usr_44.txt Per Vim version 7.3. Ultima modifica: 2008 Dec 28

VIM USER MANUAL - di Bram Moolenaar
Traduzione di questo capitolo: Giuliano Bordonaro

Evidenziazione della vostra sintassi

Vim nasce con la proprietà di evidenziare circa duecento tipi di file diversi.
Se il file che state modificando non fosse incluso, leggete questo capitolo
per scoprire come ottenere che anche questo tipo di file venga evidenziato.
Guardate anche |:syn-define| nel manuale di riferimento.

  Capitolo seguente: |usr_45.txt| Selezionate la vostra lingua
Capitolo precedente: |usr_43.txt| Utilizzo dei tipi di file
             Indice: |usr_toc.txt|

==========

44.1 Fondamentali comandi della sintassi

Usando un file di sintassi esistente per iniziare risparmierete un sacco di
tempo. Provateci trovando un file di sintassi entro $VIMRUNTIME/syntax per un
linguaggio che sia simile. Questi file vi mostreranno l'aspetto normale di
un file di sintassi. Per capirlo dovete leggere quanto segue.

Cominciamo con gli argomenti fondamentali. Prima incominciamo definendo una
nuova sintassi, dovete togliere qualsiasi vecchia definizione: >

```
:syntax clear
```

Ciò non è necessario con il file di sintassi finale, ma molto utile quando
state sperimentando.

In questo capitolo ci sono ulteriori semplificazioni. Se scrivete un file di
sintassi che venga usato da altri, leggete comunque tutto sino alla fine per
scoprire i dettagli.

ELENCARE GLI ELEMENTI DEFINITI

Per vedere quali elementi di sintassi siano attualmente definiti usate questo
comando: >

```
:syntax
```

Potete usarlo per vedere quali elementi di sintassi siano stati sinora
definiti. Risulta utile per fare esperimenti con un nuovo file di sintassi.
Mostra anche i colori usati per ciascun elemento, il che aiuta a comprendere
di cosa si tratta.
   Per elencare gli elementi di un gruppo di sintassi specifico usate: >

```
:syntax list {group-name}
```

Ciò può anche venire impiegato per elencare dei cluster (spiegati in |44.8|).
Soltanto aggiungendo un @ nel nome.

CONFRONTO DEI CARATTERI

Alcuni linguaggi non sono sensibili al carattere come il Pascal. Altri,

come il C, sono sensibili al carattere. Dovete specificare quale sia il tipo con i seguenti comandi: >

```
:syntax case match
:syntax case ignore
```

L'argomento "match" significa che Vim confronterà il carattere degli elementi di sintassi. Perciò "int" è diverso da "Int" ed "INT". Se l'argomento "ignore" viene impiegato tutti gli esempi seguenti saranno equivalenti: "Procedure", "PROCEDURE" e "procedure".
I comandi ":syntax case" possono apparire dovunque in un file di sintassi e riferirsi alle definizioni di sintassi che seguono. Il più delle volte avete solo un comando ":syntax case" nel vostro file di sintassi; se lavoraste con un tipo insolito di linguaggio contenente sia elementi sensibili al carattere che altri che non lo siano, comunque, avete la facoltà di disseminare il comando ":syntax case" per tutto il file.

==========

44.2 Parole chiave

I più diffusi elementi fondamentali della sintassi sono le parole chiave. Per definire una parola chiave usate la seguente forma: >

```
:syntax keyword {gruppo} {keyword} ...
```

Il **{gruppo}** è il nome del gruppo di sintassi. Con il comando ":highlight" potete assegnare un colore ad un **{gruppo}**. L'argomento **{keyword}** è una vera parola chiave. Ecco qualche esempio: >

```
:syntax keyword xType int long char
:syntax keyword xStatement if then else endif
```

