

usr_41.txt Per Vim version 7.1. Ultima modifica: 2007 Dic 29

VIM USER MANUAL - di Bram Moolenaar
Traduzione di questo capitolo: Antonio Colombo

Preparare uno script Vim

Il linguaggio script di Vim è usato per il file di inizializzazione vimrc, nei file di sintassi, e molto altro. Questo capitolo spiega gli elementi che possono venir usati in uno script di Vim. Ce ne sono molti, così questo capitolo è lungo.

41.1	Introduzione
41.2	Variabili
41.3	Espressioni
41.4	Condizioni
41.5	Esecuzione di una espressione
41.6	Utilizzo funzioni
41.7	Definizione funzioni
41.8	Liste e Dizionari
41.9	Eccezioni
41.10	Osservazioni varie
41.11	Scrivere un plugin
41.12	Scrivere un plugin per un tipo_file
41.13	Scrivere un plugin per un compilatore
41.14	Scrivere un plugin veloce a caricarsi
41.15	Scrivere librerie di script
41.16	Distribuzione di script Vim

Capitolo seguente:	usr_42.txt	Aggiungere nuovi menù
Capitolo precedente:	usr_40.txt	Definire nuovi comandi
Indice:	usr_toc.txt	

=====

41.1 Introduzione *vim-script-intro* *script*

Il vostro primo incontro con gli script di Vim è il file vimrc. Vim lo legge all'avvio e ne esegue i comandi. Potete impostare delle opzioni con il valore che preferite, ed usare al suo interno ogni comando che cominci per ":" (questi comandi sono talora designati come comandi Ex o comandi della linea di comando).

Anche i file di sintassi sono degli script Vim. E lo sono pure i file che impostano opzioni per uno specifico tipo di file. Una macro complessa può essere definita da un file script di Vim separato. Potete pensare ad altri usi voi stessi.

Partiamo da un semplice esempio: >

```
:let i = 1
:while i < 5
:  echo "il contatore è" i
:  let i += 1
:endwhile
```

<

Note:

I ":" non sono obbligatori in questo caso. Vanno necessariamente usati solo se immettete un comando direttamente. In un file di script Vim possono essere omessi. Noi li useremo comunque, per sottolineare che questi sono comandi coi ":" e per distinguerli dai comandi che si danno in Normal mode.

Note:

Potete provare gli esempi copiando (comando yank) le linee da questo testo ed eseguendole con :@"

L'output dell'esempio precedente è:

```
il contatore è 1 ~
il contatore è 2 ~
il contatore è 3 ~
il contatore è 4 ~
```

Nella prima linea, Il comando ":let" assegna un valore a una variabile. La

forma generale è: >

```
:let {variabile} = {espressione}
```

In questo caso il nome della variabile è "i" e l'espressione è un valore semplice, il numero 1.

Il comando ":while" inizia un ciclo. In forma generale: >

```
:while {condizione}
: {comandi}
:endwhile
```

I comandi fino ad ":endwhile" che chiude il ciclo sono eseguiti finché la condizione rimane verificata. La condizione qui usata è l'espressione "i < 5". Questa espressione è vera finché la variabile i assume valori inferiori a 5.

Note:

Se scriveste un ciclo che duri più del previsto, potete interromperlo premendo **CTRL-C** (**CTRL-Break** in ambiente MS-Windows).

Il comando ":echo" visualizza gli argomenti che gli vengono passati. In questo caso la stringa di caratteri "il contatore è" ed il valore della variabile i. Poiché i vale 1, visualizzerà:

```
il contatore è 1 ~
```

Poi c'è il comando ":let i += 1". E' equivalente ad usare ":let i = i+1". Questo aggiunge 1 alla variabile i ed assegna il nuovo valore alla variabile stessa.

L'esempio è stato dato per spiegare i comandi, ma se voleste davvero eseguire un simile ciclo, lo si può scrivere in maniera molto più concisa: >

```
:for i in range(1, 4)
: echo "il contatore è" i
:endfor
```

Il funzionamento di |:for| e |range()| verrà spiegato più oltre. Seguite i puntatori se siete impazienti.

TRE TIPI DI NUMERI

I numeri possono essere decimali, esadecimali od ottali. Un numero esadecimale inizia con "0x" o con "0X". Ad esempio "0x1f" è il numero decimale 31. Un numero ottale inizia con uno 0. "017" è il numero decimale 15. Attenzione: non mettete uno zero davanti ad un numero decimale, verrebbe interpretato come un numero ottale!

Il comando ":echo" stampa sempre numeri decimali. Ad es.: >

```
:echo 0x7f 036
< 127 30 ~
```

Un numero diventa negativo quando è preceduto da un segno "-". Questo vale anche per numeri esadecimali ed ottali. Un segno "-" è usato anche per indicare una sottrazione. Confrontate questo esempio con il precedente: >

```
:echo 0x7f -036
< 97 ~
```

Gli spazi bianchi in una espressione vengono ignorati. Comunque se ne raccomanda l'uso per separare gli elementi e rendere l'espressione di più facile lettura. Per esempio, per evitare nell'esempio la confusione con un numero negativo, mettete un spazio fra il segno "-" e il numero che lo segue: >

```
:echo 0x7f - 036
```

41.2 Variabili

Un nome di variabile è composto di lettere ASCII, cifre ed il carattere "_". Il primo carattere del nome non può essere un numero. Nomi validi di variabile sono:

```

contatore
_aap3
nome_molto_lungo_di_variabile_con_sottolineature
LunghezzaFunz
LUNGHEZZA

```

Nomi di variabile non validi sono "pippo+pluto" e "6var".

Queste variabili sono globali [valgono per tutti i buffer di una sessione Vim - NdT]. Per vedere una lista di tutte le variabili correntemente definite usate questo comando: >

```
:let
```

Potete usare variabili globali in qualsiasi posto. Questo implica anche che quando la variabile "contatore" è usata in un file di script, potrebbe anche essere usata in un altro file. Questo porta quanto meno a confusione, od al peggio crea problemi veri. Per evitarli, potete usare una variabile locale per un file di script, mettendogli come prefisso "s:". Ad esempio, uno script contiene questi comandi: >

```

:let s:contatore = 1
:while s:contatore < 5
:  source altro.vim
:  let s:contatore += 1
:endwhile

```

Poiché "s:contatore" vale solo per questo script, potete essere certi del fatto che lo script "altro.vim" non ne cambierà il valore. Se "altro.vim" usa una variabile "s:contatore", si tratterà di una copia diversa, propria di quello script. Per saperne di più sulle variabili locali ad uno script si veda: |[script-variable](#)|.

Ci sono ulteriori tipi di variabili, si veda |[internal-variables](#)|. Quelle usate più spesso sono:

b:name	variabile locale propria di un buffer
w:name	variabile locale propria di una finestra (window)
g:name	variabile globale (anche in una funzione)
v:name	variabile predefinita da Vim

ANNULLARE VARIABILI

Le variabili occupano memoria e vengono visualizzate nell'output del comando ":let". Per cancellare una variabile usate il comando ":unlet". Ad es.: >

```
:unlet s:contatore
```

Questo cancella la variabile di script locale "s:contatore" e libera la memoria che essa occupava. Se non siete sicuri dell'esistenza della variabile, e non volete vedere un messaggio di errore nel caso la variabile non esista, aggiungete al comando il "!" >

```
:unlet! s:contatore
```

Quando uno script Vim termina, le variabili locali che ha usato non saranno annullate automaticamente. La prossima volta che lo script verrà eseguito, potrà ancora usare il vecchio valore della variabile. Ad es.: >

```

:if !exists("s:contatore_chiamate")
:  let s:contatore_chiamate = 0
:endif
:let s:contatore_chiamate = s:contatore_chiamate + 1
:echo "Chiamato" s:contatore_chiamate "volte"

```

La funzione "exists()" controlla l'esistenza di una variabile. Il suo argomento è il nome della variabile di cui volete controllare l'esistenza. Non la variabile in sé! Se scriveste: >

```
:if !exists(s:contatore_chiamate)
```

Allora il valore di s:contatore_chiamate sarebbe usato come nome della

variabile la cui esistenza viene controllata da `exists()`. Questo non è ciò che volete.

Il punto esclamativo `!` nega un valore. Quando il valore della funzione era "vero" (`true`) diventa "falso" (`false`). Quando era "falso" diventa "vero". Potete leggerlo come la parola "non" (`not`). Ossia `"if !exists()"` si può leggere come `"if not exists()"`, ovvero "se è falso il valore ritornato da `exists()`".

Vim considera "vero" qualsiasi valore diverso da zero. Zero è falso.

Note:

Vim automaticamente converte una stringa in un numero quando sta cercando un numero. Usando una stringa che non comincia con un numero, il numero risultante è zero. Guardando l'esempio seguente: >

```
:if "vero"
```

```
<      Il "vero" verrà interpretato come uno zero, ossia come falso!
```

STRINGHE VARIABILI E COSTANTI

Finora alle variabili sono stati assegnati solo valori numerici. Si possono usare come valori anche stringhe di caratteri. Numeri e stringhe sono i tipi fondamentali di costanti che Vim consente. Il tipo della variabile è dinamico, viene impostato ogni volta che si assegna un valore alla variabile stessa con `":let:"`. Per ulteriori informazioni sul tipo di variabile vedere [|41.8|](#).

Per assegnare il valore di una stringa ad una variabile, dovete usare una costante di tipo stringa. Ne esistono due tipi. La prima è una stringa di caratteri, racchiusa in doppi apici: >

```
:let nome = "pietro"
```

```
:echo nome
```

```
<      petro ~
```

Se volete inserire un doppio apice all'interno della vostra stringa, metteteci davanti un `"\"` (backslash, barra retroversa). >

```
:let nome = "\"pietro\""
```

```
:echo nome
```

```
<      "pietro" ~
```

Per evitare di usare `"\"`, potete racchiudere la stringa fra apici: >

```
:let nome = '"pietro"'
```

```
:echo nome
```

```
<      "pietro" ~
```

All'interno di una stringa racchiusa fra apici singoli, tutti i caratteri restano invariati. Solo l'apice singolo stesso è speciale: ne dovete inserire due per ottenerne uno. Un `"\"` viene anch'esso preso alla lettera, così non lo potete usare per cambiare il significato del carattere che lo segue.

Nelle stringhe racchiuse fra doppi apici è possibile usare i caratteri speciali. Eccone qualcuno utile:

<code>\t</code>	<Tab>	tabulatore
<code>\n</code>	<NL>	a capo
<code>\r</code>	<CR>	<Invio>
<code>\e</code>	<Esc>	escape
<code>\b</code>	<BS>	"cancella un carattere"
<code>\"</code>	<code>"</code>	doppi apici
<code>\\</code>	<code>\</code>	barra retroversa (backslash)
<Esc>	<Esc>	escape (forma alternativa)
<C-W>	CTRL-W	carattere "iniziale" per spostarsi tra finestre

Gli ultimi due sono solo a titolo di esempio. La forma `"\<nome_car>"` si può usare per inserire il carattere speciale "nome_car".

Si veda [|expr-quote|](#) per la lista completa degli elementi speciali di una stringa.

41.3 Espressioni

Vim ha un modo ricco ma semplice per gestire le espressioni. Potete leggere qui la definizione: [|expression-syntax|](#). Qui mostreremo gli elementi più comuni.

I numeri, le stringhe e le variabili citate, sono già espressioni. Così ovunque ci si aspetti un'espressione, potete usare un numero, una stringa od

una variabile. Altri elementi fondamentali in una espressione sono:

\$NOME	variabile d'ambiente
&nome	opzione
@r	registro

Esempio: >

```
:echo "Il valore di 'tabstop' è" &ts
:echo "La tua home directory [Unix] è" $HOME
:if @a > 5
```

La forma &nome si può usare per salvare il valore di una opzione, impostare l'opzione a un nuovo valore, fare qualcosa, e ripristinare il valore precedente. >

```
:let salva = &ic
:set noic
:/L'Inizio/, $delete
:let &ic = save_ic
```

Questo serve ad essere certi che l'espressione "L'Inizio" sia usata con l'opzione '**ignorecase**' non impostata. Alla fine, prende il valore che l'utente aveva impostato. (Un altro modo di fare lo stesso potrebbe essere l'aggiunta di "\C" all'espressione da ricercare, vedere `|/\C|`.)

