LaTeX3: un nuovo gioco per i maghi e per diventarlo

Enrico Gregorio

Sommario

Esaminiamo con esempi alcune delle funzioni messe a disposizione dal nucleo di LaTeX3. In particolare ci concentreremo sull'uso delle variabili di tipo *sequence* e faremo un cenno sull'uso del modulo l3keys.

Sommario

We shall examine, by means of examples, some of the functions provided by the LaTeX3 kernel. In particular, we shall focus on the usage of the *sequence* variable type and show something about the l3keys module.

1 Introduzione

Lo sviluppo di LaTeX3 sta procedendo piuttosto bene, sebbene sia ancora limitato allo strato di programmazione. Tuttavia la lista dei pacchetti che già lo usano è lunga: fontspec, siunitx fra i più noti.

Qual è lo scopo del *LaTeX3 team*? Lo scopo finale è, naturalmente, sviluppare la versione 3 di LaTeX. Ma l'indirizzo attualmente preminente è di fornire un ambiente di programmazione unitario e coerente. La versione sperimentale, su cui sono basati i pacchetti citati, è disponibile anche all'utente comune con

```
\usepackage{expl3}
```

e l'ambiente speciale di programmazione si attiva e disattiva con

```
\ExplSyntaxOn
\ExplSyntaxOff
```

che possiamo immaginare analoghi della coppia formata da `\makeatletter` e `\makeatother`. Si veda Gregorio (2012) per una rapida introduzione al nuovo linguaggio.

Lo scopo di questo articolo non è di descrivere il linguaggio, ma di farne vedere qualche potenzialità e beneficio.

In tutti gli esempi supporremo di essere nell'ambiente di programmazione dove, è importante saperlo, gli spazi e i fine riga sono ignorati. Questo evita quasi tutti gli inghippi del vecchio tipo di programmazione in cui gli spazi spuri erano e sono un incubo dell'autore di pacchetti, ma anche dell'utente finale.

2 Definire nuovi comandi a livello utente

Il pacchetto che provvisoriamente fornisce l'infrastruttura per definire comandi a livello utente è xparse. Facciamo un esempio che ci guiderà per un po' alla scoperta delle nuove funzionalità.

Nello studio dell'algebra dei gruppi le permutazioni sono uno degli strumenti fondamentali, del resto è studiandole che la teoria dei gruppi si è sviluppata. Un metodo per denotare le permutazioni è molto interessante da un punto di vista teorico, perché il *nome* della funzione contiene anche la regola per calcolarne i valori. Questo si ottiene con la cosiddetta *decomposizione in cicli disgiunti*. Per esempio, la permutazione

$$1 \mapsto 5, \quad 2 \mapsto 3, \quad 3 \mapsto 2, \quad 4 \mapsto 1, \quad 5 \mapsto 4$$

si può denotare con

$$(1\,5\,4)(2\,3)$$

e, per sapere chi associare a un elemento basta vedere quale sta alla sua destra (se c'è una parentesi si torna al primo elemento della lista).

Ovviamente, scrivere un ciclo come $(1\,2\,3\,4\,5\,6\,7\,8)$ con il codice

```
(1\,2\,3\,4\,5\,6\,7\,8)
```

è noioso, ma non solo: potremmo voler cambiare i delimitatori o ciò che va fra un elemento e l'altro. Ecco un modo:

```
\NewDocumentCommand{\cycle}
 { >{\SplitList{,}}m }
 {(\ProcessList{#1}{\cycleargument})}

\newcommand{\maybespace}{%
  \renewcommand{\maybespace}{\,}}
\newcommand{\cycleargument}[1]{%
  \maybespace#1}
```

Il comando `\cycle` ha un argomento che verrà automaticamente suddiviso nelle parti delimitate da virgole; ciascuna parte verrà passata al secondo argomento di `\ProcessList`. Quindi l'effetto è, alla fine, di eseguire

```
\cycleargument{1}
\cycleargument{2}
\cycleargument{3}
\cycleargument{4}
\cycleargument{5}
```

e il trucco ben noto di ridefinire `\maybespace` inserirà `\,` solo fra gli elementi dopo il primo. Ecco il risultato:

`\cycle{1,2,3,4,5}` → (1 2 3 4 5)

Si può fare meglio? Certo che si può, ma occorrono strumenti un po' più raffinati di `\ProcessList`.

