VIM 5.6 Reference Guide
version 0.7

Bram Moolenaar        Oleg Raisky

May 25, 2000

Conventions:

:marks denotes VIM command typed in Ex mode

visual denotes VIM command typed in Visual mode

[..] denotes an optional part of the command

file denotes command argument(s)

{} denotes a set of characters

Space means pressed key/combination of keys

this command/feature is VIM specific (not found in Vi)

Contents

1 Movement Commands
  1.1 Left-right motions .................................. 1
  1.2 Up-down motions .................................. 2
  1.3 Text object motions ................................ 2
  1.4 Scrolling ....................................... 3
  1.5 Various motions ................................... 3
  1.6 Marks and motions ................................ 3
  1.7 Using tags ...................................... 4

2 Editing Commands
  2.1 Inserting text .................................... 4
  2.2 Keys in Insert mode ................................ 4
  2.3 Special keys in Insert mode ......................... 5
  2.4 Digraphs© ..................................... 5
  2.5 Special inserts ................................... 5
  2.6 Deleting text ..................................... 6
  2.7 Copying and moving text .............................. 6
  2.8 Changing text ..................................... 6
  2.9 Complex changes .................................. 7
  2.10 Visual mode© ................................... 8
  2.11 Text objects ...................................... 9
  2.12 Repeating Commands ................................ 10
  2.13 Undo/Redo Commands ................................ 10
  2.14 Command-line editing ............................. 10
  2.15 Encryption ...................................... 11
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Key Mappings Abbreviations</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Key mapping</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Abbreviations</td>
<td>13</td>
</tr>
<tr>
<td>3.3 User-defined commands</td>
<td>13</td>
</tr>
<tr>
<td>4 Options</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Setting Options</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Option explanation</td>
<td>15</td>
</tr>
<tr>
<td>5 Other Commands</td>
<td>20</td>
</tr>
<tr>
<td>5.1 Shell Commands</td>
<td>20</td>
</tr>
<tr>
<td>5.2 QuickFix Commands</td>
<td>20</td>
</tr>
<tr>
<td>5.3 Viminfo Commands</td>
<td>21</td>
</tr>
<tr>
<td>5.4 Various Commands</td>
<td>22</td>
</tr>
<tr>
<td>6 <code>ex</code> ranges and search patterns</td>
<td>22</td>
</tr>
<tr>
<td>6.1 Ranges</td>
<td>22</td>
</tr>
<tr>
<td>6.2 Special <code>ex</code> characters</td>
<td>23</td>
</tr>
<tr>
<td>6.3 Pattern searches</td>
<td>23</td>
</tr>
<tr>
<td>6.4 Special characters in search patterns</td>
<td>23</td>
</tr>
<tr>
<td>6.5 Offsets allowed after search command</td>
<td>25</td>
</tr>
<tr>
<td>7 Starting, Writing and Quitting Commands</td>
<td>25</td>
</tr>
<tr>
<td>7.1 Starting VIM</td>
<td>25</td>
</tr>
<tr>
<td>7.2 Editing a file</td>
<td>26</td>
</tr>
<tr>
<td>7.3 Using the argument list</td>
<td>27</td>
</tr>
<tr>
<td>7.4 Writing and quitting</td>
<td>27</td>
</tr>
<tr>
<td>8 Windows and Buffers functions</td>
<td>28</td>
</tr>
<tr>
<td>8.1 Multi-window functions</td>
<td>28</td>
</tr>
<tr>
<td>8.2 Buffer list functions</td>
<td>29</td>
</tr>
<tr>
<td>9 Script Language</td>
<td>29</td>
</tr>
<tr>
<td>9.1 Variables</td>
<td>29</td>
</tr>
<tr>
<td>9.2 Expression syntax</td>
<td>30</td>
</tr>
<tr>
<td>9.3 Functions</td>
<td>31</td>
</tr>
<tr>
<td>9.4 User-Defined Functions</td>
<td>38</td>
</tr>
<tr>
<td>9.5 Commands</td>
<td>39</td>
</tr>
<tr>
<td>10 GUI</td>
<td>40</td>
</tr>
<tr>
<td>10.1 Mouse Control</td>
<td>40</td>
</tr>
<tr>
<td>10.2 Window Position</td>
<td>41</td>
</tr>
<tr>
<td>10.3 Menus</td>
<td>41</td>
</tr>
<tr>
<td>10.4 Miscellaneous</td>
<td>43</td>
</tr>
<tr>
<td>11 Syntax highlighting</td>
<td>43</td>
</tr>
<tr>
<td>11.1 Syntax files</td>
<td>43</td>
</tr>
<tr>
<td>11.2 Defining a syntax</td>
<td>44</td>
</tr>
<tr>
<td>11.3 Syntax arguments</td>
<td>46</td>
</tr>
<tr>
<td>11.4 Syntax patterns</td>
<td>46</td>
</tr>
<tr>
<td>11.5 Synchronizing</td>
<td>47</td>
</tr>
<tr>
<td>11.6 Highlight command</td>
<td>49</td>
</tr>
</tbody>
</table>
11.7 Linking groups ........................................ 51

12 Automatic Commands 51
12.1 Defining autocommands .................................... 51
12.2 Removing autocommands .................................. 51
12.3 Listing autocommands .................................... 51
12.4 Events ........................................ 51
12.5 Patterns ....................................... 51
12.6 Filetypes ....................................... 51
12.7 Groups ........................................ 51
12.8 Executing autocommands ................................. 51
12.9 Using autocommands .................................. 51

13 Miscellany 57
13.1 VIM modes ....................................... 57
13.2 VIM registers ..................................... 57

1 Movement Commands

1.1 Left-right motions

\([n]\) h  left (also:  \[\text{CTRL-H}\], \(\text{BS}\), or \(\leftarrow\) key)

\([n]\) l  right (also:  \(\text{Space}\) or \(\rightarrow\) key)

0  to first character in the line (also:  \(\text{Home}\) key)

\(\wedge\)  to first non-blank character in the line

\([n]\) $  to the last character in the line (\(n\)-1 lines lower) (also:  \(\text{End}\) key)

\([n]\) g0  to first character in screen line (differs from 0 when lines wrap)

\([n]\) g\(\wedge\)  to first non-blank character in screen line (differs from \(\wedge\) when lines wrap)

\([n]\) g$  to last character in screen line (differs from \$ when lines wrap)

\([n]\) gm  to middle of the screen line

\([n]\) |  to column \(n\) (default: 1)

\([n]\) f char  to the \(n\)-th occurrence of char to the right

\([n]\) F char  to the \(n\)-th occurrence of char to the left

\([n]\) t char  till before the \(n\)-th occurrence of char to the right

\([n]\) T char  till before the \(n\)-th occurrence of char to the left

\([n]\) ;  repeat the last f, F, t, or T \(n\) times

\([n]\) ,  repeat the last f, F, t, or T \(n\) times in opposite direction

1.2 Up-down motions

\([n]\) k  up \(n\) lines (also:  \[\text{CTRL-P}\] and \([\uparrow]\])

\([n]\) j  down \(n\) lines (also:  \[\text{CTRL-J},\ \text{CTRL-N}\text{NL},\ \text{and}\ [\downarrow]\])

\([n]\) –  up \(n\) lines, on the first non-blank character

\([n]\) +  down \(n\) lines, on the first non-blank character (also:  \[\text{CTRL-M}\] and \([\text{Ret}]\))

\([n]\) _  down \(n\)-1 lines, on the first non-blank character

\([n]\) G  goto line \(n\) (default: last line), on the first non-blank character

\([n]\) gg  goto line \(n\) (default: first line), on the first non-blank character
n % goto line n percentage down in the file. n must be given, otherwise it is the % command

[n] gk or g↑ up n screen lines (differs from k when line wraps, and when used with an operator, because it’s not linewise.)

[n] gj or g↓ down n screen lines (differs from j when line wraps, and when used with an operator, because it’s not linewise.)

:[range]go[to] [count] Go to count byte in the buffer. Default count is zero, start of the file. When giving range, the last number in it used. End-of-line characters are counted depending on the current fileformat setting.

1.3 Text object motions

[n] w n words
[n] W n blank-separated WORDS forward
[n] e forward to the end of the n-th word
[n] E forward to the end of the n-th blank-separated WORD
[n] b n words backward
[n] B n blank-separated WORDS backward
[n] ge backward to the end of the n-th word
[n] gE backward to the end of the n-th blank-separated WORD
[n] ) n sentences forward
[n] ( n sentences backward
[n] } n paragraphs forward
[n] { n paragraphs backward
[n][] n sections forward, at start of section
[n][[ n sections backward, at start of section
[n][] n sections forward, at end of section
[n][] n sections backward, at end of section
[n][{ n times back to unclosed (
[n][{ n times back to unclosed {
[n][] n times forward to unclosed )
[n][] n times forward to unclosed }
[n][# n times back to unclosed #if or #else
[n][# n times forward to unclosed #else or #endif
[n][* n times back to start of comment /*
[n][* n times forward to end of comment */

1.4 Scrolling

[n] CTRL-E window n lines downwards (default: 1)
[n] CTRL-D window n lines Downwards (default: 1/2 window)
[n] CTRL-F window n pages Forwards (downwards)
[n] CTRL-Y window n lines upwards (default: 1)
[n] CTRL-U window n lines Upwards (default: 1/2 window)
[n] CTRL-B window n pages Backwards (upwards)

1For definition of word, WORD, sentence, paragraph and section see Section 2.11
z (Ret) or zt  redraw, current line at top of window
z. or zz  redraw, current line at center of window
z– or zb  redraw, current line at bottom of window
[n] zh  scroll screen n characters to the right
[n] zl  scroll screen n characters to the left
[n] zH  scroll screen half a screenwidth to the right
[n] zL  scroll screen half a screenwidth to the left

1.5 Various motions

%  find the next brace, bracket, comment, or #if/#else/#endif in this line and go to its match
[n] H  go to the n-th line in the window, on the first non-blank
M  go to the middle line in the window, on the first non-blank
[n] L  go to the n-th line from the bottom, on the first non-blank
[n] go  go to n-th byte in the buffer
: [range] go[to] [off]  go to [off] set byte in the buffer

1.6 Marks and motions

m{a-zA-Z}  mark current position with mark {a-zA-Z}
‘{a-z}  go to mark {a-z} within current file
‘{A-Z}  go to mark {A-Z} in any file
‘{0-9}  go to the position where VIM was last exited
“  go to the position before the last jump
“  go to the position when last editing this file
‘  go to the start of the previously operated or put text
’  go to the end of the previously operated or put text
‘<  go to the start of the (previous) Visual area
’>  go to the end of the (previous) Visual area
’{a-zA-Z0-9}’<>  same as ‘, but on the first non-blank in the line
:marks  display the active marks
[n] CTRL-O  go to n-th older position in jump list
[n] CTRL-I  go to n-th newer position in jump list
:ju[m]ps  display the jump list

1.7 Using tags

:tag  jump to tag
: [tag]  jump to n-th newer tag in tag list
CTRL-]  jump to the tag under cursor, unless changes have been made
[n] CTRL-T  jump back from n-th older tag in tag list
:ti[ump][!] [tag]  Jump to tag tag or select from list when there are multiple matches
:ts[elect][!] [tag]  list matching tags and select one to jump to
:n] tn[ext][!]  jump to n-th next matching tag
:n] tp[revious][!]  jump to n-th previous matching tag
2 Editing Commands

2.1 Inserting text

\[\text{[n]} \text{tr[ewind]][l]}\] jump to \text{n}-th matching tag
\[\text{[n]} \text{po[p][l]}\] jump back from \text{n}-th older tag in tag list
\text{tags} print tag list
\text{pt[ag]} \text{tag} open a preview window to show tag \text{tag}
\text{CTRL-W} \} like \text{CTRL-]} but show tag in preview window
\text{pts[elect]} like \text{:select} but show tag in preview window
\text{ptj[ump]} like \text{:tjump} but show tag in preview window
\text{pc[lose]} close tag preview window
\text{CTRL-W} z close tag preview window

2.2 Keys in Insert mode

\text{char} action in Insert mode
\text{Esc} end Insert mode, back to Normal mode
\text{CTRL-C} like \text{Esc}, but do not complete an abbreviation begun
\text{CTRL-A} insert previously inserted text
\text{CTRL-@} insert previously inserted text and stop insert
\text{CTRL-O} \text{command} execute \text{command} and return to Insert mode
\text{CTRL-R} \{0-9a-zA-Z%#:.--=*\} insert the contents of a register
\text{NL} or \text{Ret} or \text{CTRL-M} or \text{CTRL-J} begin new line
\text{CTRL-E} insert the character from below the cursor
\text{CTRL-Y} insert the character from above the cursor
\text{CTRL-V} \text{char} insert character literally, or enter decimal byte value
\text{CTRL-N} insert next match of identifier before the cursor
\text{CTRL-P} insert previous match of identifier before the cursor
\text{CTRL-X} \ldots complete the word before the cursor in various ways:
\text{CTRL-X} \text{CTRL-D} complete the definition or macro
\text{CTRL-X} \text{CTRL-F} complete the file name
\text{CTRL-X} \text{CTRL-I} complete the word searching the current and included files.
\text{CTRL-X} \text{CTRL-K} complete the word using \text{dictionary} files.
\text{CTRL-X} \text{CTRL-L} complete the whole line searching the current file
\text{CTRL-X} \text{CTRL-N} complete the word searching the current file

2See Section 13.1 for description of VIM modes
3See Section 13.2 for description of VIM registers
2.3 Special keys in Insert mode

cursor keys: move cursor left/right/up/down
SHIFT←/→: one word left/right
SHIFT↑/↓: one screenful backward/forward
CTRL-O: execute command
End: cursor after last character in the line
Home: cursor to first character in the line

2.4 Digraphs

Digraphs are used to enter characters that normally cannot be entered by an ordinary keyboard. These are mostly accented characters which have the eighth bit set.

:digraphs show current list of digraphs
:digraphs char1 char2 number ... add digraph(s) to the list
CTRL-K char1 [char2] enter digraph
char1 BS char2 enter digraph if digraph option set

2.5 Special inserts

:r file insert the contents of file below the cursor
:r! command insert the standard output of command below the cursor

2.6 Deleting text

["x"] [n] x delete n into register "x" characters under and after the cursor
["x"] [n] Del delete n into register "x" characters under and after the cursor
["x"] [n] X delete n into register "x" characters before the cursor
["x"] [n] d motion delete into register "x" the text that is moved over with motion
visual ["x"] d delete into register "x" the highlighted text
["x"] [n] dd delete n into register "x" lines
["x"] [n] D delete into register "x" to <EOL> (and n-1 more lines)
[n] J join n-1 lines (delete <EOL>)
[n] :[oin][!] same as J, except with [!] the join does not insert or delete any spaces.

\(^4\)for definition of motion see Section 1.3
**visual J**
join the highlighted lines

\[n\] gJ
like J, but without inserting spaces

**visual gJ**
like **visual J**, but without inserting spaces

\:[range] d[x]
delete range lines [into register x]

### 2.7 Copying and moving text

:\reg
show the contents of all registers

:\reg arg
show the contents of registers mentioned in arg

\[n\] ['x] y motion
yank the text moved over with motion into a register ['x]

**visual ['x] y**
yank the highlighted text into a register ['x]

['x] [n] yy
yank n lines into a register ['x]

['x] [n] Y
yank n lines into a register ['x]

['x] [n] p
put a register ['x] after the cursor position (n times)

['x] gp
like p but leave the cursor just after the new text. the cursor position (n times)

['x] [n] P
put a register ['x] before the cursor position (n times)

['x] gP
like P but leave the cursor just after the new text.:

['x] [MiddleMouse]
like p, but adjust indent to current line

['x] [n] ]p
like p, but adjust indent to current line

['x] [n] [p
like P, but adjust indent to current line

### 2.8 Changing text

\[n\] R
enter Replace mode (repeat the entered text n times)

gr R
Enter Virtual replace mode: Each character you type replaces existing characters in screen space.

\[n\] c motion
delete motion text [into register “x”] and start insert.

**visual c**
change the highlighted text

\[n\] cc
change n lines

\[n\] S
change n lines

\[n\] C
change to the end of the line (and n-1 more lines)

\[n\] s
change n characters

gr char
replace the virtual characters under the cursor with char. This replaces in screen space, not file space.

\[n\] r char
replace n characters with char

\[n\] gr char
replace n characters with char without affecting layout

\[n\] ~
switch case for n characters and advance cursor

**visual ~**
switch case for highlighted text

**visual U**
make highlighted text lowercase

**visual U**
make highlighted text uppercase

g~ motion
switch case for the text that is moved over with motion

\[n\] g~ or g~g~
switch case of current line.

gu motion
make the text that is moved over with motion lowercase

grU motion
make the text that is moved over with motion uppercase

\[n\] gugu or guu
make current line uppercase.

g?motion
Rot13 encode motion text.
visual g?    Rot13 encode the highlighted text.

visual g??   Rot13 encode current line.

[n] gUU or gUgU make current line uppercase.

CTRL-A   add n to the number at or after the cursor

CTRL-X   subtract n from the number at or after the cursor

[motion] move the lines that are moved over with motion one shiftwidth left

<<< move n lines one shiftwidth left

[motion] move the lines that are moved over with motion one shiftwidth right

>>> move n lines one shiftwidth right

gqq format the current line.

[n] gq format the lines that are moved over with motion to textwidth length

[:range] ce[nter] [width] center the lines in range

[:range] le[ft] [indent] left-align the lines in range with indent

[:range] ri[ght] [width] right-align the lines in range

2.9 Complex changes

[motion] ! command [Ret] filter the lines that are moved over through command

visual [range] command [Ret] filter n lines through command

visual ! command [Ret] filter the highlighted lines through command

[:]command [Ret] filter range lines through command

[n] = motion filter the lines that are moved over through indent

[n] == filter n lines through indent

visual = filter the highlighted lines through indent

[:range] s[substitute] /pattern/string/[c e g p r i l] [n] substitute pattern by string in range lines [n times]; with:

c confirm each replacement

e when the search pattern fails, do not issue an error message and, in particular, continue in maps as if no error occurred

g replace all occurrences of pattern

i Ignore case for the pattern.