OPERAZIONI MATEMATICHE

Diventa più interessante se combiniamo questi elementi fondamentali. Cominciamo con la operazioni matematiche.

a + b	addizione
a - b	sottrazione
a * b	moltiplicazione
a / b	divisione
a % b	modulo (resto della divisione)

Valgono le solite regole di precedenza. Ad es.: >

```
< :echo 10 + 5 * 2
20 ~
```

Il raggruppamento viene fatto con le parentesi. Niente sorprese qui. Ad es.: >

```
< :echo (10 + 5) * 2
30 ~
```

Le stringhe si possono riunire con ".". Ad es.: >

```
< :echo "foo" . "bar"
foobar ~
```

Quando si danno più argomenti al comando ":echo", questi vengono separati tra loro con uno spazio. Nell'esempio l'argomento è una singola espressione, così non viene inserito alcuno spazio.

L'espressione condizionale viene presa dal linguaggio C:

a ? b : c

Se "a" è vera, si usa "b", altrimenti si usa "c". Ad es.: >

```
< :let i = 4
:echo i > 5 ? "i è grande" : "i è piccolo"
i è piccolo ~
```

Le tre parti dei costrutti vengono sempre valutate prima, per cui potreste vederle funzionare come:

(a) ? (b) : (c)

=====

41.4 Condizioni

Il comando `:"if` esegue i comandi che lo seguono, fino all'`:"endif`", solo se la condizione è verificata. La forma generica è:

```
:"if {condizione}
    {comandi}
:"endif
```

Solo quando l'espressione `{condizione}` viene considerata vera (valore diverso da zero) i `{comandi}` vengono eseguiti. Questi debbono essere comandi validi. Se contengono spazzatura, Vim non riesce a trovare l'`:"endif`".

Potete anche usare `:"else`". La forma generica è:

```
:"if {condizione}
    {comandi}
:"else
    {comandi}
:"endif
```

Il secondo `{comandi}` viene eseguito solo se il primo non lo è. Infine, abbiamo l'`:"elseif`":

```
:"if {condizione}
    {comandi}
:"elseif {condizione}
    {comandi}
:"endif
```

Questo equivale ad usare `:"else`" e poi `:"if`", ma senza la necessità di inserire un ulteriore `:"endif`".

Un esempio utile per il vostro file vimrc è verificare l'opzione `'term'` e fare qualcosa in funzione del suo valore: >

```
:"if &term == "xterm"
:  " Fai qualcosa per xterm
:"elseif &term == "vt100"
:  " Fai qualcosa per un terminale vt100
:"else
:  " Fai qualcosa per gli altri tipi di terminale
:"endif
```

OPERAZIONI LOGICHE

Ne abbiamo già utilizzate alcune negli esempi precedenti. Quelle di più frequente utilizzo sono:

<code>a == b</code>	uguale a
<code>a != b</code>	diversa da
<code>a > b</code>	maggiore di
<code>a >= b</code>	maggiore o uguale a
<code>a < b</code>	minore di
<code>a <= b</code>	minore o uguale a

Il risultato è 1 se la condizione è verificata, altrimenti è 0. Ad es.: >

```
:"if v:version >= 700
:  echo "Congratulazioni"
:"else
:  echo "State usando una vecchia, versione, passate alla nuova!"
:"endif
```

Qui `"v:version"` è una variabile definita da Vim, che assume il valore della versione di Vim. 600 sta per la versione 6.0. La versione 6.1 ha il valore 6.01. Questo è molto utile per scrivere uno script che debba funzionare con molte versioni di Vim. `|v:version|`

Gli operatori logici lavorano sia con numeri che con stringhe. Confrontando due stringhe, viene usata la differenza matematica. Questa confronta il valore dei byte, ma può non andare bene per alcuni linguaggi.

Confrontando una stringa con un numero, la stringa viene prima convertita a numero. Questo è un po' arbitrario, perché quando una stringa non appare come

un numero, viene utilizzato il numero 0. Ad es.: >

```
:if 0 == "uno"
:  echo "sì"
:endif
```

Ciò darà come risultato "sì", poiché "uno" non appare come un numero, e così viene convertito nel numero zero.

Per le stringhe ci sono due ulteriori elementi;

```
a =~ b          corrisponde a
a !~ b          non corrisponde a
```

L'elemento di sinistra "a" viene usato come una stringa. L'elemento di destra "b" viene usato come espressione, simile a quelle usate per la ricerca. Ad es.: >

```
:if str =~ " "
:  echo "str contiene uno spazio"
:endif
:if str !~ '\.$'
:  echo "str non finisce con un punto"
:endif
```

Notate l'uso della stringa inclusa fra apici singoli nell'espressione. Ciò è utile perché i "\" andrebbero scritti due volte all'interno di una stringa inclusa fra doppi apici, e le espressioni regolari tendono a contenere parecchi "\".

L'opzione '**ignorecase**' viene usata nel confronto tra stringhe. Se non volete ciò, aggiungete "#" per tener conto di maiuscole e minuscole, e "?" per non farlo. Così "==" verifica se due stringhe siano uguali, trascurando maiuscole e minuscole. E "!~#" controlla se una espressione non corrisponde, controllando anche maiuscole e minuscole. Per la tabella completa degli operatori, si veda |**expr==**|.

ALTRI CICLI

Il comando ":while" è già stato citato. Ci sono altri due comandi che si possono usare tra ":while" e ":endwhile":

```
:continue      Ritorna all'inizio del ciclo "while"; il ciclo
                inizia una nuova iterazione.
:break         Salta in avanti fino a ":endwhile"; il ciclo
                termina l'esecuzione.
```

Ad es.: >

```
:while contatore < 40
:  call fa_qualche_controllo()
:  if salta_un_giro
:    continue
:  endif
:  if abbiamo_finito
:    break
:  endif
:  sleep 50m
:endwhile
```

Il comando ":sleep" fa fare a Vim un pisolino. "50m" significa "per cinquanta millisecondi". Un altro esempio è ":sleep 4", che fa dormire per quattro secondi.

Altri cicli si possono fare con il comando ":for", vedere più oltre in |**41.8**|.

=====

41.5 Esecuzione di una espressione

Finora i comandi nello script venivano eseguiti direttamente da Vim. Il comando ":execute" permette di eseguire il risultato di una espressione. È questa una maniera molto potente per costruire comandi ed eseguirli.

Un esempio è per saltare ad un tag, che sia contenuto in una variabile:

```
>
:execute "tag " . nome_tag
```

Il "." si usa per concatenare la stringa "tag " con il valore della variabile "nome_tag". Supponiamo che "tag_name" abbia come valore "get_cmd", allora il comando che verrà eseguito è: >

```
:tag get_cmd
```

Il comando ":execute" può solo eseguire i comandi che iniziano con ":". Il comando ":normal" esegue i comandi che si possono dare in Normal mode. Comunque, l'argomento di ":normal" non è una espressione, ma i caratteri del comando letterale. Ad es.: >

```
:normal gg=G
```

Questo comando salta alla prima linea e formatta tutte le linee usando l'operatore "=".

Per far funzionare ":normal" con un'espressione, combinatelo con ":execute".
Ad es.: >

```
:execute "normal " . comandi_normali
```

La variabile "comandi_normali" deve contenere comandi del Normal mode.

Assicuratevi che l'argomento di ":normal" sia un comando completo. Altrimenti Vim raggiungerà la fine dell'argomento, ma non eseguirà il comando. Ad es. se voi iniziate in Insert mode, dovete anche uscire da Insert mode. Ad esempio: >

```
:execute "normal Inuovo testo \<Esc>"
```

Ciò inserisce "nuovo testo " nella linea corrente. Si noti l'uso del tasto speciale "\<Esc>". Ciò evita di dover immettere un vero carattere <Esc> nel vostro script.

Se non volete eseguire una stringa, ma valutarla per utilizzarne il valore, potete usare la funzione eval(): >

```
:let optname = "path"
:let optval = eval('&' . optname)
```

Un carattere "&" è anteposto a "path", quindi l'argomento di eval() è in questo esempio "&path". Il risultato sarà quindi il valore della opzione 'path'.

Lo stesso risultato si può ottenere con: >

```
:exe 'let optval = &' . optname
```

=====

41.6 Utilizzo funzioni

Vim definisce parecchie funzioni e fornisce un grande numero di funzionalità in questo modo. Qualche esempio verrà dato in questa sezione. Potete trovare qui la lista completa: |[functions](#)|.

Una funzione si invoca col comando ":call". I parametri vengono passati alla funzione fra parentesi, separati da virgole. Ad es.: >

```
:call search("Data: ", "W")
```

Questo invoca la funzione search(), con argomenti "Date: " e "W". La funzione search() usa il proprio primo argomento come una espressione da ricercare e il secondo come un indicatore. L'indicatore "W" significa che la ricerca non aggiri la fine del file.

Una funzione può essere invocata all'interno di una espressione. Ad es.: >

```
:let line = getline(".")
:let repl = substitute(line, '\a', "*", "g")
:call setline(".", repl)
```

La funzione getline() ottiene una linea dal buffer corrente. Il suo argomento è una specificazione del numero di linea. In questo caso viene usato "." che

significa la linea in cui è posizionato il cursore.

La funzione `substitute()` fa qualcosa di simile al comando `:substitute`. Il primo argomento è la stringa in cui effettuare la sostituzione. Il secondo è l'espressione da cercare, il terzo argomento la stringa con cui rimpiazzarla. Gli ultimi argomenti, per finire, sono indicatori.

La funzione `setline()` imposta la linea, specificata dal primo argomento, ad una nuova stringa, il secondo argomento. In questo esempio, la linea sotto il cursore viene rimpiazzata con il risultato della funzione `substitute()`. Così l'effetto dei tre comandi è uguale a: >

```
:substitute/\a/*/g
```

L'uso delle funzioni si fa più interessante quando fate più lavoro prima e dopo la invocazione di `substitute()`.

FUNZIONI

function-list

Ci sono parecchie funzioni. Le citiamo qui sotto, raggruppate secondo il loro impiego. Potete trovare una lista in ordine alfabetico qui: [|functions|](#). Usate **CTRL-]** sul nome di funzione per saltare ad una documentazione più dettagliata su di essa.