Il nucleo di LaTeX3 fornisce decine di funzioni che astraggono compiti ripetitivi, di cui `\SplitList` e `\ProcessList` sono solo la superficie. Al di sotto sta una nuova *struttura di dati*, cioè le variabili di tipo *sequence*.[1]

Per poterle usare occorre però entrare nell'ambiente di programmazione di LaTeX3 che, al momento, richiede di racchiudere il codice fra

```
\ExplSyntaxOn
...
\ExplSyntaxOff
```

dove un nuovo mondo si apre. Vediamo l'esempio.

```
\NewDocumentCommand{\cycle}{m}
 {
  \perms_print_cycle:n { #1 }
 }
\seq_new:N \l_perms_cycle_seq
\cs_new_protected:Npn
 \perms_print_cycle:n #1
 {
  \seq_set_split:Nnn
   \l_perms_cycle_seq { , } { #1 }
  (
  \seq_use:Nnnn \l_perms_cycle_seq
   { \, } { \, } { \, }
  )
 }
```

La funzione `\seq_use:Nnnn` dice che fare della *sequence* data come primo argomento: i tre argomenti successivi dicono che cosa mettere fra i vari oggetti che compongono la *sequence*; rispettivamente se ce ne sono solo due, fra i primi $n-1$ se ce ne sono $n > 2$ e fra gli ultimi due in questo caso. La funzione nasce per risolvere il noto problema "apples and pears" o "apples, pears, and oranges" come vuole un certo stile di scrittura diffuso principalmente negli USA.

Vediamo passo passo. La prima dichiarazione traduce `\cycle` in un comando interno, come raccomandano le linee guida se l'esecuzione non può essere ridotta a comandi "esterni" come nel caso precedente. Meglio parlare di *funzioni* che di comandi interni: sono *funzioni* tutti quei comandi il cui nome contiene i due punti, dopo i quali vanno indicati gli argomenti della funzione, in questo caso uno. Prima di definire la nostra funzione dichiariamo la variabile *sequence* che useremo: le linee guida specificano che ogni variabile deve essere dichiarata. Le funzioni di tipo `set` ne stabiliscono un valore.

La funzione `\perms_print_cycle:n` è definita come `protected` perché a sua volta esegue funzioni dello stesso tipo (lo sono tutte quelle di tipo `set`). Ha un argomento di tipo normale, cioè espresso tra graffe. Il suo primo compito è di assegnare il valore alla variabile spezzando l'argomento alle virgole; il secondo è di stampare via via gli elementi della *sequence* separati da `\,`, mettendo in testa e in coda le parentesi.

Vediamo il risultato con questo esempio

`\cycle{1,2,3,4,5}` → (1 2 3 4 5)

che funziona, ma ci lascia ancora insoddisfatti. Vorremmo poter cambiare il delimitatore fra gli elementi del ciclo: alcuni preferiscono la virgola, per esempio. Così sperimentiamo il codice seguente

```
\NewDocumentCommand{\cycle}{O{\,} m}
 {
  \perms_print_cycle:nn { #1 } { #2 }
 }
\seq_new:N \l_perms_cycle_seq
\cs_new_protected:Npn
 \perms_print_cycle:nn #1 #2
 {
  \seq_set_split:Nnn \l_perms_cycle_seq
   { , } { #2 }
  (
  \seq_use:Nnnn \l_perms_cycle_seq
   { #1 } { #1 } { #1 }
  )
 }
```

dove specifichiamo che `\cycle` ha un primo argomento facoltativo, il cui valore normale è `\,`; così avremo

`\cycle{1,2,3,4,5}` → (1 2 3 4 5)
`\cycle[,]{1,2,3,4,5}` → $(1, 2, 3, 4, 5)$

In realtà la definizione ufficiale che uso è nel codice 1. La funzione `\egreg_seq_use:Nn` evita di dover scrivere tre volte lo stesso argomento; ha un prefisso diverso perché la uso in varie situazioni (in un'applicazione reale farebbe parte di un pacchetto). Ma perché definire la funzione interna con quattro argomenti invece che con due? Perché possiamo fare molto di più:

```
\NewDocumentCommand{\pcycle}{O{\,} m}
 {
  \perms_print_cycle:nnnn
   { ( } { ) } { #1 } { #2 }
 }
\NewDocumentCommand{\bcycle}{O{\,} m}
 {
  \perms_print_cycle:nnnn
   { [ } { ] } { #1 } { #2 }
 }
```