I Don’t ignore case for the pattern.

p print the line containing the last substitute

r Only useful in combination with :& or :S without arguments. :Sr works the same way as :~.

[:range] sno[magic] . . . same as :substitute, but always use nomagic.

[:range] sm[agic] . . . same as :substitute, but always use magic.

Some characters in string have a special meaning:

<table>
<thead>
<tr>
<th>magic</th>
<th>nomagic</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td>replaced with the whole matched pattern</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td>replaced with &amp;</td>
</tr>
<tr>
<td>\0</td>
<td>\0</td>
<td>replaced with the whole matched pattern</td>
</tr>
<tr>
<td>\1</td>
<td>\1</td>
<td>replaced with the matched pattern in the first pair of ()</td>
</tr>
<tr>
<td>\2</td>
<td>\2</td>
<td>replaced with the matched pattern in the second pair of ()</td>
</tr>
<tr>
<td>\9</td>
<td>\9</td>
<td>replaced with the matched pattern in the ninth pair of ()</td>
</tr>
<tr>
<td>magic</td>
<td>nomagic</td>
<td>action</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>replaced with the string of the previous substitute</td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>replaced with ~</td>
</tr>
<tr>
<td>\u</td>
<td>\u</td>
<td>next character made uppercase</td>
</tr>
<tr>
<td>\U</td>
<td>\U</td>
<td>following characters made uppercase</td>
</tr>
<tr>
<td>\l</td>
<td>\l</td>
<td>next character made lowercase</td>
</tr>
<tr>
<td>\L</td>
<td>\L</td>
<td>following characters made lowercase</td>
</tr>
<tr>
<td>\e</td>
<td>\e</td>
<td>end of /u, /U, /l and /L (NOTE: not &lt;Esc&gt;!)</td>
</tr>
<tr>
<td>\E</td>
<td>\E</td>
<td>end of /u, /U, /l and /L</td>
</tr>
<tr>
<td>&lt;CR&gt;</td>
<td>&lt;CR&gt;</td>
<td>split line in two at this point</td>
</tr>
<tr>
<td>\r</td>
<td>\r</td>
<td>idem</td>
</tr>
<tr>
<td>CTRL-V &lt;CR&gt;</td>
<td>CTRL-V &lt;CR&gt;</td>
<td>insert a carriage-return &lt;CTRL-M&gt;</td>
</tr>
</tbody>
</table>

:\[range\] &[c e c r i l] \[n\]  repeat previous :s \[n\] times with new range and options
:\[range\] [range] s[substitute] [c e c r i l] \[n\]  repeat previous :s \[n\] times with same substitute string but with last used search pattern.
&\[range\]  repeat previous :s on current line without options
:\[range\] ret[ab][!] \[tabstop\]  set tabstop to new value and adjust white space accordingly

### 2.10 Visual mode

v  start or stop highlighting characters
V  start or stop highlighting linewise
CTRL-V  start or stop highlighting blockwise
o  exchange cursor position with start of highlighting
gv  start highlighting on previous visual area

**Blockwise operators**

**lstring** With a blockwise selection, lstringEsc will insert string at the start of block on every line of the block, provided that the line extends into the block. TABs are split to retain visual columns.

**Astring** With a blockwise selection, AstringEsc will append string to the end of block on every line of the block. There is some differing behavior where the block RHS is not straight, due to different line lengths.

c  All selected text in the block will be replaced by the same text string. When using c the selected text is deleted and Insert mode started. You can then enter text (without a line break). When you hit Esc, the same string is inserted in all previously selected lines.

C  Like using c, but the selection is extended until the end of the line for all lines.

> or <  The block is shifted by shiftwidth. The RHS of the block is irrelevant. The LHS of the block determines the point from which to apply a right shift, and padding includes TABs optimally according to ts and et. The LHS of the block determines the point up to which to shift left.
Every screen char in the highlighted region is replaced with the same char, i.e. TABs are split and the virtual whitespace is replaced, maintaining screen layout.

**Virtual Replace mode**

Virtual replace mode (enter it with `gR`) is similar to Replace mode, but instead of replacing actual characters in the file, you are replacing screen real estate, so that characters further on in the file never appear to move.

This mode is very useful for editing `<Tab>` separated columns in tables, for entering new data while keeping all the columns aligned.

## 2.11 Text objects (only in Visual mode or after an operator)

- **word**: A `word` consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space (spaces, tabs, `<EOL>`). This can be changed with the `iskeyword` option.

- **WORD**: A `WORD` consists of a sequence of non-blank characters, separated with white space. An empty line is also considered to be a `word` and a `WORD`.

- **sentence**: A `sentence` is defined as ending at a “.”, “!” or “?” followed by either the end of a line, or by a space. Any number of closing “)”, “]”, “"” and “’” characters may appear after the “.”, “!” or “?” before the spaces or end of line. A paragraph and section boundary is also a `sentence` boundary. The definition of a `sentence` cannot be changed.

- **paragraph**: A `paragraph` begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the `paragraphs` option. The default is “IPLPPPQPP LIpplpipbp”, which corresponds to the macros “.IP”, “.LP”, etc. (These are `nroff` macros, so the dot must be in the first column). A section boundary is also a paragraph boundary. Note that this does not include a “{" or “}” in the first column.

- **section**: A `section` begins after a form-feed (<C-L>) in the first column and at each of a set of section macros, specified by the pairs of characters in the `sections` option. The default is “SHNHH HUnhsh”, which defines a `section` to start at the `nroff` macros “.SH”, “.NH”, “.H”, “.HU”, “.rh” and “.sh”.

```
[n] aw  select a word          
[n] iw  select inner word     
[n] aW  select a WORD         
[n] iW  select inner WORD     
[n] as  select a sentence     
[n] is  select inner sentence 
[n] ap  select a paragraph    
[n] ip  select inner paragraph
[n] a[ or a] select [n] “[“] ” blocks.             
[n] i[ or i] select [n] inner “[“] ” blocks.        
[n] a), a( or ab select a block (from ( to ))      
[n] i), i( or ib select inner block (from ( to ))  
[n] a< or a> select [n] <> blocks.                   
```

5“inner” means that white spaces between words are included in count `n`
2.12 Repeating Commands

\[ n \] . repeat last change (with n replaced with \( n \))

q(a-z) \( n \) record typed characters into register \{a-z\}

q(A-Z) \( n \) record typed characters, appended to register \{a-z\}

q \( n \) stop recording

\[ n \] @a-z execute the contents of register \{a-z\} \( n \) times

\[ n \] @@ repeat previous @a-z \( n \) times

:\[ range \] g\{global\}/pattern/\{cmd\} execute Ex command \{cmd\} (default: :p) on the lines within \[ range \] where pattern matches

:\[ range \] g\{global\}/pattern/\{cmd\} execute Ex command \{cmd\} (default: :p) on the lines within \[ range \] where pattern does NOT match

:so\{urce\} file read Ex commands from \{file\}

:so\{urce\}! file read VIM commands from \{file\}

:\[ range \] sl\{ee\}p \[ n \] \[ m \] don’t do anything for \( n \) seconds. If \( m \) is included, sleep for \( n \) milliseconds.

\[ n \] gs goto Sleep for \( n \) seconds

2.13 Undo/Redo Commands

\[ n \] u \( n \) undo last \( n \) changes

\[ n \] CTRL-R \( n \) redo last \( n \) undone changes

U restore last changed line

2.14 Command-line editing

Esc \( n \) abandon command-line (if wildchar is \( n \), type it twice)

CTRL-V char insert char literally

CTRL-V number enter decimal value of character (up to three digits)

CTRL-K char1 char2 \( n \) enter digraph

CTRL-R {0-9a-z"%#:=*} \( n \) insert the contents of a register\( ^6 \)

CTRL-R (CTRL-R) {0-9a-z"%#:=*} \( n \) insert the contents of a register. Works like using a single \( n \), but the text is inserted literally, not as if typed. This differs when the register contains characters like <BS>.

← / → cursor left/right

SHIFT ← / SHIFT → cursor one word left/right

CTRL-B / CTRL-E cursor to beginning/end of command-line

BS delete the character in front of the cursor

Del delete the character under the cursor

CTRL-W delete the word in front of the cursor

\(^6\)See Section 13.2 for description of VIM registers
3 Key Mappings Abbreviations

3.1 Key mapping

These commands are used to map a key or key sequence to a string of characters. There are five sets of mappings:

Normal mode: when typing commands. (e.g. :map <F3> o#include)

2.15 Encryption

Vim is able to write files encrypted, and read them back. The encrypted text cannot be read without the right key. The normal way to work with encryption, is to use the :X command, which will ask you to enter a key. A following write command will use that key to encrypt the file. If you later edit the same file, Vim will ask you to enter a key. The algorithm used is breakable.

Warning: The swapfile and text in memory are not encrypted. A system administrator will be able to see your text while you are editing it. Text you copy or delete goes to the numbered registers. The registers can be saved in the “.viminfo” file, where they could be read. Change your viminfo option to be safe. If you make a typo when entering the key and then write the file and exit, the text will be lost!

:X Prompt for an encryption key. The typing is done without showing the actual text, so that someone looking at the display won’t see it. The typed key is stored in the key option, which is used to encrypt the file when it is written.

CTRL-U
remove all characters

SHIFT-↑/SHIFT-↓ recall older/newer command-line from history

:his[ory][name] [first] [last] List the contents of history name which can be:

- c[md] or : command-line history
- s[earch] or / search string history
- e[xpr] or = expression register history
- i[nput] or @ input line history
- a[ll] all of the above

Context-sensitive completion on the command-line: ☝️

If there are multiple matches, beep and show the first one; further wildchar will show the next ones

wildchar

CTRL-A insert all names that match pattern in front of cursor
CTRL-D list all names that match the pattern in front of the cursor
CTRL-L insert longest common part of names that match pattern
after wildchar with multiple matches: go to next match
CTRL-P after wildchar with multiple matches: go to previous match
CTRL-R ... insert the object under the cursor:

CTRL-F the “filename” under the cursor
CTRL-P the “filename” under the cursor, expanded with path
CTRL-W the “word” under the cursor
CTRL-A the “WORD” under the cursor

CTRL-F
Visual mode: when typing commands while the Visual area is highlighted.
Operator-pending mode: when an operator is pending (after “d”, “y”, “c”, etc.).
Insert mode: these are also used in Replace mode.
Command-line mode: when entering a “;” or “/” command.

Everything from the first non-blank after lhs up to the end of the line \(<\text{EOL}>)\) (or “|”) is considered to be part of rhs. Inclusion of lhs in rhs results in a recursive mapping. Recursion depth is controlled by maxmapdepth option. Use “nore” versions of mapping commands to avoid recursion.

:\(\text{map}\) lhs rhs \(\text{map}\) lhs to rhs in Normal and Visual mode
:\(\text{map}!\) lhs rhs \(\text{map}\) lhs to rhs in Insert and Command-line mode
:\(\text{no[remap][!]\hspace{1em}}\text{lhs rhs}\) same as :map, no remapping for is rhs
:\(\text{unm[ap]\hspace{1em}}\text{lhs}\) remove the mapping of lhs for Normal and Visual mode
:\(\text{unm[ap]!\hspace{1em}}\text{lhs}\) remove the mapping of lhs for Insert and Command-line mode
:\(\text{map[!\hspace{1em}}\text{[lhs]}\) list mappings (starting with lhs) for Normal and Visual mode
:\(\text{map[!\hspace{1em}} [\text{lhs}]\) list mappings (starting with lhs) for Insert and Command-line mode
:\(\text{cmap}:/\text{cunmap}:/\text{cnoremap}\) like :map!/unmap!/noremap! but for Command-line mode only
:\(\text{imap}/\text{inmap}/\text{inoremap}\) like :map!/unmap!/noremap! but for Insert mode only
:\(\text{nmap}/\text{unmap}/\text{nnoremap}\) like :map!/unmap!/noremap but for Normal mode only
:\(\text{vmap}/\text{unmap}/\text{vnoremap}\) like :map!/unmap!/noremap but for Visual mode only
:\(\text{omap}/\text{unmap}/\text{onoremap}\) like :map!/unmap!/noremap but only for when an operator is pending
:\(\text{mk[exrc][!]\hspace{1em}}\text{file}\) write current mappings, abbreviations, and settings to file
:\(\text{mkv[imrc][!]\hspace{1em}}\text{file}\) same as :mkexrc, but with default .vimrc
:\(\text{mks[ession][!]\hspace{1em}}\text{[file]}\) like :mkvimrc, but store current files and directory too
:\(\text{mapc[lear]}\) remove mappings for Normal and Visual mode
:\(\text{mapc[lear]}!\) remove mappings for Insert and Cmdline mode
:\(\text{imapc[lear]}\) remove mappings for Insert mode
:\(\text{vmapc[lear]}\) remove mappings for Visual mode
:\(\text{omapc[lear]}\) remove mappings for Operator-pending mode
:\(\text{nmapc[lear]}\) remove mappings for Normal mode
:\(\text{cmapc[lear]}\) remove mappings for Cmdline mode

### 3.2 Abbreviations

Abbreviations are used in Insert, Replace and Command-line modes. Abbreviations are never recursive. There are three types of abbreviations:

- **full-id** this type consists entirely of keyword characters (letters and characters from iskeyword option). (e.g. \(\text{foo}, \text{g3}, -1\))
- **end-id** this type ends in a keyword character, but all the other characters are not keyword characters. (e.g. \(#i, \ldots f, $/7\))
- **non-id** this type ends in a non-keyword character, the other characters may be of any type, excluding \(<\text{Space}>\) and \(<\text{Tab}>\). (e.g. \(\text{def#}, 4/7\$\))

:\(\text{ab[breviate]}\hspace{1em}\text{lhs rhs}\) add abbreviation for lhs to rhs
### 3.3 User-defined commands

It is possible to define your own Ex commands. A user-defined command can act just like a built-in command, except that when the command is executed, it is transformed into a normal Ex command and then executed. All user-defined commands must start with an uppercase letter, to avoid confusion with built-in commands. User-defined commands can have arguments, which are subject to completion as filenames, buffers, etc. Exactly how this works depends upon the command’s attributes, which are specified when the command is defined.

- **:com[mand]** list all user-defined commands. When listing commands, the characters in the first two columns are:
  - ! Command has the -bang attribute
  - " Command has the -register attribute
- **:com[mand]** cmd list the user-defined commands that start with cmd
- **:com[mand][!] attr ...** cmd rep define a user command. The name of the command is cmd and its replacement text is rep. The command’s attributes (see below) are attr. If the command already exists, an error is reported, unless a ! is specified, when the command is redefined.
- **:delc[ommand]** cmd delete the user-defined command cmd.
- **:comc[lear]** delete all user-defined commands.

**Command attributes**

Command attributes split into four categories:

1. **argument handling**: -nargs=char, where char can be 0,1, *, ?, or +
2. **completion behaviour**: -complete=word, where word can be any of the following: au-group buffer command dir event file help highlight menu option tag
3. **range handling**: -range=n,% or -count=n
4. **special cases**: -bang – the command can take a ! modifier, -register – the first argument to the command can be an optional register name

**Replacement text**

The replacement text for a user-defined command is scanned for special escape sequences, using <...> notation. Escape sequences are replaced with values from the entered command line, and all other text is copied unchanged. The resulting string is executed as an Ex command. The valid escape sequences are
4 OPTIONS

4.1 Setting Options

`:set` show all modified options

`:set all` show all options

`:set option` toggle `option` on, show string or number option

`:set no option` toggle `option` off

`:set inv option` invert `option`

`:set option=value` set string or number `option` to `value`

`:set option?` show `value` of `option`

`:set option+=value` Add the `value` to a number option, or concatenate the `value` to a string option.

`:set option^=value` Multiply the `value` to a number option, or prepend the `value` to a string option.

`:set option-=value` Subtract the `value` from a number option, or remove the `value` from a string option, if it is there.