Manipolazione di stringhe:

<code>nr2char()</code>	ottiene un carattere dal suo valore ASCII
<code>char2nr()</code>	ottiene il valore ASCII di un carattere
<code>str2nr()</code>	converte una stringa a un numero
<code>printf()</code>	formatta una stringa secondo elementi %
<code>escape()</code>	premette '\' a caratteri in una stringa
<code>shellescape()</code>	protegge una stringa per uso con comando shell
<code>tr()</code>	traduce caratteri da un assieme all'altro
<code>strtrans()</code>	modifica una stringa rendendola stampabile
<code>tolower()</code>	passa una stringa a caratteri minuscoli
<code>toupper()</code>	passa una stringa a caratteri maiuscoli
<code>match()</code>	si posiziona ad una espressione in una stringa
<code>matchend()</code>	si posiziona a fine espressione in una stringa
<code>matchstr()</code>	trova una espressione in una stringa
<code>matchlist()</code>	come <code>matchstr()</code> ma opera usando una lista
<code>stridx()</code>	primo indice ad una stringa in una stringa
<code>strridx()</code>	ultimo indice ad una stringa in una stringa
<code>strlen()</code>	lunghezza di una stringa
<code>substitute()</code>	sostituisce una espressione in una stringa
<code>submatch()</code>	ottiene un dato valore in una <code>:substitute</code>
<code>strpart()</code>	ottiene una parte di una stringa
<code>expand()</code>	espande caratteri speciali
<code>iconv()</code>	converte testo da una codifica a un'altra
<code>byteidx()</code>	restituisce indice a carattere in una stringa
<code>repeat()</code>	ripete una stringa più volte
<code>eval()</code>	ottiene valore di espressione in una stringa

Manipolazione di Liste:

<code>get()</code>	ottiene elemento, ignora errori di indice
<code>len()</code>	numero di elementi in una Lista
<code>empty()</code>	controlla se una Lista è vuota
<code>insert()</code>	inserisce elemento in una lista
<code>add()</code>	aggiunge elemento in fondo a una Lista
<code>extend()</code>	aggiunge una Lista a una Lista
<code>remove()</code>	toglie elemento/i da una Lista
<code>copy()</code>	fa una copia nominale di una Lista
<code>deepcopy()</code>	fa una copia vera di una Lista
<code>filter()</code>	toglie elementi scelti da una Lista
<code>map()</code>	cambia ogni elemento della Lista
<code>sort()</code>	ordina Lista
<code>reverse()</code>	ordina all'indietro una Lista
<code>split()</code>	genera Lista da una Stringa
<code>join()</code>	congiunge elementi da Lista in una Stringa
<code>range()</code>	restituisce Lista con una sequenza di numeri
<code>string()</code>	rappresentazione di una Lista con una Stringa
<code>call()</code>	chiama una funzione con una Lista per argomenti
<code>index()</code>	indice di un valore in una Lista
<code>max()</code>	massimo valore in una Lista
<code>min()</code>	minimo valore in una Lista
<code>count()</code>	numero occorrenze di un valore in una Lista

repeat() ripete una Lista più volte

Manipolazione di Dizionari:

get()	ottiene elemento, ignorando errori di indice
len()	numero di elementi in un Dizionario
has_key()	controlla esistenza chiave in un Dizionario
empty()	controlla se un Dizionario è vuoto
remove()	toglie elemento da un Dizionario
extend()	aggiunge elementi di un Dizionario a un altro
filter()	toglie elementi scelti da un Dizionario
map()	cambia ogni elemento del Dizionario
keys()	ottiene Lista di chiavi del Dizionario
values()	ottiene Lista di valori del Dizionario
items()	ottiene Lista coppie chiave-valore del Diz.
copy()	fa una copia nominale di un Dizionario
deepcopy()	fa una copia vera di un Dizionario
string()	rappresentazione di un Diz. con una Stringa
max()	massimo valore in un Dizionario
min()	minimo valore in un Dizionario
count()	numero occorrenze di un valore in un Diz.

Variabili:

type()	tipo di una variabile
islocked()	controllo se una variabile è non modificabile
function()	ottiene Funcref per un nome funzione
getbufvar()	imposta valore variabile da un dato buffer
setbufvar()	imposta variabile in un dato buffer
getwinvar()	ottiene variabile da una data finestra
gettabwinvar()	ottiene variabile da una data fin. & linguetta
setwinvar()	imposta variabile in una data finestra
settabwinvar()	imposta variabile in una data fin. & linguetta
garbagecollect()	libera memoria non più usata, se possibile

Posizione del cursore e del marcatore:

col()	numero di colonna del cursore o marcatore
virtcol()	colonna sullo schermo del cursore o marcatore
line()	numero di linea del cursore o marcatore
wincol()	numero colonna del cursore nella finestra
winline()	numero linea del cursore nella finestra
cursor()	posiziona il cursore a linea/colonna
getpos()	ottieni posizione cursore, marcatore, etc.
setpos()	imposta posizione cursore, marcatore, etc.
byte2line()	dice a che linea si trova un certo byte
line2byte()	dice posizione in byte di una data linea
diff_filler()	ottieni num. linee riempimento sopra una linea

Modificare il testo nel buffer corrente:

getline()	ottiene una linea o Lista linee dal buffer
setline()	sostituisce una linea nel buffer
append()	aggiungere una linea o Lista linee al buffer
indent()	fa rientrare una data linea
cindent()	fa rientrare una data linea (stile C)
lispindent()	fa rientrare una data linea (stile Lisp)
nextnonblank()	trova la prossima linea non vuota
prevnonblank()	trova la linea non vuota precedente
search()	trova una espressione nel testo
searchpos()	trova una espressione nel testo
searchpair()	trova l'altro capo, di start/skip/end
searchpairpos()	trova l'altro capo, di start/skip/end
searchdecl()	cerca la dichiarazione di un nome

Funzioni di sistema e manipolazione di file:

glob()	valorizza espressioni regolari (wildcard)
globpath()	valorizza espressione lista-di-directory
findfile()	trova un file in una lista di directory
finddir()	trova una directory in una lista di direct.
resolve()	trova nome file dato puntatore (shortcut)
fnamemodify()	modifica nome file
pathshorten()	accorcia nomi directory in un path
simplify()	semplifica un path senza cambiarne il valore
executable()	controlla esistenza programma eseguibile
filereadable()	controlla se possibile leggere un file
filewritable()	controlla se possibile scrivere un file

getfperm()	ottiene permessi utilizzo di un file
getftype()	ottiene il tipo di un file
isdirectory()	controlla se una data directory esiste
getfsize()	ottiene lunghezza di un file
getcwd()	ottiene directory di lavoro corrente
haslocaldir()	controlla la finestra corrente ha usato :lcd
tempname()	ottiene nome di un file temporaneo
mkdir()	crea una nuova directory
delete()	cancella un file
rename()	rinomina un file
system()	ottiene codice ritorno da comando shell
hostname()	ottiene nome del computer
readfile()	legge un file in una Lista linee
writefile()	scrive una Lista linee su un file

Data e Ora:

getftime()	ottiene data ultima modifica di un file
localtime()	ottiene la data corrente
strftime()	converte una data in formato stampabile
reltime()	ottiene accurato tempo corrente o trascorso
reltimestr()	converte risultati reltime() in stringa

Buffer, finestre e lista argomenti:

argc()	numero argomenti nella lista argomenti
argidx()	posizionamento corrente nella lista argomenti
argv()	ottiene un argomento dalla lista argomenti
bufexists()	controlla se un buffer esiste
buflisted()	controlla se un buffer esiste ed è listato
bufloaded()	controlla se un buffer esiste ed è caricato
bufname()	ottiene nome di un dato buffer
bufnr()	ottiene numero buffer di un dato buffer
tabpagebuflist()	ottiene Lista di buffer in una linguetta
tabpagenr()	ottiene il numero di una linguetta
tabpagewinnr()	come winnr() per una data linguetta
winnr()	ottiene numero finestra di finestra corrente
bufwinnr()	ottiene numero finestra di un dato buffer
winbufnr()	ottiene numero buffer di una data finestra
getbufline()	ottiene una lista di linee da un dato buffer

Linea comando:

getcmdline()	ottiene la linea comando corrente
getcmdpos()	ottiene posizione cursore su linea comando
setcmdpos()	imposta posizione cursore su linea comando
getcmdtype()	ottiene il tipo della linea comando corrente

Quickfix e lista locazione:

getqflist()	ottiene lista errori quickfix
setqflist()	modifica una lista quickfix
getloclist()	ottiene lista elementi locazione
setloclist()	modifica lista elementi locazione

Completamento in modalità Insert:

complete()	imposta corrispondenze trovate
complete_add()	aggiungi a corrispondenze trovate
complete_check()	controlla se completamento va interrotto
pumvisible()	controlla se il menù popup è visualizzato

Piegature:

foldclosed()	controlla se piegatura chiusa in una linea data
foldclosedend()	come foldclosed() ma restituisce ultima linea
foldlevel()	controlla livello piegatura in una linea data
foldtext()	specifiche linea che indica piegatura chiusa
foldtextresult()	ottiene testo visualizzato in piegatura chiusa

Sintassi ed evidenziazione:

clearmatches()	annulla tutte le corrispondenze definite dai comandi matchadd() e :match
getmatches()	ottiene tutte le corrispondenze definite dai comandi matchadd() e :match
hlexists()	controlla se esiste gruppo evidenziazione
hlID()	ottiene ID di un gruppo evidenziazione
synID()	ottiene ID sintattico ad una data posizione
synIDattr()	ottiene un dato attributo di un ID sintattico

synIDtrans() diff_hlID() matchadd() matcharg() matchdelete() setmatches()	ottiene traduzione di un ID sintattico ottiene ID evidenz. modo diff in una posizione definisce espressione regolare da evidenziare (un "match", ossia una corrispondenza) ottiene informazioni su argomenti :match annulla una corrispondenza definita dai comandi matchadd() o :match ripristina una lista di corrispondenze salvata da getmatches()
Controllo ortografico: spellbadword() spellsuggest() soundfold()	trova parole errate al cursore od oltre ottiene correzioni ortografiche suggerite ottiene parole di suono simile a una parola
Storia: histadd() histdel() histget() histnr()	aggiunge un elemento a una storia toglie un elemento a una storia ottiene un elemento da una storia ottiene indice più alto di una storia
Interazione: browse() browseDir() confirm() getchar() getcharmod() feedkeys() input() inputlist() inputsecret() inputdialog() inputsave() inputrestore()	richiedi file da visualizzare richiedi directory da visualizzare chiede all'utente di fare una scelta ottiene un carattere dall'utente ottiene modificatori ultimo carattere immesso mette caratteri nella coda dei caratteri immessi (ma non ancora trattati) [typeahead queue] ottiene una linea dall'utente fa scegliere all'utente elemento in una lista ottiene una linea dall'utente senza mostrarla ottiene una linea dall'utente usando GUI salva caratteri immessi e pulisce ripristina caratteri immessi
GUI: getfontname() getwinposx() getwinposy()	ottiene nome font attualmente in uso posizione X della finestra GUI Vim posizione Y della finestra GUI Vim
Vim server: serverlist() remote_send() remote_expr() server2client() remote_peek() remote_read() foreground() remote_foreground()	restituisce lista nomi server invia caratteri comando a server Vim valuta una espressione in un server Vim invia risposta a un cliente di un server Vim controlla se esiste risposta da un server Vim legge risposta da Vim server sposta finestra Vim in primo piano sposta finestra server Vim in primo piano
Posizione e dimensioni finestra: winheight() winwidth() winrestcmd() winsaveview() winrestview()	ottiene altezza di una data finestra ottiene larghezza di una data finestra ottiene comando per ripristinare dimens. fin. ottiene veduta della finestra corrente ripristina vista salvata d. finestra corrente
Varie: type() mode() visualmode() hasmapto() mapcheck() maparg() exists() has() changenr() cscope_connection() did_filetype() eventhandler()	restituisce il tipo di una variabile ottiene modalità di edit corrente dice ultima modalità visuale utilizzata controlla se esiste mappatura controlla se esiste mappatura corrispondente ottiene valore di una mappatura controlla se variabile, funzione, etc. esiste controlla se Vim supporta una estensione data ottiene numero della modifica più recente controlla se esiste una connessione cscope controlla se già usato autocomando tipo_file controlla se invocato da un gestore eventi

libcall()	richiama una funzione in una libreria esterna
libcallnr()	come sopra, la funzione restituisce un numero
getreg()	ottieni contenuto di un registro
getregtype()	ottieni tipo di un registro
setqflist()	crea una Lista quickfix
setreg()	imposta contenuto e tipo di un registro
taglist()	ottieni Lista di tag corrispondenti
tagfiles()	ottieni Lista di file di tag

=====

41.7 Definizione funzioni

Vim vi consente di definire le proprie funzioni. La dichiarazione di una funzione comincia così: >

```
:function {nome}({var1}, {var2}, ...)
: {corpo_della_funzione}
:endfunction
```

<

Note:

I nomi di funzione devono cominciare con una lettera maiuscola.

Definiamo una breve funzione che restituisce il minore di due numeri. Comincia con questa linea:

```
:function Min(num1, num2)
```

Ciò dice a Vim che la funzione si chiama "Min" ed ha due argomenti: "num1" e "num2".

La prima cosa di cui avete bisogno è di tentare di vedere quale numero sia il più piccolo:

```
>
: if a:num1 < a:num2
```

Il prefisso speciale "a:" dice a Vim che la variabile è un argomento della funzione. Assegniamo alla variabile "minore", il valore del numero più piccolo: >

```
: if a:num1 < a:num2
:   let minore = a:num1
: else
:   let minore = a:num2
: endif
```

La variabile "minore" è una variabile locale. Variabili usate all'interno di una funzione sono locali se non hanno prefissi come "g:", "a:", o "s:".