1. Preferisco non tradurre i nomi dei tipi di variabili.

```
\NewDocumentCommand{\cycle}{O{\,} m}
 {
  \perms_print_cycle:nnnn { ( } { ) } { #1 } { #2 }
 }

\cs_new:Npn \perms_seq_use:Nn #1 #2
 {
  \seq_use:Nnnn #1 { #2 } { #2 } { #2 }
 }
\seq_new:N \l_perms_cycle_seq
\cs_new_protected:Npn \perms_print_cycle:nnnn #1 #2 #3 #4
 {
  \seq_set_split:Nnn \l_perms_cycle_seq { , } { #4 }
  #1
  \egreg_seq_use:Nn \l_perms_cycle_seq { #3 }
  #2
 }
```

Codice 1: Versione ufficiale per `\cycle`

```
\NewDocumentCommand{\vcycle}{O{\,} m}
 {
  \perms_print_cycle:nnnn
   { \lvert } { \rvert } { #1 } { #2 }
 }
```

Il tutto senza dover definire nuove funzioni interne. Vedremo più avanti che si può fare ancora meglio.

3 Cicli dentro cicli

Una notazione comune per le matrici in certi linguaggi come MATLAB è

```
{{1,2,3},{4,5,6}}
```

per denotare la matrice

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

senza dover introdurre le due dimensioni. Ci sembra un ottimo candidato per sfruttare le variabili *sequence*.

Qui c'è una piccola complicazione: siccome dobbiamo costruire una tabella con `&` e `\\`, dovremo operare in due passi. Il primo è di memorizzare tutto il materiale in una variabile di tipo *token list* e poi liberarne il contenuto nell'ambiente `array`.

Una delle funzioni importanti al proposito è `\seq_map_inline:Nn` che agisce sulla *sequence* data come primo argomento, ripetendo il codice dato come secondo argomento per ogni elemento della *sequence*. Tanto per fare un esempio

```
\seq_clear:N \l_tmpa_seq
\seq_put_right:Nn \l_tmpa_seq {a}
\seq_put_right:Nn \l_tmpa_seq {b}
\seq_put_right:Nn \l_tmpa_seq {x}
```

prima azzera la *sequence* data e poi aggiunge i tre elementi. Si noti che dobbiamo prima 'svuotare' la variabile `\l_tmpa_seq` perché le operazioni di tipo `put` aggiungono a ciò che c'è già. Ora, con

```
\fbox{
 \seq_map_inline:Nn \l_tmpa_seq { (#1) }
}
```

otterremmo $\boxed{(a)(b)(x)}$.

Il nostro scopo è di definire un comando `\matlabmatrix` che prenda la notazione MATLAB e la trasformi in una matrice vera. Vediamo che cosa ci potrebbe servire: un argomento facoltativo che ci indichi quali delimitatori vogliamo e la matrice stessa.

```
\NewDocumentCommand{\matlabmatrix}
 { O{b} m }
 {
  \matlabmatr_print:nn { #1 } { #2 }
 }
```

Finora è facile, no? Ogni comando a livello utente dovrebbe avere una sua versione interna senza argomenti facoltativi, ma solo normali. La `b` sarà usata per comporre l'ambiente `bmatrix`, potremo dare anche `p` o `v`.

Il primo passo è di memorizzare la lista esterna in una *sequence*:

```
\seq_set_split:Nnn
 \l_matlabmatr_rows_seq
 { , } { #2 }
```

e poi applicare un *mapping*

```
\seq_map_inline:Nn
 \l_matlabmatr_rows_seq
 {
  \matlabmatr_makerow:n { ##1 }
 }
```

e, per finire, comporre la matrice:

```
\begin{#1matrix}
\l_matlabmatr_matrix_tl
\end{#1matrix}
```

