello della macro è trasformata in semplice stringa con `\token_get_replacement_spec:N`.

Ma questa semplificazione non è tutto. Si può usare l3regex anche per il rappezzo! E quindi fornire anche una versione che modifica tutte le occorrenze di una certa lista di token con altre. L'esempio è quello del pacchetto semantic che non è compatibile con LuaLaTeX; la modifica richiede di sostituire in `\mathligsoff` (che è un comando robusto) e in `\@addligto` ogni occorrenza di `\mathcode` con `\Umathcodenum` che compare una volta nel primo e tre nel secondo. Si può fare con etoolbox con

```
\makeatletter
\expandafter\patchcmd
  \csname mathligsoff \endcsname
  {\mathcode}{\Umathcodenum}
  {}{}
\patchcmd\@addligto
  {\mathcode}{\Umathcodenum}{}{}
\patchcmd\@addligto
  {\mathcode}{\Umathcodenum}{}{}
\patchcmd\@addligto
  {\mathcode}{\Umathcodenum}{}{}
\makeatother
```

e xpatch renderebbe solo più facile il primo rappezzo. Con regexpatch il codice diventa

3. Si veda http://tex.stackexchange.com/questions/155358.

Tabella 2: Codice per un *here file*.

```
\documentclass{article}
\usepackage{xparse,l3regex}

\ExplSyntaxOn
\NewDocumentCommand{\tofile}{m}
 {
  \manual_tofile:n { #1 }
 }

\iow_new:N \g_manual_write_stream
\tl_const:Nn \c_manual_specials_tl
 { \  \\ \{ \} \$ \& \# \^ \_ \% \~ }
\tl_new:N \l_manual_argument_tl
\tl_new:N \l_manual_line_tl
\seq_new:N \l_manual_arguments_seq
\int_new:N \l_manual_arguments_int

\cs_new_protected:Npn \manual_dospecials:
 {
  \tl_map_inline:Nn \c_manual_specials_tl
   {
    \char_set_catcode_other:N ##1
   }
 }

\cs_new_protected:Npn \manual_tofile:n #1
 {
  \group_begin:
  \tex_endlinechar:D `\^^J
  \iow_open:Nn \g_manual_write_stream { #1 }
  \manual_tofile_aux:nw
 }

\cs_new_protected:Npn \manual_tofile_aux:nw #1
     #2 ~%
 {% #1 is the list of arguments,
  % #2 is the terminator
  \tl_set:Nn \l_manual_eof_tl { #2 }
  \tl_trim_spaces:N \l_manual_eof_tl
  \seq_set_split:Nnn \l_manual_arguments_seq
   { } { #1 }
  \int_set:Nn \l_manual_arguments_int
   { \seq_count:N \l_manual_arguments_seq }
  \manual_dospecials:
  \manual_readline:w
 }

\cs_new_protected:Npn \manual_readline:w #1 ^^J
 {
  \str_if_eq:VnTF \l_manual_eof_tl { #1 }
   {
    \iow_close:N \g_manual_write_stream
    \group_end:
   }
   {
    \manual_replace_write:n { #1 }
    \manual_readline:w
   }
 }
```

```
\cs_new_protected:Npn \manual_replace_write:n #1
 {
  \tl_set:Nn \l_manual_line_tl { #1 }
  \int_step_inline:nnnn
   { 1 }
   { 1 }
   { \l_manual_arguments_int }
   {
    \tl_set:Nx \l_manual_argument_tl
     {
      \seq_item:Nn \l_manual_arguments_seq
       { ##1 }
     }
    \regex_replace_all:nnN
     { \# ##1 }
     { \u{l_manual_argument_tl} }
     \l_manual_line_tl
   }
  \iow_now:NV \g_manual_write_stream
   \l_manual_line_tl
 }

\cs_generate_variant:Nn \iow_now:Nn { NV }
\cs_generate_variant:Nn \str_if_eq:nnTF { V }

\ExplSyntaxOff

\begin{document}
\tofile{test.c}{{One}{Two}{Three}}<<EOF
/* #1, #2 */
#include <stdio.h>
main() {
  printf("#3",\n);
}
<<EOF

\end{document}
```

test.c

```
/* One, Two */
#include <stdio.h>
main() {
  printf("Three",\n);
}
```

```
\makeatletter
\xpatchcmd\mathligsoff
  {\mathcode}{\Umathcodenum}{}{}
\xpatchcmd*{\@addligto}
  {\mathcode}{\Umathcodenum}{}{}
\makeatother
```

che è certamente più comodo. La versione con l'asterisco è quella che 'cambia tutto'.

I comandi `\xpatchcmd`, `\xpretocmd` e `\xapptocmd` sono volutamente gli stessi usati da xpatch, perché l'idea è di sostituire i rappezzi basati su etoolbox con quelli basati su l3regex, ma il pacchetto è ancora in versione sperimentale dal momento che lo è ancora l3regex. C'è anche un comando nuovo, `\regexpatchcmd` che negli argomenti di ricerca e sostituzione vuole appunto espressioni regolari.