Note:

Per accedere ad una variabile globale dall'interno di una funzione dovete premettere "g:" alla variabile stessa. Così "g:contatore" all'interno di una funzione viene usato per la variabile globale "contatore", e "contatore" è un'altra variabile, locale per la funzione.

Usate il comando ":return" per restituire il numero più piccolo all'utente. Infine, indicate la fine della funzione: >

```
: return minore
:endfunction
```

La definizione completa della funzione è la seguente: >

```
:function Min(num1, num2)
: if a:num1 < a:num2
:   let minore = a:num1
: else
:   let minore = a:num2
: endif
: return minore
:endfunction
```

Se preferite funzioni concise, questa fa la stessa cosa: >

```
:function Min(num1, num2)
:   if a:num1 < a:num2
:       return a:num1
:   endif
:   return a:num2
:endfunction
```

<

Una funzione definita dall'utente viene invocata esattamente nello stesso modo con cui vengono invocate le funzioni predefinite di Vim. Solo il nome è diverso. La funzione Min si può usare così: >

```
:echo Min(5, 8)
```

Solo adesso la funzione verrà eseguita, e le linee saranno interpretate da Vim. Se ci sono errori, come usare una variabile o una funzione indefinita, si otterrà un messaggio di errore. Quando si definisce una funzione, errori di questo tipo non vengono rilevati.

Quando una funzione trova ":endfunction" o se ":return" viene usato senza un argomento, la funzione restituisce zero.

Per ridefinire una funzione già esistente, usate il "!" nel comando ":function": >

```
:function! Min(num1, num2, num3)
```

USARE UN INTERVALLO

Al comando ":call" si può dare un intervallo di linee su cui agire. Ciò può avere uno o due significati. Quando una funzione è stata definita con la parola chiave "range", terrà conto dell'intervallo di linee stesso.

Alla funzione verranno passate le variabili "a:firstline" e "a:lastline". Queste avranno i numeri di linea dall'intervallo in cui la funzione è stata chiamata. Ad es: >

```
:function Conta_parole() range
:   let n = a:firstline
:   let quante = 0
:   while n <= a:lastline
:       let quante = quante + Wordcount(getline(n))
:       let n = n + 1
:   endwhile
:   echo "trovate " . quante . " parole"
:endfunction
```

Potete chiamare questa funzione con: >

```
:10,30call Conta_parole()
```

Verrà eseguita una volta sola, e scriverà il numero di parole.

L'altro modo per usare un intervallo di linee consiste nel definire una funzione senza la parola chiave "range". La funzione sarà chiamata una volta per ogni linea nell'intervallo, con il cursore posizionato su quella linea.

Ad es.: >

```
:function Numero()
:   echo "La linea numero " . line(".") . " contiene: " . getline(".")
:endfunction
```

Se chiamate questa funzione con: >

```
:10,15call Numero()
```

la funzione verrà invocata sei volte.

NUMERO VARIABILE DI ARGOMENTI

Vim vi consente di definire funzione che abbia un numero variabile di argomenti. Il comando seguente, ad es., definisce una funzione che deve avere

almeno 1 argomento (inizio) e può avere più di 20 argomenti in aggiunta al primo: >

```
:function Mostra(inizio, ...)
```

La variabile "a:1" contiene il primo argomento facoltativo, "a:2" il secondo, e così via. La variabile "a:0" contiene il numero di argomenti aggiuntivi. Ad es.: >

```
:function Mostra(inizio, ...)
:  echohl Title
:  echo "Mostra è " . a:inizio
:  echohl None
:  let indice = 1
:  while indice <= a:0
:    echo "  Arg " . indice . " è " . a:{index}
:    let indice = indice + 1
:  endwhile
:  echo ""
:endifunction
```

La funzione usa il comando ":echohl" per specificare l'evidenziazione da usare per il successivo comando ":echo". ":echohl None" torna alla visualizzazione normale. Il comando ":echon" si comporta come ":echo", ma non scrive il carattere per andare a capo.

Si può anche usare la variabile a:000, che è una Lista di tutti gli argomenti "...". Vedere |a:000|.

LISTA DELLE FUNZIONI

Il comando ":function" lista i nomi e gli argomenti di tutte le funzioni definite dall'utente: >

```
< :function
function Show(start, ...) ~
function GetVimIndent() ~
function SetSyn(name) ~
```

Per vedere cosa fa una funzione, usate il suo nome come argomento per ":function": >

```
< :function SetSyn
1   if &syntax == '' ~
2     let &syntax = a:name ~
3   endif ~
endfunction ~
```

DEBUGGING

Il numero linea è utile quando ricevete un messaggio di errore o in fase di debug. Si veda |debug-scripts| a proposito del debug.

Potete anche impostare l'opzione '**verbose**' a 12 o più per vedere tutte le chiamate di funzione. Impostatelo a 15 o più per visualizzare ogni linea eseguita.

CANCELLARE UNA FUNZIONE

Per cancellare la funzione Mostra(): >

```
:delfunction Mostra
```

Se la funzione non esiste, viene visualizzato un messaggio di errore.

REFERENZE A FUNZIONI

Talvolta può servire che una variabile punti a una funzione o a un'altra. Si può fare con la funzione function(). Cambia il nome di una funzione in una referencia: >

```

:let result = 0          " o 1
:function! Right()
:  return 'Right!'
:endfunc
:function! Wrong()
:  return 'Wrong!'
:endfunc
:
:if result == 1
:  let Afunc = function('Right')
:else
:  let Afunc = function('Wrong')
:endif
:echo call(Afunc, [])
< Wrong! ~

```

Note Il nome di una variabile che contiene una referenza a una funziona deve iniziare con una lettera maiuscola. Altrimenti potrebbe essere confusa col nome di una funzione predefinita.

La maniera per far eseguire una funzione il cui nome è contenuto in una variabile è quello di usare la funzione call(). Il suo primo argomento è la referenza alla funzione, il secondo argomento è una Lista che contiene gli argomenti.

Riferimenti a funzioni sono molto utili utilizzando un Dizionario, come viene spiegato nella sezione che segue.

=====

41.8 Liste e Dizionari

Finora abbiamo usato i tipi fondamentali di variabili, Stringhe e Numeri. Vim permette anche due tipi composti: Liste e Dizionari.

Una Lista è una sequenza ordinata di cose. Le cose possono essere ogni tipo di valore, quindi si può fare una Lista di numeri, una lista di Liste e perfino una lista di elementi differenti. Per creare una lista con tre stringhe: >

```
:let alist = ['aap', 'mies', 'noot']
```

Gli elementi della lista sono rinchiusi fra parentesi quadre e separati da virgole. Per creare una Lista vuota: >

```
:let alist = []
```

Potere aggiungere elementi a una lista con la funziona add(); >

```

:let alist = []
:call add(alist, 'foo')
:call add(alist, 'bar')
:echo alist
< ['foo', 'bar'] ~

```

La concatenazione di elementi ad una lista si fa col +: >

```

:echo alist + ['foo', 'bar']
< ['foo', 'bar', 'foo', 'bar'] ~

```

O, se volete estendere una Lista direttamente: >

```

:let alist = ['one']
:call extend(alist, ['two', 'three'])
:echo alist
< ['one', 'two', 'three'] ~

```

Si noti che usando add() si ottiene un risultato diverso: >

```

:let alist = ['one']
:call add(alist, ['two', 'three'])
:echo alist
< ['one', ['two', 'three']] ~

```

Il secondo argomenti di add() è aggiunto come un elemento singolo.

CICLO FOR

Una delle cose carine che si possono fare con una Lista è visitarla: >

```

:let alist = ['one', 'two', 'three']
:for n in alist
:  echo n
:endfor
< one ~
two ~
three ~

```

In questo modo si passa attraverso ogni elemento della Lista "alist", assegnandone il valore alla variabile "n". La forma generica di un ciclo for è >

```

:for {nome_variabale} in {designazione_di_Lista}
:  {comandi}
:endfor

```

Per visitare la lista un dato numero di volte vi serve una Lista di una lunghezza specifica. La funzione range() ne crea una per voi: >

```

:for a in range(3)
:  echo a
:endfor
< 0 ~
1 ~
2 ~

```

Si noti che il primo elemento della Lista prodotta da range() è zero, quindi l'ultimo elemento è inferiore di uno alla lunghezza della lista.

Potete anche specificare il valore massimo, l'incremento, e pure andare "all'indietro": >

```

:for a in range(8, 4, -2)
:  echo a
:endfor
< 8 ~
6 ~
4 ~

```

Un esempio più utile, che percorre le linee di un buffer: >

```

:for line in getline(1, 20)
:  if line =~ "Date: "
:    echo matchstr(line, 'Date: \zs.*')
:  endif
:endfor

```

Si esaminano qui le linee da 1 a 20 (incluso) visualizzando ogni data trovata.

DIZIONARI

Un Dizionario contiene coppie chiave-valore. Potete velocemente consultare un Dizionario se conoscete la chiave. Un Dizionario è creato fra parentesi graffe:

```

:let uk2nl = {'one': 'een', 'two': 'twee', 'three': 'drie'}

```

Ora potete consultare elementi (parole) mettendo la chiave fra parentesi quadre: >

```

:echo uk2nl['two']
< twee ~

```

La forma generica di definizione di un Dizionario è: >

```

{<chiave> : <valore>, ...}

```

Un Dizionario vuoto non contiene alcuna chiave: >

```
{}
```

Numerose sono le possibilità offerte dai Dizionari. Ci sono anche varie funzioni che si possono utilizzare con essi. Ad esempio, potete ottenere una lista delle chiavi e visitarla: >

```
:for key in keys(uk2nl)
:  echo key
:endfor
< three ~
one ~
two ~
```

Noterete che le chiavi non sono in ordine. Potete ottenere la lista in un ordine da voi specificato: >

```
:for key in sort(keys(uk2nl))
:  echo key
:endfor
< one ~
three ~
two ~
```

Non c'è modo di ottenere l'ordine nel quale gli elementi sono stati introdotti. A questo scopo è necessario utilizzare una Lista, che memorizza gli elementi in una sequenza ordinata.

FUNZIONI DEI DIZIONARI

Gli elementi in un Dizionario di ottengono di solito fornendo un indice fra parentesi quadre: >

```
:echo uk2nl['one']
< een ~
```

Per ottenere lo stesso risultato, usando meno segni di interpunzione: >

```
:echo uk2nl.one
< een ~
```

Questa notazione funziona solo se la chiave è composta da lettere ASCII, numeri e dal carattere "_" (underscore). Potete anche assegnare valori allo stesso modo: >

```
:let uk2nl.four = 'vier'
:echo uk2nl
< {'three': 'drie', 'four': 'vier', 'one': 'een', 'two': 'twee'} ~
```

E ora qualcosa di speciale: potete definire direttamente una funzione ed inserire una referenza a quella funzione nel Dizionario: >

```
:function uk2nl.translate(line) dict
:  return join(map(split(a:line), 'get(self, v:val, "???)'))
:endfunction
```

Cominciamo a provarla: >

```
:echo uk2nl.translate('three two five one')
< drie twee ??? een ~
```

La prima cosa speciale che si nota è il "dict" alla fine della linea ":function". Questo segnala che la funzione è usata da un Dizionario. La variabile locale "self" farà riferimento a quel Dizionario.

Ora spezziamo il complesso comando che genera il risultato: >

```
split(a:line)
```

La funzione split() piglia una stringa, la spezza in parole separate da spazi e restituisce una lista delle parole così ottenute. Quindi nell'esempio restituisce: >

```

:echo split('three two five one')
< ['three', 'two', 'five', 'one'] ~

```

Questa lista è il primo argomento alla funzione map(). map() visiterà la lista, valutando il suo secondo argomento con la variabile "v:val" impostata al valore di ogni elemento. Si tratta della scrittura abbreviata di un ciclo "for". Questo comando: >

```
:let alist = map(split(a:line), 'get(self, v:val, "??")')
```

è equivalente a: >

```

:let alist = split(a:line)
:for idx in range(len(alist))
:  let alist[idx] = get(self, alist[idx], "??")
:endfor

```

La funzione get() controlla se una chiave è presente in un Dizionario. Se esiste, ne ritrova il valore. Se non esiste, restituisce un valore di default, '???' in questo caso. E' questa una maniera comoda per gestire situazioni in cui una chiave può non essere presente, e non si desidera un messaggio di errore.