dove useremo la *token list* che conterrà la matrice nella forma con `&` e `\\`; dobbiamo ricordarci di svuotarla del suo valore prima di operare. Ci occuperemo più avanti della funzione che costruisce le righe; intanto ecco il codice per `\matlabmatr_print:nn` tutto insieme:

```
\seq_new:N \l_matlabmatr_rows_seq
\tl_new:N \l_matlabmatr_matrix_tl
\cs_new_protected:Npn
 \matlabmatr_print:nn #1 #2
 {
  \tl_clear:N \l_matlabmatr_matrix_tl
  \seq_set_split:Nnn
   \l_matlabmatr_rows_seq
   { , } { #2 }
  \seq_map_inline:Nn
   \l_matlabmatr_rows_seq
   {
    \matlabmatr_makerow:n { ##1 }
   }
  \begin{#1matrix}
  \l_matlabmatr_matrix_tl
  \end{#1matrix}
 }
```

Perché `##1`? Perché con `#1` nel codice di prima si denota l'elemento della *sequence* che viene passato alla funzione di *mapping* e qui siamo all'interno di una definizione. Abbiamo anche allocato una nuova variabile *token list* e la azzeriamo prima di cominciare il lavoro.

Ora la nostra funzione che costruisce le righe è già nota: abbiamo risolto lo stesso problema con i cicli. Qui c'è una differenza: non vogliamo stampare subito gli elementi, ma memorizzare la matrice nella *token list*.

```
\seq_new:N \l_matlabmatr_arow_seq
\cs_new_protected:Npn
 \matlabmatr_makerow:n #1
 {
  \seq_set_split:Nnn
   \l_matlabmatr_arow_seq
   {,} { #1 }
  \tl_put_right:Nx
   \l_matlabmatr_matrix_tl
   {
    \egreg_seq_use:Nn
     \l_matlabmatr_arow_seq
     { & }
   }
  \tl_put_right:Nn
   \l_matlabmatr_matrix_tl
   { \\ }
 }
```

La funzione `\tl_put_right:Nn` accoda qualcosa a quanto già memorizzato nella *token list*: inseriamo `&` fra i vari elementi e un `\\` alla fine. La usiamo anche nella forma `\tl_put_right:Nx` in modo che non venga accodato

```
\egreg_seq_use:Nn
 \l_matlabmatr_arow_seq { & }
```

ma il suo sviluppo usando il valore attuale della *sequence*. Se consultiamo il manuale di LaTeX3 (The LaTeX3 Team, 2012) scopriamo che accanto a `\seq_use:Nnnn` c'è una stellina: dice della funzione che è *espandibile*, cioè può essere usata dove l'argomento è denotato con `x` dando il risultato sperato. Le funzioni dichiarate come `protected` non sono *mai* espandibili e, infatti, `\egreg_seq_use:Nn` è stata definita con il semplice `\cs_new:Npn` per non perdere l'espandibilità nel modo abbastanza ovvio

```
\cs_new:Npn \egreg_seq_use:Nn #1 #2
 {
  \seq_use:Nnnn #1 { #2 } { #2 } { #2 }
 }
```

Il discorso sull'espandibilità è un po' troppo tecnico e ci porterebbe troppo lontano. Ciò che conta è che il comando funziona!

```
\matlabmatrix{{1,2,3},{4,5,6}}
```

↓

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$$

Proviamo con l'argomento facoltativo:

```
\matlabmatrix[v]{{1,2,3},{4,5,6}}
```

↓

$$\begin{vmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{vmatrix}$$

4 Un'applicazione frivola

Tanto per scoprire nuove funzioni, vediamo come trasformare una lista di numeri interi separati da virgole nella loro somma. Per rendere l'argomento un po' più interessante, scriveremo anche la somma in colonna: si veda il codice 2. Vediamo, come al solito, che funziona:

```
\colsum{11,2,43,84}
```

→

$$\begin{array}{r} 11+\\ 2+\\ 43+\\ 84=\\ \hline 140 \end{array}$$

Qui non c'è molto di nuovo, se non il fatto che possiamo usare una *sequence* quante volte vogliamo e in modi diversi; nel secondo impiego la diamo in pasto a `\int_eval:n` che si occupa di eseguire la somma.

E se volessimo scrivere una moltiplicazione in colonna? Si guardi il codice 3.