Questo esempio usa i nomi di gruppo "xType" e "xStatement". Per convenzione ogni nome di gruppo è preceduto dal tipo di file per il linguaggio che viene definito. Questo esempio definisce la sintassi per il linguaggio x (linguaggio di esempio senza un nome particolarmente significativo). In un file di sintassi per script "csh" potrà essere usato il nome "cshType". Così il prefisso è uguale al valore di **'filetype'**.
Questi comandi fanno si che le parole "int", "long" e "char" vengano evidenziate in un modo e le parole "if", "then", "else" ed "endif" lo siano diversamente. Ora dovete collegare i nomi del gruppo x ai nomi standard di Vim. Potete farlo con i seguenti comandi: >

```
:highlight link xType Type
:highlight link xStatement Statement
```

Ciò dice a Vim di evidenziare "xType" come "Type" e "xStatement" come "Statement". Vedere |**group-name**| per i nomi standard.

KEYWORD INSOLITE

I caratteri usati in una parola chiave debbono essere entro l'opzione **'iskeyword'**. Se usate un altro carattere la parola non potrà trovare mai una corrispondenza. Vim non fornirà un messaggio per informarvi di ciò.
Il linguaggio x usa il carattere '-' nelle parole chiave. Ecco come viene fatto:
>

```
:setlocal iskeyword+=-
:syntax keyword xStatement when-not
```

Il comando ":setlocal" viene usato per cambiare **'iskeyword'** soltanto per il buffer corrente. Ora cambierà il comportamento di comandi come "w" e "*". Se non volete ciò, non definite una parola chiave ma usate un confronto (spiegato nella prossima sezione).

Il linguaggio x ammette delle abbreviazioni. Ad esempio "next" può venire abbreviato in "n", "ne" o "nex". Potete definirlo usando questo comando:
>

```
:syntax keyword xStatement n[ext]
```

Questo non funzionerà per "nextone", le parole chiave corrispondono sempre e soltanto con parole intere.

==
44.3 Raffronti

Pensiamo di definire qualcosa di più complesso. Volete verificare le corrispondenze di identificatori ordinari. Per farlo definite un elemento di confronto per la sintassi. Questo trova corrispondenza con ogni parola che consista di soli caratteri minuscoli: >

```
	:syntax match xIdentifier /\<\l\+\>/
<
```

	Nota:
	Le keyword prevalgono su di ogni altro elemento di sintassi. Così le parole chiave "if", "then", etc., saranno keyword, come definito prima dal comando ":syntax keyword", anche se mantengono il modello per xIdentifier.

La parte finale è un modello, come esso viene usato per la ricerca. Il // viene impiegato in aggiunta al modello (come avviene in un comando ":substitute". Potete utilizzare qualsiasi altro carattere, come un più o le virgolette.

Adesso definiamo un confronto per un commento. Nel linguaggio x si tratta di qualsiasi cosa a partire da # sino alla fine della linea: >

```
	:syntax match xComment /#.*/
```

Poiché potete usare qualunque modello di ricerca, potete evidenziare cose molto complicate con un elemento di confronto. Vedere |pattern| sui modelli per la ricerca.

==
44.4 Regioni

Nel linguaggio x di esempio le stringhe vengono incluse entro virgolette doppie ("). Per evidenziare stringhe definite una regione. Serve una regione di inizio (double quote) ed una di fine (double quote). La definizione è come segue: >

```
	:syntax region xString start=/"/ end=/"/
```

Le direttive "start" ed "end" definiscono i modelli usati per trovare l'inizio o la fine della regione. Ma come fare con stringhe come questa?

```
	"A string with a double quote (\") in it" ~
```

Ciò crea un problema: Le virgolette doppie nel mezzo della stringa faranno terminare la regione. Dovrete dire a Vim di saltare ogni virgoletta doppia preceduta da \ entro la stringa. Fatelo con la parola chiave skip: >

```
	:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

La backslash doppia trova una backslash singola, poiché la backslash è un carattere speciale nei modelli per la ricerca.

Quando usare una regione in luogo di una verifica di corrispondenza? La differenza principale consiste nel fatto che un elemento di confronto è costituito da un unico modello che deve trovare una corrispondenza esatta. Una regione inizia dove si trova il modello "start". Se il modello "end" viene trovato o no non importa. Così quando l'elemento dipendesse dalla corrispondenza con il modello "end" non potreste utilizzare le regioni. Altrimenti le regioni sono generalmente più facili da definire. Ed è più semplice per impiegare elementi annidati, come spiegato nella prossima sezione.

==
44.5 Elementi annidati

Osservate questo commento:

```
	%Get input  TODO: Skip white space ~
```

Volete evidenziare TODO in grosse lettere gialle, nonostante si trovi dentro

un commento che viene evidenziato in blu. Per consentire a Vim di saperlo definite i seguenti gruppi di sintassi: >

```
:syntax keyword xTodo TODO contained
:syntax match xComment /%.*/ contains=xTodo
```

Nella prima linea l'argomento "contained" dice a Vim che questa keyword può esistere soltanto entro altri elementi di sintassi. La linea successiva ha "contains=xTodo". Ciò indica che l'elemento di sintassi xTodo è dentro di essa. Il risultato è che la linea di commento nel suo insieme viene riscontrata con "xComment" e viene fatta blu. La parola TODO entro essa trova corrispondenza con xTodo ed evidenziata in giallo (L'evidenziazione per xTodo era stata fatta per questo).

ANNIDAMENTO RICORSIVO

Il linguaggio x definisce blocchi di codice tra parentesi graffe. Ed un blocco di codice può contenere altri blocchi di codice. Ciò può venir definito così: >

```
:syntax region xBlock start=/{/ end=/}/ contains=xBlock
```

Supponiamo che abbiate questo testo:

```
while i < b { ~
        if a { ~
                b = c; ~
        } ~
} ~
```

Inizialmente uno xBlock comincia dalla { nella prima linea. Nella seconda linea si trova un'altra {. Poiché ci troviamo entro un elemento xBlock ed esso contiene solo se stesso, un elemento annidato xBlock inizierà qui. Così la linea "b = c" lè dentro la regione xBlock di secondo livello. Allora una } viene trovata nella linea successiva e corrisponde con il modello di fine della regione. Ciò termina l'xBlock annidato. Poiché la } si trova nella regione annidata, viene nascosta dalla regione del primo xBlock. Così all'ultima } termina la regione del primo xBlock.

TROVARE LA FINE

Considerate i seguenti due elementi di sintassi: >

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment
```

Definite un commento come qualsiasi cosa a partire da % sino al termine della linea. Una direttiva di preprocessore è qualunque cosa a partire da # sino al termine della linea. Poiché potete avere un commento su di una linea di preprocessore, la definizione di preprocessore include un argomento "contains=xComment". Adesso vediamo cosa succede con questo testo:

```
#define X = Y  % Comment text ~
int foo = 1; ~
```

Ciò che vedete è che anche la seconda linea viene evidenziata come xPreProc. La direttiva preprocessore potrebbe terminare alla fine della linea. Ciò perché avete usato "end=/$/". Così cosa è andato male?

Il problema è il commento contenuto. Il commento parte con % e finisce alla fine della linea. Dopo la fine del commento la sintassi del preprocessore continua. Ciò succede dopo che si è vista la fine della linea, così viene inclusa anche la linea successiva.

Per evitare questo problema e che un elemento di sintassi contenuto si mangi la necessaria fine della linea, impiegate l'argomento "keepend". Ciò si occuperà del fatto di trovare una doppia fine della linea: >

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

CONTENIMENTO DI MOLTI ELEMENTI

Potete usare l'argomento contains per specificare che tutto può essere compreso. Ad esempio: >

```
:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

Tutti gli elementi di sintassi verranno compresi entro soltanto questo. Esso contiene anche se stesso, ma non alla stessa posizione (che potrebbe causare un loop senza fine).
Potete specificare che alcuni gruppi non siano compresi. Così comprende tutti i gruppi ma solo quelli che vengono elencati:
>

```
:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

Con l'elemento "TOP" potete inserire tutti gli elementi che non abbiano un argomento "contained". "CONTAINED" viene usato solo per includere elementi con un argomento "contained". Vedere |:syn-contains| per i dettagli.