La funzione join() fa il lavoro inverso rispetto a split(): unisce assieme una lista di parole, mettendo uno spazio fra ognuna di esse.

Questa combinazione di split(), map() e join() è una maniera elegante di filtrare una linea di parole in maniera molto concisa.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Ora che potete inserire sia valori che funzioni in un Dizionario, potete in pratica utilizzare un Dizionario come un oggetto.

Sopra abbiamo usato un Dizionario per tradurre da Olandese a Inglese. Potremmo voler fare lo stesso per altri linguaggi. Creiamo prima un oggetto (noto anche come Dizionario) che abbia la funzione di traduzione, ma senza parole da tradurre: >

```

:let transdict = {}
:function transdict.translate(line) dict
:  return join(map(split(a:line), 'get(self.words, v:val, "??")'))
:endfunction

```

E' lievemente differente dalla funzione di prima, usa 'self.words' per consultare le traduzioni delle parole. Ma noi non abbiamo un 'self.words'. Quindi la si potrebbe definire una classe astratta.

Ora possiamo realizzare un oggetto che traduca in Olandese: >

```

:let uk2nl = copy(transdict)
:let uk2nl.words = {'one': 'een', 'two': 'twee', 'three': 'drie'}
:echo uk2nl.translate('three one')
< drie een ~

```

E un traduttore in Tedesco: >

```

:let uk2de = copy(transdict)
:let uk2de.words = {'one': 'ein', 'two': 'zwei', 'three': 'drei'}
:echo uk2de.translate('three one')
< drei ein ~

```

Potete vedere che la funzione copy() è usata per fare una copia del dizionario "transdict" e quindi la copia è modificata per aggiungere le parole. L'originale resta immutato, naturalmente.

Ora potete far un passo ulteriore, e usare il vostro traduttore preferito: >

```

:if $LANG =~ "de"
:  let trans = uk2de
:else
:  let trans = uk2nl
:endif

```

```

:echo trans.translate('one two three')
< een twee drie ~

```

Qui "trans" fa riferimento a uno dei due oggetti (Dizionari). Nessuna copia è qui fatta. Ulteriori informazioni su Liste e Dizionari può essere trovata a [|list-identity|](#) e [|dict-identity|](#).

Ora potreste usare un linguaggio non supportato. Potete forzare la funzione `translate()` a non fare alcunché: >

```

:let uk2uk = copy(transdict)
:function! uk2uk.translate(line)
:  return a:line
:endfunction
:echo uk2uk.translate('three one wladivostok')
< three one wladivostok ~

```

SI noti che un ! è stato usato per ricoprire il riferimento a una funzione esistente. A questo punto usate "uk2uk" quando il linguaggio non viene riconosciuto: >

```

:if $LANG =~ "de"
:  let trans = uk2de
:elseif $LANG =~ "nl"
:  let trans = uk2nl
:else
:  let trans = uk2uk
:endif
:echo trans.translate('one two three')
< one two three ~

```

Per ulteriori letture vedere [|Lists|](#) e [|Dictionaries|](#).

```

=====
*41.9* Eccezioni

```

Cominciamo con un esempio: >

```

:try
:  read ~/templates/pascal.tmpl
:catch /E484:/
:  echo "Spiacente, il file_modello Pascal non è disponibile."
:endtry

```

Il comando `:read` non può essere eseguito se il file non esiste. Invece di lasciar visualizzare un messaggio di errore, questo codice cattura l'errore e dà all'utente un messaggio più comprensibile.

Per i comandi compresi fra `:try` e `:endtry` gli errori sono trasformati in eccezioni. Una eccezione è una stringa di caratteri. Nel caso di un errore la stringa contiene il messaggio di errore. E ogni messaggio di errore ha un identificativo di messaggio. In questo caso, l'errore che intercettiamo contiene l'identificativo "E484:". Questo identificativo è una costante (il testo del messaggio può variare, ad es. può essere tradotto).

Quando il comando `:read` genera un altro errore, la stringa "E484:" non sarà contenuta nel messaggio di errore. Ragion per cui questa eccezione non sarà intercettata, e quindi riceveremo il relativo messaggio di errore.

Si potrebbe essere tentati di scrivere: >

```

:try
:  read ~/templates/pascal.tmpl
:catch
:  echo "Spiacente, il file modello Pascal non è disponibile."
:endtry

```

Così si intercettano tutti gli errori. In questo modo però non si vedono errori "utili" tipo: "E21: Non posso fare modifiche, 'modifiable' è inibito"

Un altro meccanismo utile è il comando `:finally`: >

```

:let tmp = tempname()

```

```

:try
:  exe ".,$write " . tmp
:  exe "!filter " . tmp
:  .,$delete
:  exe "$read " . tmp
:finally
:  call delete(tmp)
:endtry

```

Questo script Vim fa passare le linee dal cursore fino alla fine del file attraverso il comando "filter", che ha come argomento un nome di file.

Sia che il filtro funzioni, sia se qualche errore accade tra ":try" e ":finally" o l'utente cancella l'operazione di filtro premendo **CTRL-C**, la chiamata "call delete(tmp)" è eseguita sempre. Questo consente di essere sicuri di non lasciarsi dietro file temporanei di lavoro.

Maggiore informazione a riguardo della gestione eccezioni si trova nel manuale: |[exception-handling](#)|.

=====

41.10 Osservazioni varie

Diamo qui una serie di osservazioni relative agli script Vim. Sono spiegate anche altrove, ma sono qui riunite per convenienza.

Il carattere di fine linea dipende dal sistema operativo. Per Unix si usa un solo carattere **<NL>**. Per MS-DOS, Windows, OS/2 etc. si usa, **<CR><LF>**.

Questo è importante quando si usano mappature che finiscono con **<CR>**.

Si veda |[source_crnl](#)|.

SPAZI BIANCHI

Linee bianche possono essere inserite dappertutto, e vengono ignorate.

Spazi bianchi prima di un testo (spazi e/o TAB) sono sempre ignorati. Gli spazi bianchi fra parametri (ad es. tra il '**set**' e '**cptions**' nell'esempio qui sotto) sono ridotti ad un unico spazio, che serve da separatore.

Gli spazi bianchi dopo l'ultimo carattere (visibile) possono essere ignorati oppure no, a seconda della situazione. Si veda sotto.

Per un comando ":set" che usa il segno "=" (uguale), tipo: >

```
:set cptions    =aABceFst
```

lo spazio bianco subito PRIMA del segno "=" è ignorato. Ma non è consentito alcuno spazio bianco DOPO il segno "="!

Per inserire uno spazio bianco nel valore di una opzione, bisogna farlo precedere da un "\" (barra retroversa) come nell'esempio seguente: >

```
:set tags=il\ mio\ bel\ file
```

Lo stesso esempio scritto come >

```
:set tags=Il mio bel file
```

produrrà un errore, perché viene interpretato come: >

```

:set tags=il
:set mio
:set bel
:set file

```

COMMENTI

Il carattere " (il doppio apice) inizia un commento. A partire da questo carattere il resto della linea è considerato un commento ed è ignorato, tranne che nei comandi che non considerano i commenti, come mostrato negli esempi qui sotto. Un commento può cominciare in qualsiasi posizione della linea.

Bisogna fare attenzione ai commenti per alcuni comandi. Ad es.: >

```

:abbrev dev development      " abbreviazione
:map <F3> o#include          " inserisci include
:execute cmd                  " esegui
:!ls *.c                      " lista i file C

```

La abbreviazione '**dev**' sarà espansa come 'development' " abbreviazione'. La mappatura di **<F3>** sarà costituita dall'intera linea dopo 'o#' compreso '" inserisci include'. Il comando "execute" darà un errore. Il comando "!" passerà tutto quel che lo segue allo shell, che darà un errore perché manca un carattere '"' che chiuda il primo ''.

Non sono permessi commenti dopo i comandi ":map", ":abbreviate", ":execute" e "!" (ci sono alcuni altri comandi con la stessa limitazione). Per i comandi ":map", ":abbreviate" e ":execute" si può però scrivere: >

```

:abbrev dev development|" abbreviazione
:map <F3> o#include|" inserisci include
:execute cmd              |" esegui

```

Il carattere '|' separa un comando da quello successivo. Ed il comando successivo è solo un commento. Per l'ultimo comando bisogna fare due cose: una è |:execute| e l'altra è l'uso di '|': >

```

:exe '!ls *.c'           |" list C files

```

Si noti che non c'è alcuno spazio prima del '|' nella abbreviazione e nella mappatura. Per questi comandi, ogni carattere fino a fine linea o a '|' è compreso. Per questo motivo, non sempre è chiaro se sono presenti degli spazi dopo l'ultimo carattere visibile sulla linea; >

```

:map <F4> o#include

```

Per evidenziare questi problemi, potete attivare l'opzione '**list**' mentre state modificando degli script Vim.

In Unix si può commentare la prima linea di uno script in una maniera speciale in modo da rendere lo script vim eseguibile: >

```

#!/usr/bin/env vim -S
echo "questo è uno script Vim"
quit

```

Il comando "#" di per sé lista una linea col suo numero di linea. L'aggiunta del punto esclamativo lo rende un comando che non fa nulla, ed allo stesso tempo permette di inserire il comando shell che consente [a Vim] di eseguire [come comandi Vim] le successive righe del file. |:#!| |-S|

TRABOCCHETTI

Problemi anche maggiori sono presenti nell'esempio seguente: >

```

:map ,ab o#include
:unmap ,ab

```

Il comando "unmap" non funzionerà, perché tenta di togliere la mappatura di ",ab " (c'è uno spazio bianco dopo il "b" - NdT). Questa mappatura non esiste. Sarà quindi visualizzato un messaggio di errore, difficile da comprendere, perché lo spazio bianco finale in ":unmap ,ab " non è visibile.

Lo stesso capita quando si usa un commento dopo un comando '**unmap**': >

```

:unmap ,ab      " commento

```

Qui il commento verrà ignorato. Comunque, Vim tenterà di togliere la mappatura ',ab ', che non esiste. Riscrivetelo come: >

```

:unmap ,ab|     " commento

```

RITORNARE DOVE ERAVAMO

Talora volete fare una modifica, e ritornare al punto in cui si trovava il cursore prima della modifica. Anche un ripristino della posizione relativa dello schermo sarebbe gradita, con la stessa linea di prima visualizzata in

cima alla finestra.

L'esempio che segue copia la linea sotto il cursore, la mette sopra la prima linea del file, e torna al punto di partenza: >

```
map ,p ma"aYHmbgg"aP`bzt`a
```

Spiegazione: >

<pre>< ma "aY Hmb gg "aP `b zt `a</pre>	<pre>marcatatura "a" di posizione corrente copia linea corrente nel registro "a" marcatatura "b" su linea in cima alla pagina vai alla prima linea del file mettici sopra la linea nel registro "a" vai alla linea in cima alla pagina posiziona testo nella finestra come prima torna alla posizione in cui era il cursore</pre>
---	---

PER LA DISTRIBUZIONE

Per evitare che i vostri nomi di funzione entrino in conflitto con funzioni che altri hanno preparato, usate questo schema:

- Premettete una vostra stringa ad ogni nome funzione. Io uso spesso una abbreviazione. Ad es., "OW_" è usato per le funzioni Opzioni Window.
 - Mettete la definizione delle vostre funzioni in un unico file. Impostate una variabile globale per indicare che la funzione è stata caricata. Quando volete caricare di nuovo il file, cancellate prima le funzioni.
- Ad es: >

```
" Questa è la funzione XXX

if exists("XXX_caricata")
  delfun XXX_uno
  delfun XXX_due
endif

function XXX_uno(a)
  ... corpo della funzione ...
endfun

function XXX_due(b)
  ... corpo della funzione ...
endfun

let XXX_caricata = 1
```

```
=====
*41.11* Scrivere un plugin                                     *write-plugin*
```

Potete scrivere uno script Vim in modo tale che chi vuole lo possa usare.