Per esempio, supponiamo di voler eliminare dal comando `\chaptermark` della classe book il malefico `\MakeUppercase`. La definizione è

```
\def\chaptermark#1{%
  \markboth{\MakeUppercase{%
    \ifnum\c@secnumdepth>\m@ne
      \if@mainmatter
        \@chapapp\ \thechapter. \ %
      \fi
  \fi
  #1}}{}}
```

e l'ovvio rappezzo sarebbe

```
\makeatletter
\xpatchcmd{\chaptermark}
  {\MakeUppercase}
  {\@firstofone}
  {}{}
\makeatother
```

perché così eliminiamo anche la coppia di graffe, a spese di un'espansione in più. Volendo eliminare anche le graffe, il rappezzo sarebbe più complicato che riscriversi la macro. Con `\regexpatchcmd` si può fare meglio:

```
\makeatletter
\regexpatchcmd{\chaptermark}
  {\c{MakeUppercase}\cB.(.*?)\cE.}
  {\1}
  {}{}
\makeatother
```

e vale la pena di capire che cosa succede. L'espressione di ricerca si combina con `\MakeUppercase` seguito da una graffa aperta, da qualsiasi token (con ricerca pigra) e una graffa chiusa. I token tra le graffe sono ricordati e infatti l'espressione di sostituzione dice proprio di lasciare solo ciò che c'è tra le graffe.

Un esempio fittizio di qualcosa che non è possibile eseguire con etoolbox. Supponiamo di avere una macro `\abc` il cui comportamento deve cambiare leggermente nell'appendice, nella quale si deve modificare l'uso di uno degli argomenti. È possibile, con `\patchcmd`, avere `#1` in uno degli argomenti di ricerca o sostituzione, ma non se `\patchcmd` appare nell'argomento di un altro comando; questo è un problema se si deve eseguire un rappezzo in `\AtBeginDocument`. Si potrebbe semplicemente eseguire il rappezzo dopo `\appendix`, ma una buona programmazione vuole il codice tutto nel preambolo. Ecco come:

```
\newcommand{\abc}[2]{#1\ #2}
\newcommand{\patchabc}{%
  \regexpatchcmd{\abc}
    {\cP.1}{\cP\#2}
    {}{}%
}
\xapptocmd{\appendix}{\patchabc}{}{}
```

In altre parole, definiamo `\abc` e anche `\patchabc` che ha come unico scopo quello di rappezzare `\abc`. Con `\xapptocmd` diciamo quando eseguire il rappezzo; si potrebbe evitare il comando ausiliario, ma in questo modo sembra più pulito. Il rappezzo può andare nell'argomento di `\newcommand` perché non si usa mai un `#` esplicito, ma solo la sua rappresentazione in un'espressione regolare.

Un altro esempio è un rappezzo all'ambiente `rubric` di CurVe[4] per togliere la riga di 'continuazione'; l'ambiente è basato su `longtable` e occorreva togliere tutto quanto va da `\endfirsthead` (escluso) a `\endhead` (incluso). Con `\regexpatchcmd` è semplicissimo:

```
\regexpatchcmd{\rubric}
  {\c{endfirsthead}.*\c{endhead}}
  {\c{endfirsthead}}
  {}{}
```

dove non occorre la ricerca pigra perché `\endhead` compare solo una volta. È chiaro che avere a disposizione funzioni molto potenti non esime dallo studio della macro da rappezzare.

Il pacchetto è in grado di eseguire facilmente altri compiti che sarebbero molto complicati con altri metodi e, in sostanza, sarebbero risolubili solo riscrivendosi le macro. La funzione `\xpatchparametertext` è in grado di modificare il *parameter text* di una macro. Un caso d'uso è legato a babel per il ceco e lo slovacco.[5] Con una di queste lingue, il carattere `-` è un prefisso (come `"` per l'italiano o il tedesco) e questo ha come conseguenza che le macro `\cline` e `\cmidrule` non funzionano, perché quando si dà `\cline{1-2}`, la macro separa i due numeri cercando un carattere `-` di categoria 12, mentre ne trova uno di categoria 13. La definizione è alla riga 5185 di `latex.ltx`

4. http://www.guitex.org/home/en/forum/5-tex-e-latex/68612.
5. http://tex.stackexchange.com/questions/111999.

```
\def\cline#1{\@cline#1\@nil}
\def\@cline#1-#2\@nil{%
  \omit
  ...
}
```

e ce n'è una simile in booktabs per `\cmidrule`. Occorre sostituire il carattere di categoria 12 con lo stesso di categoria 13; il secondo rappezzo ovviamente richiede booktabs:

```
\makeatletter
\xpatchparametertext\@cline
  {-}{\cA-}
  {}{}
\xpatchparametertext\@@@cmidrule
  {-}{\cA-}
  {}{}
\makeatother
```

Ovviamente dovrebbe essere babel a occuparsi di questo; visto che non lo fa, possiamo facilmente correre ai ripari.

Riferimenti bibliografici

THE LATEX TEAM (2014a). «expl3». CTAN.

— (2014b). «l3regex». CTAN.

VAN DER LAAN, K. (1993). «Syntactic sugar». *TUGboat*, **14** (3), pp. 310–318.

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