==========

44.6 Seguendo i gruppi

Il linguaggio x comprende dichiarazioni in questa forma:

```
if (condition) then ~
```

Volete evidenziare i tre elementi in modo diverso. Ma "(condition)" e "then" potrebbero apparire anche altrove, dove sono richieste diverse evidenziazioni. Potete fare così: >

```
:syntax match xIf /if/ nextgroup=xIfCondition skipwhite
:syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
:syntax match xThen /then/ contained
```

L'argomento "nextgroup" specifica quale argomento possa essere il successivo. Ciò non è necessario. Se nessuno degli elementi che avete specificato venisse trovato non succederebbe nulla. Ad esempio, in questo testo:

```
if not (condition) then ~
```

L'"if" corrisponde con xIf. "not" non trova corrispondenze con la specificata xIfCondition di nextgroup, così solo l'"if" viene evidenziato.

L'argomento "skipwhite" dice a Vim che gli spazi bianchi (spazi e tabulazioni) possono apparire tra gli elementi. Argomenti analoghi sono "skipnl", che consente un'interruzione di linea tra gli elementi, e "skipempty", che permette linee vuote. Notate che "skipnl" non salta una linea vuota, talvolta trova corrispondenza dopo l'interruzione di linea.

==========

44.7 Altri argomenti

MATCHGROUP

Quando definite una regione l'intera regione viene evidenziata secondo il nome di gruppo specificato. Per evidenziare il testo incluso tra parentesi () con il gruppo xInside, ad esempio, usate il comando seguente: >

```
:syntax region xInside start=/(/ end=/)/
```

Immaginiamo che vogliate evidenziare le parentesi in modo diverso. Potete farlo con molte contorte dichiarazioni di regione, o potete usare l'argomento "matchgroup". Ciò dice a Vim di evidenziare l'inizio e la fine di una regione con un diverso gruppo di evidenziazione (in questo caso, il gruppo xParen): >

```
:syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

L'argomento "matchgroup" viene applicato alla corrispondenza di start e di end che lo seguono. Nell'esempio precedente sia start che end vengono evidenziati con xParen. Per evidenziare end con xParenEnd: >

```
:syntax region xInside matchgroup=xParen start=/(/
        \ matchgroup=xParenEnd end=/)/
```

Un effetto collaterale dell'uso di "matchgroup" è che gli elementi contenuti

non trovino corrispondenza all'inizio od alla fine della regione. L'esempio per "transparent" usa ciò.

TRANSPARENT

In un file in linguaggio C vorreste evidenziare il testo () dopo un "while" in modo differente dal testo tra parentesi dopo un "for". In entrambi i casi potrebbero esservi annidati degli elementi (), che dovrebbero essere evidenziati allo stesso modo. Dovete essere certi che l'evidenziazione di () cessi con la corrispondenza). Ecco un modo per farlo:
>

```
:syntax region cWhile matchgroup=cWhile start=/while\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cCondNest start=/(/ end=/)/ contained transparent
```

Ora potete dare con cWhile e cFor diverse evidenziazioni. L'elemento cCondNest può apparire in uno di essi, ma escludere l'evidenziazione dell'elemento che è contenuto. L'argomento "transparent" causa ciò.
Notare che l'argomento "matchgroup" ha lo stesso gruppo dell'elemento stesso. Perché definirlo allora? Bene, l'effetto collaterale di usare matchgroup è che elementi contenuti non vengono trovati allora con l'elemento start. Ciò evita che il gruppo cCondNest trovi la (appena dopo il "while" od il "for". Se ciò avvenisse potrebbe accoppiare tutto il testo sino alla corrispondenza con) e la regione continuerebbe dopo di esso. Adesso cCondNest trova la corrispondenza solo con il modello start, dopo la prima (.

OFFSET

Poniamo che vogliate definire una regione per il testo tra (e) dopo un "if". Ma non volete includere l'"if" o le (e). Potete fare questo specificando gli offset per i modelli. Esempio: >