Questo si chiama un "plugin". Gli utenti Vim possono copiare il vostro script Vim nella loro directory dei plugin e cominciare ad usare le sue funzioni. Si veda |[add-plugin](#)|.

Ci sono di fatto due tipi di plugin:

global plugin: Per ogni tipo di file.
 filetype plugin: Solo per file di un dato tipo.

In questa sezione è spiegato solo il "global plugin". La maggior parte delle cose dette vale anche per i "filetype plugin". Le informazioni relative ai "filetype plugin" sono nella sezione seguente |[write-filetype-plugin](#)|.

NOME

Innanzitutto dovete scegliere un nome per il vostro plugin. Lo scopo per cui è stato scritto il plugin dovrebbe essere riconoscibile dal nome che ha. E dovrebbe essere poco probabile che qualcun altro scriva un plugin con lo stesso nome, ma che faccia cose differenti. E per favore, limitate il nome a 8 caratteri, per evitare problemi sui vecchi sistemi Windows.

Uno script che corregge errori di battitura potrebbe essere chiamato

"typecorr.vim". Lo useremo qui come esempio.

Se il plugin deve poter essere usato da tutti, deve seguire alcune linee guida. Le vediamo una alla volta. L'esempio completo del plugin è riportato alla fine.

CORPO

Cominciamo dal corpo del plugin, le linee che fanno il lavoro vero: >

```
14      iabbrev li il
15      iabbrev altor altro
16      iabbrev voule vuole
17      iabbrev soprattutto
18          \ soprattutto
19      let s:contatore = 4
```

La lista vera dovrebbe essere molto più lunga, naturalmente.

La numerazione delle linee è stata aggiunta per uso didattico, non dovete metterla nel vostro file plugin!

INTESTAZIONE

Probabilmente aggiungerete nuove correzioni al vostro plugin, e vi ritroverete con diverse versioni sparse qua e là. E quando distribuirete il vostro file plugin, la gente vorrà sapere chi ha scritto questo meraviglioso plugin e a chi possono essere inviate considerazioni al riguardo. Ragion per cui, dovrete mettere una intestazione all'inizio del vostro plugin: >

```
1      " Plugin generale di Vim per correggere errori di battitura
2      " Ultima Modifica:      2000 Ott 15
3      " Manutentore:  Bram Moolenaar <Bram@vim.org>
```

A proposito di copyright e licenze: Poiché i plugin sono molto utili, e non ha gran senso restringerne la distribuzione, siete invitati a considerare l'idea di dichiarare il vostro plugin di pubblico dominio, oppure di usare la licenza Vim |[license](#)|. Una breve nota al riguardo vicino all'inizio del plugin dovrebbe essere sufficiente. Ad es.: >

```
4      " Licenza:      Questo file è di pubblico dominio.
```

CONTINUAZIONE LINEA, AL NETTO DA EFFETTI SECONDARI *use-cpo-save*

Nella linea 18 qui sopra, il meccanismo usato per la continuazione è descritto in |[line-continuation](#)|. Utenti che hanno attivato l'opzione '[compatible](#)' avranno dei problemi, e otterranno un messaggio di errore. Non possiamo semplicemente cambiare l'opzione '[compatible](#)', perché questo avrebbe molti altri effetti oltre a quello desiderato. Per evitare di toccare '[compatible](#)' imposteremo invece l'opzione '[cpoptions](#)' al suo valore predefinito in Vim e la riporteremo poi al valore che aveva. Questo ci consente di usare le linee di continuazione e nello stesso tempo rende lo script utilizzabile da un numero molto maggiore di utenti. Si fa in questo modo: >

```
11      let s:salva_cpo = &cpo
12      set cpo&vim
..
42      let &cpo = s:salva_cpo
```

Prima si memorizza il valore precedente di '[cpoptions](#)' nella variabile s:salva_cpo. Alla fine del plugin questo valore viene ripristinato.

Notate l'uso di una variabile locale allo script |[s:var](#)|. Una variabile globale con lo stesso nome potrebbe essere già in uso da qualche altra parte. Usate sempre delle variabili locali allo script per azioni limitate solo all'interno dello script.

NON CARICARE

È possibile che un utente non voglia sempre caricare questo plugin. Oppure l'amministratore del sistema l'ha messo nella directory dei plugin di uso comune a tutti gli utenti, ma un utente vuole usare il suo proprio plugin. Quindi all'utente deve essere offerta la possibilità di evitare l'uso di questo specifico plugin. Questo si ottiene così: >

```
6   if exists("loaded_typecorr")
7     finish
8   endif
9   let loaded_typecorr = 1
```

Si evita così anche che, nel caso lo script sia caricato una seconda volta, si riceva un messaggio di errore perché la funzione erà già definita, e possano presentarsi dei problemi per degli autocomandi aggiunti più di una volta.

MAPPATURE

Per rendere il plugin più interessante, introduciamo una mappatura che aggiunga una correzione per la parola che si trova sotto il cursore. Potremmo scegliere una combinazione di tasti per questa mappatura, ma l'utente magari la usa già per fare qualcosa d'altro. Per permettere all'utente di definire la sequenza di tasti che vuole usare per una mappatura all'interno di un plugin, si può usare l'elemento **<Leader>**: >

```
22      map <unique> <Leader>a <Plug>TypecorrAdd
```

Il comando "**<Plug>**TypecorrAdd" rende questo possibile, come vedremo dopo.

L'utente può dare alla variabile "mapleader" la combinazione di tasti con cui vuole che cominci questa mappatura. Se l'utente ha impostato: >

```
let mapleader = "_"
```

la mappatura definirà "_a". Se l'utente non ha fatto nulla, il valore predefinito (che è "\", barra retroversa) sarà utilizzato. In quel caso la mappatura definita sarà "\a".

Note Poiché **<unique>** è usato, sarà inviato un messaggio di errore nel caso la stessa mappatura sia già stata definita. |:map-<unique>|

E se l'utente volesse definire una sua sequenza di tasti? Glielo si può permettere con questo meccanismo: >

```
21   if !hasmapto('<Plug>TypecorrAdd')
22     map <unique> <Leader>a <Plug>TypecorrAdd
23   endif
```

Qui controlliamo se una mappatura per "**<Plug>**TypecorrAdd" esiste già, e definiamo la mappatura con "**<Leader>**a" solo se non esiste. L'utente ha quindi la possibilità di mettere nel suo file vimrc: >

```
map ,c <Plug>TypecorrAdd
```

E quindi la sequenza di tasti mappata sarà ",c" invece che "_a" o "\a".

PARTI

Se uno script si allunga, potrebbe essere desiderabile dividere il lavoro in parti. Potete usare sia delle funzioni che delle mappature per questo. Ma non volete che queste funzioni e mappature interferiscano con altre, contenute in altri script. Per esempio, potreste definire una funzione Add(), ma un altro script potrebbe tentare di definire la stessa funzione. Per evitare questo, definiamo la funzione come locale, esistente solo all'interno del nostro script, premettendo "s:" al suo nome.

Definiamo una funzione che aggiunge una nuova correzione di errore di battitura: >

```
30   function s:Add(errato, corretto)
31     let buono = input("Battere correzione per " . a:errato . ": ")
```

```

32     exe ":iabbrev " . a:errato . " " . buono
..
36     endfunction

```

Ora possiamo richiamare la funzione s:Add() dall'interno dello script. Se un altro script definisce ancora s:Add(), esisterà solo dentro quello script, e può essere chiamata solo dallo script in cui è stata definita. Ci può anche essere una funzione globale. Ci può anche essere una funzione globale Add() (senza "s:"), che è distinta dalle altre due.

<SID> si può usare senza mappature. Genera un ID dello script, che identifica lo script in cui si trova. Nel nostro plugin di correzione errori di battitura lo usiamo in questo modo: >

```

24     noremap <unique> <script> <Plug>TypecorrAdd <SID>Add
..
28     noremap <SID>Add :call <SID>Add(expand("<cword>"), 1)<CR>

```

Ovvero, quando un utente batte "\a", viene richiamata questa sequenza: >

```
\a -> <Plug>TypecorrAdd -> <SID>Add -> :call <SID>Add()
```

Se un altro script mappasse **<SID>Add**, un altro script ID sarebbe generato, e quindi verrebbe definito un'altra mappatura.

Note Usiamo qui **<SID>Add()** invece che s:Add(). Questo perché la mappatura viene immessa dall'utente, ossia dall'esterno dello script. Il **<SID>** si traduce nell'ID dello script, in modo che Vim sappia in che script cercare la funzione Add().

La cosa è un po' complicata, ma serve per far sì che il plugin funzioni assieme ad altri plugin. La regola fondamentale è che voi usiate **<SID>Add()** nelle mappature e s:Add() da altre parti (nello script stesso, in autocomandi o in comandi utente).

Possiamo anche rendere disponibile la mappatura in un menù: >

```
26     noremenu <script> Plugin.Aggiungi\ Correzione <SID>Add
```

Il menù "Plugin" è raccomandato per aggiungere elementi di menù a plugin. In questo caso c'è solo un elemento. Se si aggiungono più elementi, la creazione di un sottomenù è auspicabile. Ad esempio, "Plugin.CVS" potrebbe essere usato per un plugin che offre le operazioni CVS "Plugin.CVS.checkin", "Plugin.CVS.checkout", etc. [CVS è un software per gestire sorgenti, "checkin" inserisce un sorgente, "checkout" lo preleva per aggiornarlo - NdT]

Note Nella linea 28 ":noremap" è usato per evitare che qualsiasi altra mappatura provochi problemi. Qualcuno potrebbe aver ridefinito ":call", ad esempio. Nella linea 24 usiamo anche ":noremap", ma vogliamo che "**<SID>Add**" sia ri-mappato. Per questo "**<script>**" è usato qui. Questo permette mappature che valgono solo all'interno dello script. |:map-<script>| La stessa cosa è fatta nella linea 26 per ":noremenu". |:menu-<script>|

<SID> E **<Plug>** ***using-<Plug>***

Sia **<SID>** che **<Plug>** sono usati per evitare che mappature di combinazioni di tasti entrino in conflitto con altri che devono essere usati solo da altre mappature. Note Questa è la differenza fra usare **<SID>** e **<Plug>**:

<Plug> è visibile all'esterno dello script. È usato per mappature per cui l'utente voglia utilizzare una particolare sequenza di tasti. **<Plug>** è un codice speciale che un tasto della tastiera non è in grado di produrre. Per rendere molto difficile che altri plugin usino la stessa sequenza di caratteri, usate questa struttura:
<Plug> nome_script nome_mappatura
 Nel nostro esempio il nome_script è "Typecorr" e il nome_mappatura è "Add".
 Il risultato è "**<Plug>**TypecorrAdd". Solo il primo carattere del nome_script e del nome_mappatura deve essere maiuscolo, in modo da poter capire dove comincia il nome_mappatura.

<SID> è l'ID dello script, un identificatore unico a quello script. Internamente Vim traduce **<SID>** in "**<SNR>123_**", dove "123" può essere un numero qualsiasi. Per cui una funzione "**<SID>Add()**" avrà come nome "**<SNR>11_Add()**" in uno script, e "**<SNR>22_Add()**" in un altro. Potete vederlo se usate il comando ":function" per ottenere una lista di funzioni. La traduzione di **<SID>** nelle mappature è esattamente la stessa, ed è questo il modo con cui potete invocare una funzione locale ad uno script da una mappatura.