`:set option&` reset `option` to its default `value`

`:fix[del]` set `value` of “t_kD” according to `value` of “t_kb”

4.2 Short explanation of each option:

`in ()` – an abbreviated version

- `aleph (al)` ASCII code of the letter Aleph (Hebrew)
- `allowrevins (ari)` Allow CTRL-_ in Insert and Command-line mode. See `revins`
- `altkeymap (akm)` for default second language (Farsi/Hebrew)
- `autoindent (ai)` take indent for new line from previous line
- `autowrite (aw)` automatically write file if changed
- `background (bg)` “dark” or “light”, used for highlight colors
- `backspace (bs)` how backspace works at start of line
backup (bk) keep backup file after overwriting a file
backupdir (bdir) list of directories for the backup file
backupext (bex) extension used for the backup file
binary (bin) edit binary file mode
bioskey (biosk) MS-DOS: use bios calls for input characters
breakat (brk) characters that may cause a line break
browsedir (bsdir) (only for GUI) which directory to start browsing in
cindent (cin) do C program indenting
cinkeys (cink) keys that trigger indent when cindent is set
cinoptions (cino) how to do indenting when cindent is set
cinwords (cinw) words where si and cin add an indent
cmdheight (ch) number of lines to use for the command-line
columns (co) number of columns in the display
comments (com) patterns that can start a comment line
compatible (cp) behave Vi-compatible as much as possible
complete (cpt) specify how Insert mode completion works
confirm (cf) confirm certain operations that would normally fail because of unsaved changes to a buffer
conskey (consk) get keys directly from console (MS-DOS only)
cpoptions (cpo) flags for Vi-compatible behaviour
cscopeprg (csprg) command to execute cscope
c scopetag (cst) use cscope for tag commands
c scopetagorder (csto) determines :cstag search order
c scopeverbose (csverb) give messages when adding a cscope database
define (def) pattern to be used to find a macro definition
dictionary (dict) list of file names used for keyword completion
digraph (dg) enable the entering of digraphs in Insert mode
directory (dir) list of directory names for the swap file
display (dy) list of flags for how to display text
d compatible (ed) toggle flags of :substitute command
d ead (ed) endofline (eol) write <EO L> for last line in file
equalalways (ea) windows are automatically made the same size
equalprg (ep) external program to use for = command
erробells (eb) ring the bell for error messages
errorfile (ef) name of the “errorfile” for the QuickFix mode
errorformat (efm) description of the lines in the error file
esckeys (ek) recognize function keys in Insert mode
eventignore (ei) a list of autocommand event names, which are to be ignored
expandtab (et) use spaces when [Tab] is inserted
exrc (ex) read .vimrc and .exrc in the current directory
fileencoding (fe) file encoding for multi-byte text
fileformat (ff) file format used for file I/O
fileformats (ffs) automatically detected values for fileformat
filetype (ft) type of file, used for autocommands
fkmap (fk) Farsi keyboard mapping
formatoptions (fo) ☼ how automatic formatting is to be done
formatprg (fp) ☼ name of external program used with :gq command
gdefault (gd) ☼ the :substitute flag g is default on
greppformat (gfm) ☼ format of grepprog output
grepprg (gp) ☼ program to use for :grep
guicursor (gcr) ☼ GUI: settings for cursor shape and blinking
guifont (gfn) ☼ GUI: Name(s) of font(s) to be used
guifontset (gfs) ☼ GUI: Names of multi-byte fonts to be used
guheadroom (ghr) ☼ GUI: pixels room for window decorations
guioptions (go) ☼ GUI: Which components and options are used
guipy ☼ GUI: try to use a pseudo-tty for :! commands
helpfile (hf) ☼ name of this help file
helpheight (hh) ☼ minimum height of a new help window
hidden (hid) ☼ don’t unload buffer when it is abandoned
highlight (hl) ☼ sets highlighting mode for various occasions
hlsearch (hls) ☼ highlight matches with last search pattern
history (hi) ☼ number of command-lines that are remembered
hkmapp (hkp) ☼ Hebrew keyboard mapping
icon ☼ set icon of the window to the name of the file
iconstring ☼ string to use for the VIM icon
ignorecase (ic) ☼ ignore case in search patterns
include (inc) ☼ pattern to be used to find an include file
incsearch (is) ☼ highlight match while typing search pattern
infercase (inf) ☼ adjust case of match for keyword completion
insertmode (im) ☼ start the edit of a file in Insert mode
isfname (isf) ☼ characters included in file names and pathnames
isident (isi) ☼ characters included in identifiers
isprint (isp) ☼ printable characters
iskeyword (isk) ☼ characters included in keywords
joinspaces (js) ☼ two spaces after a period with a join command
key ☼ encryption key
keymodel (km) ☼ enable starting/stopping selection with keys
keywordprg (kp) ☼ program to use for the K command
langmap (lmap) ☼ alphabetic characters for other language mode
laststatus (ls) ☼ tells when last window has status lines
lazyredraw (l2) ☼ don’t redraw while executing macros
linebreak (lbr) ☼ wrap long lines at a blank
lines ☼ number of lines in the display
lisp ☼ automatic indenting for Lisp
list ☼ show [Tab] and <EOL>
listchars (lcs) ☼ characters for displaying in list mode
magic ☼ changes special characters in search patterns
makeef (mef) ☼ name of the errorfile for :make
makeprg (mp) ☼ program to use for the :make command
matchpairs (mps) ☼ pairs of characters that “%” can match
matchtime \( (\text{mat}) \) \( \odot \) tenths of a second to show the matching parenthesis, when \texttt{showmatch} is set

\textit{maxfuncdepth} \( (\text{mfd}) \) \( \odot \) maximum recursive depth for user functions

\textit{maxmapdepth} \( (\text{mmd}) \) \( \odot \) maximum recursive depth for mapping

\textit{maxmem} \( (\text{mm}) \) \( \odot \) maximum memory (in Kbyte) used for one buffer

\textit{maxmemtot} \( (\text{mmt}) \) \( \odot \) maximum memory (in Kbyte) used for all buffers

\textit{modeline} \( (\text{ml}) \) \( \odot \) recognize modelines at start or end of file

\textit{modelines} \( (\text{mls}) \) \( \odot \) number of lines checked for modelines

\textit{modified} \( (\text{mod}) \) \( \odot \) buffer has been modified

\textit{more} \( \odot \) pause listings when the whole screen is filled

\textit{mouse} \( \odot \) enable the use of mouse clicks

\textit{mousefocus} \( (\text{mousef}) \) \( \odot \) keyboard focus follows the mouse

\textit{mousehide} \( (\text{mh}) \) \( \odot \) hide mouse pointer while typing

\textit{mousemodel} \( (\text{mousem}) \) \( \odot \) changes meaning of mouse buttons

\textit{mousetime} \( (\text{mouset}) \) \( \odot \) max time between mouse double-click

\textit{nrformats} \( (\text{nf}) \) \( \odot \) number formats recognized for \texttt{CTRL-A} command

\textit{number} \( (\text{nu}) \) \( \odot \) print the line number in front of each line

\textit{osfiletype} \( (\text{oft}) \) \( \odot \) operating system-specific filetype information

\textit{paragraphs} \( (\text{para}) \) \( \odot \) nroff macros that separate paragraphs

\textit{paste} \( \odot \) allow pasting text

\textit{pastetoggle} \( (\text{pt}) \) \( \odot \) key code that causes \texttt{paste} to toggle

\textit{patchmode} \( (\text{pm}) \) \( \odot \) keep the oldest version of a file

\textit{path} \( (\text{pa}) \) \( \odot \) list of directories searched with \texttt{gf} \textit{et al}

\textit{previewheight} \( (\text{pvh}) \) \( \odot \) height of the preview window

\textit{readonly} \( (\text{ro}) \) \( \odot \) disallow writing the buffer

\textit{remap} \( \odot \) allow mappings to work recursively

\textit{report} \( \odot \) threshold for reporting number of lines changed

\textit{restorescreen} \( (\text{rs}) \) \( \odot \) Win32: restore screen when exiting

\textit{revins} \( (\text{ri}) \) \( \odot \) inserting characters will work backwards

\textit{rightleft} \( (\text{rl}) \) \( \odot \) window is right-to-left oriented

\textit{ruler} \( (\text{ru}) \) \( \odot \) show cursor line and column in the status line

\textit{rulerformat} \( (\text{ruf}) \) \( \odot \) custom format for the ruler

\textit{scroll} \( (\text{scr}) \) \( \odot \) lines to scroll with \texttt{CTRL-U} and \texttt{CTRL-D}

\textit{scrollbind} \( (\text{scb}) \) \( \odot \) scroll in window as other windows scroll

\textit{scrolljump} \( (\text{sj}) \) \( \odot \) minimum number of lines to scroll

\textit{scrolloff} \( (\text{so}) \) \( \odot \) minimum number of lines above and below cursor

\textit{scrollopt} \( (\text{sbo}) \) \( \odot \) how \texttt{scrollbind} should behave

\textit{sections} \( (\text{sect}) \) \( \odot \) nroff macros that separate sections

\textit{secure} \( \odot \) secure mode for reading \texttt{.vimrc} in current dir

\textit{selection} \( (\text{sel}) \) \( \odot \) what type of selection to use

\textit{selectmode} \( (\text{slm}) \) \( \odot \) when to use Select mode instead of Visual mode

\textit{sessionoptions} \( (\text{ssop}) \) \( \odot \) options for \texttt{:mksession}

\textit{shell} \( (\text{sh}) \) name of shell to use for external commands

\textit{shellcmdflag} \( (\text{shcf}) \) \( \odot \) flag to shell to execute one command

\textit{shellpipe} \( (\text{sp}) \) \( \odot \) string to put output of \texttt{:make} in error file
shellquote (shq) .quote character(s) for around shell command
shellredir (srr) .string to put output of filter in a temp file
shellslash (ssl) .use forward slash for shell file names
shellytype (st)  Amiga: influences how to use a shell
shellxquote (sxq)  like shellquote, but include redirection
shiftround (sr)  round indent to multiple of shiftwidth
shiftwidth (sw)  number of spaces to use for (auto)indent step
shortmess (shm)  list of flags, reduce length of messages
shortname (sn)  non-MS-DOS: Filenames assumed to be 8.3 chars
showbreak (sbr)  string to use at the start of wrapped lines
showcmd (sc)  show (partial) command in status line
showfulltag (sft)  show full tag pattern when completing tag
showmatch (sm)  briefly jump to matching bracket if insert one
sidescroll (ss)  minimum number of columns to scroll horizontal
smartcase (scs)  no ignore case when pattern has uppercase
smartindent (si)  smart autoindenting for C programs
smarttab (sta)  use shiftwidth when inserting Tab
softtabstop (sts)  number of spaces that Tab uses while editing
splitbelow (sb)  new window from split is below the current one
startofline (sol)  commands move cursor to first blank in line
statusline (stl)  custom format for the status line
suffixes (su)  suffixes that are ignored with multiple match
swapfile (swf)  whether to use a swapfile for a buffer
swapsync (sws)  how to sync the swap file
switchbuf (swb)  sets behavior when switching to another buffer
syntax (syn)  syntax to be loaded for current buffer
tabstop (ts)  number of spaces that Tab in file uses
tagsearch (tbs)  use binary searching in tags files
taglength (tl)  number of significant characters for a tag
tagrelative (tr)  file names in tag file are relative
tags (tag)  list of file names used by the tag command
tagstack (tgst)  push tags onto the tag stack
term  name of the terminal
terse  shorten some messages
textauto (ta)  obsolete, use fileformats
textmode (tx)  obsolete, use fileformat
textwidth (tw)  maximum width of text that is being inserted
tildeop (top)  tilde command ~ behaves like an operator
timeout (to)  time-out on mappings and key codes
timeoutlen (tm)  time-out time in milliseconds
title  set title in window to the name of the file
titlelen (tm)  gives the percentage of “columns” to use for the length of the window title
titleold  old title, restored when exiting
5 OTHER COMMANDS

5.1 Shell Commands

:sh[ell] start a shell
:!! command execute command with a shell
K lookup keyword under the cursor with keywordprg program (default: “man”)
5.2 QuickFix Commands

VIM has a special mode to speedup the edit-compile-edit cycle. The idea is to save the error messages from the compiler in a file and use VIM to jump to the errors one by one. The errorformat option should be set to match the error messages from your compiler (see below).

```
:cc[!] [num]  display error num (default is the same again). Without “!” this doesn’t work when jumping to another buffer, the current buffer has been changed, there is the only window for the buffer and both hidden and autowrite are off. When jumping to another buffer with “!” any changes to the current buffer are lost, unless hidden is set or there is another window for this buffer.
:[n] cn[ext][!] display the n next error in the list that includes a file name. If there are no file names at all, go to the n next error. See :cc for “!”.
:[n] cp[revious][!] display the n previous error in the list that includes a file name. If there are no file names at all, go to the n previous error.
:[n] cnf[ile][!] display the first error in the n next file in the list that includes a file name. If there are no file names at all or if there is no next file, go to the n next error.
:cl[ist] [from] [, [to]] list all errors that include a filename
:cl[ist]! list all errors
:cf read errors from the file “errorfile”
:cr[ewind][!] [nr] display error [nr]. If [nr] is omitted, the FIRST error is displayed.
:cla[st]! [nr] display error [nr]. If [nr] is omitted, the LAST error is displayed.
:cq quit without writing and return error code (to the compiler)
:make [args] start make, read errors, and jump to first error
:gr[ep] [args] execute grepprg to find matches and jump to the first one.
:col[der] [n] go to older error list [n times].
:cnew[er] [n] go to newer error list [n times].
```

**Errorformat option syntax**

<table>
<thead>
<tr>
<th>Spec</th>
<th>Description</th>
<th>Spec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>column number (a number)</td>
<td>%f</td>
<td>file name (a string)</td>
</tr>
<tr>
<td>%l</td>
<td>line number (a number)</td>
<td>%m</td>
<td>error message (a string)</td>
</tr>
<tr>
<td>%n</td>
<td>error number (a number)</td>
<td>%r</td>
<td>matches the rest of a single-line</td>
</tr>
<tr>
<td></td>
<td>error type (single character)</td>
<td>%*&lt;conv&gt;</td>
<td>any scanf non-assignable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>conversion</td>
</tr>
<tr>
<td>%%</td>
<td>the single “%” character</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3 Viminfo Commands

The viminfo_file is used to store:
- The command line history.
- The search string history.
- The input-line history.
- Contents of registers.
- Marks for several files.
- File marks, pointing to locations in files.
- Last search/substitute pattern (for “*” and “&”).
• The buffer list.
• Global variables.

```
5 OTHER COMMANDS

viminfo_file
:rv[iminfo] file
:rv[iminfo]! file
:wv[iminfo] file
:wv[iminfo]! file

Viminfo option syntax

The format of "viminfo" string: char string or char number, where char can be:

' – maximum number of previously edited files for which the marks are remembered.
f – whether file marks need to be stored. If zero, file marks ('0 to '9, 'A to 'Z) are not stored. When not present or when non-zero, they are all stored.
r – removable media. The argument is a string (up to the next ";"). This parameter can be given several times. Each specifies the start of a path for which no marks will be stored. Maximum length of each "r" argument is 50 characters.
" – maximum number of lines saved for each register. If zero then registers are not saved. When not included, all lines are saved.
: – maximum number of items in the command line history to be saved. When not included, the value of history is used.
/ – maximum number of items in the search pattern history to be saved. If non-zero, then the previous search and substitute patterns are also saved. When not included, the value of history is used.
n – name of the viminfo file. The name must immediately follow the "n". Must be the last one! If the "-i" argument was given when starting Vim, that file name overrides the one given here with viminfo. Environment variables are expanded when opening the file, not when setting the option.
% – save and restore the buffer list. If Vim is started with a file name argument, the buffer list is not restored. If Vim is started without a file name argument, the buffer list is restored from the viminfo file. Buffers without a file name and buffers for help files are not written to the viminfo file.
```

Automatic option setting when editing a file

```
vim: set-arg: ..
```
in the first and last lines of the file (see ml option), set-arg is given as an argument to :set

5.4 Various Commands

```
:h[elp] ☺ split the window and display the help file in read-only mode. If there is a help window open already, use that one.
:h[elp] subject
Like :help, additionally jump to the tag subject. subject can include wildcards like "*", "?" and " [a-z] "
CTRL-L     clear and redraw the screen
CTRL-G     show current file name (with path) and cursor position
```

6.1 Ranges

, separates two line numbers
; idem, set cursor to the first line number before interpreting the second one

number an absolute line number
. the current line
$ the last line in the file
% equal to 1,$ (the entire file)
* equal to ’<,’ (visual area)
’t position of mark t
/pattern[/] the next line where pattern matches
?q[?pattern] the previous line where pattern matches
+num add num to the preceding line number (default: 1)
-num subtract num from the preceding line number (default: 1)

6.2 Special Ex characters

| separates two commands (not for :global and :)" begins comment
% current file name (only where a file name is expected)
#number alternate file name number (only where a file name is expected)

Note: The next six are typed literally; these are not special keys!
<cword> word under the cursor (only where a file name is expected)
<cWORD> WORD under the cursor (only where a file name is expected)
<cfile> filename under the cursor (only where a file name is expected)
<afile> filename for autocommand (only where a file name is expected)
when executing autocommands, is replaced with the currently effective buffer number.

<amatch> when executing autocommands, is replaced with the match for which this autocommand was executed.