```
:syntax region xCond start=/if\s*(/ms=e+1 end=/)/me=s-1
```

L'offset per il modello di start è "ms=e+1". "ms" sta per Match Start. Ciò definisce un offset per l'inizio della corrispondenza. Normalmente la verifica parte dal modello di ricerca. "e+1" significa che la verifica adesso partirà al termine del modello di ricerca e precisamente da un carattere dopo.
L'offset per il modello end è "me=s-1". "me" sta per Match End. "s-1" significa l'inizio dal modello di ricerca ed allora un carattere prima. Il risultato è che in questo testo:

```
if (foo == bar) ~
```

Soltanto il testo "foo == bar" verrà evidenziato come xCond.

Altro sugli offset qui: |**:syn-pattern-offset**|.

ONELINE

L'argomento "oneline" indica che la regione non attraversa il limite di una linea. Ad esempio: >

```
:syntax region xIfThen start=/if/ end=/then/ oneline
```

Ciò definisce una regione che inizia dall'"if" e termina al "then". Ma se non ci fosse un "then" dopo l'"if", la regione non corrisponderebbe.

> Nota:
> Usando "oneline" la regione non inizia se il modello end non si trova nella stessa linea. Senza "oneline" Vim non potrà trovare una corrispondenza per il modello di end pattern. La regione inizia lo stesso quando il modello di end non venisse trovato nel resto del file.

LINEE CHE VANNO A CAPO E COME EVITARLE

Adesso le cose diventano un po' più complesse. Definiamo una linea di preprocessore. Questa inizia con un # nella prima colonna e continua sino alla fine della linea. Una linea che finisca con \ rende la linea successiva una linea di continuazione. Il modo per gestire ciò è consentire che un elemento di sintassi contenga un modello di continuazione: >

```
:syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
:syntax match xLineContinue "\\$" contained
```

In questo caso sebbene xPreProc verifichi normalmente una sola linea, il gruppo in essa contenuto (vale a dire xLineContinue) gli consente di procedere per più di una sola linea. Ad esempio, verificherà entrambe queste linee:

```
#define SPAM  spam spam spam \ ~
                bacon and spam ~
```

In questo caso è quanto volevate. Se non fosse così potreste imporre alla regione di essere su di una sola linea aggiungendo "excludenl" al modello contenuto. Ad esempio, volete evidenziare "end" in xPreProc, ma soltanto alla fine della linea. Per evitare che xPreProc continui anche sulla linea successiva, come fa xLineContinue, usate "excludenl" in questa maniera: >

```
:syntax region xPreProc start=/^#/ end=/$/
        \ contains=xLineContinue,xPreProcEnd
:syntax match xPreProcEnd excludenl /end$/ contained
:syntax match xLineContinue "\\$" contained
```

"excludenl" deve essere posto prima del modello. Se c'è "xLineContinue" non si può usare "excludenl", una corrispondenza con esso estenderebbe xPreProc alla prossima linea come prima.

==========

44.8 Cluster

Una delle cose che dovete sapere quando iniziate a scrivere un file di sintassi è che state per generare molti gruppi di sintassi. Vim vi consente di definire una collezione di gruppi di sintassi che viene chiamata cluster.

Immaginate di avere un linguaggio che contenga cicli di for, dichiarazioni if, cicli while e funzioni. Ognuno di essi contiene gli stessi elementi di sintassi: numeri ed identificatori. Potete definirli così: >

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent
:syntax match xIf /^if.*/ contains=xNumber,xIdent
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

Vi tocca ripetere lo stesso "contains=" ogni volta. Se volete aggiungere un altro elemento contenuto, dovete aggiungerlo tre volte. I cluster di sintassi semplificano queste definizioni consentendovi di avere un solo cluster che stia per molti gruppi di sintassi.

Per definire un cluster per i due elementi che i tre gruppi contengono usate il seguente comando: >