Qui dobbiamo solo ricordarci di rovesciare la *sequence*, perché l'operazione si esegue da destra

```
\NewDocumentCommand{\colsum}{m}
 {
  \colsum_do:n { #1 }
 }
\tl_new:N \l_colsum_list_tl
\seq_new:N \l_colsum_list_seq
\cs_new_protected:Npn \colsum_do:n #1
 {
  \tl_clear:N \l_colsum_list_tl
  \seq_set_split:Nnn \l_colsum_list_seq { , } { #1 }
  \tl_put_right:Nx \l_colsum_list_tl
   { \egreg_seq_use:Nn \l_colsum_list_seq { & + \\ } }
  \tl_put_right:Nn \l_colsum_list_tl { & = \\ \midrule}
  \tl_put_right:Nx \l_colsum_list_tl
   {
    \int_eval:n { \egreg_seq_use:Nn \l_colsum_list_seq { + } }
   }
  \begin{array}{r@{\;}l}
  \l_colsum_list_tl
  \end{array}
 }
```

Codice 2: Somma in colonna

```
\NewDocumentCommand{\colmult}{mm}
 {
  \colmult_do:nn { #1 } { #2 }
 }
\tl_new:N \l_colmult_spaces_tl
\tl_new:N \l_colmult_partial_summands_tl
\seq_new:N \l_colmult_multiplier_seq
\cs_new_protected:Npn \colmult_do:nn #1 #2
 {
  \tl_clear:N \l_colmult_partial_summands_tl
  \tl_clear:N \l_colmult_spaces_tl
  \seq_set_split:Nnn \l_colmult_multiplier_seq { } { #2 }
  \seq_reverse:N \l_colmult_multiplier_seq
  \seq_map_inline:Nn \l_colmult_multiplier_seq
   {
    \tl_put_right:Nx \l_colmult_partial_summands_tl
     {
      \int_eval:n { #1 * ##1 }
     }
    \tl_put_right:NV \l_colmult_partial_summands_tl \l_colmult_spaces_tl
    \tl_put_right:Nn \l_colmult_partial_summands_tl { \\ }
    \tl_put_right:Nn \l_colmult_spaces_tl { \hphantom{0} }
   }
  \begin{array}{r@{\;}l}
  #1 & \times \\
  #2 & = \\
  \midrule
  \l_colmult_partial_summands_tl
  \midrule
  \int_eval:n { #1 * #2 }
  \end{array}
 }
```

Codice 3: Moltiplicazione in colonna

a sinistra; la funzione si chiama, in modo piuttosto intuitivo, `\seq_reverse:N`. Vediamo anche che un argomento può essere spezzato "a niente": così il numero 294 diventa la *sequence* `{2}{9}{4}`. Vediamo l'esempio:

$$\begin{array}{rr} & 412\times \\ & 294= \\ \hline & 1648 \\ & 3708 \\ & 824 \\ \hline \texttt{\textbackslash colmult\{412\}\{294\}}\rightarrow & 121128 \end{array}$$

A ogni passo del *mapping* aggiungiamo alla *token list* ausiliaria un `\hphantom{0}` per mantenere il corretto allineamento.

Meglio nemmeno provare a scrivere una cosa del genere per LaTeX 2ε: ci si perderebbe la testa in poco tempo: dovremmo sistemare la ricorsività, invertire il moltiplicatore, scrivere le macro per i calcoli. Invece la macro `\colmult` è stata scritta in circa cinque minuti, compresa la correzione degli errori delle versioni iniziali.

5 Altre interfacce

Il modo con cui abbiamo scritto `\perms_print_cycle:nnnn` ci permette di definire nuove interfacce: per esempio

```
\xcycle[delims=(),
 separator=\,,]
 {1,2,3,4,5}
\xcycle[delims={[]},
 separator={;},]
 {1,2,3,4,5}
```

ma senza dover ridefinire il comando interno. Vediamo come:

```
\NewDocumentCommand{\xcycle}{O{} m}
 {
  \keys_set:nn {perms}
   {delims=(),separator=\,}
  \keys_set:nn {perms} { #1 }
  \perms_print_cycle:VVVn
   \l_perms_left
   \l_perms_right
   \l_perms_separator
   { #2 }
 }
\cs_generate_variant:Nn
 \perms_print_cycle:nnnn
 {VVV}
\tl_new:N \l_perms_left
\tl_new:N \l_perms_right
\tl_new:N \l_perms_separator
```

La funzione `\perms_print_cycle:VVVn` è definita in termini di `\perms_print_cycle:nnnn` dicendo che i primi tre argomenti devono essere presi come il valore delle variabili indicate.