<sfile> filename of a `source`d file, within that file (only where a file name is expected)

After %, #, <cfile>, <sfile> or <afile>

: p full path  : h head (file name removed)
:t tail (file name only)  : r root (extension removed)
:e extension  : . reduce file name to be relative to
current directory, if possible
: ∼ reduce file name to be relative to
the home directory, if possible  : s/pat/sub/ substitute pat with sub

6.3 Pattern searches

[n] / pattern / offset [Ret]  search forward for the n-th occurrence of pattern
[n] [Ret] repeat last search, in the forward direction
[n] ?[Ret] repeat last search, in the backward direction
[n] n  repeat last search
[n] N  repeat last search, in opposite direction
[n] *  search forward for the identifier under the cursor
[n] #  search backward for the identifier under the cursor
[n] g*  like *, but also find partial matches
[n] g#  like #, but also find partial matches
gd  goto local declaration of identifier under the cursor
gD  goto global declaration of identifier under the cursor

6.4 Special characters in search patterns

<table>
<thead>
<tr>
<th>magic</th>
<th>nomagic</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td>matches any single character</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
<td>at beginning of pattern or after “\” or “(“ , matches start of line; at other positions, matches literal “^”</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
<td>at any position, matches literal “^”</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>matches &lt;EOL&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>matches start of word</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>matches end of word</td>
</tr>
<tr>
<td>[a-z]</td>
<td>[a-z]</td>
<td>matches a single char from the range</td>
</tr>
<tr>
<td>[^a-z]</td>
<td>[^a-z]</td>
<td>matches a single char not in the range</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
<td>matches an identifier char ⊇</td>
</tr>
<tr>
<td>l</td>
<td>l</td>
<td>idem but excluding digits ⊇</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>matches a keyword character ⊇</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>idem but excluding digits ⊇</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>matches a file name character ⊇</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>idem but excluding digits ⊇</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>matches a printable character ⊇</td>
</tr>
</tbody>
</table>
### Magic vs. Nomagic

<table>
<thead>
<tr>
<th>Magic</th>
<th>Nomagic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\P</td>
<td>\P</td>
<td>idem but excluding digits</td>
</tr>
<tr>
<td>\s</td>
<td>\s</td>
<td>matches a white space character</td>
</tr>
<tr>
<td>\S</td>
<td>\S</td>
<td>matches a non-white space character</td>
</tr>
<tr>
<td>\d</td>
<td>\d</td>
<td>digit: [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>\D</td>
<td>non-digit: [^0-9]</td>
</tr>
<tr>
<td>\x</td>
<td>\x</td>
<td>hex digit: [0-9A-Fa-f]</td>
</tr>
<tr>
<td>\X</td>
<td>\X</td>
<td>non-hex digit: [^0-9A-Fa-f]</td>
</tr>
<tr>
<td>\o</td>
<td>\o</td>
<td>octal digit: [0-7]</td>
</tr>
<tr>
<td>\O</td>
<td>\O</td>
<td>non-octal digit: [^0-7]</td>
</tr>
<tr>
<td>\W</td>
<td>\W</td>
<td>word character: [0-9A-Za-z_]</td>
</tr>
<tr>
<td>\w</td>
<td>\w</td>
<td>matches 0 or more of the preceding atom</td>
</tr>
<tr>
<td>\W</td>
<td>\W</td>
<td>non-word character: [^0-9A-Za-z_]</td>
</tr>
<tr>
<td>\h</td>
<td>\h</td>
<td>head-of-word character: [A-Za-z_]</td>
</tr>
<tr>
<td>\H</td>
<td>\H</td>
<td>non-head-of-word character: [^A-Za-z_]</td>
</tr>
<tr>
<td>\a</td>
<td>\a</td>
<td>alphabetic character: [A-Za-z]</td>
</tr>
<tr>
<td>\A</td>
<td>\A</td>
<td>non-alphabetic character: [^A-Za-z]</td>
</tr>
<tr>
<td>\l</td>
<td>\l</td>
<td>lowercase character: [a-z]</td>
</tr>
<tr>
<td>\L</td>
<td>\L</td>
<td>non-lowercase character: [^a-z]</td>
</tr>
<tr>
<td>\u</td>
<td>\u</td>
<td>uppercase character: [A-Z]</td>
</tr>
<tr>
<td>\U</td>
<td>\U</td>
<td>non-uppercase character: [^A-Z]</td>
</tr>
</tbody>
</table>

**Note:** using the atom is faster than the [] form

- \e matches <Esc>
- \t matches <Tab>
- \r matches <Ret>
- \b matches <BS>
- \| separates two branches
- \(\) group a pattern into an atom
- \~ matches the last given substitute string

1, 2, .., 9 matches the same string that was matched by the first, second .. ninth sub-expression in \( and \).

### Quantifiers

- \* matches 0 or more of the preceding atom
- \+ matches 1 or more of the preceding atom
- \= matches 0 or 1 of the preceding atom
- \{n,m\} matches \(n\) to \(m\) of the preceding atom, as much as possible
- \{n\} matches \(n\) of the preceding atom
- \{n,\} matches at least \(n\) of the preceding atom, as much as possible
- \{,m\} matches 0 to \(m\) of the preceding atom, as much as possible
- \{\} matches the same as *  
- \{-n,m\} matches \(n\) to \(m\) of the preceding atom, as few as possible
- \{-n\} matches \(n\) of the preceding atom
- \{-n,\} matches at least \(n\) of the preceding atom, as few as possible
- \{,-m\} matches 0 to \(m\) of the preceding atom, as few as possible
- \{-\} matches 0 or more of the preceding atom, as few as possible

---

7 see option magic, page 17
Character class expression

A character class expression is evaluated to the set of characters belonging to that character class. The brackets in character class expressions are additional to the brackets delimiting a range. The following character classes are supported:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:alnum:]</td>
<td>letters and digits</td>
<td>[:alpha:]</td>
<td>letters</td>
</tr>
<tr>
<td>[:ascii:]</td>
<td>ASCII characters</td>
<td>[:blank:]</td>
<td>space and tab characters</td>
</tr>
<tr>
<td>[:cntrl:]</td>
<td>control characters</td>
<td>[:digit:]</td>
<td>decimal digits</td>
</tr>
<tr>
<td>[:graph:]</td>
<td>printable characters</td>
<td>[:lower:]</td>
<td>lowercase letters</td>
</tr>
<tr>
<td>[:print:]</td>
<td>printable characters</td>
<td>[:punct:]</td>
<td>punctuation characters</td>
</tr>
<tr>
<td>[:space:]</td>
<td>whitespace characters</td>
<td>[:upper:]</td>
<td>uppercase letters</td>
</tr>
<tr>
<td>[:xdigit:]</td>
<td>hexadecimal digits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.5 Offsets allowed after search command

- `[num]` *num* lines downwards, in column 1
- `+[num]` *num* lines downwards, in column 1
- `–[num]` *num* lines upwards, in column 1
- `e[+num]` *num* characters to the right of the end of the match
- `e[–num]` *num* characters to the left of the end of the match
- `s[+num]` *num* characters to the right of the start of the match
- `s[–num]` *num* characters to the left of the start of the match
- `b[+num]` *num* characters to the right of the start (begin) of the match
- `b[–num]` *num* characters to the left of the start (begin) of the match

;search command  execute search command next

7 Starting, Writing and Quitting Commands

7.1 Starting VIM

- `vim options` start editing with an empty buffer
- `vim options file . . .` start editing one or more files
- `vim options –` read file from stdin
- `vim options –t tag` edit the file associated with tag
- `vim options –q [file]` start editing in QuickFix mode, display the first error
  
  **VIM arguments:**
  
  +/pattern file . . . ☺ put the cursor at the first occurrence of pattern
  + command   execute command after loading the file
  +[num]       put the cursor at line [num] (default: last line)
  ––           end of options, other arguments are file names
  –b ☺         binary mode
  –C ☺         compatible, set the compatible option
  –d device ☺ Amiga: open device to be used as a console
  –e ☺         E± mode, start VIM in E± mode
  –F ☺         Farsi mode (fkmap and rightleft are set)
GUI: foreground process, don’t fork; Amiga: do not restart VIM to open a window

start GUI (also allows other options)

Hebrew mode (hkmap and rightleft are set)

read info from viminfo instead of other files

Lisp mode

modifications not allowed

do not create a swap file

nocompatible, reset the compatible option

open n windows (default: one for each file)

recover aborted edit session

give list of swap files

read-only mode, implies -n

first read commands from the file scriptin

set terminal name

idem, for when starting the GUI

read inits from vimrc instead of other inits

verbose, give informative messages

Vi mode, start Ex in Normal mode

write typed chars to file scriptout (append)

write typed chars to file scriptout (overwrite)

use encryption to read/write files. Will prompt for a key, which is then stored in the key option.

read file from stdin

7.2 Editing a file

:edit edit the current file, unless changes have been made

:edit! edit the current file always. Discard any changes

:edit file edit file, unless changes have been made

:edit! file edit file always. Discard any changes

:ex [+cmd] [file] same as :edit, but also switch to Ex mode.

:find[!] [+cmd] [file] Find file in “$path” and then :edit it. not in Vi

:vi[ual][!] [+cmd] [file] when entered in Ex mode: Leave Ex mode, go back to Normal mode.

:vi[w] [+cmd] file when entered in Ex mode: Leave Ex mode, go back to Normal mode.

edit alternate file n (equivalent to :e n)

gf or f edit the file whose name is under the cursor

print the current directory name

change the current directory to path

back to previous current directory.

print the current file name and the cursor position

set the current file name to name

show alternate file names
7.3 Using the argument list

- `:ar[gs]` print the argument list, with the current `file` in [ ]
- `:all` or `:sall` open a window for every `file` in the arg list
- `:wn[ext][!]` write `file` and edit next `file`
- `:wn[ext][!] file` write to `file` and edit next `file`, unless `file` exists. With `!`, overwrite existing `file`
- `:wN[ext][!] file` write `file` and edit previous `file`

<table>
<thead>
<tr>
<th>in current window:</th>
<th>in new window:</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:argu[ment] [n]</code></td>
<td><code>:sar[ument] n</code></td>
<td>edit <code>file</code> <code>n</code></td>
</tr>
<tr>
<td><code>:n[ext]</code></td>
<td><code>:sn[ext]</code></td>
<td>edit next <code>file</code></td>
</tr>
<tr>
<td><code>:n[ext] arglist</code></td>
<td><code>:sn[ext] arglist</code></td>
<td>define new <code>arglist</code> and edit first <code>file</code></td>
</tr>
<tr>
<td><code>:N[ext]</code></td>
<td><code>:sN[ext]</code></td>
<td>edit previous <code>file</code></td>
</tr>
<tr>
<td><code>:rew[ind][!]</code></td>
<td><code>:srew[ind]</code></td>
<td>edit first <code>file</code></td>
</tr>
<tr>
<td><code>:last</code></td>
<td><code>:slast</code></td>
<td>edit last <code>file</code></td>
</tr>
</tbody>
</table>

7.4 Writing and quitting

- `:[range]w[rite][!]` write to the current file
- `:[range]w[rite] file` write to `file`, unless it already exists
- `:[range]w[rite]! file` write to `file`. Overwrite an existing file
- `:[range]w[rite][!] >>` append to the current file
- `:[range]w[rite][!] >> file` append to `file`
- `:[range]w[rite]! cmd` execute `cmd` with `range` lines as standard input
- `:[range]up[date][!]` write to current file if modified
- `:wall[!]` write all changed buffers
- `:q[uit]` quit current buffer, unless changes have been made. Exit VIM when there are no other non-help buffers
- `:q[uit]!` quit current buffer always, discard any changes. Exit VIM when there are no other non-help buffers
- `:conf[irm] qa[ll]` exit VIM. Bring up a prompt when some buffers have been changed.
- `:qa[ll]` exit VIM, unless changes have been made
- `:qa[ll]!` exit VIM always, discard any changes
- `:cq` quit without writing and return error code
- `:wq[!]` write the current file and exit
- `:wq[!] file` write to `file` and exit
- `:x[it][!] file` like `:wq` but write only when changes have been made
- `ZZ` same as `:x`
- `ZQ` same as `:q!`
- `:conf[irm] wqa[ll]` or `:conf[irm] xa[ll]` write all changed buffers and exit VIM. Bring up a prompt when some buffers are readonly or cannot be written for another reason.
- `:xa[ll][!]` or `:wqa[ll][!]` write all changed buffers and exit
- `:st[op][!]` suspend VIM or start new shell. If `aw` option is set and `[]` not given write the buffer
- `CTRL-Z` same as `:stop!`
8 Windows and Buffers functions

8.1 Multi-window functions

**CTRL-W s** or **:split**  split window into two parts

`:n` **split** or **new** `[+cmd] file`  split window and edit file in one of them. Execute the command `+cmd` when the file has been loaded. Make new window `n` high.

`:n` **split** or **new** `[+cmd] file`  Same as **:split**, but set `readonly` option for this buffer.

`:n` **split** or **new** `[+cmd] file`  Same as **:split**, but search for file in “$path”. Doesn’t split if file is not found.

**CTRL-W ]**  split window and jump to tag under cursor

:tag[!] [tagname]  Does `tag[!] [tagname]` and shows the found tag in a “Preview” window without changing the current buffer or cursor position. If a “Preview” window already exists, it is re-used (like a help window is). If a new one is opened, `previewheight` is used for the height of the window.

**CTRL-W z** or **:pclose[!]**  Close any “Preview” windows currently open. When the `hidden` option is set, or when the buffer was changed and the `![]` is used, the buffer becomes hidden (unless there is another window editing it). The command fails if any “Preview” buffer cannot be closed.

:pop[!]  Does `pop[!]` in the preview window.

**CTRL-W }**  Use identifier under cursor as a tag and perform a :ptag on it. Make the new “Preview” window (if required) N high. If N is not given, `previewheight` is used.

**CTRL-W g }**  Use identifier under cursor as a tag and perform a :ptjump on it. Make the new “Preview” window (if required) N high. If N is not given, `previewheight` is used.

**CTRL-W g ]**  split current window in two. Use identifier under cursor as a tag and perform :tselect on it in the new upper window. Make new window N high.

**CTRL-W g **CTRL-]**  split current window in two. Use identifier under cursor as a tag and perform :tjump on it in the new upper window. Make new window N high.

**CTRL-W f**  split window and edit file name under the cursor

**CTRL-W**  split window and edit alternate file

**CTRL-W n** or **new**  create new empty window

**CTRL-W q** or **:quit**  quit editing and close window

**CTRL-W c** or **:close**  make buffer hidden and close window

**CTRL-W o** or **:only[!]**  make current window only one on the screen

**CTRL-W j**  move cursor to window below

**CTRL-W k**  move cursor to window above

**CTRL-W**  move cursor to window below (wrap)

**CTRL-W W**  move cursor to window above (wrap)

**CTRL-W t**  move cursor to top window

**CTRL-W b**  move cursor to bottom window

**CTRL-W p**  move cursor to previous active window

**CTRL-W r**  rotate windows downwards

**CTRL-W R**  rotate windows upwards

**CTRL-W x**  exchange current window with next one

**CTRL-W =**  make all windows equal height

**CTRL-W –**  decrease current window height
9.2 Buffer list functions

:buffers or :files    list all known buffer and file names
:ball or :sball       edit all args/buffers
:unhide or :sunhide   edit all loaded buffers
:bad[d] [+lnum] fname  add file name fname to the list, without loading it. If lnnum is specified, the cursor will be positioned at that line when the buffer is first entered.
:bunload[!] [n]       unload buffer n from memory
:bd[delete][!] [n] or [n] bd[delete]  unload buffer n and delete it from the buffer list

in current window: in new window: description

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:n] buffer [n]</td>
<td>[:n] sbuffer [n]</td>
<td>to arg/buf n</td>
</tr>
<tr>
<td>[:n] bnext [n]</td>
<td>[:n] sbnext [n]</td>
<td>to n-th next arg/buf</td>
</tr>
<tr>
<td>[:n] bNext [n]</td>
<td>[:n] sbNext [n]</td>
<td>to n-th previous arg/buf</td>
</tr>
<tr>
<td>[:n] bprevious [n]</td>
<td>[:n] sbprevious [n]</td>
<td>to n-th previous arg/buf</td>
</tr>
<tr>
<td>:brewind</td>
<td>:sbrewind</td>
<td>to first arg/buf</td>
</tr>
<tr>
<td>:blast</td>
<td>:sblast</td>
<td>to last arg/buf</td>
</tr>
<tr>
<td>[:n] bmod [n]</td>
<td>[:n] sbmod [n]</td>
<td>to n-th modified buf</td>
</tr>
</tbody>
</table>

9 Script Language

9.1 Variables

VIM supports two types of variables: Number—a 32 bit signed number and String—a NULL terminated string of 8-bit unsigned characters. They are converted automatically, depending on how they are used. For boolean operators Numbers are used. Zero is FALSE, non-zero is TRUE.

A VIM variable name can be made up of letters, digits and underscore (“_”), but it cannot start with a digit. An internal variable is created with the :let and destroyed with the :unlet command. A variable name that is preceded with b: and w: is local to the current buffer and window, respectively. Inside functions global variables are accessed with g:.

Built-in variables

v:count          The count given for the last Normal mode command. Can be used to get the count before a mapping. Read-only.
v:count1         Just like “v:count”, but defaults to one when no count is used.
v:errmsg         Last given error message. This variable may be set.
v:warningmsg     Last given warning message. It’s allowed to set this variable.
v:statusmsg      Last given status message. It’s allowed to set this variable.
v:shell_error    Result of the last shell command. When non-zero, the last shell command had an error. When zero, there was no problem.
v:this_session   Full filename of the last loaded or saved session file. See :mksession.
9.2 Expression syntax

Operators:

<table>
<thead>
<tr>
<th>#</th>
<th>Operator</th>
<th>Description</th>
<th>#</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>[..]</td>
<td>logical OR</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>&amp;&amp; [..]</td>
<td>logical AND</td>
<td>10</td>
<td>!~</td>
<td>regexp doesn’t match</td>
</tr>
<tr>
<td>3</td>
<td>==</td>
<td>equal</td>
<td>11</td>
<td>+</td>
<td>number addition</td>
</tr>
<tr>
<td>4</td>
<td>!=</td>
<td>not equal</td>
<td>12</td>
<td>− [..]</td>
<td>number subtraction</td>
</tr>
<tr>
<td>5</td>
<td>&gt;</td>
<td>greater than</td>
<td>13</td>
<td>. [..]</td>
<td>string concatenation</td>
</tr>
<tr>
<td>6</td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>14</td>
<td>* [..]</td>
<td>number multiplication</td>
</tr>
<tr>
<td>7</td>
<td>&lt;</td>
<td>smaller than</td>
<td>15</td>
<td>/ [..]</td>
<td>number division</td>
</tr>
<tr>
<td>8</td>
<td>&lt;=</td>
<td>smaller than or equal</td>
<td>16</td>
<td>% [..]</td>
<td>number modulo</td>
</tr>
<tr>
<td>17</td>
<td>! expr</td>
<td>logical NOT</td>
<td>18</td>
<td>− expr</td>
<td>unary minus</td>
</tr>
</tbody>
</table>

Description:
- All expressions within one level are parsed from left to right.
- Comparison operators can be appended with # – to “match case” or ? – to “ignore case” of compared expressions.
- The arguments of *, +, −, %, /, || and && operations are (converted to) Numbers. When comparing a String with a Number, the String is converted to a Number, and the comparison is done on Numbers.
- Comparing two Strings is done with strcmp(). This results in the mathematical difference, not necessarily the alphabetical difference in the local language.
- The =~ and !~ operators match the left hand argument with the right hand argument, which is used as a pattern. This matching is always done like magic was set, no matter what the actual value of magic is. The value of ignorecase does matter though. To avoid backslashes in the regexp pattern to be doubled, use a single-quote string.