```
:syntax cluster xState contains=xNumber,xIdent
```

I cluster vengono usati entro altri elementi di sintassi proprio come ogni gruppo di sintassi. Il loro nome inizia con @. Potete definire i tre gruppi in questo modo: >

```
:syntax match xFor /^for.*/ contains=@xState
:syntax match xIf /^if.*/ contains=@xState
:syntax match xWhile /^while.*/ contains=@xState
```

Potete aggiungere nuovi nomi di gruppo a questo cluster con l'argomento "add": >

```
:syntax cluster xState add=xString
```

Potete togliere dei gruppi di sintassi da questo elenco altrettanto facilmente: >

```
:syntax cluster xState remove=xNumber
```

==========

44.9 Inserimento di un altro file di sintassi

La sintassi del linguaggio C++ include quella del linguaggio C. Poiché non volete scrivere due file di sintassi potrete fare leggere il file di sintassi per C++ entro uno per C usando il comando seguente: >

```
:runtime! syntax/c.vim
```

Il comando ":runtime!" cerca in 'runtimepath' tutti i file "syntax/c.vim". Ciò fa sì che la parte C della sintassi C++ sia definita come per i file C. Se aveste sostituito il file di sintassi c.vim, o aggiunto elementi con un file extra, questi verrebbe pure caricati.

Dopo aver caricato gli elementi di sintassi C gli elementi specifici per C++ potranno essere definiti. Ad esempio, aggiungere keyword che non vengono usate in C: >

```
:syntax keyword cppStatement    new delete this friend using
```

Funzionerà esattamente come in ogni altro file di sintassi.

Adesso consideriamo il linguaggio Perl. Uno script Perl consiste di due parti distinte: una sezione di documentazione in formato POD, e il programma vero e proprio, scritto in Perl. La sezione POD comincia con "=head" e finisce con "=cut".

Volete definire la sintassi POD in un solo file ed usarlo dal file di sintassi di Perl. Il comando ":syntax include" legge in un file di sintassi ed immagazzina gli elementi che questo ha definito in un cluster di sintassi. Per Perl le dichiarazioni sono come segue: >

```
:syntax include @Pod <sfile>:p:h/pod.vim
:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

Quando in un file Perl viene trovato "=head", la regione perlPOD inizia. In questa regione è contenuto il cluster @Pod. Tutti gli elementi definiti come elementi di massimo livello nei file di sintassi pod.vim si troveranno qui. Quando verrà trovato "=cut", la regione finirà e ritorneremo agli elementi definiti nel file di Perl.

Il comando ":syntax include" è abbastanza intelligente da ignorare un comando ":syntax clear" nel file incluso. Ed un argomento come "contains=ALL" conterrà solo elementi definiti nel file in esso incluso, non nel file che lo include.

La parte "<sfile>:p:h/" usa il nome del file corrente (<sfile>), lo espande al suo path completo (:p) ed allora prende l'intestazione (:h). Ciò risulta nel nome della directory del file. Ciò causa che il file pod.vim venga incluso nella directory stessa.

==

44.10 Sincronizzazione

I compilatori la fanno semplice. Partono dall'inizio del file e lo interpretano direttamente. Vim non fa ciò così facilmente. Deve partire dal mezzo, dove si è iniziato a lavorare. Come fare a dirgli dove si trova?

Il segreto sta nel comando ":syntax sync". Spiega a Vim come capire dove si trovi. Ad esempio, il comando che segue dice a Vim di cercare all'indietro l'inizio o la fine di un commento in stile C ed iniziare da li a colorare la sintassi: >

```
:syntax sync ccomment
```

Potete affinare questo processo con qualche argomento. L'argomento "minlines" dice a Vim il minimo numero di linee da vedere all'indietro, e "maxlines" dice all'editor il massimo numero di linee da scandire.