COMANDI UTENTE

Aggiungiamo ora un comando utente per aggiungere una correzione: >

```
38     if !exists("Correzione")
39         command -nargs=1 Correzione :call s:Add(<q-args>, 0)
40     endif
```

Il comando utente viene definito solo se non esiste già un comando con lo stesso nome. Altrimenti otterremmo un messaggio di errore. Sostituire il comando utente già esistente col nostro, scrivendo ":command!" non è una buona idea, in quanto l'utente si domanderebbe perché il comando che LUI ha definito non funziona. |:command|

VARIABILI NEGLI SCRIPT

Quando una variabile ha un nome che comincia con "s:" è una variabile dello script. Si può solo usare all'interno di uno script, Non esiste all'esterno dello script. Questo consente di evitare problemi quando si utilizza lo stesso nome di variabile in script differenti. La variabile sarà disponibile per tutto il tempo in cui si usa Vim. E la stessa variabile verrà riutilizzata se si ricarica lo stesso script una seconda volta. |s:var|

Il bello è che queste variabili possono anche essere usate in funzioni, autocomandi e comandi utente che sono definiti nello script. Nel nostro esempio di script possiamo aggiungere alcune linee per contare il numero di correzioni: >

```
19     let s:contatore = 4
..
30     function s:Add(from, correct)
..
34         let s:contatore = s:contatore + 1
35         echo s:contatore . " correzioni finora"
36     endfunction
```

Dapprima s:contatore è inizializzato a 4 nello script stesso. Quando, più tardi, la funzione s:Add() viene invocata, incrementa s:count. Non importa da DOVE la funzione è stata chiamata, poiché, essendo stata definita nello script, userà solo variabili interne a questo script.

IL RISULTATO

Questo è lo script completo risultante: >

```
1     " Plugin generale di Vim per correggere errori di battitura
2     " Ultima Modifica:      2000 Ott 15
3     " Manutentore:  Bram Moolenaar <Bram@vim.org>
4     " Licenza:      Questo file è di pubblico dominio.
5
6     if exists("loaded_typecorr")
7         finish
8     endif
9     let loaded_typecorr = 1
10
11     let s:salva_cpo = &cpo
12     set cpo&vim
13
14     iabbrev li il
15     iabbrev altor altro
16     iabbrev vuole vuole
```

```

17     iabbrev soprattutto
18         \ soprattutto
19     let s:contatore = 4
20
21     if !hasmapto('<Plug>TypecorrAdd')
22         map <unique> <Leader>a <Plug>TypecorrAdd
23     endif
24     noremap <unique> <script> <Plug>TypecorrAdd <SID>Add
25
26     noremenu <script> Plugin.Aggiungi\ Correzione <SID>Add
27
28     noremap <SID>Add :call <SID>Add(expand("<cword>"), 1)<CR>
29
30     function s:Add(errato, corretto)
31         let buono = input("Battere correzione per " . a:errato . ": ")
32         exe ":iabbrev " . a:errato . " " . buono
33         if a:corretto | exe "normal viws\<C-R>\\" \b\e" | endif
34         let s:contatore = s:contatore + 1
35         echo s:contatore . " correzioni finora"
36     endfunction
37
38     if !exists(":Correzione")
39         command -nargs=1 Correzione :call s:Add(<q-args>, 0)
40     endif
41
42     let &cpo = s:salva_cpo

```

La linea 33 non è stata ancora spiegata. Applica la nuova correzione alla parola sotto il cursore. Il comando |:normal| è usato per fare uso della nuova abbreviazione. Note mappature e abbreviazioni sono espresse qui, anche se la funzione è stata invocata da una mappatura definita come ":noremap".

L'uso di "unix" come opzione '**fileformat**' è raccomandato. Gli script Vim in questo modo funzioneranno anche in tutti gli altri ambienti software. Script che abbiano '**fileformat**' impostato a "dos" non funzionano sotto Unix.

Si veda anche |:source_crnl|. Per essere sicuri di usarlo, scrivete, prima di scrivere il file che contiene lo script: >

```
:set fileformat=unix
```

DOCUMENTAZIONE

```
*write-local-help*
```

E' una buona idea preparare anche un po' di documentazione per il vostro plugin. Questo è ancora più necessario quando il suo comportamento può essere personalizzato dall'utente. Si veda |**add-local-help**| per come aggiungere la vostra documentazione a quella di Vim.

Qui vedete un semplice esempio di file di aiuto, di nome "typecorr.txt": >

```

1     *typecorr.txt*  Plugin per correzione di errori di battitura.
2
3     Se vi capita di fare errori di battitura, questo plugin li correggerà
4     automaticamente.
5
6     Al momento le correzioni sono poche. Potete aggiungerne, se volete.
7
8     Mappatura:
9     <Leader>a      o    <Plug>TypecorrAdd
10         Aggiunge una correzione per la parola sotto il cursore.
11
12     Comandi:
13     :Correzione {parola}
14         Aggiunge una correzione per {parola}.
15
16                                     *typecorr-settings*
17     Questo plugin non ha alcuna impostazione particolare.

```

La prima linea è l'unica per la quale il formato è importante. Sarà questa ad essere estratta dal file di aiuto per essere inserita nella sezione "LOCAL ADDITIONS" del file help.txt |**local-additions**|. Il primo "*" deve essere nella prima colonna della prima linea di testo. Dopo aver aggiunto il vostro file di aiuto date il comando ":help" per controllare che le linee in

questione siano correttamente allineate.

Potete aggiungere altri tag (racchiusi fra **) nel vostro file di aiuto. Ma state attenti ad evitare nomi di tag di help già esistenti. Potreste inserire il nome del vostro plugin nei loro nome, come "typecorr-settings" nel nostro esempio.

L'uso di puntatori ad altre parti del file di help (racchiusi fra ||) è auspicabile. Questo facilita l'utente nel reperimento dell'aiuto a loro associato.

DETERMINAZIONE DEL TIPO FILE

plugin-filetype

Se il tipo del vostro file non è già determinato da Vim, dovreste creare un piccolo script che lo determini in un file a parte. Di solito lo script consiste in un autocomando che imposta il tipo file quando il nome del file corrisponde a un dato modello.

Ad es.: >

```
au BufNewFile,BufRead *.foo
```

```
set filetype=foofoo
```

Scrivete il file contenente quest'unica linea come "ftdetect/foofoo.vim" nella prima directory che compare in '**runtimepath**'. In Unix questa sarebbe "~/vim/ftdetect/foofoo.vim". La convenzione è di usare il nome del tipo file come nome dello script.

I controlli possono essere molto più sofisticati, se volete, come ad es. una ispezione del contenuto del file per riconoscere il linguaggio. SI veda anche **|new-filetype|**.

SOMMARIO

plugin-special

Lista delle particolarità che contraddistinguono un plugin:

s:name variabili interne allo script.

<SID> Script-ID, usato per mappature e funzioni interne allo script.

hasmapto() Funzione per accertare se l'utente ha già definito mappature per funzionalità contenute nello script.

<Leader> Valore di "mapleader", che l'utente definisce come sequenza di tasti con cui iniziano le mappature del plugin.

:map **<unique>** Emette un messaggio se una mappatura esiste già.

:noremap **<script>** Usa solo mappature interne allo script, non mappature valide globalmente.

exists(":Cmd") Controlla se un comando utente esiste già.

=====

ftplugin

41.12 Scrivere un plugin per un tipo_file

write-filetype-plugin

A plugin per un tipo_file è simile a un plugin globale, ma imposta opzioni e definisce mappature solo per il buffer corrente. Si veda **|add-filetype-plugin|** per informazioni sull'uso di questo tipo di plugin.

Leggete prima la sezione precedente sui plugin globali **|41.11|**. Tutto quel che si dice lì vale anche per i plugin per un tipo_file. Ci sono alcune regole in più, che sono spiegate qui di seguito. La caratteristica fondamentale è che un plugin per un tipo_file dovrebbe avere effetto solo sul buffer corrente.

DISABILITAZIONE

Se state scrivendo un plugin per un tipo_file che potrà essere usato da molta

gente, dovete concedere loro la possibilità di evitare di usarlo. Mettete questo in testa al plugin: >

```
" Usa solo se non esiste già un altro plugin attivo
if exists("b:did_ftplugin")
    finish
endif
let b:did_ftplugin = 1
```

Occorre usare questa tecnica per evitare che lo stesso plugin sia eseguito due volte per lo stesso buffer (succede quando si usa un comando ":edit" senza fornirgli alcun argomento).

A questo punto gli utenti possono disabilitare completamente il plugin predefinito creando un plugin per un tipo_file composto solo da questa linea:

```
let b:did_ftplugin = 1
```

Per ottenere questo, la directory che contiene il vostro plugin per un tipo_file deve venire prima della directory \$VIMRUNTIME nella list di directory contenuta nella opzione 'runtimepath'!

Se volete usare il plugin predefinito, ma cambiare una delle impostazioni, potete inserire l'impostazione da cambiare in uno script: >

```
setlocal textwidth=70
```

Poi lo scrivete nella directory "after" ["dopo"], così che venga eseguito DOPO il plugin di tipo_file predefinito (in questo caso "vim.vim").

Si veda |after-directory|. In Unix questa directory è: "~/vim/after/ftplugin/vim.vim". Note Il plugin predefinito avrà impostato "b:did_ftplugin", ma la cosa viene qui ignorata.

OPZIONI

Per far sì che il plugin per un tipo_file agisca solo sul buffer corrente usate il comando:

```
:setlocal
```

per impostare le opzioni. Ed impostate solo le opzioni che agiscono solo sul buffer corrente (si veda l'aiuto relativo a ogni singola opzione per controllare quali lo facciano). Quando si usa |:setlocal| per opzioni globali o per opzioni che riguardano una particolare finestra, il valore potrebbe cambiare in numerosi buffer, e non è questo l'obiettivo che si dovrebbe avere in un plugin per un tipo_file.

Quando una opzione ha un valore che è una lista di indicatori o elementi, potete utilizzare la sintassi "+=" e "-=" per non modificare i valori già impostati. Tenete presente che l'utente possa aver già cambiato il valore di una opzione. Ripristinare il valore predefinito e poi cambiarlo come serve è spesso una buona idea. Ad es.: >

```
:setlocal formatoptions& formatoptions+=ro
```

MAPPATURE

Per far sì che le mappature siano applicate solo nel buffer corrente usate il comando: >

```
:map <buffer>
```

Questo va combinato con la mappatura in due passi spiegata più sopra.

Ecco un esempio di come si definisce una funzionalità in un plugin per un tipo_file: >

```
if !hasmapto('<Plug>JavaImport')
    map <buffer> <unique> <LocalLeader>i <Plug>JavaImport
endif
noremap <buffer> <unique> <Plug>JavaImport oimport ""<Left><Esc>
```

|**hasmapto()**| è usato per controllare se l'utente ha già definito una mappatura per **<Plug>JavaImport**. In caso negativo, il plugin per un tipo_file definisce la mappatura predefinita. Questa inizia con |**<LocalLeader>**|, che permette all'utente di scegliere il tasto (o i tasti) con cui vuole che inizino le mappature dei plugin per un tipo_file. Il valore predefinito è la barra rovesciata ("****").

"**<unique>**" è usato per mandare un messaggio di errore se la mappatura esiste già o entra in conflitto con una mappatura esistente.

|**:noremap**| è usato per evitare che ogni altra mappatura che l'utente abbia definito possa interferire. Potreste voler usare "**:noremap <script>**" per far sì che si possano ri-mappare mappature definite in questo script, che comincino con **<SID>**.

L'utente deve avere la possibilità di disabilitare una mappatura in un plugin per un tipo_file, senza disabilitarlo completamente. Ecco un esempio di come si può fare questo per il tipo_file "mail" (messaggi di posta elettronica): >

```
" Aggiungi mappature, a meno che l'utente preferisca evitarlo.
if !exists("no_plugin_maps") && !exists("no_mail_maps")
  " Citate un testo inserendo "> "
  if !hasmapto('<Plug>MailQuote')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote
    nmap <buffer> <LocalLeader>q <Plug>MailQuote
  endif
  vnoremap <buffer> <Plug>MailQuote :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote :.,$s/^/> /<CR>
endif
```

Due variabili globali sono usate:

no_plugin_maps	disabilita mappature per ogni plugin di un tipo_file
no_mail_maps	disabilita mappature per un dato tipo_file

COMANDI UTENTE

Per aggiungere un comando utente per un dato tipo_file, in modo che possa solo essere usato in un buffer, usate l'argomento "**-buffer**" di |**:command**|.