Operands:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1[expr2]</td>
<td>index in String</td>
<td>number</td>
<td>number constant</td>
</tr>
<tr>
<td>&quot;string&quot;</td>
<td>string constant</td>
<td>’string’</td>
<td>literal string constant</td>
</tr>
<tr>
<td>&amp;option</td>
<td>option value</td>
<td>(expr1)</td>
<td>nested expression</td>
</tr>
<tr>
<td>variable</td>
<td>internal variable</td>
<td>$VAR</td>
<td>environment variable</td>
</tr>
<tr>
<td>@r</td>
<td>contents of register “r”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description:

"numbered in order of increasing precedence
"[..] indicates that the operations in this level can be concatenated.
expr[expr2] This results in a String that contains the expr2’th single character from expr1. expr1 is used as a String, expr2 as a Number. The index starts with 0 (like in C).

Careful: column numbers start with one!

If the length of the String is less than the index, the result is an empty String.

"string"

A string constant may contain these special characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3-digit octal number</td>
<td>..</td>
<td>2-digit octal number</td>
</tr>
<tr>
<td>.</td>
<td>1-digit octal number</td>
<td>\x..</td>
<td>2-character hex number</td>
</tr>
<tr>
<td>\x.</td>
<td>1-character hex number</td>
<td>X..</td>
<td>same as \x..</td>
</tr>
<tr>
<td>\X.</td>
<td>same as \x.</td>
<td>b</td>
<td>backspace</td>
</tr>
<tr>
<td>\e</td>
<td>escape</td>
<td>f</td>
<td>formfeed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
<td>r</td>
<td>return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
<td>&lt;xxx&gt;</td>
<td>Special key name “xxx”</td>
</tr>
</tbody>
</table>

Note that \000 and \x00 force the end of the string.

'string' This string is taken literally. No backslashes are removed or have a special meaning. A literal string cannot contain a single quote. Use a normal string for that.

@r The result is the contents of the named register, as a single string. Newlines are inserted where required. To get the contents of the unnamed register use “@@”. The “=” register can not be used here.

9.3 Functions

argc() The result is the number of files in the argument list. See arglist.
argv(n) The result is the n-th file in the argument list.
browse(save, title, initdir, default) Put up a file requester. This only works only in some GUI versions. The input fields are:
  save when non-zero, select file to write title title for the requester
  initdir directory to start browsing in default default file name
append(lnum, string) Append the text string after line lnum in the current buffer. Lnum can be zero, to insert a line before the first one. Returns 1 for failure (lnum out of range) or 0 for success.
bufexists(var) The result is a Number, which is non-zero if a buffer called var exists. If the var argument is a string, it must match a buffer name exactly. If the var argument is a number, buffer numbers are used. Use buffer_exists(0) to test for the existence of an alternate file name.
bufloaded(expr) The result is a Number, which is non-zero if a buffer called expr exists and is loaded (shown in a window or hidden). The expr argument is used like with bufexists().
bufname(expr) The result is the name of a buffer, as it is displayed by the :ls command. If expr is a Number, that buffer number’s name is given. If expr is a String, it is used as a regexp pattern to match with the buffer names.

10must be followed by non-digit
12must be followed by non-hex
bufnr(expr)  The result is the number of a buffer, as it is displayed by the :ls command.
bufwinr(expr)  The result is a Number, which is the number of the first window associated
              with buffer expr. For the use of expr, see bufname() above. If buffer expr
doesn’t exist or there is no such window, -1 is returned.
byte2line(byte)  Return the line number that contains the character at byte count byte in the
current buffer. This includes the end-of-line character, depending on the ‘file-
format’ option for the current buffer. The first character has byte count one.
char2nr(expr)  Return ASCII value of the first char in expr.
col(expr)  The result is a Number, which is the column of the file position given with
epr. The accepted positions are:
  .  the cursor position
  ‘x  position of mark “x” (if the mark is not set, 0 is returned).

Only marks in the current file can be used. The first column is 1. 0 is returned
for an error.
confirm(msg, choices[, default [, type ]])  msg is displayed in a dialog with choices as the
alternatives. default is the number of the choice that is made if the user hits
CR. If default is omitted, 0 is used. The optional type argument gives the
type of dialog.
delete(fname)  Deletes the file by the name fname. The result is a Number, which is 0 if the
file was deleted successfully, and non-zero when the deletion failed.
did_filetype()  Returns non-zero when autocommands are being executed and the FileType
event has been triggered at least once. Can be used to avoid triggering the
FileType event again in the scripts that detect the file type.
escape(string, chars)  Escape the characters in chars that occur in string with a backslash.
exists(expr)  The result is a Number, which is 1 if var is defined, zero otherwise. The expr
argument is a string, which contains one of these:
  &option-name  VIM option
  $ENVNAME     environment variable
  varname      internal variable.
expand(expr [, flag])  Expand the file wildcards in expr. The result is a String. When the result
of expr starts with %, # or <, the expansion is done like for the cmdline-
special variables with their associated modifiers. There cannot be a white
space between the variables and the following modifier. When the current or
alternate file name is not defined, % or # use an empty string. Using %:p in a
buffer with no name results in the current directory, with a “/” added.
filereadable(fname)  The result is a Number, which is TRUE when a file with the name fname
exists, and can be read. If fname doesn’t exist, or is a directory, the result is
FALSE. fname is any expression, which is used as a String.
fnamemodify(fname, mods)  Modify file name fname according to mods. mods is a string of
characters like it is used for file names on the command line.
getcwd()  The result is a String, which is the name of the current working directory.
getftime(fname)  The result is a Number, which is the last modification time of the given file
fname. The value is measured as seconds since 1st Jan 1970, and may be
passed to strftime().
getline(lnum)  The result is a String, which is line lnum from the current buffer.
getwinposx()  The result is a Number, which is the X coordinate in pixels of the left hand
side of the GUI vim window. The result will be -1 if the information is not
available.
getwinposy()  The result is a Number, which is the Y coordinate in pixels of the top of the GUI vim window. The result will be -1 if the information is not available.

glob(expr)  Expand the file wildcards in expr. The result is a String. When there are several matches, they are separated by [NL] characters. If the expansion fails, the result is an empty string.

has(feature)  The result is a Number, which is 1 if the feature is supported, zero otherwise. The feature argument is a string. See Feature-list below.

hostname()  The result is a String, which is the name of the machine on which VIM is currently running. Machine names greater than 256 characters long are truncated.

histadd(history, item)  Add the String item to the history history which can be one of:
- cmd or : command line history
- search or / search pattern history
- expr or = typed expression history
- input or @ input line history

If item does already exist in the history, it will be shifted to become the newest entry. The result is a Number: 1 if the operation was successful, otherwise 0 is returned.

histdel(history [, item])  Clear history, i.e. delete all its entries. If the parameter item is given as String, this is seen as regular expression. All entries matching that expression will be removed from the history (if there are any). If item is a Number, it will be interpreted as index. The respective entry will be removed if it exists. The result is a Number: 1 for a successful operation, otherwise 0 is returned.

histget(history [, index])  The result is a String, the entry with Number index from history. See hist-names for the possible values of history, and :history-indexing for index. If there is no such entry, an empty String is returned. When index is omitted, the most recent item from the history is used.

histnr(history)  The result is the Number of the current entry in history. See hist-names for the possible values of history. If an error occurred, -1 is returned.

hlexists(name)  The result is a Number, which is non-zero if a highlight group called name exists. The group may have been defined as a highlight group or as a syntax item or both. Not necessarily when highlighting has been defined for it, it may also have been used for a syntax item.

hIID(name)  The result is a Number, which is the ID of the highlight group with name name. When the highlight group doesn’t exist, zero is returned.

input(prompt)  The result is a String, which is whatever the user typed on the command-line. The parameter is either a prompt string, or a blank string (for no prompt). A \n can be used in the prompt to start a new line.

isdirectory(directory)  The result is a Number, which is TRUE when a directory with the name directory exists. If directory doesn’t exist, or isn’t a directory, the result is FALSE. directory is any expression used as a String.

libcall(libname, funcname, argument)  Call function funcname in the run-time library libname with argument argument. The result is the String returned. If argument is a number, it is passed to the function as an int; if param is a string, it is passed as a null-terminated string. If the function returns NULL, this will appear as an empty string to Vim. WARNING: If the function returns a non-valid pointer, Vim will crash! This also happens if the function returns a number. For Win32 systems, libname should be the filename of the DLL without the “.dll” suffix. A full path is only required if the DLL is not in the usual places.
line(expr)  The result is a Number, which is the line number of the file position given with expr. The accepted positions are:

- the cursor position
- \$ the last line in the current buffer
- ‘x’ position of mark “x” (if the mark is not set, 0 is returned)

Only marks in the current file can be used.

line2byte(lnum)  Return the byte count from the start of the buffer for line lnum. This includes the end-of-line character, depending on the ‘fileformat’ option for the current buffer. The first line returns 1. When lnum is invalid -1 is returned.

localtime()  Return the current time, measured as seconds since 1st Jan 1970.

maparg(name[, mode])  Return the rhs of mapping name in mode mode. When there is no mapping for name, an empty String is returned. These characters can be used for mode:

- “n” Normal
- “v” Visual
- “o” Operator-pending
- “i” Insert
- “c” Cmd-line
- ”” Normal, Visual and Operator-pending

When mode is omitted, the modes from ”” are used. The name can have special key names, like in the ‘:map’ command. The returned String has special characters translated like in the output of the ‘:map’ command listing.

mapcheck(name[, mode])  Check if there is a mapping that matches with name in mode mode. When there is no mapping that matches with name, and empty String is returned. If there is one, the rhs of that mapping is returned. If there are several matches, the rhs of one of them is returned. This function can be used to check if a mapping can be added without being ambiguous.

match(expr, pat)  The result is a Number, which gives the index in expr where pat matches. If there is no match, -1 is returned. See pattern for the patterns that are accepted.

matchend(expr, pat)  Same as match(), but return the index of first character after the match.

matchstr(expr, pat)  Same as match(), but return the matched string.

nr2char(expr)  Return a string consisting of a single character with the ASCII value expr.

rename(from, to)  Rename the file by the name from to the name to. This should also work to move files across file systems. The result is a Number, which is 0 if the file was renamed successfully, and non-zero when the renaming failed.

setline(lnum, line)  Set line lnum of the current buffer to line. If this succeeds, 0 is returned. If this fails (most likely lnum is invalid) 1 is returned.

strftime(format[, time])  The result is a String, which is the current date and time, as specified by the format string. See the manual page of the C function strftime() for the format. The maximum length of the result is 80 characters.

strlen(expr)  The result is a Number, which is the length of the String expr.

strpart(src, start, len)  The result is a String, which is part of src, starting from character start, with the length len. When characters beyond the length of the string are implied, this doesn’t result in an error, the characters are simply omitted.

strtrans(expr)  The result is a String, which is expr with all unprintable characters translated into printable characters.

substitute(expr, pat, sub, flags)  The result is a String, which is a copy of expr, in which the first match of pat is replaced with sub. This works like the :substitute command (without any flags). But the magic option is ignored, the pat is always processed as if magic is set. When pat does not match in expr, expr is
returned unmodified. When flags is g, all matches of pat in expr are replaced. Otherwise flags should be "".

**synID(line, col, trans)** The result is a Number, which is the syntax ID at the position line and col in the current window. The syntax ID can be used with synIDattr() and synIDtrans() to obtain syntax information about text. col is 1 for the leftmost column, line is 1 for the first line. When trans is non-zero, transparent items are reduced to the item that they reveal. This is useful when wanting to know the effective color. When trans is zero, the transparent item is returned. This is useful when wanting to know which syntax item is effective (e.g. inside parentheses).

**synIDattr(synID, what) [, mode]** The result is a String, which is the what attribute of syntax ID synID. This can be used to obtain information about a syntax item. mode can be gui, cterm or term, to get the attributes for that mode.

**synIDtrans(synID)** The result is a Number, which is the translated syntax ID of synID. This is the syntax group ID of what is being used to highlight the character. Highlight links are followed.

**system(expr)** Get the output of the shell command expr. Note: newlines in expr may cause the command to fail. This is not to be used for interactive commands. The result is a String. To make the result more system-independent, the shell output is filtered to replace <CR> with <NL> for Macintosh, and <CR><NL> with <NL> for DOS-like systems.

**tempname()** The result is a String, which is the name of a file that doesn’t exist. It can be used for a temporary file. The name is different for each least 26 consecutive calls a unique file.

**visualmode()** The result is a String, which describes the last Visual mode used. Initially it returns an empty string, but once Visual mode has been used, it returns “v”, “V”, or “CTRL-V” (a single CTRL-V character) for character-wise, line-wise, or block-wise Visual mode respectively.

**virtcol(expr)** The result is a Number, which is the screen column of the file position given by expr. The column number is return as if the screen were of infinite width. If there is a <Tab> at that position, the returned Number is the last column occupied by the <Tab>. For example, for a <Tab> in column 1, with ts set to 8, it returns 8; The accepted positions are:

```
  . the cursor position
  ’x  position of mark “x” (if the mark is not set, 0 is returned)
```

Only marks in the current file can be used.

**winbufnr(n)** The result is a Number, which is the number of the buffer associated with window n. When n is zero, the number of the buffer in the current window is returned. When window n doesn’t exist, −1 is returned.

**winheight(n)** The result is a Number, which is the height of window nr. When n is zero, the height of the current window is returned. When window n doesn’t exist, −1 is returned. An existing window always has a height of zero or more.

**winnr()** The result is a Number, which is the number of the current window. The top window has number 1.

**Feature-list:**
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_builtin_terms</td>
<td>all builtin terminals enabled.</td>
</tr>
<tr>
<td>amiga</td>
<td>Amiga version of VIM.</td>
</tr>
<tr>
<td>arp</td>
<td>ARP support (Amiga).</td>
</tr>
<tr>
<td>autocmd</td>
<td>autocommands support.</td>
</tr>
<tr>
<td>beos</td>
<td>BeOS version of VIM.</td>
</tr>
<tr>
<td>browse</td>
<td>:browse support, and browse() will work.</td>
</tr>
<tr>
<td>builtin_terms</td>
<td>some builtin terminals.</td>
</tr>
<tr>
<td>byte_offset</td>
<td>support for “o” in statusline</td>
</tr>
<tr>
<td>cindent</td>
<td>cindent support.</td>
</tr>
<tr>
<td>clipboard</td>
<td>clipboard support.</td>
</tr>
<tr>
<td>cmdline_compl</td>
<td>cmdline-completion support.</td>
</tr>
<tr>
<td>cmdline_info</td>
<td>showcmd and ruler support.</td>
</tr>
<tr>
<td>comments</td>
<td>comments support.</td>
</tr>
<tr>
<td>cryptv</td>
<td>encryption support encryption.</td>
</tr>
<tr>
<td>cscope</td>
<td>:cscope support.</td>
</tr>
<tr>
<td>compatible</td>
<td>Compiled to be very Vi compatible.</td>
</tr>
<tr>
<td>debug</td>
<td>DEBUG defined.</td>
</tr>
<tr>
<td>dialog_con</td>
<td>console dialog support.</td>
</tr>
<tr>
<td>dialog_gui</td>
<td>GUI dialog support.</td>
</tr>
<tr>
<td>digraphs</td>
<td>support for digraphs.</td>
</tr>
<tr>
<td>dos32</td>
<td>32 bits DOS (DJGPP) version of VIM.</td>
</tr>
<tr>
<td>dos16</td>
<td>16 bits DOS version of VIM.</td>
</tr>
<tr>
<td>emacs-tags</td>
<td>support for Emacs tags.</td>
</tr>
<tr>
<td>eval</td>
<td>expression evaluation support.</td>
</tr>
<tr>
<td>ex_extra</td>
<td>extra Ex commands.</td>
</tr>
<tr>
<td>extra_search</td>
<td>support for incsearch and hlsearch</td>
</tr>
<tr>
<td>farsi</td>
<td>Farsi support (farsi).</td>
</tr>
<tr>
<td>file_in_path</td>
<td>support for gf and &lt;cfile&gt;</td>
</tr>
<tr>
<td>find_in_path</td>
<td>support for include file searches</td>
</tr>
<tr>
<td>fname_case</td>
<td>Case in file names matters (Unix only).</td>
</tr>
<tr>
<td>fork</td>
<td>Compiled to use fork() /exec() instead of system()</td>
</tr>
<tr>
<td>gui</td>
<td>GUI enabled.</td>
</tr>
<tr>
<td>gui_athena</td>
<td>Athena GUI.</td>
</tr>
<tr>
<td>gui_beos</td>
<td>BeOs GUI.</td>
</tr>
<tr>
<td>gui_gtk</td>
<td>GTK+ GUI.</td>
</tr>
<tr>
<td>gui_mac</td>
<td>Macintosh GUI.</td>
</tr>
<tr>
<td>gui_motif</td>
<td>Motif GUI.</td>
</tr>
<tr>
<td>gui_win32</td>
<td>MS Windows Win32 GUI.</td>
</tr>
<tr>
<td>gui_win32s</td>
<td>ibid, and Win32s system being used (Windows 3.1)</td>
</tr>
<tr>
<td>gui_running</td>
<td>VIM is running in the GUI, or it will start soon.</td>
</tr>
<tr>
<td>hangul_input</td>
<td>Hangul input support.</td>
</tr>
<tr>
<td>insert_expand</td>
<td>support for CTRL-X expansion commands in Insert mode.</td>
</tr>
<tr>
<td>langmap</td>
<td>langmap support.</td>
</tr>
<tr>
<td>linebreak</td>
<td>linebreak, breakat and showbreak</td>
</tr>
<tr>
<td>support. lispindent</td>
<td>support for lisp indenting.</td>
</tr>
<tr>
<td>mac</td>
<td>Macintosh version of VIM.</td>
</tr>
<tr>
<td>menu</td>
<td>support for :menu.</td>
</tr>
<tr>
<td>mksession</td>
<td>support for :mksession.</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>modify_fname</td>
<td>file name modifiers.</td>
</tr>
<tr>
<td>mouse</td>
<td>support mouse.</td>
</tr>
<tr>
<td>mouse_dec</td>
<td>support for Dec terminal mouse.</td>
</tr>
<tr>
<td>mouse_gpm</td>
<td>support for gpm (Linux console mouse)</td>
</tr>
<tr>
<td>mouse_netterm</td>
<td>support for netterm mouse.</td>
</tr>
<tr>
<td>mouse_xterm</td>
<td>support for xterm mouse.</td>
</tr>
<tr>
<td>multi_byte</td>
<td>support for Korean et al.</td>
</tr>
<tr>
<td>multi_byte_ime</td>
<td>support for IME input method</td>
</tr>
<tr>
<td>ole</td>
<td>OLE automation support for Win32.</td>
</tr>
<tr>
<td>os2</td>
<td>OS/2 version of Vim.</td>
</tr>
<tr>
<td>osfiletype</td>
<td>support for osfiletypes.</td>
</tr>
<tr>
<td>perl</td>
<td>Perl interface.</td>
</tr>
<tr>
<td>python</td>
<td>Python interface.</td>
</tr>
<tr>
<td>quickfix</td>
<td>quickfix support.</td>
</tr>
<tr>
<td>rightleft</td>
<td>rightleft support.</td>
</tr>
<tr>
<td>scrollbind</td>
<td>scrollbind support.</td>
</tr>
<tr>
<td>smartindent</td>
<td>smartindent support.</td>
</tr>
<tr>
<td>sniff</td>
<td>SniFF interface support.</td>
</tr>
<tr>
<td>statusline</td>
<td>support for statusline, rulerformat and special formats of titlestring and</td>
</tr>
<tr>
<td></td>
<td>iconstring.</td>
</tr>
<tr>
<td>syntax</td>
<td>syntax highlighting support.</td>
</tr>
<tr>
<td>syntax_items</td>
<td>There are active syntax highlighting items for the current buffer.</td>
</tr>
<tr>
<td>system</td>
<td>Compiled to use system() instead of fork()/exec().</td>
</tr>
<tr>
<td>tag_binary</td>
<td>binary searching in tags files.</td>
</tr>
<tr>
<td>tag_old_static</td>
<td>support for old static tags.</td>
</tr>
<tr>
<td>tag_any_white</td>
<td>support for any white characters in tags files.</td>
</tr>
<tr>
<td>tcl</td>
<td>Tcl interface.</td>
</tr>
<tr>
<td>terminfo</td>
<td>terminfo instead of termcap.</td>
</tr>
<tr>
<td>textobjects</td>
<td>support for text-objects.</td>
</tr>
<tr>
<td>tgetent</td>
<td>tgetent support, able to use a termcap or terminfo file.</td>
</tr>
<tr>
<td>title</td>
<td>window title support title.</td>
</tr>
<tr>
<td>unix</td>
<td>Unix version of VIM.</td>
</tr>
<tr>
<td>user-commands</td>
<td>User-defined commands.</td>
</tr>
<tr>
<td>viminfo</td>
<td>viminfo support.</td>
</tr>
<tr>
<td>vim_starting</td>
<td>True while initial source’ing takes place.</td>
</tr>
<tr>
<td>visualextra</td>
<td>extra Visual mode commands.</td>
</tr>
<tr>
<td>vms</td>
<td>VMS version of Vim.</td>
</tr>
<tr>
<td>wildignore</td>
<td>wildignore option.</td>
</tr>
<tr>
<td>win32</td>
<td>Win32 version of VIM (Windows 95/NT).</td>
</tr>
<tr>
<td>wildmenu</td>
<td>wildmenu option.</td>
</tr>
<tr>
<td>wildignore</td>
<td>wildignore option.</td>
</tr>
<tr>
<td>winaltkeys</td>
<td>winaltkeys option.</td>
</tr>
<tr>
<td>win16</td>
<td>Win16 version of Vim (Windows 3.1).</td>
</tr>
<tr>
<td>win32</td>
<td>Win32 version of Vim (Windows 95/NT).</td>
</tr>
<tr>
<td>writebackup</td>
<td>writebackup default on.</td>
</tr>
<tr>
<td>xim</td>
<td>X input method support.</td>
</tr>
<tr>
<td>xfontset</td>
<td>X fontset support.</td>
</tr>
<tr>
<td>xterm_clipboard</td>
<td>support for xterm clipboard.</td>
</tr>
</tbody>
</table>
9.4 User-Defined Functions