Ad esempio, il comando seguente dice a Vim di osservare almeno 10 linee prima della cima dello schermo: >

```
:syntax sync ccomment minlines=10 maxlines=500
```

Se non riesce a capire dove si trovi entro questo spazio comincerà a guardare sempre più distante sino a riuscirvi. Ma non guarderà indietro più di 500 linee. (Un grosso "maxlines" rallenta il processo. Uno piccolo potrebbe causare il fallimento di sincronizzazione.)

Per sincronizzare un po' più rapidamente dite a Vim quali elementi di sintassi possono essere saltati. Tutte le corrispondenze e le regioni che

richiedono solo di essere usate mentre viene mostrato il testo possono essere date con l'argomento "display".
   Di default, il commento che deve essere trovato verrà colorato come parte del gruppo Comment syntax. Se voleste colorare le cose in altro modo potete specificare un gruppo di sintassi differente: >

```
:syntax sync ccomment xAltComment
```

Se il vostro linguaggio di programmazione non prevede commenti in stile C potete provare un altro metodo di sincronizzazione. Il modo più semplice consiste nel dire a Vim di spaziare all'indietro di un certo numero di linee e tentare di riuscire a capire le cose da li. Il comando seguente dice a Vim di andare indietro di 150 linee ed iniziare ad interpretare da lì: >

```
:syntax sync minlines=150
```

Un valore grosso di "minlines" può rendere Vim più lento, specialmente scorrendo il file all'indietro.
   In ultimo, potete specificare un gruppo di sintassi per cercare usando questo comando: >

```
:syntax sync match {sync-group-name}
        \ grouphere {group-name} {pattern}
```

Esso dice a Vim che quando vede **{pattern}** il gruppo di sintassi chiamato **{group-name}** comincia subito dopo il modello dato. Il **{sync-group-name}** viene usato per dare un nome a questa specifica di sincronizzazione. Ad esempio negli script del programma sh una dichiarazione if inizia con "if" e si conclude con "fi": >

```
if [ --f file.txt ] ; then ~
        echo "File exists" ~
fi ~
```

Per definire una direttiva "grouphere" per questa sintassi usate il seguente comando: >

```
:syntax sync match shIfSync grouphere shIf "\<if\>"
```

L'argomento "groupthere" dice a Vim che il modello finisce un gruppo. Ad esempio, la fine del gruppo if/fi è come segue: >

```
:syntax sync match shIfSync groupthere NONE "\<fi\>"
```

In questo esempio il NONE dice a Vim che non siete entro una regione di sintassi speciale. Particolarmente non vi trovate entro un blocco if.

Potete anche definire corrispondenze e regioni che siano senza "grouphere" od argomenti di "groupthere". Questi gruppi sono per gruppi di sintassi saltati durante la sincronizzazione. Ad esempio quanto segue salta ogni cosa entro {}, anche se corrisponderebbe normalmente con altri metodi di sincronizzazione: >

```
:syntax sync match xSpecial /{.*}/
```

Di più circa la sincronizzazione nel manuale di riferimento : |**:syn-sync**|.

==========

44.11 Installare un file di sintassi

Quando il vostro nuovo file di sintassi è pronto per essere usato mettetelo entro una directory chiamandola "syntax", nel **'runtimepath'**. Per Unix potrebbe essere "~/.vim/syntax".
  Il nome del file di sintassi deve essere uguale al tipo di file con aggiunto ".vim". Così per il linguaggio x il path completo potrebbe essere:

```
~/.vim/syntax/x.vim ~
```

Dovete anche far sì che il tipo di file venga riconosciuto. Vedere |**43.2**|.

Se il vostro file lavora bene potreste desiderare di renderlo disponibile per altri utenti di Vim. Prima leggete la prossima sezione per essere certi che il vostro file funzioni bene per altri. Allora inviate un messaggio

e-mail al manutentore di Vim: <maintainer@vim.org>. Così spiegate come il tipo di file può essere riconosciuto. Con un po' di fortuna il vostro file verrà incluso nella prossima versione di Vim!