Ad es.: >

```
:command -buffer Make make %:r.s
```

VARIABILI

Un plugin per un tipo_file sarà utilizzato in ogni buffer contenente un file di quel tipo. La variabile interna di script |**s:var**| sarà la stessa per tutti i buffer in cui il plugin viene utilizzato. Usate variabili interne al buffer |**b:var**| se volete che una variabile sia usata solo all'interno di un unico buffer.

FUNZIONI

Quando si definisce una funzione, questa va definita soltanto una volta. Ma il plugin di tipo_file verrà invocato ogni volta che verrà aperto un file con quel dato tipo file. La tecnica seguente fa sì che la funzione venga definita una volta sola. >

```
:if !exists("*s:Func")
:  function s:Func(arg)
:    ...
:  endfunction
:endif
```

<

ANNULLARE

undo_ftplugin

Quando l'utente imposta "**:setfiletype xyz**" l'effetto del tipo_file "precedente" dovrebbe essere annullato. Impostate la variabile **b:undo_ftplugin** con i comandi che annulleranno le impostazioni di un vostro plugin per un tipo_file. Ad es.: >

```
let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<
```

```
\ . "| unlet b:match_ignorecase b:match_words b:match_skip"
```

L'uso di ":setlocal" con "<" subito dopo il nome dell'opzione reimposta per quell'opzione il suo valore globale. Nella maggior parte dei casi è questa la maniera migliore di ripristinare i valori "precedenti" delle opzioni.

Occorre togliere la indicazione "C" da 'coptions' per consentire la continuazione della linea, come detto prima |use-cpo-save|.

NOME FILE

Il tipo_file deve far parte del nome del plugin |ftplugin-name|. Usate una di queste tre forme:

```
.../ftplugin/roba.vim
.../ftplugin/roba_mia.vim
.../ftplugin/roba/tua.vim
```

"roba" è il tipo_file, "mia" e "tua" sono nomi a piacere.

SOMMARIO

ftplugin-special

Lista delle particolarità che contraddistinguono un plugin di tipo_file:

<LocalLeader>	Valore di "maplocalleader", che l'utente definisce come la combinazione di tasti con cui iniziano le mappature per un plugin di tipo_file.
:map <buffer>	Definisce una mappatura che vale solo all'interno del buffer.
:noremap <script>	Ri-mappa solo mappature definite in questo script e che cominciano con <SID> .
:setlocal	Imposta una opzione solo per il buffer corrente.
:command -buffer	Definisce un comando utente valido solo in questo buffer.
exists("*s:Func")	Controlla se una funzione è stata già definita.

Si veda anche |plugin-special|, che descrive le proprietà specifiche dei plugin.

=====

41.13 Scrivere un plugin per un compilatore

write-compiler-plugin

Un plugin per un compilatore imposta le opzioni da usare con un dato compilatore. L'utente può invocarlo con il comando |:compiler|. L'uso principale che se ne fa è per impostare le opzioni 'errorformat' e 'makeprg'.

Un esempio è la maniera più semplice di introdurlo. Questo comando modificherà tutti i plugin predefiniti per i compilatori: >

```
:next $VIMRUNTIME/compiler/*.vim
```

Usate |:next| per passare al file di plugin successivo.

Ci sono due caratteristiche notevoli di questi file. La prima è un meccanismo che permette a un utente di fare modifiche o aggiunte al file di plugin predefinito. I file predefiniti cominciano con: >

```
:if exists("current_compiler")
:  finish
:endif
:let current_compiler = "mine"
```

Quando scrivete un file per un compilatore e lo mettete nella vostra propria directory di file di esecuzione ("runtime directory") (ad es., ~/.vim/compiler in Unix), voi potete impostare la variabile "current_compiler"

in modo che il file predefinito non esegua (in seguito) le impostazioni che contiene.

:CompilerSet

Il secondo metodo consiste nell'usare ":set" per ":compiler!" e ":setlocal" per ":compiler". Vim fornisce il comando utente ":CompilerSet" a questo scopo. D'altra parte, precedenti versioni di Vim non ne sono dotate, e per questo motivo un vostro plugin dovrebbe definirla solo dopo aver controllato. Ecco un esempio: >

```
if exists(":CompilerSet") != 2
  command -nargs=* CompilerSet setlocal <args>
endif
CompilerSet errorformat&          " usa 'errorformat' predefinito
CompilerSet makeprg=nmake
```

<

Quando scrivete un plugin per un compilatore da distribuire con Vim o da mettere in una directory di esecuzione ("runtime directory") a livello di sistema, usate il meccanismo delineato più sopra. Se "current_compiler" era già stato impostato da un plugin dell'utente, questo verrà ritenuto valido e non sostituito in alcun modo.

Quando scrivete un plugin di compilatore per modificare impostazioni contenute nel plugin predefinito, non controllate la variabile "current_compiler". Un plugin di questo tipo dovrebbe venire eseguito per ultimo, e quindi dovrebbe trovarsi in una directory in fondo alla lista di directory specificata in 'runtimepath'. In Unix il suo nome potrebbe essere ~/.vim/after/compiler.

=====

41.14 Scrivere un plugin veloce a caricarsi *write-plugin-quickload*

Un plugin più crescere e divenire piuttosto lungo. Il ritardo alla partenza può diventare notevole, e magari il plugin non vi serve quasi mai. E' a questo punto che può venir buono un plugin veloce a caricarsi.

L'idea di fondo è che il plugin è caricato due volte. La prima volta per definire i comandi utenti e le mappature che offrono la funzionalità. la seconda volta per definire le funzioni che implementano la funzionalità.

Può sembrare sorprendente che "caricamento veloce" voglia dire caricare il plugin due volte. Quel che si vuol dire è che si carica velocemente all'inizio, rimandando la parte più sostanziosa dello script alla seconda volta, il che vuol dire solo quando ne avete veramente bisogno. Quando la funzionalità viene utilizzata spesso, il risultato è in effetti più lento!

Note Da Vim 7 c'è una altra possibilità: usate la funzionalità |autoload| |41.15|.

L'esempio che segue mostra il procedimento: >

```
" Vim global plugin for demonstrating quick loading
" Last Change: 2005 Feb 25
" Maintainer:  Bram Moolenaar <Bram@vim.org>
" License:     This file is placed in the public domain.

if !exists("s:did_load")
  command -nargs=* BNRead  call BufNetRead(<f-args>)
  map <F19> :call BufNetWrite('something')<CR>

  let s:did_load = 1
  exe 'au FuncUndefined BufNet* source ' . expand('<sfile>')
  finish
endif

function BufNetRead(...)
  echo 'BufNetRead(' . string(a:000) . ')'
  " read functionality here
endfunction

function BufNetWrite(...)
  echo 'BufNetWrite(' . string(a:000) . ')'
  " write functionality here
endfunction
```

Quando lo script è caricato la prima volta "s:did_load" non è impostato. I comandi fra l' "if" e l' "endif" vengono eseguiti. Questi terminano con un comando |:finish|, e quindi il resto dello script non viene eseguito.

La seconda volta che lo script viene caricato "s:did_load" esiste e i comandi dopo l' "endif" vengono eseguiti. Qui sono definite le funzioni (che possono essere "pesanti") BufNetRead() e BufNetWrite().

Se copiate questo script nella vostra directory dei plugin Vim lo eseguirà alla partenza. Questa è la sequenza degli eventi che si producono:

1. Il comando "BNRead" è definito e il tasto <F19> è mappato quando lo script viene letto alla partenza. Un autocomando |FuncUndefined| è definito. Il comando ":finish" fa concludere velocemente lo script.
2. L'utente immette il comando BNRead o preme il tasto <F19>. La funzione BufNetRead() o BufNetWrite() sarà invocata.
3. Vim non trova la funzione e chiama l'autocomando per l'evento |FuncUndefined|. Poiché l'espressione "BufNet*" corrisponde alla funzione invocata, il comando "source fname" sarà eseguito. "fname" sarà uguale al nome dello script (in qualsiasi posto si trovi), perché proviene dalla espansione di "<sfile>" (vedere |expand()|).
4. Lo script è caricato ancora, la variabile "s:did_load" esiste e le sue funzioni vengono a questo punto definite.

Si noti che le funzioni che sono caricate in seguito sono quelle che corrispondono all'espressione dell'autocomando |FuncUndefined|. Dovete accertarvi che nessun altro plugin definisca funzioni che corrispondono alla stessa espressione regolare.

=====

41.15 Scrivere librerie di script *write-library-script*

Alcune funzionalità serviranno in parecchi posti. Quando questo supera il limite delle "poche linee", vorrete metterle in un unico script, e usarle da parecchi altri script. Chiameremo quell'unico script uno script di libreria.

Caricare uno script di libreria è possibile, ma dovete evitare di caricarlo ancora, una volta che sia stato caricato. Questo si può fare con la funzione |exists()|. Esempio: >

```
if !exists('*MyLibFunction')
    runtime library/mylibscript.vim
endif
call MyLibFunction(arg)
```

Occorre qui sapere che MyLibFunction() sia definita in uno script "library/mylibscript.vim" in una delle directory in 'runtimepath'.

Per semplificare la faccenda, Vim fornisce il meccanismo di autoload. Così il nostro esempio diventerebbe: >

```
call mylib#myfunction(arg)
```

Molto più semplice, vero? Vim riconoscerà il nome di funzione e quando non è definito cercherà lo script "autoload/mylib.vim" in 'runtimepath'. Quello script deve contenere la definizione della funzione "mylib#myfunction()".

Potete mettere molte altre funzioni nello script mylib.vim, potete organizzare le vostre funzioni in uno script di libreria. Ma dovete usare nomi di funzione in cui la parte prima del simbolo "#" indichi il nome dello script. Altrimenti Vim non saprebbe quale script caricare.

Se, presi dall'entusiasmo, scrivete un mucchio di script di libreria, potreste voler usare delle sotto-directory. Esempi: >

```
call netlib#ftp#read('somefile')
```

Sotto Unix lo script di libreria usato qui potrebbe essere:

```
~/.vim/autoload/netlib/ftp.vim
```

Dove le funzioni sono definite così: >

```
function netlib#ftp#read(fname)
    " Read the file fname through ftp
endfunction
```

Si noti che il nome con cui è definita la funzione è esattamente lo stesso che il nome usato per richiamare la funzione. E la parte prima dell'ultimo "#" riflette esattamente il nome della sotto-directory e dello script.

Potete usare lo stesso meccanismo per delle variabili: >

```
let weekdays = dutch#weekdays
```

Questo farà caricare lo script "autoload/dutch.vim", che dovrebbe contenere qualcosa del tipo di: >

```
let dutch#weekdays = ['zondag', 'maandag', 'dinsdag', 'woensdag',
    \ 'donderdag', 'vrijdag', 'zaterdag']
```

Per ulteriori letture vedere: |autoload|.

```
=====
*41.16* Distribuzione di script Vim
```

```
*distribute-script*
```

Gli utenti Vim possono cercare script sul sito Web vim: <http://www.vim.org>. Se avete fatto qualcosa che potrebbe essere utile ad altri, condividetelo!

Gli script Vim si possono usare su ogni sistema. I comandi tar e zip potrebbero non essere disponibili in alcuni sistemi. Se volete riunire più file e/o comprimerli, si raccomanda di usare il comando di utilità "zip".

Per la massima portabilità, usate Vim stesso per impaccare i vostri script. E' possibile farlo con il programma di utilità vimball. Si veda |vimball|.

E' una buona cosa se aggiungete una linea che permetta un aggiornamento automatico. Si veda |glvs-plugins|.

```
=====
Capitolo seguente: |usr_42.txt| Aggiungere nuovi menù
```

Copyright: vedere |manual-copyright| vim:tw=78:ts=8:ft=help:norl:

Per segnalazioni scrivere a vimdoc.it at gmail dot com
oppure ad Antonio Colombo azc100 at gmail dot com