New functions can be defined. They can be called with “Name ( )”, just like built-in functions. The name must start with an uppercase letter, to avoid confusion with builtin functions.

:fu[nction] List all functions and their arguments.
:fu[nction] name List function name.
:fu[nction][!] name ([arguments]) [range] [abort] Define a new function by the name name. The name must be made of alphanumeric characters and underscore, and must start with a capital. An argument can be defined by giving its name. In the function this can then be used as “a:name” (“a:” for argument). Up to 20 arguments can be given, separated by commas. An argument “...” can be specified, which means that more arguments may be following. In the function they can be used as “a:1”, “a:2”, etc. “a:0” is set to the number of extra arguments (which can be 0). When not using “...”, the number of arguments in a function call must be equal the number of named arguments. When using “...”, the number of arguments may be larger. The body of the function follows in the next lines, until “:endfunction”. When a function by this name already exists and [!] is not used an error message is given. When [!] is used, an existing function is silently replaced. When the range argument is added, the function is expected to take care of a range itself. The range is passed as “a:firstline” and “a:lastline”. If range is excluded, a “:call” with a range will call the function for each line, with the cursor on the start of each line. When the [abort] argument is added, the function will abort as soon as an error is detected.

:endf[uunction] The end of a function definition.
:delf[uunction] name Delete function name.
:retu[rn] [expr] Return from a function. When expr is given, it is evaluated and returned as the result of the function. If expr is not given, the number 0 is returned. When a function ends without an explicit “:return”, the number 0 is returned.

Inside a function variables can be used. These are local variables, which will disappear when the function returns. Global variables need to be accessed with g:.

9.5 Commands

:let var-name = expr Set internal variable var-name to the result of the expression expr. The variable will get the type from the expr. If var-name didn’t exist yet, it is created.
:let $env-name = expr Set environment variable env-name to the result of the expression expr. The type is always String.
:let @reg-name = expr Write the result of the expression expr in register reg-name. reg-name must be a single letter, and must be the name of a writable register. “@@” can be used for the unnamed register. If the result of expr ends in a <CR> or <NL>, the register will be linewise, otherwise it will be set to characterwise.
let &option-name = expr  Set option option-name to the result of the expression expr. The type of the option is always used.

:unl[et][!] var-name  Remove the internal variable var-name. Several variable names can be given, they are all removed. With [!] no error message is given for non-existing variables.

:if expr ... :endif  Execute the commands until the next matching :else or :endif if expr evaluates to non-zero. Note: from VIM version 4.5 until 5.0, every Ex command between the :if and :endif is ignored.

:else  Execute the commands until the next matching :else or :endif if they were not already being executed.

:elseif[!] expr  Short for :else :if, with the addition that there is no extra :endif.

:while expr ... :endwhile  Repeat the commands between :while and :endwhile, as long as expr evaluates to non-zero. When an error is detected from a command inside the loop, execution continues after the :endwhile.

Note: The :append and :insert commands don’t work properly inside a :while loop.

:continue  When used inside a :while, jumps back to the :while.

:break  When used inside a :while, skips to the command after the matching :endwhile.

:echo expr ...  Echoes each expr, with a space in between and a terminating <EOL>. See also :comment.

:echon expr ...  Echoes each expr, without anything added. Also see :comment.

:echohl name  Use the highlight group name for the following :echo[n] commands.

:execute expr ...  Executes the string that results from the evaluation of expr as an Ex command. Multiple arguments are concatenated, with a space in between.

Note: :execute, :echo and :echon cannot be followed by a comment directly, because they see the " as the start of a string. But, you, however, can use “|” followed by a comment.

10  GUI

10.1 Mouse Control

The mouse only works if the appropriate flag in the mouse option is set. When the GUI is switched on, the mouse option is set to a, enabling it for all modes except for the “hit return to continue” message. This can be changed from the gvimrc file. A quick way to set these is with the “:behave” command.

:behave model  set behavior for mouse and selection. Valid arguments are: mswin (MS-Windows behavior) and xterm (Xterm behavior)

Using “:behave” changes these options:

<table>
<thead>
<tr>
<th>option</th>
<th>mswin</th>
<th>xterm</th>
<th>option</th>
<th>mswin</th>
<th>xterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectmode</td>
<td>mouse,key</td>
<td>–</td>
<td>mousemodel</td>
<td>popup</td>
<td>extend</td>
</tr>
<tr>
<td>keymodel</td>
<td>startsel,stopsel</td>
<td>–</td>
<td>selection</td>
<td>exclusive</td>
<td>inclusive</td>
</tr>
</tbody>
</table>
Visual Selection with Mouse

The mouse can be used to start a selection. How depends on the mousemodel option: If selectmode contains mouse, then the selection will be in Select mode. This means that typing normal text will replace the selection. Otherwise, the selection will be in Visual mode.

Right button: Click the right button to extend the visual selection to the position pointed to with the mouse. In Visual mode the closest end will be extended, otherwise Visual mode is started and extends from the old cursor position to the new one.

Left button: Double clicking may be done to make the selection word-wise, triple clicking makes it line-wise, and quadruple clicking makes it rectangular block-wise.

X11 vs. Win32 GUI

X11 GUI: In Visual mode, the highlighted text may be pasted into other windows. Likewise, the selected text from other windows may be pasted into VIM in Normal mode, Insert mode, or on the Command line by clicking the middle mouse button.

Win32 GUI: Visually selected text is only copied to the clipboard when using a y command, or another operator when the “*” register is used.

Other Text Selection with Mouse

In Command-line mode, at the hit-return prompt or if the mouse option is turned off, a different kind of selection is used: the left button selects, the right button extends the selection and the middle one pastes the text back.

Various Mouse Clicks

Left or right click on the status line makes that window current. Drag the status line to resize the windows above and below.

- S-LeftMouse  Search forward for the word under the mouse click.
- S-RightMouse  Search backward for the word under the mouse click.
- C-LeftMouse  Jump to the tag name under the mouse click.
- C-RightMouse  Jump back to position before the previous tag jump

GUI Selections

A special register “*” is used for storing GUI selection. Nothing is put in there unless the information about what text is selected is about to change, or when another application wants to paste the selected text. Similarly, when we want to paste a selection from another application, the selection is put in the “*” register first, and then put like any other register.

Note: when pasting text from one VIM into another separate VIM, the type of selection (character, line, or block) will also be copied.

Mouse Mappings

The mouse events, complete with modifiers, may be mapped.

```
Example

:map <S-LeftMouse> <RightMouse>
:map <S-LeftDrag> <RightDrag>
```

Note: Mouse mapping with modifiers does not work for xterm-like selection.
10.2 Window Position

Vim tries to make the window fit on the screen when it starts up. This avoids that you can’t see part of it. You can change the height that is used for the window title and a task bar with the guiheadroom option.

:\winpos[os] Display current position of the top left corner of the GUI vim window in pixels. Does not work in all versions.

:\winpos[os] \(X\ Y\) Put the GUI vim window at the given \(X\) and \(Y\) coordinates. The coordinates should specify the position in pixels of the top left corner of the window. Does not work in all versions.

10.3 Menus

The default menus are read from the file “\$VIMRUNTIME/menu.vim”. Motif and Win32 GUIs support Tear-off menus.

Creating New Menus

:\me \:menu \:noreme \:noremenu \:am \:amenu \:an \:anoremenu
:\nme \:nmenu \:nnoreme \:nnoremenu \:ome \:omenu \:onoreme \:onoremenu
:\vme \:vmenu \:vnoreme \:vnoremenu \:ime \:imenu \:inoreme \:inoremenu
:\cme \:cmenu \:cnoreme \:cnoremenu

To create a new menu item, use the \:menu commands. They work exactly like the \:map set of commands but the first argument is a menu item name, given as a path of menus and sub-menus with a “.” between them.

The \:amenu command can be used to define menu entries for all modes at once. To make the command work correctly, a character is automatically inserted for modes: Normal ⇒ nothing, Insert ⇒ <CTRL-O>, Cmdline ⇒ <CTRL-C>, Visual ⇒ <Esc>, Op-pending ⇒ <Esc>.

**Careful:** In Insert mode this only works for a SINGLE Normal mode command, because of the [CTRL-O]. If you have two or more commands, you will need to use the \:imenu command.

Special characters in a menu name:

& The next character is the shortcut key. Make sure each shortcut key is only used once in a (sub)menu.

<Tab> Separates the menu name from right-aligned text. This can be used to show the equivalent typed command.

Menu-priority

The position of a menu item on the menu bar is determined by its “priority”. The priority is given as a number before the \:menu command. Menus with a higher priority go more to the right. When no priority is given, 500 is used. The highest possible priority is about 32000. Currently, you can only give a priority for the location of the menu in the menu bar, not for the location of a menu item in a menu. The default menus have these priorities: File → 10; Edit → 20; Tools → 40; Syntax → 50; Buffers → 60; Window → 70; Help → 9999; The same mechanism can be used to position a submenu. The priority is then given as a dot-separated list of priorities, before the menu name.
Toolbar

Currently, the toolbar is only available in the Win32 and gtk+ GUI. It should turn up in other
GUIs in due course. The display of the toolbar is controlled by the guioptions letter T. The toolbar
is defined as a special menu called ToolBar, which only has one level.

Tooltips & Menu tips

These are currently only supported for the Win32 GUI.

:tmenu menupath rhs Define a tip for a menu or tool. When a tip is defined for a menu item,
it appears in the command-line area when the mouse is over that item. When a tip is defined
for a toolbar item, it appears as a tooltip when the mouse pauses over that button.
:tunmenu menupath Remove a tip for a menu or tool.

Showing What Menus Are Mapped To

To see what an existing menu is mapped to, use just one argument after the menu commands
(similar to the :map commands). If the menu specified is a submenu, then all menus in that
hierarchy will be shown. If no argument is given after :menu at all, then ALL menu items are
shown for the appropriate mode (e.g. Command-line mode for :cmenu).

Note: while entering a menu name after a menu command, Tab may be used to complete the
name of the menu item.

Deleting Menus

:unme :unmenu :aun :aunmenu :unme :unmenu
:cunme :cunmenu

To delete a menu item or a whole submenu, use the :unmenu commands, which are analogous to
the :unmap commands. To remove all menus use:

:unmenu "*" remove all menus in Normal and visual mode
:unmenu! "*" remove all menus in Insert and Command-line mode

10.4 Miscellaneous

This section describes other features which are related to the GUI.
- Typing ^V followed by a special key in the GUI will insert <Key>, since the internal string
  used is meaningless. Modifiers may also be held down to get <Modifiers-Key>.
- In the GUI, the modifiers <SHIFT>, <CTRL>, and <ALT> (or <META>) may be used
  within mappings of special keys and mouse events
- In the GUI, several normal keys may have modifiers in mappings etc, these are <Space>,
  <Tab>, <NL>, <CR>, <Esc>.
- Executing an external command from the GUI will not always work. “Normal” commands
  like ls, grep and make mostly work fine. Commands that require an intelligent terminal
  like less and ispell won’t work. Some may even hang and need to be killed from
  another terminal. For the X11 GUI the external commands are executed inside the gvim
  window. For the Win32 GUI the external commands are executed in a separate window.
- Normally, Vim takes control of all `<Alt>-<Key>` combinations, to increase the number of possible mappings. This clashes with the standard use of Alt in Win32 as the key for accessing menus. The quick way of getting standard behavior is to set the `winaltkeys` option to `yes`. This however prevents you from mapping <Alt> keys at all. Another way is to set `winaltkeys` to `menu`. Menu shortcut keys are then handled by windows, other ALT keys can be mapped. This doesn’t allow a dependency on the current state though. To get round this, the `:simalt key` command allows Vim (when `winaltkeys` is not set to `yes`) to fake a Windows-style <Alt> keypress.

This example shows how to add and remove a menu item for the keyword under the cursor. The register z is used.

```
:menu Words.Add\ Var wb"zye:menu! Words.<C-R>z <C-R>z<CR>
:menu Words.Remove\ Var wb"zye:unmenu! Words.<C-R>z<CR>
```

### 11 Syntax highlighting 😊

Syntax highlighting provides the possibility of showing parts of the text in another font or color. To start using syntax highlighting, type this command: `:syntax on`. This will enable automatic syntax highlighting. The type of highlighting will be selected using the file name extension, and sometimes using the first line of the file. The name of the active syntax is stored in the “current_syntax” variable.

#### 11.1 Syntax files

The syntax and highlighting commands for one language are normally stored in a syntax file named “`name.vim`”, where `name` is the [abbreviated] name of the language. The syntax file can contain any Ex commands.