Si tratta ora di associare i valori. Ci occupiamo prima di assegnare i valori normali e poi, eventualmente, di cambiarli se viene espresso l'argomento facoltativo.

```
\keys_define:nn { perms }
 {
  delims .code:n = \perms_set_delims:NN #1,
  separator .tl_set:N = \l_perms_separator,
 }
\cs_new_protected:Npn
 \perms_set_delims:NN #1 #2
 {
  \tl_set:N \l_perms_left #1
  \tl_set:N \l_perms_right #2
 }
```

Una dichiarazione di chiave, cioè data in `\keys_define:nn`, come

```
delims .code:n = \perms_set_delims:NN #1,
```

dice che il valore a destra dell'uguale dopo la chiave `delims`, rappresentato da `#1`, deve essere passato alla funzione `\perms_set_delims:NN`. Invece

```
separator .tl_set:N = \l_perms_separator,
```

è una forma abbreviata di

```
separator .code:n =
 \tl_set:N \l_perms_separator { #1 },
```

Vediamo le possibili combinazioni:

`\xcycle{1,2,3}`	(1 2 3)
`\xcycle[delims={[]}]{1,2,3}`	[1 2 3]
`\xcycle[separator=;]{1,2,3}`	(1; 2; 3)
`\xcycle[delims={[]},separator=;]{1,2,3}`	[1; 2; 3]

Per fare un esempio un po' più reale, si pensi al comando `\parbox` che accetta ben tre argomenti facoltativi:

`\parbox[`⟨*pos*⟩`][`⟨*height*⟩`][`⟨*innerpos*⟩`]{`⟨*text*⟩`}`

e ci si dimentica sempre quali siano. Una sintassi del tipo

```
\xparbox[pos=⟨pos⟩,
  height=⟨height⟩,
  innerpos=⟨innerpos⟩]{⟨text⟩}
```

oppure una sintassi per `\chapter` del tipo

```
\xchapter[numbered=⟨boolean⟩,
  toc=⟨text⟩,
  header=⟨text⟩]{⟨text⟩}
```

sarebbero certamente le benvenute. Non è in realtà troppo difficile scrivere `\xchapter` in termini del comando attuale, si veda il codice 4. Ovviamente definire `\xchapter` in termini di `\chapter` esegue parecchi compiti più volte; ma la potenza del nuovo nucleo è evidente.

```
\NewDocumentCommand{\xchapter}{ O{} m }
 {
  \keys_set:nn {chapter}
   {
    numbered=true,
    toc=#2,
    head=#2,
    #1
   }
  \class_xchapter:VVn \l_class_toctitle_tl \l_class_headtitle_tl { #2 }
 }
\bool_new:N \l_chapter_numbered_bool
\tl_new:N \l_chapter_toctitle_tl
\tl_new:N \l_chapter_headtitle_tl
\keys_define:nn {chapter}
 {
  numbered .bool_set:N = \l_chapter_numbered_bool,
  toc .tl_set:N = \l_chapter_toctitle_tl,
  head .tl_set:N = \l_chapter_headtitle_tl,
 }
\cs_new_protected:Npn \class_xchapter:nnn #1 #2 #3
 {
  \bool_if:NTF \l_chapter_numbered_bool
   {
    \chapter[#1]{#3}\markboth{\MakeUppercase{#2}}{}
   }
   {
    \chapter*{#3}
   }
 }
\cs_generate_variant:Nn \class_xchapter:nnn {VV}
```

Codice 4: Il comando `\xchapter`

Riferimenti bibliografici

Gregorio, E. (2012). «Scrivere un pacchetto per LaTeX3 – il pacchetto kantlipsum». *ArsTEXnica*, **13**.

The LaTeX3 Team (2012). «The LaTeX3 interfaces». Documentation manual, The LaTeX3 Project. URL `http://texdoc.net/texmf-dist/doc/latex/l3kernel/interface3.pdf`. Accessibile con `texdoc interface3`.

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