AGGIUNGERE AD UN FILE DI SINTASSI ESISTENTE

Stiamo immaginando che stiate aggiungendo un file di sintassi completamente nuovo. Quando un file di sintassi esistente funziona, ma sono stati persi alcuni elementi, potete aggiungere degli elementi in un file separato. Ciò evita di cambiare i file di sintassi distribuiti che possono andare perduti installando una nuova versione di Vim.
Scrivete comandi di sintassi nel vostro file possibilmente usando nomi di gruppo dalla sintassi esistente. Ad esempio per aggiungere nuovi tipi di variabile al file di sintassi di C:
>

```
:syntax keyword cType off_t uint
```

Scrivete il file con lo stesso nome del file di sintassi originale. In questo caso "c.vim". Piazzatelo entro una directory vicino alla fine di **'runtimepath'**. Ciò fa sì che venga caricato dopo il file di sintassi originale. Per Unix potrebbe essere:

```
~/.vim/after/syntax/c.vim ~
```

==========

44.12 Aspetto di un file di sintassi portatile

Non sarebbe bello se tutti gli utenti di Vim si scambiassero i file di sintassi? Per renderlo possibile il file di sintassi deve seguire poche linee guida.

Cominciate con un header che spieghi a cosa serve il file di sintassi, chi lo mantiene e quale sia l'ultimo aggiornamento. Non includete troppe informazioni circa la storia delle modifiche, poca gente lo leggerà. Esempio: >

```
" Vim syntax file
" Language:     C
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" Last Change:  2001 Jun 18
" Remark:       Included by the C++ syntax.
```

Usate lo stesso aspetto degli altri file di sintassi. Usando un file di sintassi esistente come esempio risparmierete un mucchio di tempo.

Scegliete un buono, descrittivo nome per il vostro file di sintassi. Usate lettere minuscole e cifre. Non fatelo troppo lungo, viene usato in molti posti: Il nome del file di sintassi "nome.vim", **'filetype'**, b:current_syntax e l'inizio di ogni gruppo di sintassi (nameType, nameStatement, nameString, etc.).

Iniziate con una prova per "b:current_syntax". Se è definito, qualche altro file di sintassi, trovato prima in **'runtimepath'** era già stato caricato. >

```
if exists("b:current_syntax")
  finish
endif
```

Per mantenere la compatibilità con Vim 5.8 usate: >

```
if version < 600
  syntax clear
elseif exists("b:current_syntax")
  finish
endif
```

Mettete "b:current_syntax" al nome della sintassi alla fine. Non dimenticate che i file inclusi fanno ciò anche, potreste dover resettare "b:current_syntax" se includete due file.

Se volete che il vostro file di sintassi funzioni con Vim 5.x, aggiungete una prova per v:version. Vedere yacc.vim per un esempio.

Non includete alcunché sia una preferenza dell'utente. Non impostate
'tabstop', **'expandtab'**, etc. Ciò appartiene al plugin di un filename.

Non includete mappature od abbreviazioni. Aggiungete soltanto l'impostazione
'iskeyword' se è realmente necessario per riconoscere le keyword.

Per consentire agli utenti di scegliere i propri colori preferiti, definite un
nome di gruppo differente per ogni tipo di elemento evidenziato. Poi
collegate ognuno di essi ad uno dei gruppi di evidenziazione standard. Ciò
ne renderà possibile il funzionamento con ogni schema di colore. Se scegliete
dei colori specifici, potrebbero risultare sgradevoli con alcuni schemi di
colore. E non dimenticate che alcuni usano un differente colore di fondo,
oppure hanno a disposizione solo otto colori.

Per il collegamento usate "hi def link", in modo che l'utente possa scegliere
evidenziazioni diverse prima che il vostro file di sintassi sia caricato.
Esempio: >

```
hi def link nameString        String
hi def link nameNumber        Number
hi def link nameCommand       Statement
... etc ...
```

Aggiungete l'argomento "display" agli elementi che non si usano quando si
effettua una sincronizzazione, per velocizzare lo scorrimento all'indietro
e **CTRL-L**.

==========

Capitolo seguente: |**usr_45.txt**| Selezionate la vostra lingua

 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