**Naming Conventions**

To allow each user to pick his favorite set of colors, a set of pre-defined names for highlight groups common for many languages has been chosen. These are the preferred names for different highlight groups:

<table>
<thead>
<tr>
<th>Name</th>
<th>Used for:</th>
<th>Name</th>
<th>Used for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Comment</td>
<td>any comment</td>
<td>*Constant</td>
<td>any constant</td>
</tr>
<tr>
<td>*Error</td>
<td>any erroneous construct</td>
<td>*Identifier</td>
<td>any variable name</td>
</tr>
<tr>
<td>*Ignore</td>
<td>left blank, hidden</td>
<td>*PreProc</td>
<td>generic Preprocessor</td>
</tr>
<tr>
<td>*Special</td>
<td>any special symbol</td>
<td>*Statement</td>
<td>any statement</td>
</tr>
<tr>
<td>*Todo</td>
<td>anything that needs extra attention</td>
<td>*Type</td>
<td>int, long, char...</td>
</tr>
<tr>
<td>Boolean</td>
<td>a boolean constant</td>
<td>Character</td>
<td>a character constant</td>
</tr>
<tr>
<td>Conditional</td>
<td>if, then, else ...</td>
<td>Debug</td>
<td>debugging statements</td>
</tr>
<tr>
<td>Define</td>
<td>#define</td>
<td>Delimiter</td>
<td>delimiting character</td>
</tr>
<tr>
<td>Exception</td>
<td>try, catch, throw</td>
<td>Float</td>
<td>a floating point constant</td>
</tr>
<tr>
<td>Name</td>
<td>Used for:</td>
<td>Name</td>
<td>Used for:</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Function</td>
<td>function and class method names</td>
<td>Include</td>
<td>#include</td>
</tr>
<tr>
<td>Keyword</td>
<td>any other keyword</td>
<td>Label</td>
<td>case, default</td>
</tr>
<tr>
<td>Macro</td>
<td>same as Define</td>
<td>Number</td>
<td>a number constant</td>
</tr>
<tr>
<td>Operator</td>
<td>sizeof, +, * ...</td>
<td>PreCond</td>
<td>#if, #else ...</td>
</tr>
<tr>
<td>Repeat</td>
<td>for, do, while ...</td>
<td>SpecialChar</td>
<td>special character in a constant</td>
</tr>
<tr>
<td>SpecialComment</td>
<td>special things inside a comment</td>
<td>StorageClass</td>
<td>static, register ...</td>
</tr>
<tr>
<td>String</td>
<td>a string constant</td>
<td>Structure</td>
<td>struct, union, enum ...</td>
</tr>
<tr>
<td>Tag</td>
<td>use [CTRL-] on this</td>
<td>Typedef</td>
<td>a typedef</td>
</tr>
</tbody>
</table>

The names marked with “*” are the preferred groups, the other are minor groups. For the preferred groups, the “syntax.vim” file contains default highlighting. The highlight group names are not case sensitive. The following names are reserved and cannot be used as a group name: NONE ALL ALLBUT contains contained.

### 11.2 Defining a syntax

VIM understands three types of syntax items:

- **keyword** can only contain keyword characters, defined by the `iskeyword` option. Keywords cannot contain other syntax items. It will only be recognized when there is a complete match (there are no keyword characters before or after the match), e.g. if would match in `if(a=b)`, but not in `ifdef x`.
- **match** a match with a single regexp pattern. It must be within one line.
- **region** starts at a match of the `start` regexp pattern and ends with a match of the `end` regexp pattern. A `skip` regexp pattern can be used to avoid matching the `end` pattern.

Several syntax items can be put into one syntax group. For a syntax group you can provide highlighting attributes. You are free to make a highlight group for one syntax item, or to put all items into one group. In case where more than one item matches at the same position, the one that was defined last wins. A keyword always goes before a match or region. A keyword with matching case always goes before a keyword with ignoring case.

`:syntax case [match|ignore]` defines whether the following `:syntax` commands will work with matching case, when using `match`, or with ignoring case, when using `ignore`. Note that any items before this are not affected, and all items until the next `:syntax case` command are affected.

`:syntax keyword group-name [options] keyword ... [options]` defines a number of keywords, where:

- `group-name` – syntax group name, e.g. `Comment`.
- `options` – See “Syntax arguments” below.
- `keyword` ... – list of keywords which belong to this group.

The `options` can be given anywhere in the line. They will apply to all keywords given, also for options that come after a keyword. When you have a keyword with an optional tail, like Ex
commands in VIM, you can put the optional characters inside [], to define all the variations at once.

A keyword always has higher priority than a match or region; the keyword is used if more than one item matches. Keywords do not nest and a keyword can’t contain anything else. The maximum length of a keyword is 80 characters. The same keyword can be defined multiple times, when its containment differs.

```vim
:syntax match group-name [options] [excludenl] pattern [options]
```
defines one match, where

- `pattern` is the search pattern that defines the match. `excludenl` don’t make a pattern with the end-of-line “$” extend a containing match or item. Only useful for end patterns.

```vim
:syntax region group-name [options] [matchgroup=group_name] [keepend] start = start_pattern
...[skip = skip_pattern] end = end_pattern ...[options]
```
defines one region, where:

- `[matchgroup=group-name]` – the syntax group to use for the following `start` or `end` pattern matches only. Not used for the text in between the matched `start` and `end` patterns. Use NONE to reset to not using a different group for the `start` or `end` match.
- `keepend` – doesn’t allow contained matches to go past a match with the `end` pattern.
- `start=start_pattern` – the search pattern that defines the start of the region.
- `skip=skip_pattern` – the search pattern that defines text inside the region where not to look for the `end` pattern.
- `end=end_pattern` – the search pattern that defines the end of the region.

The `start/skip/end` patterns and the options can be given in any order. There can be zero or one skip pattern. There must be one or more start and end patterns.

### Cleaning up

```vim
:syntax clear
```
switches off syntax highlighting. It’s a good idea to include this command at the beginning of a syntax file.

```vim
:syntax off
```
disables syntax highlighting for all buffers

```vim
:syntax clear sync-group-name ...
```
removes all patterns and keywords for `group-name` in the current buffer.

### Listing syntax items

```vim
:syntax [list]
```
lists all the syntax items

```vim
:syntax list group-name
```
shows the syntax items for one syntax group

```vim
:syntax list grouplist-name
```
shows the syntax groups for one group list

### 11.3 Syntax arguments

The `:syntax` commands that define syntax items take a number of arguments. The common ones are explained here. The arguments may be given in any order and may be mixed with the patterns.

- `contained` when the `contained` argument is given, this item will not be recognized at the top level, but only when it is mentioned in the `contains` field of another match.
- `transparent` if the `transparent` argument is given, this item will not be highlighted itself, but will take the highlighting of the item it is contained in. This is useful for syntax items that don’t need any highlighting but are used only to skip over a part of the text. The same groups as the item it is contained in are used, unless a `contains` argument is given too.
oneline  the oneline argument indicates that the region does not cross a line boundary. It must match completely in the current line. However, when the region has a contained item that does cross a line boundary, it continues on the next line anyway. A contained item can be used to recognize a line continuation pattern.

contains=groupname,...  the contains argument is followed by a list of syntax group names. These groups will be allowed to begin inside the item (they may extend past the containing group’s end). This allows for recursive nesting of matches and regions. If there is no contains argument, no groups will be contained in this item. The group names do not need to be defined before they can be used here.

contains=ALL  if the only item in the contains list is ALL, then all groups will be accepted inside the item.

contains=ALLBUT, group-name,...  if the first item in the contains list is ALLBUT, then all groups will be accepted inside the item, except the ones that are listed, and the contained items.

The group-name in the contains list can be a pattern. All group names that match the pattern will be included (or excluded, if ALLBUT is used). The pattern cannot contain white space or a comma.

nextgroup=groupname,...  the nextgroup argument is followed by a list of syntax group names, separated by commas (just like with contains, so you can also use patterns). If the nextgroup argument is given, the mentioned syntax groups will be tried for a match, after the match or region ends. If none of the groups match, highlighting continues normally. If there is a match, this group will be used, even when it is not mentioned in the contains field of the current group. This is like giving the mentioned group priority over all other groups

skipwhite  skip over <Space> and <Tab> characters. When skipwhite is present, the white space is only skipped if there is no next group that matches the white space.

skipnl  skip over the end of a line. When skipnl is present, the match with nextgroup may be found in the next line. This only happens when the current item ends at the end of the current line. When skipnl is not present, the nextgroup will only be found after the current item in the same line.

skipempty  skip over empty lines (implies a skipnl)

Note: the skipwhite, skipnl and skipempty are only used in combination with nextgroup.

11.4 Syntax patterns

In the syntax commands, a pattern must be surrounded by two identical characters (delimiters). Syntax patterns are always interpreted as if the magic option is set and the “1” flag is not included in cpoptions. The pattern can be followed by a character offset, which can be used to change the highlighted part and to change the text area included in the match or region. Note: no white space is allowed between the pattern and the character offset(s).

The offset takes the form of what=offset, where what can be one of six strings:

- ms  Match Start   offset for the start of the matched text
- me  Match End     offset for the end of the matched text
- hs  Highlight Start offset for where the highlighting starts
- he  Highlight End  offset for where the highlighting ends
- rs  Region Start  offset for where the body of a region starts
- re  Region End    offset for where the body of a region ends
- lc  Leading Context offset past “leading context” of pattern

The offset can be:
Leading context

The \texttt{lc} offset specifies a leading context: a part of the pattern that must be present, but is not considered part of the match. An offset of \texttt{lc=n} will cause \textsc{vim} to step back \texttt{n} columns before attempting the pattern match, allowing characters which have already been matched in previous patterns to also be used as the leading context for this match.

The \texttt{ms} offset is automatically set to the same value as the \texttt{lc} offset, unless you set \texttt{ms} explicitly.

11.5 Synchronizing

\texttt{:sy[n]tax sync [ccomment [group-name] | minlines=\texttt{N} | \ldots]}

There are three ways to synchronize. For all three methods, the line range within which the parsing can start is limited by \texttt{minlines} and \texttt{maxlines}.

If the \texttt{minlines=\texttt{N}} argument is given, the parsing always starts at least that many lines backwards. This can be used if the parsing may take a few lines before it’s correct, or when it’s not possible to use syncing.

If the \texttt{maxlines=\texttt{N}} argument is given, the number of lines that are searched for a comment or syncing pattern is restricted to \texttt{N} lines backwards (after adding \texttt{minlines}). This is useful if you have few things to sync on and a slow machine.

First syncing method:

For the first method, only the \texttt{ccomment} argument needs to be given. When \textsc{vim} finds that the line where displaying starts is inside a C-style comment, the first region syntax item with the group-name \texttt{Comment} will be used.

The \texttt{maxlines} argument can be used to restrict the search to a number of lines. The \texttt{minlines} argument can be used to start at least a number of lines back (e.g., for when there is some construct that only takes a few lines, but is hard to sync on).

Second syncing method:

For the second method, only the \texttt{minlines=\texttt{N}} argument needs to be given. \textsc{vim} will subtract \texttt{N} from the line number and start parsing there. This means \texttt{N} extra lines need to be parsed, which makes this method a bit slower.

Note: \texttt{lines} and \texttt{minlines} are equivalent.

Third syncing method:

The idea is to synchronize on the end of a few specific regions, called a sync pattern. The search starts in the line just above the one where redrawing starts. From there the search continues backwards in the file.
A line continuation pattern can be given here. It is used to decide which group of lines need to be searched as if they were a single line. This means that the search for a match with the specified items starts in the first of the following that contains the continuation pattern.

When a match with a sync pattern is found, the rest of the line (or group of adjacent lines) is searched for another match. The last match is used. This is used when a line can contain both the start and the end of a region (e.g., in a C-comment like /* this */, the last */ is used).

There are two ways how a match with a sync pattern can be used:

- Parsing for highlighting starts where redrawing starts (and where the search for the sync pattern started). The syntax group that is expected to be valid there must be specified. This works well when the regions that cross lines cannot contain other regions.

- Parsing for highlighting continues just after the match. The syntax group that is expected to be present just after the match must be specified. This can be used when the previous method doesn’t work well. It’s much slower, because more text needs to be parsed.

Both types of sync patterns can be used at the same time. Besides the sync patterns, other matches and regions can be specified, to avoid finding unwanted matches.

`:syntax` sync match `group-name groupthere sync-group-name ...` define a match that is used for syncing. `sync-group-name` is the name of a syntax group that follows just after the match. Parsing of the text for highlighting starts just after the match. A region must exist for this sync-group-name. The first one defined will be used. `NONE` can be used for when there is no syntax group after the match.

`:syntax` sync match `group-name groupthere sync-group-name ...` like `groupthere`, but `sync-group-name` is the name of a syntax group that is to be used at the start of the line where searching for the sync point started. The text between the match and the start of the sync pattern searching is assumed not to change the syntax highlighting.

`:syntax` sync match ..., :syntax sync region ... define a region or match that is skipped while searching for a sync point.

`:syntax` sync linecont `pattern` when `pattern` matches in a line, it is considered to continue in the next line. This means that the search for a sync point will consider the lines to be concatenated.

If the `maxlines=N` argument is given too, the number of lines that are searched for a match is restricted to `N`.

Clearing syntax

`:syntax` sync clear clears all sync settings
`:syntax` sync clear `group-name ...` clears specific sync patterns

11.6 Highlight command

There are two types of highlight groups:

- The groups used for specific languages. For these the name starts with the name of the language. Many of these don’t have any attributes, but are linked to a group of the second type.

- The groups used for all languages. These are also used for the `highlight` option.

`:highlight` list all the current highlight groups that have attributes set.
`:highlight` `group-name` list one highlight group.
```
:highlight clear group-name, :highlight group-name NONE  disable the highlighting for one
highlight group.

:highlight group-name key=arg ...  add a highlight group, or change the highlighting for an
existing group.

Highlight arguments for black and white terminals (vt100, xterm)

term=attr-list  attr-list is a comma separated list (without spaces) of the following items (in any
order): NONE (no attributes used), bold, underline, reverse, inverse (same as re-
verse), italic, standout.

start=term-list, stop=term-list  These lists of terminal codes can be used to get non-standard at-
tributes on a terminal.

The escape sequence specified with the start argument is written before the characters in the
highlighted area. It can be anything that you want to send to the terminal to highlight this
area. The escape sequence specified with the stop argument is written after the highlighted
area. This should undo the start argument.

The term-list can have two forms:

- A string with escape sequences. This is any string of characters, except that it can’t start
  with t_ and blanks are not allowed. The <> notation is recognized here, so you can use
  things like <Esc> and <Space>.

- A list of terminal codes. Each terminal code has the form t_xx, where xx is the name
  of the termcap entry. The codes have to be separated with commas.

  Note: white space is not allowed.

Default highlight group

These are the default highlighting groups. These groups are used by the highlight option
default.

Cursor      the character under the cursor
Directory   directory names (and other special names in listings)
ErrorMsg    error messages
IncSearch   incsearch highlighting
ModeMsg     showmode message
MoreMsg     more-prompt
NonText     ~ and at the end of the window and characters from
Question    hit-return prompt and yes/no questions
SpecialKey  Meta and special keys listed with “:map”
StatusLine  status line of current window
StatusLineNC status lines of not-current windows
Title       titles for output from :set all, “:autocmd” etc.
Visual      Visual mode selection
WarningMsg  warning messages
LineNr      line number for “:number” and “#:” commands, and when number
Normal      normal text
Search      last search pattern highlighting (see hlsearch)
```
 Highlight arguments for color terminals (MS-DOS console, color-xterm)

cterm=attr-list  The cterm argument is likely to be different from term, when colors are used. For example, in a normal terminal comments could be underlined, in a color terminal they can be made Blue.

Note: Many terminals (e.g., DOS console) can’t mix these attributes with coloring. Use only one of cterm= OR ctermfg= OR ctermbg=.

ctermfg=color-num, ctermbg=color-num  The color-num argument is a color number. It ranges from zero to the number given by the termcap entry “Co” (non-inclusive). The actual color with this number depends on the type of terminal and its settings. Sometimes the color also depends on the settings of cterm. For example, on some systems cterm=bold ctermfg=3 gives another color, on others you just get color 3.

The following names are recognized, with the color number used:

<table>
<thead>
<tr>
<th>NR-16</th>
<th>NR-8</th>
<th>Color Name</th>
<th>NR-16</th>
<th>NR-8</th>
<th>Color Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Black</td>
<td>8</td>
<td>0*</td>
<td>DarkGray</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>DarkBlue</td>
<td>9</td>
<td>4*</td>
<td>Blue, LightBlue</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>DarkGreen</td>
<td>10</td>
<td>2*</td>
<td>Green, LightGreen</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>DarkCyan</td>
<td>11</td>
<td>6*</td>
<td>Cyan, LightCyan</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>DarkRed</td>
<td>12</td>
<td>1*</td>
<td>Red, LightRed</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>DarkMagenta</td>
<td>13</td>
<td>5*</td>
<td>Magenta, LightMagenta</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>Brown</td>
<td>14</td>
<td>3*</td>
<td>Yellow</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>LightGray(^{13}), Gray</td>
<td>15</td>
<td>7*</td>
<td>White</td>
</tr>
</tbody>
</table>

The number under NR-16 is used for 16-color terminals (“t_Co” greater than or equal to 16). The number under NR-8 is used for 8-color terminals (“t_Co” less than 16). The “*” indicates that the bold attribute is set for ctermfg. In many 8-color terminals (e.g. linux), this causes the bright colors to appear. This doesn’t work for background colors. The case of the color names is ignored.

 Highlight arguments for the GUI

gui=attr-list  These give the attributes to use in the GUI mode. Note that bold can be also sed here and by specifying a bold font. It has the same effect.

font=font-name  font-name is the name of a font as it is used on the system VIM runs on. The font-name NONE can be used to revert to the default font. When setting the font for the Normal group, this becomes the default font (until the guifont option is changed; the last one set is used).

Note: all fonts used should be of the same character size as the default font!

guifg=color-name guibg=color-name  These give the foreground (guifg) and background (guibg) color to use in the GUI. There are a few special names:

NONE  no color (transparent)
background (bg)  use normal background color
foreground (fg)  use normal foreground color

You can also specify a color in the RGB format #rrggbb, where rr is the Red value, gg is the Blue value and bb is the Green value. All values are hexadecimal, range from 00 to ff.

\(^{13}\)Gray can be spelled as Grey
11.7 Linking groups

:hi[ghlight][!] link from-group to-group  If you want to use the same highlighting for several syntax groups, you can do this by linking these groups into one common highlight group, and give the color attributes only for that group.

Notes:
- If the from-group and/or to-group doesn’t exist, it is created. You don’t get an error message for a non-existent group.
- If the to-group is NONE, the link is removed from the from-group.
- As soon as you use a :highlight command for a linked group, the link is removed.
- If there are already highlight settings for the from-group, the link is not made, unless the ! is given. For a :highlight link command in a sourced file, you don’t get an error message. This can be used to skip links for groups that already have settings.

12 Automatic Commands

You can specify commands to be executed automatically for when reading or writing a file, when entering or leaving a buffer or window, and when exiting VIM.

12.1 Defining autocommands

:au[tocmd] [group] event pat [nested] cmd  Add cmd to the list of commands that will be automatically executed on event for a file matching pat. VIM always adds the cmd after existing autocommands, so that the autocommands execute in the order in which they were given.

The :autocmd command cannot be followed by another command, since any “|” is considered part of the command. Special characters (e.g. %, <cword>) in the :autocmd arguments are not expanded when the autocommand is defined. These will be expanded when the event is recognized, and the cmd is executed. When your “.vimrc” file is sourced twice, the autocommands will appear twice. To avoid this, put :autocmd! in your “.vimrc” file, before defining autocommands.

When the group argument is not given, VIM uses the current group (as defined with :autogroup); otherwise, VIM uses the group defined with [group].

Note: while testing autocommands, it might be useful to set verbose=9. This causes the executed autocommands to be echoed.

12.2 Removing autocommands

:au[tocmd]! [group] event pat [nested] cmd  Remove all autocommands associated with event and pat, and add the command cmd.

:au[tocmd]! [group] event pat  Remove all autocommands associated with event and pat.

:au[tocmd]! [group] * pat  Remove all autocommands associated with pat for all events.

:au[tocmd]! [group] event  Remove ALL autocommands for event.

:au[tocmd]! [group]  Remove ALL autocommands.

When the group argument is not given, VIM uses the current group (as defined with :autogroup); otherwise, VIM uses the group defined with group.
12.3 Listing autocommands

:au[tocmd] [group] event pat  Show the autocommands associated with event and pat.
:au[tocmd] [group] * pat    Show the autocommands associated with pat for all events.
:au[tocmd] [group] event  Show all autocommands for event.
:au[tocmd] [group]       Show all autocommands.

If you provide the group argument, VIM lists only the autocommands for group; otherwise, VIM lists the autocommands for ALL groups. Note that this argument behavior differs from that for defining and removing autocommands.

12.4 Events

The following events are recognized. Case is ignored; for example, BUFRead and bufread can be used instead of BufRead.

BufFilePre Before changing the name of the current buffer with the “:.file” command.
BufFilePost After changing the name of the current buffer with the “:.file” command.
BufNewFile When starting to edit a file that doesn’t exist. Can be used to read in a skeleton file.
BufReadPre When starting to edit a new buffer, before reading the file into the buffer. Not used if the file doesn’t exist.
BufRead or BufReadPost When starting to edit a new buffer, after reading the file into the buffer, before executing the modelines. This does NOT work for :r file. Not used when the file doesn’t exist. Also used after successfully recovering a file.
FileReadPre Before reading a file with a :read command.
FileReadPost After reading a file with a :read command. Note that VIM sets the ’[ and ’] marks to the first and last lines of the read. This can be used to operate on the lines just read.
FilterReadPre Before reading a file from a filter command. VIM checks the pattern against the name of the current buffer, not the name of the temporary file that is the output of the filter command.
FilterReadPost After reading a file from a filter command. VIM checks the pattern against the name of the current buffer as with FilterReadPre.
FileType When the filetype option has been set. <afile> can be used for the name of the file where this option was set, and <amatch> for the new value of filetype.
Syntax When the syntax option has been set. <afile> can be used for the name of the file where this option was set, and <amatch> for the new value of syntax.
StdinPre Before reading from stdin into the buffer. Only used when the “-” argument was used when VIM was started.
StdinPost After reading from stdin into the buffer, before executing the modelines. Only used when the “-” argument was used when VIM was started.
BufWrite or BufWritePre Before writing the whole buffer to a file.
BufWritePost After writing the whole buffer to a file (should undo the commands for BufWritePre).
FileWritePre Before writing to a file, when not writing the whole buffer.
FileWritePost After writing to a file, when not writing the whole buffer.
FileAppendPre Before appending to a file.
FileAppendPost  After appending to a file.

FilterWritePre  Before writing a file for a filter command. The file name of the current buffer is used to match with the pattern, not the name of the temporary file that is the input for the filter command.

FilterWritePost  After writing a file for a filter command. Like FilterWritePre, the file name of the current buffer is used.

FileChangedShell  After VIM runs a shell command and notices that the modification time of the current file has changed since editing started. Run in place of the “has been changed” message. See timestamp. Useful for reloading related buffers which are affected by a single command.

FocusGained  When Vim got input focus. Only for the GUI version and a few console versions where this can be detected.

FocusLost  When Vim lost input focus. Only for the GUI version and a few console versions where this can be detected.

CursorHold  When the user doesn’t press a key for the time specified with updatetime. Not re-triggered until the user has pressed a key (i.e. doesn’t fire every updatetime ms if you leave Vim to make some coffee. :) Note: Interactive commands and “**:normal**” cannot be used for this event.

BufEnter  After entering a buffer. Useful for setting options for a file type. Also executed when starting to edit a buffer, after the BufReadPost autocommands.

BufLeave  Before leaving to another buffer. Also when leaving or closing the current window and the new current window is not for the same buffer.

BufUnload  Before unloading a buffer. This is when the text in the buffer is going to be freed. This may be after a BufWritePost and before a BufDelete.

BufHidden  Just after a buffer has become hidden. That is, when there are no longer windows that show the buffer, but the buffer is not unloaded or deleted. NOTE: When this autocommand is executed, the current buffer “%” may be different from the buffer being unloaded `<afile>`.

BufCreate  Just after creating a new buffer. Also used just after a buffer has been renamed. NOTE: When this autocommand is executed, the current buffer “%” may be different from the buffer being deleted `<afile>`.

BufDelete  Before deleting a buffer from the buffer list. The BufUnload may be called first (if the buffer was loaded).

WinEnter  After entering another window. Not done for the first window, when VIM has been just started. Useful for setting the window height. If the window belongs to a different buffer from the one previously being edited, VIM executes the BufEnter autocommands after the WinEnter autocommands.

WinLeave  Before leaving a window. If the window to be entered next is for a different buffer, VIM executes the BufLeave autocommands before the WinLeave autocommands.

GUIEnter  After starting the GUI successfully, and after opening the window. It is triggered before VimEnter when using gvim. Can be used to position the window from a .gvimrc file:

VimEnter  After doing all the startup stuff, including loading .vimrc files, executing the “-c cmd” arguments, creating all windows and loading the buffers in them.

VimLeavePre  Before exiting Vim, just before writing the .viminfo file. This is executed only once, if there is a match with the name of what happens to be the current buffer when exiting.
**VimLeave**

Before exiting VIM, just before writing the `.viminfo` file.

**User**

Never executed automatically. To be used for autocommands that are only executed with :doautocmd.

**FileEncoding**

Fires off when you change the file encoding with `:set fileencoding`. Allows you to set up fonts or other language sensitive settings.

**TermChanged**

After the value of `term` was changed. Useful for re-loading the syntax file to update the colors, fonts and other terminal-dependent.

### 12.5 Patterns

The file pattern is tested for a match against the file name in one of two ways:

- When there is no “/” in the pattern, VIM checks for a match against only the tail part of the file name (without its leading directory path).

- When there is a “/” in the pattern, VIM checks for a match against the short file name (as you typed it) and the full file name (after expanding it to a full path, resolving symbolic links).

The pattern is interpreted like mostly used in file names. When the pattern starts with “/”, this does not mean it matches the root directory. It can match any “/” in the file name. To match the root directory, use “^/”.

For all systems the “/” character is used for path separator (even on MS-DOS and OS/2). This was done because the backslash is difficult to use in a pattern, and to make the autocommands portable across different systems.

Using “~” in a file name (for home directory) doesn’t work. Use a pattern that matches the full path name, for example “*home/user/.cshrc”.

### 12.6 Filetypes

On systems which support filetypes you can specify that a command should only be executed if the file is of a certain type. The actual type checking depends on which platform you are running Vim on. To use filetype checking in an autocmd you should put a list of types to match in angle brackets in place of a pattern.

To enable file type detection, use this command in your vimrc: `:filetype on`. This command will load the file `$VIMRUNTIME/filetype.vim`, which defines autocommands for the FileType event. If the file type is not found by the name, the file `$VIMRUNTIME/scripts.vim` is used to detect it from the contents of the file.

### 12.7 Groups

`:aug[roup] name` Define the autocmd group name for the following `:autocmd` commands. The name “end” or “END” selects the default group.

When no specific group is selected, VIM uses the default group. The default group does not have a name. You cannot execute the autocommands from the default group separately; you can execute them only by executing autocommands for all groups.

Normally, when executing autocommands automatically, VIM uses the autocommands for all groups. The group only matters when executing autocommands with `:doautocmd` or `:doautoall`, or when defining or deleting autocommands.

The group name can contain any characters except white space. The group name `end` is reserved (also in uppercase).
12.8 Executing autocmds

Autocommands can also be executed manually. This can be used after adjusting the autocommands, or when the wrong autocommands have been executed (file pattern match was wrong).

*Note:* there is currently no way to disable the autocommands.

```
:do[ autocmd ] [ group ] event [ fname ]
```

Apply the autocommands matching [fname] (default: current file name) for event to the current buffer. This can be used when the current file name does not match the right pattern, after changing settings, or to execute autocommands for a certain event. It’s possible to use this inside an autocommand too, so you can base the autocommands for one extension on another extension.

When the [group] argument is not given, VIM executes the autocommands for all groups. When the [group] argument is included, VIM executes only the matching autocommands for that group.

*Note:* if you use an undefined group name, VIM gives you an error message.

```
:doauto[ ill ] [ group ] event [ fname ]
```

Like :doautocmd, but apply the autocommands to each loaded buffer.

*Careful:* Don’t use this for autocommands that delete a buffer, change to another buffer or change the contents of a buffer; the result is unpredictable. It is only meant to perform autocommands that set options, change highlighting, and so on.

12.9 Using autocommands

Reading files

For reading files there are three possible pairs of events. VIM uses only one pair at a time:

- `BufNewFile`: start editing a non-existent file
- `BufReadPre`, `BufReadPost`: start editing an existing file
- `FilterReadPre`, `FilterReadPost`: read the temp file with filter output
- `FileReadPre`, `FileReadPost`: any other file read

Reading compressed files

```
:autocmd! BufReadPre,FileReadPre *.gz set bin
:autocmd BufReadPost,FileReadPost *.gz '/Admin/gunzip
:autocmd BufReadPost,FileReadPost *.gz set nobin
:autocmd BufReadPost,FileReadPost *.gz execute ":doautocmd BufReadPost" . %:r
```

Writing Files

For writing files there are four possible pairs of events. VIM uses only one pair at a time:

- `BufWritePre`, `BufWritePost`: write the whole buffer
- `FilterWritePre`, `FilterWritePost`: write to the temp file with filter input
- `FileAppendPre`, `FileAppendPost`: append to a file
- `FileWritePre`, `FileWritePost`: any other file write
Writing compressed files

Example

```
:autocmd! BufWritePost,FileWritePost *.gz !mv <afile> <afile>:r
:autocmd BufWritePost,FileWritePost *.gz !gzip <afile>:r
:autocmd! FileAppendPre *.gz !gunzip <afile>
:autocmd FileAppendPre *.gz !mv <afile>:r <afile>
:autocmd! FileAppendPost *.gz !mv <afile> <afile>:r
:autocmd FileAppendPost *.gz !gzip <afile>:r
```

Nesting

By default, autocommands do not nest. If you use :e or :w in an autocommand, VIM does not execute the BufRead and BufWrite autocommands for those commands. If you do want this, use the nested flag for those commands in which you want nesting. The nesting is limited to 10 levels to get out of recursive loops.

Order of execution

All matching autocommands will be executed in the order that they were specified. It is recommended that your first autocommand be used for all files by using "*" as the file pattern. This means that you can define defaults you like here for any settings, and if there is another matching autocommand it will override these. But if there is no other matching autocommand, then at least your default settings are recovered (if entering this file from another for which autocommands did match). Note that "*" will also match files starting with ".", unlike Unix shells.

Search Patterns

The search patterns are saved and restored, so that the autocommands do not change them. While executing autocommands, you can use search patterns normally, e.g. with the n command. After the autocommands finish, the patterns from before the autocommand execution are restored. This means that the strings highlighted with the hlsearch option are not affected by autocommands.

13 Miscellany

13.1 VIM modes

BASIC modes

Vim has six BASIC modes\textsuperscript{14}:

**Normal mode**
In Normal mode you can enter all the normal editor commands. If you start the editor you are in this mode. This is also known as command mode.

**Visual mode**
This is like Normal mode, but the movement commands extend a highlighted area. When a non-movement command is used, it is executed for the highlighted area.

**Select mode**
This looks most like the MS-Windows selection mode. Typing a printable character deletes the selection and starts Insert mode.

\textsuperscript{14}The type of the mode is shown on the status line if the showmode option is set
**Insert mode**
In Insert mode the text you type is inserted into the buffer.

**Command-line mode**
In Command-line mode (also called Cmdline mode) you can enter one line of text at the bottom of the window. This is for the Ex commands, “:”, the pattern search commands, “?” and “/”, and the filter command, “!”.

**Ex mode**
Like Command-line mode, but after entering a command you remain in Ex mode. Very limited editing of the command line.

**ADDITIONAL modes**
There are five ADDITIONAL modes:

**Operator-pending mode**
This is like Normal mode, but after an operator command has started, and Vim is waiting for a motion to specify the text that the operator will work on.

**Replace mode**
Replace mode is a special case of Insert mode. You can do the same things as in Insert mode, but for each character you enter, one character of the existing text is deleted.

**Insert Normal mode**
Entered when [CTRL-O] given in Insert mode. This is like Normal mode, but after executing one command Vim returns to Insert mode.

**Insert Visual mode**
Entered when starting a Visual selection from Insert mode. When the Visual selection ends, Vim returns to Insert mode.

**Insert Select mode**
Entered when starting Select mode from Insert mode. When the Select mode ends, Vim returns to Insert mode.

**Switching from mode to mode**
If for any reason you do not know which mode you are in, you can always get back to Normal mode by typing [Esc] twice.

<table>
<thead>
<tr>
<th>FROM TO</th>
<th>Normal</th>
<th>Visual</th>
<th>Select</th>
<th>Insert</th>
<th>Replace</th>
<th>Cmd-line</th>
<th>Ex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>–</td>
<td>v V</td>
<td>^V</td>
<td>–</td>
<td>–</td>
<td>R</td>
<td>: / ? ! Q</td>
</tr>
<tr>
<td>Visual</td>
<td>–</td>
<td>–</td>
<td>^G</td>
<td>c C</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Select</td>
<td>–</td>
<td>^O</td>
<td>^G</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>: –</td>
</tr>
<tr>
<td>Insert</td>
<td>–</td>
<td>Esc</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Replace</td>
<td>–</td>
<td>Esc</td>
<td>–</td>
<td>–</td>
<td>Ins</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Cmd-line</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>:start</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Ex</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**13.2 VIM registers**
There are nine types of VIM registers:
1. **Unnamed register ""**

This register is used to place all text deleted with the "d", "c", "s", "x" commands or copied with the yank command, regardless of whether or not a specific register was used (e.g. "xdd"). An exception is the _register: "_dd does not store the deleted text in any register. The contents of this register are used by any put command (p or P) which does not specify a register. It can be also accessed by the name "".

2. **Numbered registers "0"–"9**

These are filled with yank and delete commands. Register “0” is filled with the last yank command, unless another register was specified. Register “1” is filled with the text that was deleted by each delete or change command, unless another register was specified or the text is less than one line (text deleted with “x” or “dw” will be put in the small delete register). The contents of register “1” are put in “2”, “2” → “3”, and so forth. The content of register “9” is lost.

3. **Small delete register "-"**

This one is filled with delete commands that delete less than one line, except when another register was specified.

4. **Named registers "a"–"z and "A"–"Z**

These are only filled when you say so. They are named “a” to “z” normally. If you use an uppercase letter, the same register as with the lower case letter is used, but the text is appended to the previous content. With a lower case letter the previous content is lost.

5. **Read-only registers ":", ".", ":% and ":#**

They can only be used with the commands “p”, “P”, “:put” and with <CTRL-R>. 

":" Contains the last inserted text (the same as what is inserted with the insert mode commands <CTRL-A> and <CTRL-@>.

Note: this doesn’t work with <CTRL-R> on the command line.

":%" Contains the name of the current file.

":#" Contains the name of the alternate file.

":" Contains the last command line. It can be used with “@@:”, this repeats the last command line.

6. **Expression register "="**

This is not really a register that stores text, but a way to use an expression where a register can be used. It is read-only, you cannot put text into the expression register. After the "=" the cursor moves to the command line, where you can enter any expression. All normal command line editing commands are available, including a special history for expressions. When you end the command line by typing <CR>, the result of the expression is computed. If you end it with <ESC>, the expression is abandoned. If the entered command line is empty, the previous expression is used.
7. **Selection register */

This is used for storing and retrieving the selected text for the GUI.

If you use a `put` command without specifying a register, the register that was last written to is used (this is also the content of the `unnamed register`). If you are confused, use the "`.dis`" command to find out what will be `put`.

8. **Black hole register */

When writing to this register, nothing happens. This can be used to delete text without affecting the normal registers. When reading from this register, nothing is returned.

9. **Last search pattern register */

Contains the most recent search-pattern. This is used for `n` and `hlsearch`. 
VIM Distribution:

VIM is Charityware. Please, read VIM documentation for details.

VIM Guide © 1997-2000, Oleg Raisky <olrcc@scisun.sci.ccny.cuny.edu>
VIM Author, Bram Moolenaar <bram@vim.org>
Proofread by Jean Jordaan <rgo_anas@rgo.sun.ac.za>
PDF version by Nguyen-Dai Quy <quy@linuxbe.org>
VIM on WWW: http://www.vim.org
This document: http://scisun.sci.ccny.cuny.edu/~olrcc/vim/